Formal Specification and Verification of Object-Oriented Programs

The Java Modeling Language: Basic Language Features (Part I)

Organisation

Richard Bubel
EMail: bubel@cs.tu-darmstadt.de
Office: Room A225 (S2|02)
Office Hours: on appointment

Lecture URL: https://moodle.informatik.tu-darmstadt.de/course/view.php?id=440

Organisation

Richard Bubel
EMail: bubel@cs.tu-darmstadt.de
Office: Room A225 (S2|02)
Office Hours: on appointment

Lecture URL: https://moodle.informatik.tu-darmstadt.de/course/view.php?id=440

Course: Lectures & Exercises

Course Language English

Lecture/Exercise Sessions

- Monday, 11:40 - 13:20 in C110 (S2|02)
- Thursday, 13:30 - 15:10 in C110 (S2|02)

Organisation

Richard Bubel
EMail: bubel@cs.tu-darmstadt.de
Office: Room A225 (S2|02)
Office Hours: on appointment

Lecture URL: https://moodle.informatik.tu-darmstadt.de/course/view.php?id=440

Lecture: Prerequisites

- Experience with Java-like object-oriented programming language
  - Classes, interfaces, inheritance, dynamic dispatch, visibility etc.
- (Basic) knowledge about
  - First-order logic (FOL): Syntax & Semantics
  - Deduction for FOL: E.g., sequent/tableaux/natural deduction calculus
  - Hoare logic or wp-calculus (Dijkstra)

Organisation

TECHNISCHE UNIVERSITÄT DARMSTADT

Richard Bubel
EMail: bubel@cs.tu-darmstadt.de
Office: Room A225 (S2|02)
Office Hours: on appointment

Lecture URL: https://moodle.informatik.tu-darmstadt.de/course/view.php?id=440

Exercises

- Assignments are published one week before an exercise session
- Not corrected, but:
  - Discussed in exercise sessions
  - Live from active participation

Specification

TECHNISCHE UNIVERSITÄT DARMSTADT

> An act of identifying something precisely or of stating a precise requirement.
>
> (Oxford Dictionary)

Specification

An act of identifying something precisely or of stating a precise requirement.

(Oxford Dictionary)

"I need a screw."

Specification

TECHNISCHE UNIVERSITÄT DARMSTADT

> An act of identifying something precisely or of stating a precise requirement.
>
> (Oxford Dictionary)

"I need a screw."

Imprecise

- Length, diameter, thickness
- For which material? Wood, metal, concrete?

Specification

An act of identifying something precisely or of stating a precise requirement.

(Oxford Dictionary)

"I need a screw."

Imprecise

- Length, diameter, thickness
- For which material? Wood, metal, concrete?

Possible implicit assumptions

- ... and bring it to me.
- ... the one I've given you a few moments ago.

Specification

TECHNISCHE
UNIVERSITÄT
DARMSTADT

An act of identifying something precisely or of stating a precise requirement.
(Oxford Dictionary)

"I need a screw."

Imprecise

- Length, diameter, thickness
- For which material? Wood, metal, concrete?

Possible implicit assumptions

- . . . and bring it to me.
- . . . the one I've given you a few moments ago.

I do not need (framing)

- a hammer
- a sandwich, a hat . . .

Specification Target

Explicit set of requirements to be satisfied by a material, product, system, or service.

(Form and Style for ASTM Standards, Blue Book of ASTM)

Specification Target

Explicit set of requirements to be satisfied by a
material, product, system, or service.
(Form and Style for ASTM Standards, Blue Book of ASTM)

System level specification
(requirements analysis, GUI, use cases)
important, but
not subject of this course

Specification Target

TECHNISCHE UNIVERSITÄT DARMSTADT

Explicit set of requirements to be satisfied by a
material, product, system, or service.
(Form and Style for ASTM Standards, Blue Book of ASTM)

System level specification

(requirements analysis, GUI, use cases)

important, but

not subject of this course

We focus (mostly) on:

Unit specification—contracts between implementors on various levels:

- Application level $\leftrightarrow$ application level
- Application level $\leftrightarrow$ library level
- Library level $\leftrightarrow$ library level

Unit (Module) Specifications

Cf. unit testing

In Object-Oriented Setting:

Units to be specified are interfaces, classes, and their methods

Unit (Module) Specifications

Cf. unit testing

In Object-Oriented Setting:

