

## Collections

### Array

```
(* Array is passed by far reference to
   other actors *)
numbersArray1:: Array new: 10.
(* TransferArray is passed by copy to other
   actors *)
numbersArray2:: TransferArray new: 10.
(* ValueArray denotes an immutable array *)
numbersArray3:: ValueArray new: 10 withAll:
    [:i | i*i].
```

```
(* all types of arrays have the same API *)
1 to: 10 do[:i | numbersArray1 at: i put:
    i.].
numbersArray1 at: 1 → 1
numbersArray1 size → 10
```

### Vector

```
studentsVector:: Vector new: 10.
studentsVector append: 'Joe'.
(studentsVector at: 1) println. → Joe
(* iterating *)
studentsVector do: [:s | ('Student ' + s)
    println.].
studentsVector doIndexes: [:i |
    ('Student ' + (studentsArray at: i))
    println.].
```

### Dictionary

```
dictionary := Dictionary new: 10.
dictionary at: 'somns' put: 80.
dictionary containsKey: 'somns' → true
dictionary at: 'somns' → 80
```

## 4. Concurrency

### Actor Definition

```
(* createActorFromValue message creates an
   actor from Math value; it returns a far
   reference to the actor Math *)
mathFarRef:: (actors createActorFromValue:
    Math).
(* new message creates a new instance of
   the Math actor *)
mathActor:: mathFarRef <-: new.
```

## Implicit Promises

```
result:: mathActor <-: division: 27 and: 5.
(* Registering a callback for a promise;
   whenResolved: is applied when the result
   is available, onError: when an error
   happens; onError: is optional*)
result whenResolved[:div |
    ('Division result: ' + div) println.
] onError[:error |
    ('DivisionZeroError' + error) println. ].
```

## Promise Group

```
squareA:: mathActor <-: square: sideA.
squareB:: mathActor <-: square: sideB.
(* registers a promise for a group of
   promises stored in a table *)
squareA, squareB whenResolved:[
    :squaresVector | ... ].
, → concat. operator returns a table
```

## Explicit Promises

```
(* explicit promise creation *)
promisePair:: actors createPromisePair.
(* resolves the promise with a value *)
promisePair resolve: perimeter.
(* resolves the promise with an error *)
promisePair error: e.
(* accessing the promise object *)
promisePair promise
(* accessing the resolver object *)
promisePair resolver
```

## References

1. SOMNS: <https://github.com/smarr/SOMns>
2. Setup guide: <https://somns.readthedocs.io/>
3. Sample programs: <https://github.com/ctrlpz/somns-sample-programs>
4. The standard language library is accessible in the SDK of the project opened in IntelliJ: *core-lib*.

This cheat sheet has been adapted from the Smalltalk one at <http://sdmeta.gforge.inria.fr/Teaching/0809Turino/st-cheatsheet.pdf>

# SOMNS Cheat Sheet

Software Languages Lab  
Vrije Universiteit Brussel

November 2020

## 1. The SOMNS IntelliJ plugin

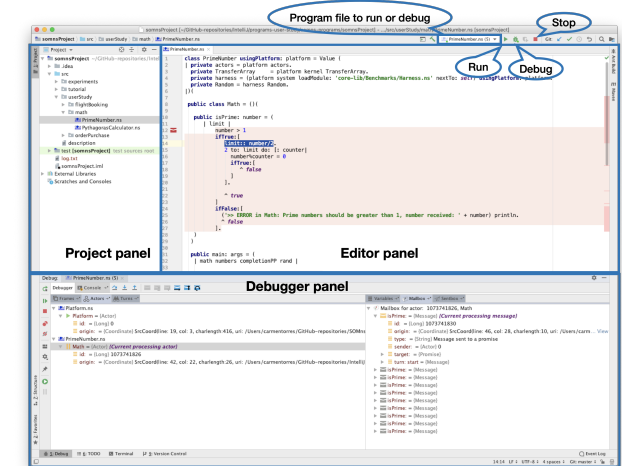


Figure 1: The SOMNS IntelliJ plugin

**Run it:** (CTRL+FN+SHIFT+F10) Evaluate selected .ns file.

**Debug it:** (CTRL+FN+SHIFT+F9) Evaluate selected .ns file step-by-step with the integrated debugger.

**Stop it:** (CMD+FN+F2) Stop program's execution, in run or debug mode.

## 2. The SOMNS Language

- Class-based OO inspired by Smalltalk: everything is an object. Everything happens by sending messages.
- Communicating Event-Loops actor model.
- Messages between objects within the same actor are sent synchronously and return a promise.
- Messages between objects in different actors are sent asynchronously.

