What is JML?

- A behavioral interface specification language for Java
  - The syntactic interface specifies the signature of a method
  - The behavior interface specifies what should happen at runtime
- Combines the practicality of DBC languages and the formality of model-based specification language
Type Specification

- JML can be used to specify an abstract data type
  - Abstract fields can be specified using model or ghost fields
  - The behavior of each method can be specified using method specifications
  - Invariants and constraints can be specified to further refine the behavior of an ADT

Method Specification

- A method specification consists of three components
  - Precondition - a logic assertion that specifies the states in which a method can be called
  - Frame Axiom - the set of locations that can be updated
  - Normal post-condition - a logic assertion that specifies the states that may result from normal return
  - Exceptional post-condition - a logic assertion that specify the states that may result when an exception occurs
An Example

```java
package org.jmlspecs.samples.jmlrefman; // line 1
// line 2
public abstract class IntHeap { // line 3
// line 4
//@ public model non_null int [] elements; // line 5
// line 6
/*@ public normal_behavior // line 7
@   requires elements.length >= 1; // line 8
@   assignable \nothing; // line 9
@   ensures \result // line 10
@     == (\max int j; // line 11
@       0 <= j && j < elements.length; // line 12
@       elements[j]); // line 13
@*/ // line 14
public abstract /*@ pure @*/ int largest(); // line 15
// line 16
//@ ensures \result == elements.length; // line 17
public abstract /*@ pure @*/ int size(); // line 18
}; // line 19
```

Why JML?

- Supports the powerful concept of design by contract
  - Record details of method responsibilities
  - Help to assign blames
  - Avoids inefficient defensive checks
  - Supports modular reasoning
  - Enables automatic tool support
  - Can be used to document detailed designs
Defensive Checks

Suppose that you are writing a method to implement binary search. Can you think of any requirement that must be satisfied before calling this method?

```c
/*@ requires a != null
@              && (\forall int i;
@                          0 < i && i < a.length;
@                          a[i-1] <= a[i]);
@ */
int binarySearch (int[] a, int x) {
}
```

Why not just code?

Code contains all the information. So, why additional specification?
Tool Support

- **jmlc** - JML Compiler which can compile annotated Java programs into bytecode. The compiled code includes runtime assertion checking.
- **jmlunit** - A unit testing tool that combines JML and JUnit. The tool uses code generated by jmlc to check the test results.
- **jmldoc** - Generate documentation for annotated programs.
- **escjava** - Check JML annotations statically for potential bugs such as null pointer and out of array bounds.
- **jml** - A faster substitute of jmlc. It checks JML annotations but does not generate code.

JML

- **Introduction**
- **Fundamental Concepts**
- **JML Specifications**
- **Summary**
**Model and Ghost**

- A feature declared with `model` is only present for the purpose of specification
  - A model field is an abstraction of one or more concrete Java fields
  - Model methods and types are used to help specification
  - A model field is connected to one or more concrete fields, whereas model methods and types are simply imaginary

- A `ghost` field is also only present for the purpose of specification
  - Unlike a model field, a ghost field has no connection to concrete fields.

**Lightweight vs Heavyweight**

- JML allows both heavyweight and lightweight specs
  - A lightweight spec. only specifies important aspects that are of interest.
  - A heavyweight spec. can only omit parts of a specification when the default behavior is believed appropriate

- A specification is heavyweight if it starts with the following keywords: `normal_behavior`, `exception_behavior`, or `behavior`
Visibility (1)

- The context of an annotation is the smallest grammatical unit that has an attached visibility and that contains the annotation
  - For instance, the annotation context of a pre-/post-condition is the method

- An annotation cannot refer to names that are more hidden than the visibility of the annotation context
  - For instance, public clients should be able to see all the names in publicly visible annotations

Visibility (2)

- An expression in a public annotation context can refer to x only if x is declared as public.

