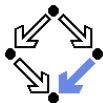
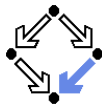


The Java Modeling Language (Part 2)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
<http://www.risc.jku.at>





JML Class Specifications

- Object invariants and history constraints.
 - `non_null`, `invariant`, `constraint`.
- Public versus private behavior.
 - `private normal_behavior`.
- Model fields and model representations.
 - `model`, `represents`.
- Data groups.
 - `in`, `maps ... \into`.
- Class refinements.
 - `refines`.

Support for programming in the large.

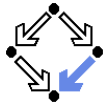
Practical Issues



Recommended use with JML-annotated Java files.

- First compile with `javac`.
 - Check syntactic and type correctness of Java source.
- Then compile with `jml`.
 - Check syntactic and type correctness of JML annotations.
- Then compile with `escjava2`.
 - Check semantic consistency of JML annotations.
 - More on ESC/Java2 later.

Errors can be made at each level.



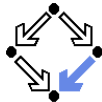
1. Basic Class Specifications

2. Classes for Modeling

3. Model-based Class Specifications

4. Rounding Things Up

A Java Class



```
class IntStack
{
    int[] stack;
    int number;

    final int N = 10;
    IntStack()
    {
        stack = new int[N];
        number = 0;
    }

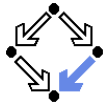
    boolean isempty()
    {
        return number == 0;
    }

    void push(int e)
    { if (number == stack.length)
      resize();
      stack[number] = e;
      number = number+1;
    }

    int pop()
    { number = number-1;
      return stack[number];
    }

    void resize()
    { int s[] = new int[2*stack.length+1];
      for (int i=0; i<stack.length; i++)
        s[i] = stack[i];
      stack = s;
    }
}
```

Object Invariants



```
class IntStack
{
    /*@ non_null */ int[] stack;
    int number;

    //@ invariant 0 <= number && number <= stack.length;
    ...
}
```

- A object invariant must hold **before and after** each method call.
 - Variable annotated by **non_null** must not be null.
 - Clause **invariant** specifies a general object invariant.
 - Private **/*@ helper */** method need not maintain invariant.

Every object invariant is automatically added to the pre- and to the postcondition of every (non-helper) method.

History Constraints

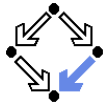


```
class IntStack
{
  ...
  // no method touches elements below the top of stack
  /*@ constraint (\forall int i; 0 <= i && i < number-1;
    @          stack[i] == \old(stack[i])); @*/
  ...
}
```

- A history constraint must hold for the pre/post-state **pair** of every method call.
 - A **constraint** condition may use **\old** to refer to the pre-state.

Every history constraint is added to the post-condition of every method.

Light-Weight Specification



```
class IntStack // V1
{
    ...
    final int N = 10;

    /*@ ensures stack.length == N
       @ && number == 0; @*/
    IntStack()
    { stack = new int[N];
      number = 0;
    }

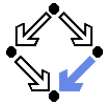
    /*@ ensures \result <==>
       @ number == 0; @*/
    boolean isempty()
    { return number == 0;
    }

    /*@ ensures number == \old(number)+1
       @ && stack[number-1] == e; @*/
    void push(int e)
    { if (number == stack.length)
      resize();
      stack[number] = e;
      number = number+1;
    }

    /*@ requires number > 0;
       @ ensures number == \old(number)-1
       @ && \result == stack[number]; @*/
    int pop()
    { number = number-1;
      return stack[number];
    }

    ...
}
```


Light-Weight Specification (Contd)



```
...

/*@ ensures stack.length > \old(stack.length)
   @ && number == \old(number)
   @ && (\forall int i;
   @     0 <= i && i < number;
   @     stack[i] == \old(stack[i])); @*/
void resize()
{ int s[] =
  new int[2*stack.length+1];
  for (int i=0; i<stack.length; i++)
    s[i] = stack[i];
  stack = s;
}
}
```

Problem: stack implementation is externally visible.

Private Implementation vs Public Interface



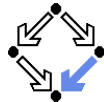
```
class IntStack
{
    private int stack[];
    private int number;
    private final int N = 10;

    public IntStack() { ... }
    public boolean isempty() { ... }
    public void push(int e) { ... }
    public int pop() { ... }

    private void resize() { ... }
}
```

Only selected methods should belong to the public interface.