Keywords

- self, the receiver.
- super, the receiver, method lookup starts in super-class.
- nil, the unique instance of the class Nil.
- true, the unique instance of the class True.
- false, the unique instance of the class False.

Literals

- Integer: 123
- Double: 123.4
- Boolean: true, false
- String: 'abc'
- Symbol: #ok
- Array:

obj:: object new.  
array:: { nil, false, #rr, obj }.

(array at: 1) —> nil.

(array at: 2) —> false.

(array at: 3) —> rr.

(array at: 4) —> instance of Object.

Message Sends

1. *Unary messages* take no argument.  
25 sqrt sends the message sqrt to the object 25.
2. *Binary messages* take exactly one argument.  
3 + 4 sends message + with argument 4 to the object 3. Binary selectors are built from one or more of the characters +, -, \*, =, <, >, etc.
3. *Keyword messages* take one or more arguments.  
2.0 pow: 6.0 sends the message named pow: with argument 6 to the object 2.

Unary messages are sent first, then binary messages and finally keyword messages:

2.0 pow: 2 + 16 sqrt —> 64

Messages are sent left to right. Use parentheses to change the order:

1 + 2 \* 3 —> 9

1 + (2 \* 3) —> 7

2

Syntax

- Comments  
(\* Comments are enclosed in parentheses and asterisks \*)
- Temporary variables  
| var1 var2 |
- Mutable variable declaration  
var ::= astatement
- Immutable variable declaration  
var = astatement
- Variable assignment  
var:: astatement
- Statements  
astatement1. astatement2

- Synchronous messages  
receiver message (unary msg)  
receiver + argument (binary msg)  
receiver message: argument (keyword msg)
- Asynchronous messages  
receiver (unary msg)  
receiver <:- message: arg (keyword msg)  
receiver <:- message: arg1 with: arg2
- Blocks  
[astatement1. astatement2]  
[:arg1: arg2 | | temp1 temp2 | statement]  
~ astatement
- Return statement  
Main class definition  
public class MainClassName usingPlatform:  
platform = Value (  
| slots |  
)  
(\* classes definitions and method definitions \*)  
public main: args = ( ^ (\* returns an integer as error code or a promise for program completion \*) )

3

Class definition

```
public class ClassName new: parameter1  
parameter2: parameter2 = (  
| slots |  
) ( body )
```

Method definition

```
messagesSelectorAndArgumentNames = (  
(* comment stating purpose of message *)  
| temporary variable names |  
statements )
```

3. Standard Classes

Logical Expressions

```
true not —> false  
1 = 2 or: [ 2 = 1 ] —> false  
1 < 2 and: [ 2 > 1 ] —> true
```

Conditionals

```
1 = 2 ifTrue: [ '1 is equal to 2' println.  
1 = 2 ifFalse: [ '1 is not equal to 2'  
println.]
```

Loops

```
(* conditional iteration *)  
[ student notNil ] whileFalse: [ 'student  
nil' ]  
[ student notNil ] whileTrue: [ (student  
name) println.]  
(* fixed iteration *)  
sum:: 0.  
100 timesRepeat: [  
sum::: sum + 1. ].
```

```
(* another fixed iteration *)  
1 to: 100 do: [ :index | index println. ].
```

Blocks (anonymous functions)

```
[ 1 + 2 ] value —> 3  
[ :x | x + 2 ] value: 1 —> 3  
[ :x: y | x + y ] value: 1 value: 2 —> 3
```

4