Units to be specified are interfaces, classes, and their methods

First focus on methods

Method specifications must comprise the following aspects:

- Result value
- Initial values of formal parameters
- Pre-state and post-state

Unit (Module) Specifications

Cf. unit testing

In Object-Oriented Setting:

Units to be specified are interfaces, classes, and their methods

First focus on methods

Method specifications must comprise the following aspects:

- Result value
- Initial values of formal parameters
- Accessible part of pre-/post-state

Meaning of Pre-/Post-Condition Pairs

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition

A **pre-/post-condition** pair for a method `m` is **satisfied by the implementation** of `m` if:

> *When `m` is called in any state that satisfies the precondition then in any terminating state of `m` the postcondition is true.*

Meaning of Pre-/Post-Condition Pairs

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition

A **pre-/post-condition** pair for a method `m` is **satisfied by the implementation** of `m` if:

> *When `m` is called in any state that satisfies the precondition then in any terminating state of `m` the postcondition is true.*

Remarks

1. No guarantee when the precondition is not satisfied
2. Termination may or may not be guaranteed
3. Terminating state may be reached by normal or by abrupt termination (e.g., exception)

Specifications as Contracts

Useful analogy to stress the different roles/obligations/responsibilities in a specification:

Specification as a contract (between method implementor and user)

"Design by Contract" methodology (Meyer, 1992, EIFFEL)

Specifications as Contracts

Useful analogy to stress the different roles/obligations/responsibilities in a specification:

Specification as a contract (between method implementor and user)

"Design by Contract" methodology (Meyer, 1992, EIFFEL)

Contract between caller and callee (called method)

Callee guarantees certain outcome provided caller guarantees prerequisites

Running Example: HealthTracker.java

```
public class HealthTracker {
 //fields:
 private String username;
 private Category[] category;
 private int nrCategories;

 // methods:
 public int getNumberCategories()
 public String getUsername()
 public Category findCategoryById(int)
 public boolean addCategory(Category)
 ...
}
```

```
public class Category {
 private int id;
 public int getId();
 ...
}
```

Informal Specification

TECHNISCHE UNIVERSITÄT DARMSTADT

Very informal specification of

boolean `addCategory(Category p_category)`

> *" Add the given category `p_category` to the list of categories and return success provided that no category with the same id exists and the maximal number of categories has not yet been reached.*
>
> *Otherwise return false to indicate failure."*

Becoming More Precise: Specification as Contract

Contract states what is guaranteed under which conditions

Becoming More Precise: Specification as Contract

Contract states what is guaranteed under which conditions

precondition `p_category`, `category` are not null, `nrCategories` is non-negative, maximal number of categories reached

Becoming More Precise: Specification as Contract

Contract states what is guaranteed under which conditions

precondition	`p_category`,`category` are not null, `nrCategories` is non-negative, maximal number of categories reached
postcondition	category is not added, `false` is returned

Becoming More Precise: Specification as Contract

Contract states what is guaranteed under which conditions

precondition	`p_category`,`category` are not null, `nrCategories` is non-negative, maximal number of categories reached
postcondition	category is not added, `false` is returned

precondition	`p_category`,`category` are not null, `nrCategories` is non-negative, category with same id is present maximal number of categories not reached
postcondition	category is not added, `false` is returned

Becoming More Precise: Specification as Contract

Contract states what is guaranteed under which conditions

precondition	`p_category`,`category` are not null, `nrCategories` is non-negative, maximal number of categories reached
postcondition	category is not added, `false` is returned
precondition	`p_category`,`category` are not null, `nrCategories` is non-negative, category with same id is present maximal number of categories not reached
postcondition	category is not added, `false` is returned
precondition	`p_category`,`category` are not null, `nrCategories` is non-negative maximal number of categories not reached, category with same id is not present

Becoming More Precise: Specification as Contract

Contract states what is guaranteed under which conditions

precondition `p_category`, `category` are not null, `nrCategories` is non-negative, maximal number of categories reached

postcondition category is not added, `false` is returned

precondition `p_category`, `category` are not null, `nrCategories` is non-negative, category with same id is present maximal number of categories not reached

postcondition category is not added, `false` is returned

precondition `p_category`, `category` are not null, `nrCategories` is non-negative maximal number of categories not reached, category with same id is not present

postcondition category is added , `nrCategories` increased by `1` and `true` is returned

