# Formal Specification and Verification

Bernhard Beckert

Adaptation of slides by
Wolfgang Ahrendt
Chalmers University, Gothenburg, Sweden

# Unit Specifications

**in the object-oriented setting:**

The units to be specified are interfaces, classes, and their methods.

We first focus on specifying methods.

Methods are specified by *potentially* referring to:

- the result value,
- the initial values of formal parameters,
-

But what do we mean by state?

# Unit Specifications

**in the object-oriented setting:**

The units to be specified are interfaces, classes, and their methods.

We first focus on specifying methods.

Methods are specified by *potentially* referring to:

- the result value,
- the initial values of formal parameters,
-

But what do we mean by state?

# Unit Specifications

**in the object-oriented setting:**

The units to be specified are interfaces, classes, and their methods.

We first focus on specifying methods.

Methods are specified by *potentially* referring to:

- the result value,
- the initial values of formal parameters,
-

But what do we mean by state?

# Unit Specifications

**in the object-oriented setting:**

The units to be specified are interfaces, classes, and their methods.

We first focus on specifying methods.

Methods are specified by *potentially* referring to:

- the result value,
- the initial values of formal parameters,
- the overall state

But what do we mean by state?

# Unit Specifications

**in the object-oriented setting:**

The units to be specified are interfaces, classes, and their methods.

We first focus on specifying methods.

Methods are specified by *potentially* referring to:

- the result value,
- the initial values of formal parameters,
- the locally visible part of the overall state

But what do we mean by state?

# Unit Specifications

**in the object-oriented setting:**

The units to be specified are interfaces, classes, and their methods.

We first focus on specifying methods.

Methods are specified by *potentially* referring to:

- the result value,
- the initial values of formal parameters,
- the locally visible part of the overall state

But what do we mean by state?

# Prerequisite: Object-oriented States

By state, we mean a 'snapshot' of the system, at any point during the the computation, described in terms of the *programmer's model*.

An *object oriented* state consists of:

- the set $\mathcal{C}$ of all loaded classes
- the values of the static fields of classes in $\mathcal{C}$
- the set $\mathcal{O}$ of references to all created objects
- the values of the instance fields of objects in $\mathcal{O}$

Here, values of *local variables* and *formal parameters* are *not* considered part of the state.

# Prerequisite: Object-oriented States

By state, we mean a 'snapshot' of the system, at any point during the the computation, described in terms of the *programmer's model*.

An *object oriented* state consists of:

- the set $\mathcal{C}$ of all loaded classes
- the values of the static fields of classes in $\mathcal{C}$
- the set $\mathcal{O}$ of references to all created objects
- the values of the instance fields of objects in $\mathcal{O}$

Here, values of *local variables* and *formal parameters* are *not* considered part of the state.

# Prerequisite: Object-oriented States

By state, we mean a 'snapshot' of the system, at any point during the the computation, described in terms of the *programmer's model*.

An *object oriented* state consists of:

- the set $\mathcal{C}$ of all loaded classes
- the values of the static fields of classes in $\mathcal{C}$
- the set $\mathcal{O}$ of references to all created objects
- the values of the instance fields of objects in $\mathcal{O}$

Here, values of *local variables* and *formal parameters* are *not* considered part of the state.

# Prerequisite: Visible State

Like implementations, specifications can only refer to the locally visible part of the state (e.g., not to `private` fields of other classes).

# Prerequisite: Visible State

In our context, we stick to the following principle:

**Same Visible State for Specifications and Implementations:**

In some local context, specifications and implementations can access the same part of the overall state.[a]

[a] Later, we'll refine this principle, and introduce well defined exceptions.

Thus, specifications talk only about those parts of the state which are accessible by:

- respecting JAVA's visibility rules (`public`, `protected`, `private`),
- following (visible) references, starting from local fields.

# Prerequisite: Visible State

In our context, we stick to the following principle:

**Same Visible State for Specifications and Implementations:**

In some local context, specifications and implementations can access the same part of the overall state.[a]

[a] Later, we'll refine this principle, and introduce well defined exceptions.