Problem with Light-Weight Specification



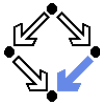
```
class IntStack
{
  private int stack[];
  private int number;
  private final int N = 10;

  /*@ ensures stack.length == N
   @ && number == 0; @*/
  public IntStack() { ... }
  ...
}
```

```
jml -Q IntStack.java
```

```
...
```

Field "stack" (private visibility) can not be referenced in a specification context of "package" visibility [JML]

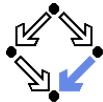


Visibility of Specifications

- Every JML specification has a visibility level.
 - Analogous to Java visibility levels.
 - Default, `private`, `protected`, `public`.
 - Light-weight specifications: default visibility.
 - Similar to `public` but restricted to package level.
- Heavy-weight specifications: visibility explicitly specified.
 - `public normal_behavior`, `private normal_behavior`.
- A specification may only access fields within its visibility.
 - Only `private` specifications may access `private` fields.

Need to use heavy-weight specifications.

Heavy-Weight Specification



```
class IntStack // V2
{
  private /*@ non_null @*/ int[] stack;
  private int number;

  /*@ private invariant 0 <= number
    @ && number <= stack.length; @*/

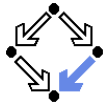
  /*@ private constraint
    @ (\forall int i;
    @   0 <= i && i < number-1;
    @   stack[i] == \old(stack[i]));
    @*/

  private final int N = 10;

  /*@ private normal_behavior
    @ assignable stack, number;
    @ ensures stack.length == N
    @   && number == 0; @*/
  public IntStack()
  { stack = new int[N];
    number = 0;
  }

  /*@ private normal_behavior
    @ assignable \nothing;
    @ ensures \result <==>
    @   number == 0; @*/
  public /*@ pure @*/
  boolean isempty()
  { return number == 0;
  }
  ...
}
```

Heavy-Weight Specification (Contd)



```
...
/*@ private normal_behavior
  @ assignable stack, stack[*], number;
  @ ensures number == \old(number)+1
  @ && stack[number-1] == e; @*/
public void push(int e)
{ if (number == stack.length)
  resize();
  stack[number] = e;
  number = number+1;
}

/*@ private normal_behavior
  @ requires number > 0;
  @ assignable number;
  @ ensures number == \old(number)-1
  @ && \result == stack[number]; @*/
public int pop()
{ number = number-1;
  return stack[number];
}

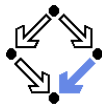
/*@ private normal_behavior
  @ assignable stack;
  @ ensures \fresh(stack)
  @ && stack.length >
  @ \old(stack.length)
  @ && number == \old(number)
  @ && (\forall int i;
  @ 0 <= i && i < number;
  @ stack[i] == \old(stack[i]));
private void resize()
{
  int s[] =
    new int[2*stack.length+1];
  for (int i=0; i<stack.length; i++)
    s[i] = stack[i];
  stack = s;
}
}
```

Heavy-Weight Specification: Considerations



- Visibility of invariants and history constraints.
 - `private invariant`, `private constraint`.
- Explicit frame conditions recommended: `assignable`.
 - Default: `assignable \everything`.
- New predicate: `\fresh(stack)`.
 - `stack` is newly allocated after `resize()`.
 - Thus assignment `stack[number] == ...` in `push` is legal.
 - Otherwise possible that `stack` refers after `resize()` to existing array.
 - Rule: assignment to location is legal in method if location appears in method `assignable` clause or if location is newly allocated in method.

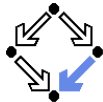
Private versus Public Specifications



Let us assess the current situation.

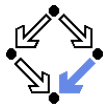
- We have constructed a **private** specification.
 - Refers to the private variables of the class.
 - Can be used in the context of the class implementation.
 - Cannot be used as a **contract** between the user and the implementor of the class.
- For use as a contract, we need a **public** specification.
 - May refer only to public class interface.
 - But this interface may be too restricted to express the desired behavior of the class.

We need a possibility to extend the public class interface for the purpose of specifying the behavior of the class.



-
1. Basic Class Specifications
 - 2. Classes for Modeling**
 3. Model-based Class Specifications
 4. Rounding Things Up

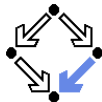
Model Fields



```
class C                                interface I
{                                        {
  //@ model T x;                          //@ instance model T x;
  //@ represents x <- E;                  //@ represents x <- E;
  ...                                    ...
}
```

- A **model** field is a **specification-only** field.
 - Considered as a normal field for the purpose of reasoning.
 - Actually not provided by the implementation.
 - In an interface, an **instance model** field, is considered a field of every class implementing the interface.
- A **represents** clause associates the model field to an implementation expression.
 - Describes how model field can be computed from actual fields.

