JML's Advanced Features

Néstor Cataño

ncatano@puj.edu.co

Faculty of Engineering

Pontificia Universidad Javeriana

Lecture Plan

1. `assignable` Clauses and Datagroups
2. Behavioural Subtyping

Lecture Plan

1. **`assignable`** Clauses and Datagroups
2. Behavioural Subtyping

Problems With `assignable` Clauses

- Tend to expose implementation details

```
private /*@ spec_public @*/ int x;
...
//@ assignable x, ....;
public void m(...) { ... }
```

- Tend to become very long

```
//@ assignable x, y, z[*], ...;
```

- Tend to accumulate

```
//@ assignable x, f.x, g.y, h.z[*], ...;
```

Problems With `assignable` Clauses

```
public class Timer {

 /*@ spec_public @*/ private int time_hrs, time_mins, time_secs;
 /*@ spec_public @*/ private int alarm_hrs, alarm_mins, alarm_secs;

 //@ assignable time_hrs, time_mins, time_secs;
 public void tick() { ... }

 //@ assignable alarm_hrs, alarm_mins, alarm_secs;
 public void setAlarm(int hrs, int mins, int secs) { ... }
}
```

Solution: Datagroups

```
public class Timer {
 //@ public model JMLDatagroup time, alarm;
  private int time_hrs, time_mins, time_secs; //@ in alarm;
  private int alarm_hrs, alarm_mins, alarm_secs; //@ in time;

 //@ assignable time;
 public void tick() { ... }

 //@ assignable alarm;
 public void setAlarm(int hrs, int mins, int secs) { ... }
}
```

time and **alarm** are specification only fields

`assignable` Clauses and Datagroups

- Datagroups provide an abstraction mechanism for **`assignable`** clauses
- A Datagroup is a set of fields referenced by a specific name
- When a Datagroup is mentioned in a method's **`assignable`** clause, all the members of the Datagroup may be assigned in the method's body

`assignable` Clauses and Datagroups

- When a `model` field is declared, a Datagroup with the same name is automatically created
- This field is always a member of the group it creates
- Class and model fields become member of a Datagroup through the clauses `in` and `maps-into` that come immediately after the field declaration

Datagroups and Fields

```
public class ArrayTimer {

  /*@ spec_public @*/ private char[] currentTime;


  //@ assignable currentTime[*];
  public void tick() { ... }
}
```

Datagroups and Fields

```
public class ArrayTimer {
  //@ public model JMLDatagroup time;
  private char[] currentTime; //@ in time;
  //@ maps currentTime[*] \into time;

  //@ assignable time;
  public void tick() { ... }
}
```

Datagroups and Interfaces

```
public interface TimerInterface {
  //@ model instance public JMLDatagroup time, alarm;
  //@ assignable time;

  public void tick();

  //@ assignable alarm;
  public void setAlarm(int hrs, int mins, int secs);
}
```

- Classes implementing this interface are free to choose which fields are in the Datagroups
- Keyword **`instance`** is needed, because fields in interfaces are by default static fields in Java. Interfaces in Java do not have instance fields, but in JML they can have model instance fields

Lecture Plan

1. `assignable` Clauses and Datagroups
2. Behavioural Subtyping

Lecture Plan

1. **`assignable`** Clauses and Datagroups
2. Behavioural Subtyping

Behavioural Subtyping

- Code will behave as expected if one provides a subclass object where a superclass object is expected
- Objects from subclasses behave like objects from superclasses

Behavioural Subtyping

To achieve behavioural subtyping, in JML subclasses inherit specifications of superclasses

Behavioural Subtyping

To achieve behavioural subtyping, in JML subclasses inherit specifications of superclasses

- Fields specifications for public and protected instance methods
  - invariants, and history constraints, **`initially`** clauses, Datagroup declarations, **`represents`** clauses

Behavioural Subtyping

To achieve behavioural subtyping, in JML subclasses inherit specifications of superclasses

- Fields specifications for public and protected instance methods
  - invariants, and history constraints, **`initially`** clauses, Datagroup declarations, **`represents`** clauses
- Methods specifications for public and protected instance methods
  - It can be thought of textual inheritance of specifications connected by **`also`** clauses

Behavioural Subtyping

To achieve behavioural subtyping, in JML subclasses inherit specifications of superclasses

- Fields specifications for public and protected instance methods
  - invariants, and history constraints, **`initially`** clauses, Datagroup declarations, **`represents`** clauses
- Methods specifications for public and protected instance methods
  - It can be thought of textual inheritance of specifications connected by **`also`** clauses

Specifications for constructors or static methods are not inherited, as they are not involved in dynamic dispatch

Behavioural Subtyping

Behavioural subtyping enforced by insisting that

- Precondition for any overriding method is weaker than precondition for the overridden method
- Postcondition for any overridden method is stronger than postcondition for the overriding method
- Invariant for the subclass is stronger than invariant for superclass

Behavioural Subtyping

- Fields declared in a supertype retain their Datagroup membership when inherited
- invariants from the superclass are inherited to the subclass
- **`initially`** clauses specified for supertype fields must also be obeyed in all subtypes
- History constraints specified in each supertype must be obeyed in the subtype
- **`represents`** clauses for **`model`** fields are also inherited

Inheritance for Fields

As in Java

- Private fields are inherited by a subtype but not visible to it
- Package fields are not accessible if the subtype is declared in a different package than the supertype declaring the field

Inheritance for Invariants

```
public interface BoundedThing {
  //@ public model instance int MAX_SIZE;

  //@ public instance invariant MAX_SIZE > 0;
  ...
}

public interface BoundedStackInterface extends BoundedThing {
  //@ public model instance JMLObjectSequence stack;

  //@ public instance invariant stack != null;
  ...
}
```

Invariant for `BoundedStackInterface` is **`stack != null && MAX_SIZE > 0`**.
Model field **`MAX_SIZE`** is inherited from `BoundedThing`.