Thus, specifications talk only about those parts of the state which are accessible by:

- respecting JAVA's visibility rules (`public`, `protected`, `private`),
- following (visible) references, starting from local fields.

# Prerequisite: Visible State

In our context, we stick to the following principle:

**Same Visible State for Specifications and Implementations:**

In some local context, specifications and implementations can access the same part of the overall state.[a]

[a] Later, we'll refine this principle, and introduce well defined exceptions.

Thus, specifications talk only about those parts of the state which are accessible by:

- respecting JAVA's visibility rules (`public`, `protected`, `private`),
- following (visible) references, starting from local fields.

# Purely Functional Specification

A purely functional specification of a (non-`void`) method talks

- only about
  - the result of a call
  - the initial value of input parameters
- but not about
  - (any part of) the state

examples:

| *interface/class:* | *method:* |
|---|---|
| `Math` | `static int abs(int a)` |
| `Math` | `static double sqrt(double a)` |

# Purely Functional Specification

A purely functional specification of a (non-`void`) method talks

- only about
  - the result of a call
  - the initial value of input parameters
- but not about
  - (any part of) the state

examples:

| *interface/class:* | *method:* |
|---|---|
| `Math` | `static int abs(int a)` |
| `Math` | `static double sqrt(double a)` |

# Purely Functional Specification: `Math::abs()`

from the JAVA API:

**Specification of `static int abs(int a)`**

Returns the absolute value of an int value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.
Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable int value, the result is that same value, which is negative.

Green: Intuitive description rather than a specification.
Red: Precise specification by case distinction, given we know what 'negative' and 'negation' mean exactly.
Blue: A consequence of the specification, i.e. a *redundant part* of it.

Red and Blue are candidates for formalisation.

# Purely Functional Specification: `Math::abs()`

from the JAVA API:

**Specification of `static int abs(int a)`**

Returns the absolute value of an int value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.
Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable int value, the result is that same value, which is negative.

Green: Intuitive description rather than a specification.
Red: Precise specification by case distinction, given we know what 'negative' and 'negation' mean exactly.
Blue: A consequence of the specification, i.e. a *redundant part* of it.

Red and Blue are candidates for formalisation.

# Purely Functional Specification: `Math::abs()`

from the JAVA API:

**Specification of `static int abs(int a)`**

Returns the absolute value of an int value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.
Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable int value, the result is that same value, which is negative.

Green: Intuitive description rather than a specification.
Red: Precise specification by case distinction, given we know what 'negative' and 'negation' mean exactly.
Blue: A consequence of the specification, i.e. a *redundant part* of it.

Red and Blue are candidates for formalisation.

# Purely Functional Specification: `Math::abs()`

from the JAVA API:

**Specification of `static int abs(int a)`**

Returns the absolute value of an int value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.
Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable int value, the result is that same value, which is negative.

Green: Intuitive description rather than a specification.
Red: Precise specification by case distinction, given we know what 'negative' and 'negation' mean exactly.
Blue: A consequence of the specification, i.e. a *redundant part* of it.

Red and Blue are candidates for formalisation.

# Going a bit more formal

```
static int abs(int a)
```

*Informal spec:*
If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.

*Semi formal:*

- Under the precondition '$a \in [0...2147483647]$', `abs` ensures the postcondition '$result = a$'.
- Under the precondition '$a \in [-2147483648... -1]$', `abs` ensures the postcondition '$result = -a$'.

# Going a bit more formal

`static int` **`abs`**`(int a)`

*Informal spec:*
If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.

*Semi formal:*

- Under the precondition '$a \in [0...2147483647]$',
  **`abs`** ensures the postcondition '$result = a$'.
- Under the precondition '$a \in [-2147483648... -1]$',
  **`abs`** ensures the postcondition '$result = -a$'.