Example



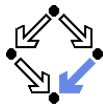
```
class IntStack
{
  private /*@ non_null @*/ int[] stack;
  private int number;

  /*@ model int len;
  /*@ represents len <- stack.length;

  /*@ invariant 0 <= number && number <= len;

  /*@ ensures len == N && number == 0; @*/
  IntStack()
  { stack = new int[N];
    number = 0;
  }
  ...
}
```

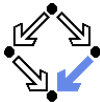
Class Specifications and Abstract Datatypes



How to specify the public behavior of a class (concrete datatype) C ?

- First mathematically axiomatize an **abstract datatype**.
 - Type name A and names of operations on A .
 - Laws (“axioms”) that the operations must obey.
- Then define C and an **abstraction function** $a : C \rightarrow A$
 - Maps a program object of type C to a mathematical object A .
 - Has as its inverse a **concretization relation** $c \subseteq A \times C$.
$$\forall x \in C : c(a(x), x) \wedge \forall y \in A : c(y, x) \Rightarrow x = a(y).$$
- Specify the methods of C in terms of the operations of A .
 - Instead of variable x of type C use term $a(x)$ of type A .
- Thus C becomes related to the well understood A .
 - Must prove that the methods satisfy the laws of the operations of A .

C.A.R. Hoare, 1972: Proof of Correctness of Data Representations.



An Abstract Datatype

The abstract datatype “integer stack”.

- Sort S .

- Operations

$empty : S, push : \mathbb{Z} \times S \rightarrow S, isempty : S \rightarrow \mathbb{B},$
 $top : S \rightarrow \mathbb{Z}, pop : S \rightarrow S.$

- $\forall s, s' \in S, x, x' \in \mathbb{Z} :$

- $empty \neq push(x, s);$

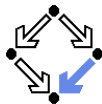
- $push(x, s) = push(x', s') \Rightarrow x = x' \wedge s = s';$

- $isempty(empty) = \text{true},$

- $isempty(push(x, s)) = \text{false};$

- $top(push(x, s)) = x;$

- $pop(push(x, s)) = s.$



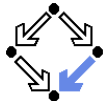
A Method Specification

Assume concrete type `Stack` and abstraction function $a : \text{Stack} \rightarrow S$.

- Input $s : \text{Stack}$.
- Input condition: $isempty(a(s)) = \text{false}$.
- Output $s' : \text{Stack}$.
- Output condition: $a(s') = pop(a(s))$.

The concrete method behaves like the abstract operation *pop*.

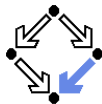
An Abstract Datatype in JML



```
public /*@ pure @*/ class IntStackModel
{
    // IntStackModel() is default constructor

    /*@ public model boolean isempty();
    /*@ public model IntStackModel push(int e);
    /*@ public model int top();
    /*@ public model IntStackModel pop();

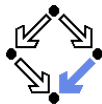
    /*@ axiom
    @ (\forall IntStackModel s, s2; s != null && s2 != null;
    @   (\forall int e, e2; ;
    @     !new IntStackModel().equals(s.push(e)) &&
    @     (s.push(e).equals(s2.push(e2)) ==> s.equals(s2) && e == e2) &&
    @     new IntStackModel().isempty() &&
    @     !s.push(e).isempty() &&
    @     e == s.push(e).top() &&
    @     s.equals(s.push(e).pop())));
    @*/
}
```



An Abstract Datatype in JML (Contd)

- A class tagged as **pure** contains only pure methods.
 - Convenient shortcut for classes describing abstract datatypes.
- A **model** method is a **specification-only** method.
 - Just for reasoning, no implementation provided.
 - Typically pure (but need not be).
 - Behavior described by axioms (or by model programs).
- `IntStackModel` is a “class for modeling”.
 - Intended for supporting specifications.
 - May use model methods without implementations.
 - Just for reasoning, no runtime checking possible.
 - May also provide method implementations.
 - Also runtime checking possible.

The JML tool suite comes with a library of pre-defined classes for modeling (but also for executing).



JML Classes for Modeling

- Package `org.jmlspecs.models.*`.
 - Directory `/zvol/formal/JML/org/jmlspecs/models`.
 - Container types:
 - `JMLObjectSet`, `JMLObjectBag`, `JMLObjectSequence`, ...
 - Numerical types:
 - `JMLInfiniteIntegerClass`, `JMLFiniteIntegerClass`, ...
- Most classes contain method implementations.
 - Useful for runtime checking.
- Usage primarily by `model import`.
 - Not linked to classes when compiled with `javac`.

```
//@ model import org.jmlspecs.models.*;
```

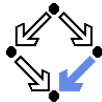
For examples, see “Leavens et al, 2004: Preliminary Design of JML”.