Formal Specification

Natural language specs are very important and widely used

Formal Specification

Natural language specs are very important and widely used, we focus on

Formal Specification

Describe contracts of units with mathematical rigour

Formal Specification

Natural language specs are very important and widely used, we focus on

Formal Specification

Describe contracts of units with mathematical rigour

Motivation

- High degree of precision
  - formalization often exhibits omissions/inconsistencies
  - avoid ambiguities inherent to natural language
- Potential for automation of program analysis
  - run-time assertion checking
  - monitoring
  - test case generation
  - program verification

Program Verification

In contrast to testing:

Program verification proves absence of bugs

Program Verification

In contrast to testing:

Program verification proves absence of bugs

We focus on deductive program verification

- Specification of properties in a formal logic language

Program Verification

TECHNISCHE
UNIVERSITÄT
DARMSTADT

In contrast to testing:

Program verification proves absence of bugs

We focus on deductive program verification

- Specification of properties in a formal logic language
- Use of semi-automated theorem proving to verify with mathematical rigour that

the implementation adheres to its specification

Program Verification

In contrast to testing:

Program verification proves absence of bugs

We focus on deductive program verification

- Specification of properties in a formal logic language
- Use of semi-automated theorem proving to verify with mathematical rigour that

  the implementation adheres to its specification

  - **Input:**

    Specification & Implementation
  - **Output:**

    Machine-checkable mathematical proof of correctness

Program Verification

In contrast to testing:

Program verification proves absence of bugs (*)

We focus on deductive program verification

- Specification of properties in a formal logic language
- Use of semi-automated theorem proving to verify with mathematical rigour that

  the implementation adheres to its specification

  - **Input:**

    Specification & Implementation
  - **Output:**

    Machine-checkable mathematical proof of correctness

(*) under the assumption that compiler, virtual machine, operating system, hardware are correct and no interference by cosmic rays

Java Modeling Language (JML)

JML is a specification language tailored to JAVA

General JML Philosophy

Integrate

- specification
- implementation

in one single language

$\Rightarrow$ JML is not external to JAVA, but an extension of JAVA

Java Modeling Language (JML)

TECHNISCHE UNIVERSITÄT DARMSTADT

JML is a specification language tailored to JAVA

General JML Philosophy

Integrate

- specification
- implementation

in one single language

⇒ JML is not external to JAVA, but an extension of JAVA

JML
is
JAVA

Java Modeling Language (JML)

JML is a specification language tailored to JAVA

General JML Philosophy

Integrate

- specification
- implementation

in one single language

⇒ JML is not external to JAVA, but an extension of JAVA

JML
is
JAVA + First-Order Logic

Java Modeling Language (JML)

JML is a specification language tailored to JAVA

General JML Philosophy

Integrate

- specification
- implementation

in one single language

⇒ JML is not external to JAVA, but an extension of JAVA

JML
is
JAVA + First-Order Logic + pre-/post-conditions, invariants

Java Modeling Language (JML)

JML is a specification language tailored to JAVA

General JML Philosophy

Integrate

- specification
- implementation

in one single language

⇒ JML is not external to JAVA, but an extension of JAVA

JML
is
JAVA + First-Order Logic + pre-/post-conditions, invariants + more . . .

JML Annotations

JML extends JAVA by annotations

In this course . . .

- Preconditions & postconditions
- Class invariants
- Additional modifiers
- Loop invariants
- Specification of interfaces, abstract classes & modular specification
  - “Specification-only” fields
  - “Specification-only” methods
  - Static & dynamic frames

JML/JAVA integration

JML annotations are attached to JAVA programs
by
writing them directly into the JAVA source code files

Ensures compatibility with standard JAVA compiler:

JML annotations live in special JAVA comments,
ignored by JAVA compiler, recognized by JML tools

JML embedded as JAVA Comments

TECHNISCHE UNIVERSITÄT DARMSTADT

Demo: `HealthTracker#addCategory`

JML embedded as JAVA Comments

TECHNISCHE UNIVERSITÄT DARMSTADT

Demo: `HealthTracker#addCategory`

```
/*@ public normal_behavior
 @ requires p_category != null
 @ requires category != null
 @ requires nrCategories >= 0;
 @ requires nrCategories >= category.length - 1;;
 @ ensures \result == false;
 @*/
public boolean addCategory(Category p_category)
```

