



# Models are the 'M' in JML

Using ADT Models in  
Formal Specification with JML

Joseph Kiniry



# Models, not Modeling

- ⊛ the ‘M’ in JML is **not** the same as the ‘M’ in UML, even if both use the term ‘model’
- ⊛ JML models are mathematical abstractions
  - ⊛ UML models are pretty pictures
- ⊛ JML models are used to specify abstract behavior independent of implementation
- ⊛ an implementation realizes a model and is verified as fulfilling the model



# Standard Models

- ✧ standard mathematical models include:
  - ✧ bag, list, map, pair, relation, sequence, set
  - ✧ variants exist for values and objects
- ✧ standard Java models include:
  - ✧ Byte, Char, Double, Float, Integer, Long, Short, String, Type
  - ✧ Collection, Comparable, Enumeration, Iterator



# Mathematical Models

- ✧ each model is realized by one Java class
  - ✧ see the package *org.jmlspecs.models*
- ✧ all methods of all models are functional
- ✧ each model has a full specification
  - ✧ spec is in OO/ADT style
  - ✧ algebraic equational axiomatic spec
- ✧ NB no models have been verified yet!



# Java Models

- ⊛ all core classes have models
- ⊛ some of these models are quite simple (e.g., Byte, Char, Integer, and String)
- ⊛ others are quite complicated (e.g., Double and Float)



# Using Models

- ✧ models are used by declaring *model fields*
- ✧ one can also declare *model methods*
- ✧ in specifications, models are used in lieu of concrete fields when at all possible
- ✧ in implementations, models are bound to implementations with a *represents* clause
  - ✧ representations can be concrete fields or abstract pure method invocations



# Example Models: JMLString

```
public /*@ pure @*/ class JMLString implements JMLComparable {  
  
    /** The contents of this object. */  
    //@ public model String theString;  
    //@ public invariant theString != null;  
  
    protected String str_  
    //@           in theString;  
    //@ protected represents theString <- str_  
  
    //@ protected invariant str_ != null;  
}
```



# Example Models: JMLInteger

```
public /*@ pure @*/ class JMLInteger implements JMLComparable {  
  
    /** The integer value of this object. */  
    //@ public model int theInt;  
  
    //@ public constraint theInt == \old(theInt);  
  
    private int intValue;  
    //@          in theInt;  
    //@ private represents theInt <- intValue;  
}
```



# JMLInteger's remainderBy()

```
/** Return a new object containing the remainder of this object's
 * integer value divided by that of the given argument.
 */
/*@ public normal_behavior
 *   @ requires i2 != null && !i2.equals(new JMLInteger(0));
 *   @ ensures \result != null
 *   @         && \result.theInt == theInt % i2.theInt;
 */
public /*@ non_null */
    JMLInteger remainderBy(/*@ non_null */ JMLInteger i2) {
    //@ assume i2.intValue != 0;
    return new JMLInteger(intValue % i2.intValue);
}
```



# Issues with Models

- \* awkward to use
  - \* all operators are functional and are methods, thus an unfamiliar prefix-notation is necessary
  - \* all mathematical models are parameterized on a type, but since Java  $\leq 1.5$  has no parameterized classes, casting is frequent
- \* execution speed with jmlrac is very slow
  - \* particularly true of mathematical models



# Verifying with Models

- ✧ models with built-in types and functional representations work in ESC/Java2
- ✧ small models with richer types and functional representations sometimes work
  - ✧ primarily complexity issue with Simplify
- ✧ medium to large models with richer types do not work at all
  - ✧ currently revising core specifications to match ESC/Java2's current capabilities