Specification Inheritance for Method Specifications

```
public interface BoundedStackInterface extends BoundedThing {
  //@ public model instance JMLObjectSequence stack;

  /*@ public normal_behavior
    @  requires !stack.isEmpty();
    @  ensures stack.equals(\old(stack.trailer()));
    @*/
  public void pop( );
}
public class BoundedStack implements BoundedStackInterface {
  /*@ also
    @  public normal_behavior
    @   requires stack.isEmpty();
    @   ensures stack.equals(\old(stack));
    @*/
    public void pop( );
}
```

Specification Inheritance for Method Specifications

```
public class BoundedStack implements BoundedStackInterface {
 /*@   public normal_behavior
   @    requires !stack.isEmpty();
   @    ensures stack.equals(\old(stack.trailer()));
   @ also
   @   public normal_behavior
   @    requires stack.isEmpty();
   @    ensures stack.equals(\old(stack));
   @*/
   public void pop( );
}
```

Specification Inheritance for Method Specifications

```
public class BoundedStack implements BoundedStackInterface {
 /*@ public normal_behavior
   @  requires !stack.isEmpty() || stack.isEmpty();
   @  ensures \old(!stack.isEmpty()) ==>
   @            stack.equals(\old(stack.trailer()));
   @  ensures \old(stack.isEmpty()) ==>
   @            stack.equals(\old(stack));
   @*/
 public void pop( );
}
```

Desugaring Method Specifications

```
public interface BoundedStackInterface extends BoundedThing {
  /*@   public normal_behavior
    @    requires !stack.isEmpty();
    @    assignable size, stack;
    @    ensures stack.equals(\old(stack.trailer()));
    @ also
    @   public exceptional_behavior
    @    requires stack.isEmpty();
    @    assignable \nothing;
    @    signals_only BoundedStackException;
    @*/
  public void pop( ) throws BoundedStackException;
}
```

Desugaring Method Specifications

```
public interface BoundedStackInterface extends BoundedThing {
  /*@   public behavior
    @    requires !stack.isEmpty();
    @    assignable size, stack;
    @    ensures stack.equals(\old(stack.trailer()));
    @    signals (java.lang.Exception) false;
    @ also
    @   public behavior
    @    requires stack.isEmpty();
    @    assignable \nothing;
    @    ensures false;
    @    signals_only BoundedStackException;
    @    signals (java.lang.Exception) true;
    @*/
  public void pop( ) throws BoundedStackException;
}
```

Desugaring Method Specifications

```
public interface BoundedStackInterface extends BoundedThing {
  /*@   public behavior
    @    requires !stack.isEmpty();
    @    assignable size, stack;
    @    ensures stack.equals(\old(stack.trailer()));
    @    signals (java.lang.Exception) false;
    @ also
    @   public behavior
    @    requires stack.isEmpty();
    @    assignable \nothing;
    @    ensures false;
    @    signals_only BoundedStackException;
    @    signals (java.lang.Exception) true;
    @*/
  public void pop( ) throws BoundedStackException;
}
```

Desugaring Method Specifications

```
public interface BoundedStackInterface extends BoundedThing {
 /*@ public behavior
   @  requires !stack.isEmpty() || stack.isEmpty();
   @  assignable size, stack;
   @  ensures \old(!stack.isEmpty()) ==>
   @            stack.equals(\old(stack.trailer()));
   @  ensures \old(stack.isEmpty()) ==>
   @            \not_assigned(size) && \not_assigned(stack);
   @  signals_only BoundedStackException;
   @  signals (java.lang.Exception) \old(!stack.isEmpty()) ==> false
   @  signals (java.lang.Exception) \old(stack.isEmpty()) ==>
   @            \not_assigned(size) && \not_assigned(stack)
   @            && true;
   @*/
 public void pop( ) throws BoundedStackException;
}
```

Desugaring Method Specifications

```
public interface BoundedStackInterface extends BoundedThing {
 /*@ public behavior
   @  requires !stack.isEmpty() || stack.isEmpty();
   @  assignable size, stack;
   @  ensures \old(!stack.isEmpty()) ==>
   @            stack.equals(\old(stack.trailer()));
   @  ensures \old(stack.isEmpty()) ==>
   @            \not_assigned(size) && \not_assigned(stack);
   @  signals_only BoundedStackException;
   @  signals (java.lang.Exception) \old(!stack.isEmpty()) ==> false
   @  signals (java.lang.Exception) \old(stack.isEmpty()) ==>
   @            \not_assigned(size) && \not_assigned(stack)
   @            && true;
   @*/
 public void pop( ) throws BoundedStackException;
}
```