Everything between /* and */ is invisible for JAVA compiler

JML embedded as JAVA Comments

TECHNISCHE UNIVERSITÄT DARMSTADT

JAVA comment lines starting with @ read and parsed by JML tools

```
/*@ public normal_behavior
  @ requires p_category != null; @ only to beautify
  @ requires category != null
  @ requires nrCategories >= 0;
  @ requires nrCategories >= category.length - 1;
  @*/
//@ ensures \result == false;
//␣@ ensures \result == false; no JML: @ not first
public boolean addCategory(Category p_category)
```

JML by Example: Public Modifier

```
/*@ public normal_behavior
  @ requires p_category != null;
  @ requires category != null
  @ requires nrCategories >= 0;
  @ requires nrCategories >= category.length - 1;
  @ ensures \result == false;
  @*/
public boolean addCategory(Category p_category)
```

This is a **public** specification case:

1. it is accessible from all classes and interfaces
2. it can only refer to `public` fields/methods of this class
   (can be problematic, come back to it later)

Visibility will become important later when discussing modular specification.

JML by Example: Specification Cases

TECHNISCHE UNIVERSITÄT DARMSTADT

```
/*@ public normal_behavior
  @ requires p_category != null;
  @ requires category != null
  @ requires nrCategories >= 0;
  @ requires nrCategories >= category.length - 1;
  @ ensures \result == false;
  @*/
public boolean addCategory(Category p_category)
```

Each keyword ending with **behavior** opens a specification case

`normal_behavior` Specification Case

The called method guarantees to terminate and to not throw an exception,
if the caller guarantees all preconditions of this specification case

JML by Example: Preconditions


```
/*@ public normal_behavior
  @ requires p_category != null;
  @ requires category != null
  @ requires nrCategories >= 0;
  @ requires nrCategories >= category.length - 1;
  @ ensures \result == false;
  @*/
public boolean addCategory(Category p_category)
```

Specification case has four preconditions (marked by `requires`)

JML by Example: Preconditions

TECHNISCHE UNIVERSITÄT DARMSTADT

```
/*@ public normal_behavior
  @ requires p_category != null;
  @ requires category != null
  @ requires nrCategories >= 0;
  @ requires nrCategories >= category.length - 1;
  @ ensures \result == false;
  @*/
public boolean addCategory(Category p_category)
```

Specification case has four preconditions (marked by `requires`)

Here:
preconditions happen to be boolean JAVA expressions

In general:
preconditions are boolean JML expressions (including quantifiers)

JML by Example: Preconditions Cont'd


```
/*@ public normal_behavior
  @ requires p_category != null;
  @ requires category != null
  @ requires nrCategories >= 0;
  @ requires nrCategories >= category.length - 1;
  @ ensures \result == false;
  @*/
```

Both preconditions must be true in prestate

Equivalent to:

```
/*@ public normal_behavior
  @ requires p_category != null && category != null  &&
  @   nrCategories >= 0 &&  nrCategories >= category.length - 1;
  @ ensures \result == false;
  @*/
```

JML by Example: Postconditions


```
/*@ public normal_behavior
  @ requires p_category != null && ... && nrCategories >= 0;
  @ requires nrCategories >= category.length - 1;
  @ ensures \result == false;
  @*/
```

Specification case has one postcondition (marked by **ensures**)

- Postconditions speaks about the poststate
- Postconditions are boolean JML expressions
- If there is more than one **ensures** clause:
  postcondition is the conjunction of all clauses

JML by Example: Accessing the Return Value

TECHNISCHE UNIVERSITÄT DARMSTADT

```
/*@ public normal_behavior
  @ requires p_category != null && ... && nrCategories >= 0;
  @ requires nrCategories >= category.length - 1;
  @ ensures \result == false;
  @*/
```

Special keyword \result to refer to the return value of a method.
Only allowed in postconditions.