JML Model Classes



```
// file "IntStackModel.jml"
/*@ public pure model class IntStackModel
   @ {
   @   public model IntStackModel();
   @   public model boolean isempty();
   @   public model IntStackModel push(int e);
   @   public model int top();
   @   public model IntStackModel pop();
   @
   @   public axiom ...
   @ }
  @*/
```

- A **model** class is a **specification-only** class.
 - Just for reasoning, no implementation provided.



-
1. Basic Class Specifications
 2. Classes for Modeling
 - 3. Model-based Class Specifications**
 4. Rounding Things Up

Specifying the Public Behavior of a Class

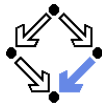


There are different styles to specify the public behavior of a class.

- Specify the public behavior in the class itself.
 - Class **adds** the public behavior to its private behavior.
- Specify the public behavior in an **abstract class**.
 - Class **inherits** from this abstract class.
- Specify the public behavior in an **interface**.
 - Class **implements** this interface.
- Specify the public behavior in an **JML specification file**.
 - Class **refines** this specification.

We will investigate these alternatives in turn.

Public Behavior in Class



```
class IntStack // V3
{
  ... // private int[] stack, int number;

  /*@ private invariant
    @ 0 <= number
    @ && number <= stack.length;

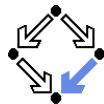
  /*@ private constraint
    @ (\forall int i;
    @ 0 <= i && i < number-1;
    @ stack[i] == \old(stack[i])); @*/

  /*@ public model
    @ non_null IntStackModel stackM;
    @ public initially stackM.isempty();
    @
    @ represents stackM <- toModel();
    @ public model
    @ pure IntStackModel toModel(); @*/

  /*@ public normal_behavior
    @ assignable stackM;
    @ ensures stackM.isempty();
    @ also private normal_behavior
    @ assignable stack, number;
    @ ensures stack.length == N
    @ && number == 0;
    @*/
  public IntStack()
  {
    stack = new int[N];
    number = 0;
  } /*@ nowarn Post;

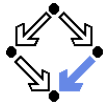
  ...
}
```

Public Behavior in Class: Considerations



- **initially** Clause:
 - Specification of initial value of model field.
- **model pure IntstackModel toModel()**
 - Pure function to convert this object to IntStackModel.
 - Implementation remains unspecified (later).
- **nowarn Post**
 - Since implementation of toModel is unspecified, ESC/Java2 cannot check postcondition of public behavior.
 - Unfortunately this also prevents checking of private behavior.
- **also ...**
 - Combine public behavior and private behavior.
 - Method must satisfy each behavior.
 - Problem with assignable clause of public behavior (later).

Public Behavior in Class (Contd)

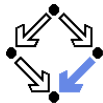


```
...
/*@ public normal_behavior
  @ assignable \nothing;
  @ ensures \result <==>
  @   stackM.isempty();
  @ also private normal_behavior
  @ assignable \nothing;
  @ ensures \result <==>
  @   number == 0;
  @*/
public /*@ pure @*/ boolean isempty()
{
  return number == 0;
} //@ nowarn Post;

/*@ public normal_behavior
  @ assignable stackM;
  @ ensures stackM ==
  @   \old(stackM.push(e));
  @ also private normal_behavior
  @ assignable
  @   stack, stack[*], number;
  @ ensures number ==
  @   \old(number)+1
  @   && stack[number-1] == e;
  @*/
public void push(int e)
{
  if (number == stack.length)
    resize();
  stack[number] = e;
  number = number+1;
} //@ nowarn Post;

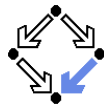
...
```

Public Behavior in Class (Contd'2)



```
...                               /*@ private normal_behavior
                                   @ assignable stack;
                                   @ ensures \fresh(stack)
                                   @ && stack.length >
                                   @   \old(stack.length)
                                   @ && number == \old(number)
                                   @ && (\forall int i;
                                   @   0 <= i && i < number;
                                   @   stack[i] ==
                                   @     \old(stack[i])); @*/
/*@ public normal_behavior        private void resize()
  @ requires !stackM.isempty();  {
  @ assignable stackM;           int s[] =
  @ ensures                       new int[2*stack.length+1];
  @   \result == \old(stackM.top());
  @ && stackM == \old(stackM.pop());
  @ also private normal_behavior  for (int i=0; i<stack.length; i++)
  @ requires number > 0;         s[i] = stack[i];
  @ assignable number;           stack = s;
  @ ensures number == \old(number)-1
  @ && \result == stack[number]; }
  @*/                             }
public int pop()                 }
{
  //@ assume number > 0;
  number = number-1;
  return stack[number];
} //@ nowarn Post;
```


Public Behavior in Class: Considerations



- `assume number > 0` in `pop()`
 - ESC/Java2 complains.
 - Due to the lack of the implementation of abstraction function, this cannot be deduced from the precondition of the public behavior.
- No separation of public and private behavior.
 - Both mixed in same file.