# Going a bit more formal

`static int` **`abs`**`(int a)`

*Redundant informal spec:*
Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable int value, the result is that same value, which is negative.

*Semi formal:*

- Under the precondition ‘a $= -2147483648$’, **`abs`** ensures the postcondition ‘result $= -2147483648$’.

*Or simply:*[1]

- **`abs`**$(-2147483648) = -2147483648$

[1] But be careful when using a method call in a formula, see below.

# Going a bit more formal

`static int` **`abs`**`(int a)`

*Redundant informal spec:*
Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable int value, the result is that same value, which is negative.

*Semi formal:*

- Under the precondition ‘a $= -2147483648$’,
  **`abs`** ensures the postcondition ‘result $= -2147483648$’.

*Or simply:*[1]

- **`abs`**$(-2147483648) = -2147483648$

[1] But be careful when using a method call in a formula, see below.

# State Aware Specification

A state aware specification of a (`void` or non-`void`) method talks about

- the result of a call (if non-`void`)
- the initial value of input parameters
- *two* states:
  - the 'pre-state' of the method call
  - the 'post-state' of the method call

examples:

| *interface/class:* | *method:* |
|---|---|
| `List` | `Object set(int index, Object element)` |
| `Collections` | `static void sort(List list)` |

# State Aware Specification

A state aware specification of a (`void` or non-`void`) method talks about

- the result of a call (if non-`void`)
- the initial value of input parameters
- *two* states:
  - the 'pre-state' of the method call
  - the 'post-state' of the method call

examples:

| *interface/class:* | *method:* |
|---|---|
| `List` | `Object set(int index, Object element)` |
| `Collections` | `static void sort(List list)` |

# State Aware Specification

A state aware specification of a (`void` or non-`void`) method talks about

- the result of a call (if non-`void`)
- the initial value of input parameters
- *two* states:
  - the 'pre-state' of the method call
  - the 'post-state' of the method call

examples:

| *interface/class:* | *method:* |
|---|---|
| `List` | `Object set(int index, Object element)` |
| `Collections` | `static void sort(List list)` |

# State Aware Specification

A state aware specification of a (`void` or non-`void`) method talks about

- the result of a call (if non-`void`)
- the initial value of input parameters
- *two* states:
  - the 'pre-state' of the method call
  - the 'post-state' of the method call

examples:

| *interface/class:* | *method:* |
|---|---|
| `List` | `Object set(int index, Object element)` |
| `Collections` | `static void sort(List list)` |

# State Aware Specification: `List::set(i,e)`

*from the Java API of* `List::set` *(simplified):*

`public Object` **`set`**`(int index, Object element)`

Replaces the element at the specified position in this list with the specified element.

**Parameters:**
`index` - index of element to replace.
`element` - element to be stored at the specified position.
**Returns:**
the element previously at the specified position.
**Throws:**
`IndexOutOfBoundsException`
- if the index is out of range (index $<$ 0 || index $>=$ size()).

*Why is the spec state aware?*
*It talks about the state, in particular about the state change.*

# State Aware Specification: `List::set(i,e)`

*from the Java API of* `List::set` *(simplified):*

`public Object` **`set`**`(int index, Object element)`

Replaces the element at the specified position in this list with the specified element.

**Parameters:**
`index` - index of element to replace.
`element` - element to be stored at the specified position.
**Returns:**
the element previously at the specified position.
**Throws:**
`IndexOutOfBoundsException`
- if the index is out of range (index $<$ 0 || index $>=$ size()).

*Why is the spec state aware?*

*It talks about the state, in particular about the state change.*

# State Aware Specification: `List::set(i,e)`

*from the Java API of* `List::set` *(simplified):*

`public Object` **`set`**`(int index, Object element)`

Replaces the element at the specified position in this list with the specified element.

**Parameters:**
`index` - index of element to replace.
`element` - element to be stored at the specified position.
**Returns:**
the element previously at the specified position.
**Throws:**
`IndexOutOfBoundsException`
- if the index is out of range (index $<$ 0 || index $>=$ size()).