JML by Example: Multiple Specification Cases

TECHNISCHE UNIVERSITÄT DARMSTADT

Multiple specification cases connected by `also`

```
/*@ public normal_behavior
  @ requires p_category != null && ... && nrCategories >= 0;
  @ requires nrCategories >= category.length - 1;
  @ ensures \result == false;
  @
  @ also
  @
  @ public normal_behavior
  @ requires p_category != null && ... && nrCategories >= 0;
  @ requires nrCategories < category.length - 1;
  @ requires (\forall int i; i>=0 && i<nrCategories;
  @                              p_category.id != category[i].id);
  @ ensures category[nrCategories - 1] == p_category;
  @ ensures nrCategories == \old(nrCategories + 1);
  @ ensures \result == true;  @*/
public boolean addCategory(Category p_category)
```

JML by Example: Quantified Expressions

```
/*@ public normal_behavior
  @ requires p_category != null && ... && nrCategories >= 0;
  @ requires nrCategories < category.length - 1;
  @ requires (\forall int i; i>=0 && i<nrCategories;
  @                                    p_category.id != category[i].id);
  @ ensures category[nrCategories - 1] == p_category;
  @ ensures nrCategories == \old(nrCategories + 1);
  @ ensures \result == true;
  @*/
public boolean addCategory(Category p_category)
```

Quantified JML Expressions (usage & more, see later)

- `(\forall t x; a; b)`: "for all `x` of type `t` fulfilling `a`, `b` is true"
- `(\exists t x; a; b)`: "there exists an `x` of type `t` fulfilling `a`, such that `b` is true"

JML by Example: Access of Prestate

TECHNISCHE UNIVERSITÄT DARMSTADT

```
/*@ public normal_behavior
  @ requires p_category != null && ... && nrCategories >= 0 &&
  @       nrCategories < category.length - 1;
  @ requires (\forall int i; i>=0 && i<nrCategories;
  @                                 p_category.id != category[i].id);
  @ ensures category[nrCategories - 1] == p_category;
  @ ensures nrCategories == \old(nrCategories + 1);
  @ ensures \result == true;
  @*/
```

Access to value of prestate in postcondition

`\old(`*E*`)` means: *E* evaluated in the prestate (of `addCategory(Category)`)

- `\old(`*E*`)` is a JML expression that is not a JAVA expression
- *E* can be any (arbitrarily complex) JAVA/JML expression

Specification Cases Complete?

TECHNISCHE UNIVERSITÄT DARMSTADT

```
@ requires p_category != null && ... && nrCategories >= 0 &&
@       nrCategories < category.length - 1;
@ requires (\forall int i; i>=0 && i<nrCategories;
@                               p_category.id != category[i].id);
@ ensures category[nrCategories - 1] == p_category;
@ ensures nrCategories == \old(nrCategories + 1);
@ ensures \result == true;
```

Specification Cases Complete?

TECHNISCHE UNIVERSITÄT DARMSTADT

```
@ requires p_category != null && ... && nrCategories >= 0 &&
@       nrCategories < category.length - 1;
@ requires (\forall int i; i>=0 && i<nrCategories;
@                              p_category.id != category[i].id);
@ ensures category[nrCategories - 1] == p_category;
@ ensures nrCategories == \old(nrCategories + 1);
@ ensures \result == true;
```

What does the specification case not tell about poststate?

Specification Cases Complete?

TECHNISCHE UNIVERSITÄT DARMSTADT

```
@ requires p_category != null && ... && nrCategories >= 0 &&
@       nrCategories < category.length - 1;
@ requires (\forall int i; i>=0 && i<nrCategories;
@                              p_category.id != category[i].id);
@ ensures category[nrCategories - 1] == p_category;
@ ensures nrCategories == \old(nrCategories + 1);
@ ensures \result == true;
```

What does the specification case not tell about poststate?

Fields of class `HealthTracker`:

`username, category, nrCategories`

What happens with `username`, `category` and `category[j]` (j<nrCategories - 1)?

Completing Specification Cases

```
@ requires ...
@ ensures category[nrCategories - 1] == p_category;
@ ensures nrCategories == \old(nrCategories + 1) &&  \result == true;
@ ensures username == \old(username);
@ ensures category == \old(category);
@ ensures (\forall int j; j>=0 && j<category.length && j != nrCategories-1;
@                                                category[j] == \old(category[j]))
 ;
```

- Similar postconditions added for the other specification cases
- Assumption that environment is unchanged unless explicitly stated: usually called frame condition

Completing Specification Cases


```
@ requires ...
@ ensures category[nrCategories - 1] == p_category;
@ ensures nrCategories == \old(nrCategories + 1) &&  \result == true;
@ ensures username == \old(username);
@ ensures category == \old(category);
@ ensures (\forall int j; j>=0 && j<category.length && j != nrCategories-1;
@                                                  category[j] == \old(category[j]))
 ;
```

- Similar postconditions added for the other specification cases
- Assumption that environment is unchanged unless explicitly stated: usually called frame condition