- An expression in a protected annotation context can refer to x only if x is declared as public or protected, and x must be visible according to Java’s rules

- An expression in a default visibility annotation context can refer to x only if x is declared as public, protected, or with default visibility, and x must be visible according to Java’s rules

- An expression in a private visibility annotation context can refer to x only if x is visible according to Java’s rule.
Visibility (3)

public class PrivacyDemoLegalAndIllegal {
    public int pub;
    protected int prot;
    int def;
    private int priv;

    //@ public invariant pub > 0;
    //@ public invariant prot > 0;
    //@ public invariant def > 0;
    //@ public invariant priv < 0;

    //@ protected invariant pub > 1;
    //@ protected invariant prot > 1;
    //@ protected invariant def > 1;
    //@ protected invariant priv < 1;

    //@ invariant pub > 1;
    //@ invariant prot > 1;
    //@ invariant def > 1;
    //@ invariant priv < 1;

    //@ private invariant pub > 1;
    //@ private invariant prot > 1;
    //@ private invariant def > 1;
    //@ private invariant priv < 1;
}

JML

- Introduction
- Fundamental Concepts
- JML Specifications
- Summary
**Visible State**

- A visible state for an object is one that occurs at one of the following moments:
  - At the end of a non-helper constructor invocation
  - At the beginning of a non-helper finalizer invocation
  - At the beginning or end of a non-helper non-static non-finalizer method invocation
  - At the beginning or end of a non-helper static method invocation
  - When no constructor, destructor, non-static or static method invocation is in progress

- A visible state for a type T if it occurs after static initialization is complete and it is a visible state for some object of type T

**Invariant**

- Properties that must hold in all visible states
  - A method or constructor assumes an invariant if the invariant must hold in its pre-state.
  - A method or constructor establishes an invariant if the invariant must hold in its post-state.
  - A method or constructor preserves an invariant if the invariant is both assumed and established.
Static vs Instance Invariants

- Invariants can be declared **static** or **instance**
  - A static invariant cannot refer to the current object or any instance field or any non-static method
  - Instance variants must be established by the constructor of an object and must be preserved by all non-helper instance methods
  - Static invariants must be established by the static initialization of a class, and must be preserved by all non-helper constructors and methods

Checking Invariants

- Invariants can be checked as follows:
  - Each non-helper constructor of a class \( C \) must preserve the static invariants of \( C \) and must establish the instance invariant of the object under construction.
  - Each non-helper, non-static, non-finalizer method of a class \( C \) must preserve the static invariants of \( C \), and must preserve the instance invariants of the receiver object.
  - Each non-helper static method of a class \( C \) must preserve the static invariant of \( C \)
Example

```java
package org.jmlspecs.samples.jmlrefman;
public abstract class Invariant {
    boolean[] b;
    //@ invariant b != null && b.length == 6
    //@ assignable b;
    Invariant() {
        b = new boolean[6];
    }
}
```

Invariants and Exceptions

- **Invariant** should be preserved in the case of both normal termination and abrupt termination.
- If it is expected for an invariant to be violated in case of an exception, then it is suggested either to *strengthen* the precondition of the method or *weaken* the invariant.
- If a method has no side effects when it throws an exception, then it automatically preserves all invariants.
**Invariants and Inheritance**

- A class inherits all the instance invariants specified in its super classes and super interfaces.

**History Constraints**

- **History** constraints or constraints are restrictions that must hold between a combination of two visible states.

- A method respects a constraint if and only if its pre-state and post-state satisfy the constraint.
  - A constraint declaration may optionally list the methods that must respect the constraint.
  - otherwise, the constraint must be respected by all the non-helper methods.
**Example**

```java
package org.jmlspecs.samples.jmlrefman;

public abstract class Constraint {
    int a;
    //@ constraint a == \old(a);
    boolean[] b;
    //@ invariant b != null;
    //@ constraint b.length == \old(b.length);
    boolean[] c;
    //@ invariant c != null;
    //@ constraint c.length >= \old(c.length);
    //@ requires bLength >= 0 && cLength >= 0;
    Constraint(int bLength, int cLength) {
        b = new boolean[bLength];
        c = new boolean[cLength];
    }
}
```

---

**Invariants vs Constraints**

What are the differences between invariants and constraints?
A few points....

- Static constraints should be respected by all constructors and methods, and instance constraints must be respected by all instance methods.
- Constraints should be respected at both normal and abrupt termination.
- Constraints in a class are inherited by its subclasses.
- A method only has to respect a constraint only if its preconditions are satisfied.