*Why is the spec state aware?*
*It talks about the state, in particular about the state change.*

# Going a bit more formal

`public Object set(int index, Object element)`

*Informal spec:*
Replaces the element at the specified position in this list with the specified element.

*Semi formal:*
`set` ensures the following postcondition:

- `element` = '`get(index)` *evaluated in the post-state*'

Does this capture the meaning of the word 'replace'?

# Going a bit more formal

`public Object set(int index, Object element)`

*Informal spec:*
Replaces the element at the specified position in this list with the specified element.

*Semi formal:*
**`set`** ensures the following postcondition:

- `element` = '`get(index)` *evaluated in the post-state*'

Does this capture the meaning of the word 'replace'?

# Going a bit more formal

`public Object set(int index, Object element)`

*Informal spec:*
Replaces the element at the specified position in this list with the specified element.

*Semi formal:*
**`set`** ensures the following postcondition:

- `element` = '`get(index)` *evaluated in the post-state*'

Does this capture the meaning of the word 'replace'?

# Going a bit more formal

`public Object set(int index, Object element)`

*Informal spec:*
Replaces the element at the specified position in this list with the specified element.

*Semi formal:*
`set` ensures the following postconditions:

- `element` = '`get(index)` *evaluated in the post-state*', and
- for all `j` $\in [0 \ldots$`size()` $- 1]$ with `j` $\neq$ `index`:
  '`get(j)` *in post-state*' = '`get(j)` *in pre-state*'

# Going a bit more formal

`public Object set(int index, Object element)`

*Informal spec:*
Replaces the element at the specified position in this list with the specified element.

*Semi formal:*
**`set`** ensures the following postconditions:

- `element` = '`get(index)` *evaluated in the post-state*', and
- for all `j` $\in [0 \ldots$`size()` $- 1]$ with `j` $\neq$ `index`:
  '`get(j)` *in post-state*' = '`get(j)` *in pre-state*'

# Going a bit more formal

`public Object set(int index, Object element)`

*Informal spec:*
Replaces the element at the specified position in this list with the specified element ... Returns the element previously at the specified position.

*Semi formal:*
`set` ensures the following postconditions:

- result = '`get(index)` *evaluated in the pre-state*', and
- `element` = '`get(index)` *evaluated in the post-state*', and
- for all `j` $\in [0...$`size()` $- 1]$ with `j` $\neq$ `index`:
  '`get(j)` *in post-state*' = '`get(j)` *in pre-state*'

# Going a bit more formal

`public Object set(int index, Object element)`

*Informal spec:*
Replaces the element at the specified position in this list with the specified element ... Returns the element previously at the specified position.

*Semi formal:*
**`set`** ensures the following postconditions:

- result = '`get(index)` *evaluated in the pre-state*', and
- `element` = '`get(index)` *evaluated in the post-state*', and
- for all `j` $\in$ [0...`size()` $-$ 1] with `j` $\neq$ `index`:
  '`get(j)` *in post-state*' = '`get(j)` *in pre-state*'

# Going a bit more formal

`public Object set(int index, Object element)`

*Informal spec:*
Replaces the element at the specified position in this list with the specified element ... Returns the element previously at the specified position ... Throws `IndexOutOfBoundsException` if the index is out of range (index $<$ 0 || index $>=$ size()).

*Semi formal:*

- Under the precondition '`index` $\in$ [0...`size()` $-$ 1]', `set` ensures the following postconditions:
  - result = '`get(index)` *evaluated in the pre-state*', and
  - `element` = '`get(index)` *evaluated in the post-state*', and
  - for all `j` $\in$ [0...`size()` $-$ 1] with `j` $\neq$ `index`: '`get(j)` *in post-state*' = '`get(j)` *in pre-state*'
- Under the precondition '`index` $\notin$ [0...`size()` $-$ 1]', `set` throws `IndexOutOfBoundsException`.