Clearly unsatisfactory to add

```
@ ensures loc == \old(loc);
```

for all locations loc which do not change

Assignable Locations

More efficient to explicitly list all locations that may change:

`@ assignable` $loc_1, \ldots, loc_n$;

Assignable clause: value of no location besides $loc_1, \ldots, loc_n$ can change (but could change temporarily during execution of method)

Assignable Locations

TECHNISCHE UNIVERSITÄT DARMSTADT

More efficient to explicitly list all locations that may change:

```
@ assignable loc_1 , . . . , loc_n ;
```

Assignable clause: value of no location besides $loc_1, \ldots, loc_n$ can change (but could change temporarily during execution of method)

Special cases of assignable clause

No location may be changed:

```
@ assignable \nothing;
```

Unrestricted, method allowed to change anything:

```
@ assignable \everything;
```

This is the default if no assignable clause is given

Specification Case with Assignable


```
@ requires ...
@ ensures category[nrCategories - 1] == p_category;
@ ensures nrCategories == \old(nrCategories + 1);
@ ensures \result == true;
@ assignable nrCategories, category[nrCategories];
```

Expressions in assignable clause evaluated in pre-state.

Specification Case with Assignable

TECHNISCHE UNIVERSITÄT DARMSTADT

```
@ requires ...
@ ensures category[nrCategories - 1] == p_category;
@ ensures nrCategories == \old(nrCategories + 1);
@ ensures \result == true;
@ assignable nrCategories, category[nrCategories];
```

Expressions in assignable clause evaluated in pre-state.

If a field of a different (i.e., not this) objectis changed (e.g., `id` of `p_category`)

```
@ assignable p_category.id;
```

Specification Case with Assignable


```
@ requires ...
@ ensures category[nrCategories - 1] == p_category;
@ ensures nrCategories == \old(nrCategories + 1);
@ ensures \result == true;
@ assignable nrCategories, category[nrCategories];
```

Expressions in assignable clause evaluated in pre-state.

If a field of a different (i.e., not this) objectis changed (e.g., `id` of `p_category`)

```
@ assignable p_category.id;
```

Does

```
@ assignable p_category;
```

make sense?

Specification Case with Assignable

```
@ requires ...
@ ensures category[nrCategories - 1] == p_category;
@ ensures nrCategories == \old(nrCategories + 1);
@ ensures \result == true;
@ assignable nrCategories, category[nrCategories];
```

Expressions in assignable clause evaluated in pre-state.

If a field of a different (i.e., not this) objectis changed (e.g., `id` of `p_category`)

```
@ assignable p_category.id;
```

Does

```
@ assignable p_category;
```

make sense?

No! Parameters in Java are passed copy-by-value.

JML Modifiers

JML extends the JAVA modifiers by additional modifiers

The most important ones are:

- `spec_public`
- `pure`
- `nullable` (next lecture)
- `non_null` (next lecture)
- `helper` (next lecture)

JML Modifiers: `spec_public`

In “`addCategory`” the specifications used class fields

But: **`public`** specifications can access only **`public`** fields

Not desired: make all fields mentioned in specification public

JML Modifiers: `spec_public`

In “`addCategory`” the specifications used class fields

But: **`public`** specifications can access only **`public`** fields

Not desired: make all fields mentioned in specification public

Control visibility with `spec_public`

- Keep visibility of JAVA fields **`private/protected`**
- If necessary make them visible in specification only by **`spec_public`**

JML Modifiers: spec_public

In “`addCategory`” the specifications used class fields

But: **`public`** specifications can access only **`public`** fields

Not desired: make all fields mentioned in specification public

Control visibility with spec_public

- Keep visibility of JAVA fields **`private/protected`**
- If necessary make them visible in specification only by **`spec_public`**

```
private /*@ spec_public @*/ Category[] category;
private /*@ spec_public @*/ int nrCategories;
```

(different solution(see lecture in 4-5 weeks): use specification-only fields)

JML Modifiers: Notions of Purity — `pure`

Specifications more concise with method calls inside JML annotations:

Examples: `o1.equals(o2)` `li.contains(elem)` `li1.max() < li2.min()`

Specifications may not themselves change the state!

Definition (Pure method)

A JAVA method is called pure iff it has no visible side effects and it always terminates.
A method is strictly pure if it must not create new objects, otherwise weakly pure.