A messy solution.

Frame Condition of Public Behavior



```
/*@ public normal_behavior
   @ assignable stackM;
   @ ensures stackM.isempty();
   @ also private normal_behavior
   @ ...
   @*/
public IntStack()
{
    stack = new int[N];
    number = 0;
} //@ nowarn Post;
```

■ assignable stackM

- Frame condition says that only model field `stackM` may be changed.
- But actually concrete fields `stack` and `number` are changed.
- ESC/Java2 complains.

Need to relate model fields to concrete fields.

Data Groups



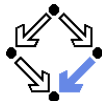
```
private /*@ non_null @*/ int[] stack; /*@ in stackM;  
/*@ maps stack[*] \into stackM;
```

```
private int number; /*@ in stackM;
```

- Declaration of field `stackM` also introduces a **data group** `stackM`.
 - A data group is a set of storage locations.
 - Initially, only the location of the declared variable is in data group.
- An **assignable** clause actually refers to data groups.
 - All storage locations in referenced data group may be changed.
- A data group may be extended.
 - **in stackM** adds declared variable to data group `stack`.
 - **maps stack[*] \into stackM** adds all elements of array `stack`.

By incorporation into the data group `stackM`, the variable `stack`, all elements of `stack` and `number` may change, when `stackM` may change.

Implementation of Abstraction Function



We have not yet defined the abstraction function `toModel()`.

```
/*@ public pure model IntStackModel toModel()
  @ {
  @   IntStackModel m = new IntStackModel();
  @   for (int i = 0; i < number; i++)
  @     m = m.push(stack[i]);
  @   return m;
  @ } @*/
```

- Practically useful for runtime checking.
 - Any reference to model variable `stackM` is replaced by `toModel()`.
 - Requires an implementation of (the methods of) `IntStackModel`.
- Principally useful for verification.
 - Requires a specification of `toModel` which uniquely determines `stackM` from `stack` and `number`.
 - Reasoner must be strong enough (ESC/Java2 is not).



Specification of Abstraction Function

```
/*@ also private normal_behavior
  @ ensures \result.length() == number
  @ && (\forall int i; 0 <= i && i < number;
  @   \result.elemAt(i) == stack[number-i-1]);
  @ public pure model IntStackModel toModel()
  @ {
  @   IntStackModel m = new IntStackModel();
  @   for (int i = 0; i < number; i++)
  @     m = m.push(stack[i]);
  @   return m;
  @ }
  @*/
```

Relates the elements of `stackM` to those of `stack`.

Generalization of Model Type

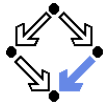


```
class IntStackModel
{
    ...
    //@ public model int length();
    //@ public model int elemAt(int i);

    /*@ public axiom
        @ (\forall IntStackModel s; s!= null;
          @   (\forall int e, i; ;
            @     new IntStackModel().length() == 0 &&
            @     s.push(e).length() == 1+s.length() &&
            @     s.elemAt(0) == s.top() &&
            @     s.elemAt(i+1) == s.pop().elemAt(i)));
        @*/
}
```

Recursive definition of `length` and of `elemAt`.

Public Behavior in Abstract Class



```
public abstract class IntStackBase // V4
{
    /*@ public model
       @ non_null IntStackModel stackM;
       @ public initially stackM.isempty();
       @
       @ represents stackM <- toModel();
       @ public model
       @ pure IntStackModel toModel();
       @*/

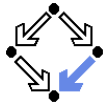
    /*@ public normal_behavior
       @ assignable stackM;
       @ ensures stackM.isempty();
       @*/
    public IntStackBase ()
    {
    } //@ nowarn Post, Invariant;
    // must be overridden

    /*@ public normal_behavior
       @ ensures \result <==>
       @ stackM.isempty(); @*/
    public abstract /*@ pure @*/
    boolean isempty();

    /*@ public normal_behavior
       @ assignable stackM;
       @ ensures stackM ==
       @ \old(stackM.push(e)); @*/
    public abstract void push(int e);

    /*@ public normal_behavior
       @ requires !isempty();
       @ assignable stackM;
       @ ensures \result ==
       @ \old(stackM.top())
       @ && stackM ==
       @ \old(stackM.pop()); @*/
    public abstract int pop();
}
```