Method Specification

- A method specification consists of three major components:
  - Pre-conditions, Post-conditions (normal and abrupt termination), and Frame conditions
- It can either be heavyweight or lightweight
Accessibility

- A **heavyweight** specification may be declared with an **explicit modifier**
  - The access modifier of a heavyweight specification case cannot allow more access than the method being specified
  - For instance, a `public` method may have a private specification, but a private method may not have a public specification

- A **lightweight** specification has the same accessibility as the method being specified

---

Example - LWS

```java
package org.jmlspecs.samples.jmlrefman;

public abstract class Lightweight {

    protected boolean P, Q, R;
    protected int X;

    /* @ requires P;
        @ assignable X;
        @ ensures Q;
        @ signals (Exception) R;
        @*/
    protected abstract int m() throws Exception;
}
```

---
Example - HWS

```java
package org.jmlspecs.samples.jmlrefman;

public abstract class Heavyweight {
    protected boolean P, Q, R;
    protected int X;

    /*@ protected behavior */
    /*@ requires P; @*/
    /*@ diverges \not_specified; @*/
    /*@ assignable X; @*/
    /*@ when \not_specified; @*/
    /*@ working_space \not_specified; @*/
    /*@ duration \not_specified; @*/
    /*@ ensures Q; @*/
    /*@ signals (Exception) R; @*/
    /*@ */
    protected abstract int m() throws Exception;
}
```

Behavior Spec. (1)

```java
behavior
    requires P;
    diverges D;
    assignable A;
    when W;
    ensures Q;
    signals (E1 e1) R1;
    ...
    signals (En en) Rn;
    also
code_contract
    accessible C
callable p ()
```
Behavior Spec. (2)

If a non-helper method is invoked in a pre-state that satisfies $P$ and all applicable invariants, then one of the following is true:

- JVM throws an error
- The method does not terminate and the predicate $D$ holds in the pre-state
- The method terminates normally or abnormally
  - Only locations that either did not exist in the pre-state, that are local to the method, or that are either named by $A$ or are dependents of such locations, can be assigned
  - In case of a normal return, $Q$ holds and so do all applicable invariants and constraints
  - In case of an exception of type $E_i$, $R_i$ holds and so do all applicable invariants and constraints
  - Only locations named in $C$ can be directly accessed by the method
  - Only methods named in $p$ are directly called by the method

Behavior Spec. (3)

The semantics for a specification case of a helper method is the same as that for a non-helper method, except that:

- The instance invariants for the current object and the static invariants for the current class are not assumed to hold in the pre-state, and do not have to be established in the post-state
- The instance constraints for the current object and the static constraints for the current class do not have to be established in the post-state
Normal Behavior Spec.

- A normal behavior spec. is just a behavior spec with an implicit signal clause: signals (java.lang.Exception) false;
  - A normal spec. is indicated by the keyword normal_behavior and cannot have any signals clause
  - Has the same semantics as the behavior spec. with the false signal clause

Exceptional Behavior Spec.

- A exceptional behavior spec. is just a behavior spec with an implicit ensures clause: ensures false;
  - An exceptional behavior spec. is indicated by the keyword exceptional_behavior and cannot have any ensures clause
  - Has the same semantics as the behavior spec. with the false ensures clause
Requires Clause

- A requires clause specifies a precondition of method or constructor.
- If there is no requires clause, then the default precondition is \not_specified for a lightweight spec., and is true for a heavyweight spec.

\begin{align*}
\text{requires P;} \\
\text{requires Q;} & \iff \text{requires P \&\& Q;}
\end{align*}

Ensures Clause

- An ensures clause specifies a normal postcondition of a method or constructor.
- An ensures clause may contain expressions of form \old(e), which is evaluated at the pre-state.
- If there is no ensures clause, then the default precondition is \not_specified for a lightweight spec., and is true for a heavyweight spec.

\begin{align*}
\text{ensures P;} \\
\text{ensures Q;} & \iff \text{ensures P \&\& Q;}
\end{align*}
**Signals Clause**

- A signals clause specifies an abnormal postcondition of a method or constructor.
- A signals clause may contain expressions of form \( \text{old}(e) \), which is evaluated at the pre-state.
- There can be multiple signal clauses, one for each possible exception.
- A signal clause specifies when an exception may be thrown, not when a certain exception must be thrown.

```plaintext
signals (E e) P;
```