JML Modifiers: Notions of Purity — `pure`

Specifications more concise with method calls inside JML annotations:

Examples: `o1.equals(o2)` `li.contains(elem)` `li1.max() < li2.min()`

Specifications may not themselves change the state!

Definition (Pure method)

A JAVA method is called pure iff it has no visible side effects and it always terminates.
A method is strictly pure if it must not create new objects, otherwise weakly pure.

JML expressions may call pure methods. These are annotated by **`pure`** (weakly pure) or **`strictly_pure`** resp.

```
public /*@ pure @*/ int max() { ... }
```

JML Modifiers: `pure` Cont'd

How do we know that a `pure` method is really pure?

- `pure` puts obligation on implementor not to cause side effects
- It is possible to formally verify that a method is pure
  - Write a contract that expresses purity and verify it
- `pure` implies `assignable \nothing;` (may create new objects)
- `assignable \strictly_nothing;` expresses that no new objects are created
- Assignable clauses can be local to a specification case while `pure` fixes behavior of a method

JML Expressions

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition (JML Expressions)

- Each side-effect free JAVA expression is a JML expression
- If *E* is a side-effect free JAVA expression, then `\old(E)` is a JML expression
- If `a` and `b` are **`boolean`** JML expressions, `x` is a variable of type `t`:
  - `!a` ("not `a`"), `a && b` ("`a` and `b`"), `a || b` ("`a` or `b`")
  - `a ==> b` ("`a` implies `b`")
  - `a <==> b` ("`a` is equivalent to `b`")
  - `(\forall t x; a)` ("for all `x` of type `t`, `a` is true")
  - `(\exists t x; a)` ("there exists `x` of type `t` such that `a`")
  - `(\forall t x; a; b)` ("for all `x` of type `t` fulfilling `a`, `b` is true")
  - `(\exists t x; a; b)` ("there exists an `x` of type `t` fulfilling `a`, such that `b` is true")

are also **`boolean`** JML expressions.

Range Predicates

Definition (Range predicate)

In the JML expressions (`\forall t x; a; b`) and (`\exists t x; a; b`) the **`boolean`** **a** is called range predicate.

Range predicates are syntactic sugar for standard FOL quantifiers:

```
(\forall t x; a; b)
```
equivalent to
```
(\forall t x; a ==> b)
```

```
(\exists t x; a; b)
```
equivalent to
```
(\exists t x; a && b)
```

Pragmatics of Range Predicates

Range predicates used to restrict range of `x` further than to its type `t`

Example

"Array `a` is sorted between indices 0 and 9":

```
(\forall int i,j; 0<=i && i<j && j<10; a[i] <= a[j])
```

Using Quantified JML Expressions

- An array int a contains only non-negative elements

JML
```
(\forall int i; 0 <= i && i < a.length; a[i] >= 0)
```
JML

Using Quantified JML Expressions

TECHNISCHE UNIVERSITÄT DARMSTADT

- An array `int a` contains only non-negative elements

JML
```
(\forall int i; 0 <= i && i < a.length; a[i] >= 0)
```
JML

- The variable `m` holds a maximal element of array `a`

JML
```
(\forall int i; 0 <= i && i < a.length; m >= a[i])
```
JML

Using Quantified JML Expressions

TECHNISCHE UNIVERSITÄT DARMSTADT

- An array `int a` contains only non-negative elements

JML
```
(\forall int i; 0 <= i && i < a.length; a[i] >= 0)
```
JML

- The variable `m` holds a maximal element of array `a`

JML
```
(\forall int i; 0 <= i && i < a.length; m >= a[i])
```
JML

Is this sufficient? Need in addition:

JML
```
(\exists int i; 0 <= i && i < a.length; m == a[i])
```
JML

Literature for this Lecture

Reading

KeY Book Andreas Roth & Peter H. Schmitt: Formal Specification. Chapter 5, Sections 5.1, 5.3, In: B. Beckert, R. Hähnle, and P. Schmitt, eds. *Verification of Object-Oriented Software: The KeY Approach*, vol 4334 of *LNCS*. Springer, 2006.

At http://link.springer.com/book/10.1007/978-3-540-69061-0

JML Reference Manual G. T. Leavens et. al, *JML Reference Manual*

JML Tutorial G. T. Leavens, Y. Cheon. *Design by Contract with JML*

JML Overview G. T. Leavens, A. L. Baker, and C. Ruby. *JML: A Notation for Detailed Design*

At www.jmplspecs.org