Public Behavior in Abstract Class (Contd)



```
class IntStack extends IntStackBase
{
    private /*@ non_null @*/
        int[] stack; /*@ in stackM;
        /*@ maps stack[*] \into stackM;

    private int number; /*@ in stackM;

    /*@ private invariant
        @ 0 <= number
        @ && number <= stack.length; @*/

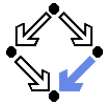
    /*@ private constraint
        @ (\forall int i;
        @     0 <= i && i < number-1;
        @     stack[i] == \old(stack[i]));
        @*/

    private final int N = 10;

    /*@ private normal_behavior
        @ assignable stackM,
        @     stack, number;
        @ ensures stack.length == N
        @     && number == 0;
        @ also public normal_behavior
        @ assignable stackM;
        @ ensures stackM.isempty(); @*/
    public IntStack()
    { stack = new int[N];
      number = 0;
    } /*@ nowarn Post, Invariant;

    ...
}
```


Public Behavior in Abstract Class (Contd'2)



```
...
/*@ also private normal_behavior
  @ assignable \nothing;
  @ ensures \result <==> number == 0; @*/
public /*@ pure @*/ boolean isempty()
{ return number == 0;
} //@ nowarn Post, Invariant;

/*@ also private normal_behavior
  @ assignable stack, stack[*], number;
  @ ensures number ==
  @   \old(number)+1
  @   && stack[number-1] == e; @*/
public void push(int e)
{
  if (number == stack.length)
    resize();
  stack[number] = e;
  number = number+1;
} //@ nowarn Post, Invariant;

/*@ also private normal_behavior
  @ requires number > 0;
  @ assignable number;
  @ ensures number ==
  @   \old(number)-1
  @   && \result ==
  @   stack[number];
  @*/
public int pop()
{
  //@ assume number > 0;
  number = number-1;
  return stack[number];
} //@ nowarn Post, Invariant;

...
```

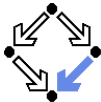
Public Behavior in Abstract Class (Contd'3)



...

```
/*@ private normal_behavior
   @ assignable stack;
   @ ensures \fresh(stack)
   @ && stack.length > \old(stack.length)
   @ && number == \old(number)
   @ && (\forall int i;
   @     0 <= i && i < number;
   @     stack[i] == \old(stack[i])); @*/
private void resize()
{
    int s[] = new int[2*stack.length+1];
    for (int i=0; i<stack.length; i++)
        s[i] = stack[i];
    stack = s;
}
}
```

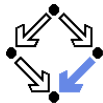
Public Behavior in Abs.Class: Considerations



- Clear separation of behaviors.
 - Public behavior in abstract superclass.
 - Private behavior in concrete subclass.
- `model stackM`
 - Model field inherited by any subclass of abstract class.
- Constructor must be specified in abstract class.
 - Abstract class always has default constructor.
- `also private normal_behavior`
 - Extension of public behavior by private behavior.
- `assignable stackM, ... in constructor IntStack()`
 - Frame condition of private behavior!
 - Constructor `IntStack()` calls constructor `InstStackBase()`.

Quite clean solution.

Public Behavior in Interface



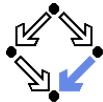
```
public interface IntStackInterface // V5
{
    /*@
     @ public instance model
     @ non_null IntStackModel stackM;
     @ public initially stackM.isempty();
     @
     @ represents stackM <- toModel();
     @ public model
     @ pure IntStackModel toModel();
     @*/

    /*@ public normal_behavior
     @ assignable \nothing;
     @ ensures \result <==>
     @ stackM.isempty();
     @*/
    public /*@ pure @*/ boolean isempty();

    /*@ public normal_behavior
     @ assignable stackM;
     @ ensures stackM ==
     @ \old(stackM.push(e));
     @*/
    public void push(int e);

    /*@ public normal_behavior
     @ requires !stackM.isempty();
     @ assignable stackM;
     @ ensures \result ==
     @ \old(stackM.top())
     @ && stackM ==
     @ \old(stackM.pop());
     @*/
    public int pop();
}
```

Public Behavior in Interface (Contd)



```
class IntStack implements IntStackInterface
{
    private /*@ non_null @*/ int[] stack;
    /*@ in stackM;
    /*@ maps stack[*] \into stackM;

    private int number; /*@ in stackM;

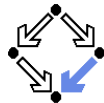
    /*@ private invariant 0 <= number
       @ && number <= stack.length; */

    /*@ private constraint
       @ (\forall int i;
       @     0 <= i && i < number-1;
       @     stack[i] == \old(stack[i]));
       @*/

    private final int N = 10;

    /*@ private normal_behavior
       @ assignable stack, number;
       @ ensures stack.length == N
       @ && number == 0;
       @ also public normal_behavior
       @ assignable stackM;
       @ ensures stackM.isEmpty();
       @*/
    public IntStack()
    {
        stack = new int[N];
        number = 0;
    } /*@ nowarn Post, Invariant;
    ...
}
```