**Diverges Clause (1)**

- A diverges clause specifies when a method may loop forever or otherwise not return to its caller.
- When a diverges clause is omitted, the default diverges condition is `\not\_specified` for a lightweight spec., and `false` for a heavyweight spec..
- The partial correctness of a method can be reasoned by setting the diverges condition to `true`. 
Diverges Clause (2)

```java
package org.jmlspecs.samples.jmlrefman;

public abstract class Diverges {
    /*@ public behavior
    @    diverges true;
    @    assignable \nothing;
    @    ensures false;
    @    signals (Exception) false;
    @*/
    public static void abort();
}
```

Assignable Clauses

- An assignable clause specifies that, from the client's point of view, only the locations named can be assigned by the method
  - Local locations and locations that are created by the method can always be assigned.
- For a lightweight spec., the default list is `\not_specified`; for a heavyweight spec., the default list is `\everything`.
- A pure method has an implicit assignable clause: `assignable \nothing`. 
**When Clause**

- A *when* clause specifies when a caller of a method will be delayed until the *when* condition holds.
- For a lightweight spec., the default when condition is `\not_specified`; for a heavyweight spec., the default when condition is `true`.
- A *when* clause is used to specify the concurrency aspects of a method. Currently, JML provides primitive support for concurrency.

---

**Built-in Operators (1)**

- `\result` - the return value, can only be used in an *ensures* clause of a non-void method
- `\old(e)` - the value obtained by evaluating e at the pre-state
- `\not_assigned (x, y, ...)` - asserts that a list of locations as well as their dependent locations are not assigned
- `\not_modified (x, y, ...)` - asserts that a list of locations are not assigned
- `\fresh (x, y, ..)` - asserts that objects are newly created, i.e., they do not exist in the pre-state
**Built-in Operators (2)**

- \texttt{\textbackslash nonnull\textbackslash elements} - asserts that an array and its elements are all non-null

\[
\text{nonnull\textbackslash elements(myArray)}
\]

\[
\text{myArray} \neq \text{null} \land \\
(\forall \text{int } i; \ 0 \leq i \land i < \text{myArray.length}; \\
\text{myArray}[i] \neq \text{null})
\]

**Built-in Operators (3)**

- \texttt{\textbackslash typeof} - returns the most-specific dynamic type of an expression
- \texttt{\textbackslash elem\textbackslash type} - returns the most-specific static type shared by all the elements of an array
- \texttt{\textbackslash type} - introduces the literals of type T:

\[
\text{\textbackslash type(PlusAccount) returns PlusAccount}
\]
**Built-in Operators (4)**

- \( \forall \) - the universal quantifier
- \( \exists \) - the existential quantifier

\[
\forall \text{ int } i, j; 0 \leq i \& \& i < j \& \& j < 10; a[i] < a[j]
\]

**Built-in Operators (5)**

- \( \max \) - returns the maximum of a range of values
- \( \min \) - returns the minimum of a range of values
- \( \text{product} \) - returns the product of a range of values
- \( \text{sum} \) - returns the sum of a range of values
- \( \text{num_of} \) - counts the number of the values that satisfy a certain condition
**Built-in Operators (6)**

\[
\begin{align*}
\sum \text{int } i; 0 \leq i \& \& i < 5; i) &= ? \\
\prod \text{int } i; 0 \leq i \& \& i < 5; i) &= ? \\
\max \text{int } i; 0 \leq i \& \& i < 5; i) &= ? \\
\min \text{int } i; 0 \leq i \& \& i < 5; i) &= ? \\
\text{num_of int } i; 0 \leq i \& \& i < 5; i < 3) &= ?
\end{align*}
\]

---

**Statements**

- An assert statement states that a condition must be true when the control flow reaches the statement.
- A set statement assigns a value to a ghost variable within annotations.
- A unreachable statement asserts that the control flow shall never reach a point.
- A debug statement helps debugging by allowing to execute an arbitrary Java expression within an annotation.
Example
//@ assert x > 0;
//@ set i = 0;
//@ set collection.elementType = \type(int);
if (true) {...} else { //@ unreachable; }
//@ debug System.out.println(x);
//@ debug x = x + 1;

JML
- Introduction
- Fundamental Concepts
- JML Specifications
- Summary
Summary

- What is JML? Why JML?
- What is a behavioral interface specification?
- How to specify the behavior of a type in JML?
- How to specify the behavior of a method in JML?
- What is a model field? And a ghost field? How do they differ?
- What is an invariant? And constraints? How do they differ?
- What is a lightweight spec.? And a heavyweight spec.? How do they differ?
- How to specify a normal behavior? How to specify an exceptional behavior?