# Going a bit more formal

`public Object set(int index, Object element)`

*Informal spec:*
Replaces the element at the specified position in this list with the specified element ... Returns the element previously at the specified position ... Throws `IndexOutOfBoundsException` if the index is out of range (index $<$ 0 || index $>=$ size()).

*Semi formal:*

- Under the precondition '`index` $\in$ [0...`size()` $-$ 1]',
  **`set`** ensures the following postconditions:
  - result = '`get(index)` *evaluated in the pre-state*', and
  - `element` = '`get(index)` *evaluated in the post-state*', and
  - for all `j` $\in$ [0...`size()` $-$ 1] with `j` $\neq$ `index`:
    '`get(j)` *in post-state*' = '`get(j)` *in pre-state*'
- Under the precondition '`index` $\notin$ [0...`size()` $-$ 1]',
  **`set`** throws `IndexOutOfBoundsException`.

# Altogether:

`public Object set(int index, Object element)`

*Informal spec:*
Replaces the element at the specified position in this list with the specified element ... Returns the element previously at the specified position ... Throws `IndexOutOfBoundsException` if the index is out of range (index $<$ 0 || index $>=$ size()).

*Semi formal:*

- Under the precondition '`index` $\in$ [0...`size()` $-$ 1]',
  **`set`** ensures the following postconditions:
  - result = '`get(index)` *evaluated in the pre-state*', and
  - `element` = '`get(index)` *evaluated in the post-state*', and
  - for all `j` $\in$ [0...`size()` $-$ 1] with `j` $\neq$ `index`:
    '`get(j)` *in post-state*' = '`get(j)` *in pre-state*'
- Under the precondition '`index` $\notin$ [0...`size()` $-$ 1]',
  **`set`** throws `IndexOutOfBoundsException`.

# Reflection

We identify elements of a framework for *Formal Specification*

- pairs of
  - preconditions
  - corresponding postconditions
- a language to express these conditions, capturing:
  - relations, equality, logical connectives
  - *quantification*
- constructs to refer to:
  - values in the *new* and in the *old* state
  - the throwing of exceptions

To identify one more element, we consider another example.

# Reflection

We identify elements of a framework for *Formal Specification*

- pairs of
  - preconditions
  - corresponding postconditions
- a language to express these conditions, capturing:
  - relations, equality, logical connectives
  - *quantification*
- constructs to refer to:
  - values in the *new* and in the *old* state
  - the throwing of exceptions

To identify one more element, we consider another example.

# Consider Class SortedIntegers

```
public class SortedIntegers {

    private int arr[];
    private int capacity, size = 0;

    public SortedIntegers(int capacity) {
        this.capacity = capacity;
        this.arr = new int[capacity];
    }

    public void add(int elem) { /*...*/ }

    public boolean remove(int elem) { /*...*/ }

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

Which methods have purely functional / state aware specifications?

# Consider Class SortedIntegers

```
public class SortedIntegers {

    private int arr[];
    private int capacity, size = 0;

    public SortedIntegers(int capacity) {
        this.capacity = capacity;
        this.arr = new int[capacity];
    }

    public void add(int elem) { /*...*/ }

    public boolean remove(int elem) { /*...*/ }

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

Which methods have purely functional / state aware specifications?

# Specifying `SortedIntegers::max()`

**Specification of `int max()`**

`max()` returns the maximum of the elements in the array `arr`.

But that is not what we wanted.

`max()` should return the maximum of the elements which were already added, and not removed thereafter.

# Specifying `SortedIntegers::max()`

**Specification of `int max()`**

`max()` returns the maximum of the elements in the array `arr`.

But that is not what we wanted.

`max()` should return the maximum of the elements which were already added, and not removed thereafter.

# Specifying `SortedIntegers::max()`

**Specification of `int max()`**

`max()` returns the maximum of the elements in the array `arr`.

But that is not what we wanted.

`max()` should return the maximum of the elements which were already added, and not removed thereafter.