Public Behavior in Interface (Contd'2)

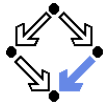


```
...
/*@ also private normal_behavior
  @ assignable \nothing;
  @ ensures \result <==> number == 0;
  @*/
public /*@ pure @*/ boolean isempty()
{ return number == 0;
} //@ nowarn Post, Invariant;

/*@ also private normal_behavior
  @ assignable stack, stack[*], number;
  @ ensures number == \old(number)+1
  @ && stack[number-1] == e; @*/
public void push(int e)
{ if (number == stack.length)2
    resize();
  stack[number] = e;
  number = number+1;
} //@ nowarn Post, Invariant;

/*@ also private normal_behavior
  @ requires number > 0;
  @ assignable number;
  @ ensures number ==
  @   \old(number)-1
  @ && \result == stack[number];
  @*/
public int pop()
{
  //@ assume number > 0;
  number = number-1;
  return stack[number];
} //@ nowarn Post, Invariant;
...
```

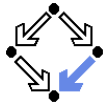
Public Behavior in Interface (Contd'3)



...

```
/*@ private normal_behavior
  @ assignable stack;
  @ ensures \fresh(stack)
  @ && stack.length > \old(stack.length)
  @ && number == \old(number)
  @ && (\forallall int i;
  @     0 <= i && i < number;
  @     stack[i] == \old(stack[i])); @*/
private void resize()
{ int s[] = new int[2*stack.length+1];
  for (int i=0; i<stack.length; i++)
    s[i] = stack[i];
  stack = s;
}
}
```

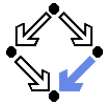
Public Behavior in Interface: Considerations



- Clear separation of behaviors.
 - Public behavior in interface.
 - Private behavior in class.
- `instance model stackM`
 - Model field of any class implementing the interface.
- No constructor in interface possible.
 - Both public and private behavior of constructor specified in class.
- `also private normal_behavior`
 - Extension of public behavior specified in interface by private behavior.

Rather clean solution.

Public Behavior in JML Specification File



```
// V6, file "IntStack.jml"
public class IntStack
{
  /*@ public model
   @ non_null IntStackModel stackM;
   @ public initially stackM.isempty();
   @
   @ represents stackM <- toModel();
   @ public model
   @ pure IntStackModel toModel(); @*/

  /*@ public normal_behavior
   @ assignable stackM;
   @ ensures stackM.isempty(); @*/
  public IntStack();

  /*@ public normal_behavior
   @ assignable \nothing;
   @ ensures \result <==> stackM.isempty(); @*/
  public /*@ pure @*/ boolean isempty();

  /*@ public normal_behavior
   @ assignable stackM;
   @ ensures stackM ==
   @   \old(stackM.push(e)); @*/
  public void push(int e);

  /*@ public normal_behavior
   @ requires !stackM.isempty();
   @ assignable stackM;
   @ ensures \result ==
   @   \old(stackM.top())
   @   && stackM ==
   @   \old(stackM.pop()); @*/
  public int pop();
}
```

Public Behavior in JML Spec. File (Contd)



```
//@ refine "IntStack.jml";
class IntStack
{
  private /*@ non_null @*/
    int[] stack; //@ in stackM;
  //@ maps stack[*] \into stackM;

  private int number; //@ in stackM;
  /*@ private invariant 0 <= number
    @ && number <= stack.length; @*/

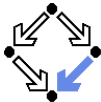
  /*@ private constraint
    @ (\forall int i;
    @   0 <= i && i < number-1;
    @   stack[i] == \old(stack[i])); @*/

  private final int N = 10;
```

```
/*@ also private normal_behavior
  @ assignable stack, number;
  @ ensures stack.length == N
  @ && number == 0; @*/
public IntStack()
{
  stack = new int[N];
  number = 0;
} //@ nowarn Post, Invariant;

/*@ also private normal_behavior
  @ assignable \nothing;
  @ ensures \result <==>
  @   number == 0; @*/
public /*@ pure @*/
boolean isempty()
{
  return number == 0;
} //@ nowarn Post, Invariant;
...
```

Public Behavior in JML Spec. File (Contd'2)



```
/*@ also private normal_behavior
   @ assignable stack, stack[*], number;
   @ ensures number == \old(number)+1
   @ && stack[number-1] == e; @*/
public void push(int e)
{ if (number == stack.length)
    resize();
  stack[number] = e;
  number = number+1;
} //@ nowarn Post, Invariant;

/*@ also private normal_behavior
   @ requires number > 0;
   @ assignable number;
   @ ensures number == \old(number)-1
   @ && \result == stack[number]; @*/
public int pop()
{ //@ assume number>0;
  number = number-1;
  return stack[number];
} //@ nowarn Post, Invariant;

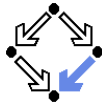
/*@ private normal_behavior
   @ assignable stack;
   @ ensures \fresh(stack)
   @ && stack.length >
   @ \old(stack.length)
   @ && number == \old(number)
   @ && (\forall int i;
   @ 0 <= i && i < number;
   @ stack[i] == \old(stack[i]));
private void resize()
{
  int s[] =
    new int[2*stack.length+1];
  for (int i=0; i<stack.length; i++)
    s[i] = stack[i];
  stack = s;
}
}
```

Public Behavior in JML File: Considerations



- Clear separation of behaviors.
 - Public behavior in JML specification file.
 - Private behavior in Java implementation file.
- `model stackM`
 - Model field of any class refining the specification.
- Also constructor specification in JML file.
 - Only private behavior of constructor in implementation file.
- `refine "IntStack.jml"`
 - All entities specified in specification file "IntStack.jml" must be implemented in implementation file "IntStack.java".
- `also private normal_behavior`
 - Extension of public behavior specified in JML file by private behavior.

Very clean solution.



-
1. Basic Class Specifications
 2. Classes for Modeling
 3. Model-based Class Specifications
 - 4. Rounding Things Up**

Desugaring Specifications



A `normal_behavior` specification is translated as follows.

<code>public normal_behavior</code>		<code>public behavior</code>
<code>requires P;</code>		<code>requires P;</code>
<code>assignable V;</code>	\Rightarrow	<code>assignable V;</code>
<code>ensures Q;</code>		<code>ensures Q;</code>
		<code>signals (Exception e) false;</code>

The method does not throw an exception.

Desugaring Specifications (Contd)



A exceptional_behavior specification is translated as follows.

<code>public exceptional_behavior</code>		<code>public behavior</code>
<code> requires P;</code>		<code> requires P;</code>
<code> assignable V;</code>	\Rightarrow	<code> assignable V;</code>
<code> signals (E e) Q;</code>		<code> ensures false;</code>
		<code> signals (E e) Q;</code>

The method does not return normally.

Desugaring Specifications (Contd'2)



Two public behavior specifications are combined as follows.

```
public behavior
  requires P1;
  assignable V1;
  ensures Q1;
  signals (E1 e) R1;
also public behavior
  requires P2;
  assignable V2;
  ensures Q2;
  signals (E2 e) R2;
⇒
public behavior
  requires P1 || P2;
  assignable V1 if P1,
                V2 if P2;
  ensures (\old(P1) ==> Q1)
          && (\old(P2) ==> Q2);
  signals (E1 e1) \old(P1) && R1;
  signals (E2 e2) \old(P2) && R2;
```

Basically the same for combining a public and a private behavior.

The Meaning of a Specification



```
public behavior
  requires  $P$ ;
  assignable  $V$  if  $M$ , ...;
  ensures  $Q$ ;
  signals ( $E1$   $e1$ )  $R1$ ;
  ...
```

- The method may be called, if P holds on the pre-state.
 - The conditions of multiple requires clauses are disjoined by $||$.
- The method may change V , if M holds.
 - And so on for the other variables in the assignable clause.
- If the method returns normally, Q holds on the pre/post-state pair.
 - The conditions of multiple ensures clauses are conjoined by $\&\&$.
- If the method throws an exception of type $E1$, $R1$ holds on the pre/post-state pair.
 - And so on for the other signals clauses.



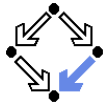
Specifications and Subtyping

Combining specifications works also for subtyping.

- If a class C_2 inherits from a class C_1 ,
 - C_2 inherits all **non-private** entities of C_1 .
- If C_2 **overrides** some non-private method m of C_1 ,
 - C_2 combines C_1 's **non-private** behavior specification of m with its own behavior specification of m .
 - This is why the new behavior specification of m in C_2 must begin with **also**.
- Thus an object of type C_2 behaves like an object of type C_1 .
 - C_2 specifies a **behavioral subtype** of C_1 .

Thus we can say “a C_2 object is a C_1 object”.

Further Features of JML



Not covered in this course ...

- Specification shortcuts
 - `\nonnullelements`, `\not_modified`, ...
- Redundant specifications and examples.
 - `ensures_redundantly`, `invariant_redundantly`,
`represents_redundantly`, `implies_that`, `for_example`, ...
- Non-functional specifications.
 - Execution time, execution space, methods invoked, ...
- Concurrency.
 - Experimental support of MultiJava.

JML is a (perhaps too) large and expressive language.