Syntax and Messages

- The syntax of Smalltalk is simple and uniform, but it can look strange at first sight!
- Literals: numbers, strings, arrays....
- Variable names
- Pseudo-variables
- Assignments, returns
- Message Expressions
- Block expressions

Made for Kids

Read it as a non-computer-literate person:

```
| bunny |
bunny := Actor fromFile: 'bunny.vrml'.
bunny head doEachFrame:
  [ bunny head
      pointAt: (camera
               transformScreenPointToScenePoint: (Sensor mousePoint)
               using: bunny)
      duration: camera rightNow ]
```

Numbers

- SmallInteger, Integer,
  - 4, 2r100 (4 in base 2),3r11 (4 in base 3), 1232
- Automatic coercion
  - 1 + 2.3 -> 3.3
  - 1 class -> SmallInteger
  - 1 class maxVal class -> SmallInteger
  - (1 class maxVal + 1) class -> LargeInteger
- Fraction, Float, Double
  - 3/4, 2.4e7, 0.75d
  - (1/3) + (2/3) -> 1
  - 1000factorial / 999 factorial -> 1000
  - 2/3 + 1 -> (5/3)

Characters

- Characters:
  - $F, $Q $U $E $N $T $i $N
- Unprintable characters:
  - Character space, Character tab, Character cr

Strings

- Strings:
  - #mac asString -> 'mac'
  - 12 printString -> '12'
  - 'This packet travelled around to the printer' 'l''idiot'
  - String with: $A
  - Collection of characters
  - 'lulu' at: 1 -> $l
- To introduce a single quote inside a string, just double it.

Symbols

- Symbols:
  - #class #mac #at:put: #+ #accept:
- Kinds of String
- Unique in the system (see after)

Comments and Tips

- "This is a comment"
- A comment can span several lines. Moreover, avoid putting a space between the " and the first character. When there is no space, the system helps you to select a commented expression. You just go after the " character and double click on it: the entire commented expression is selected. After that you can printIt or doIt, etc.

Arrays

#(1 2 3) #('lulu' (1 2 3)) -> #('lulu' #(1 2 3))

#(mac node1 pc node2 node3 lpr) an array of symbols.

When one prints it it shows #(#mac #node1 #pc #node2 #node3 #lpr)

- Byte Array (not in Squeak)
  #[1 2 255]

Arrays

- Heterogenous

  #('lulu' (1 2 3)) PrIt-> #('lulu' #(1 2 3))

  #('lulu' 1.22 1) PrIt-> #('lulu' 1.22 1)

- An array of symbols:

  #(calvin hobbes suzie) PrIt-> #(#calvin #hobbes #suzie)

- An array of strings:

  #('calvin' 'hobbes' 'suzie') PrIt-> #('calvin' 'hobbes' 'suzie')

Arrays and Literal Arrays

- Only the creation time differs between literal arrays and arrays. Literal arrays are known at compile time, arrays at run-time.
- #(Packet new) an array with two symbols and not an instance of Packet
- Array new at: 1 put: (Packet new) is an array with one element an instance of Packet
- Literal or not
  - #(...) considers elements as literals and false true and nil
  - #(1 + 2) PrIt-> #(1 #+ 2)
  - Array with: (1 +2) PrIt-> #(3)

Arrays with {} in Squeak

- { …} a shortcut for Array new: …

Array with: (1 +2) with: Packet new

<=>

{(1+2) . Packet new}

=>

#(3 aPacket)

Idioms linked to Array Weakness

- This internal representation of method objects has led to the following idioms to prevent unwanted side effects :
- Never give direct access to a literal array but only provide a copy.
- For example:

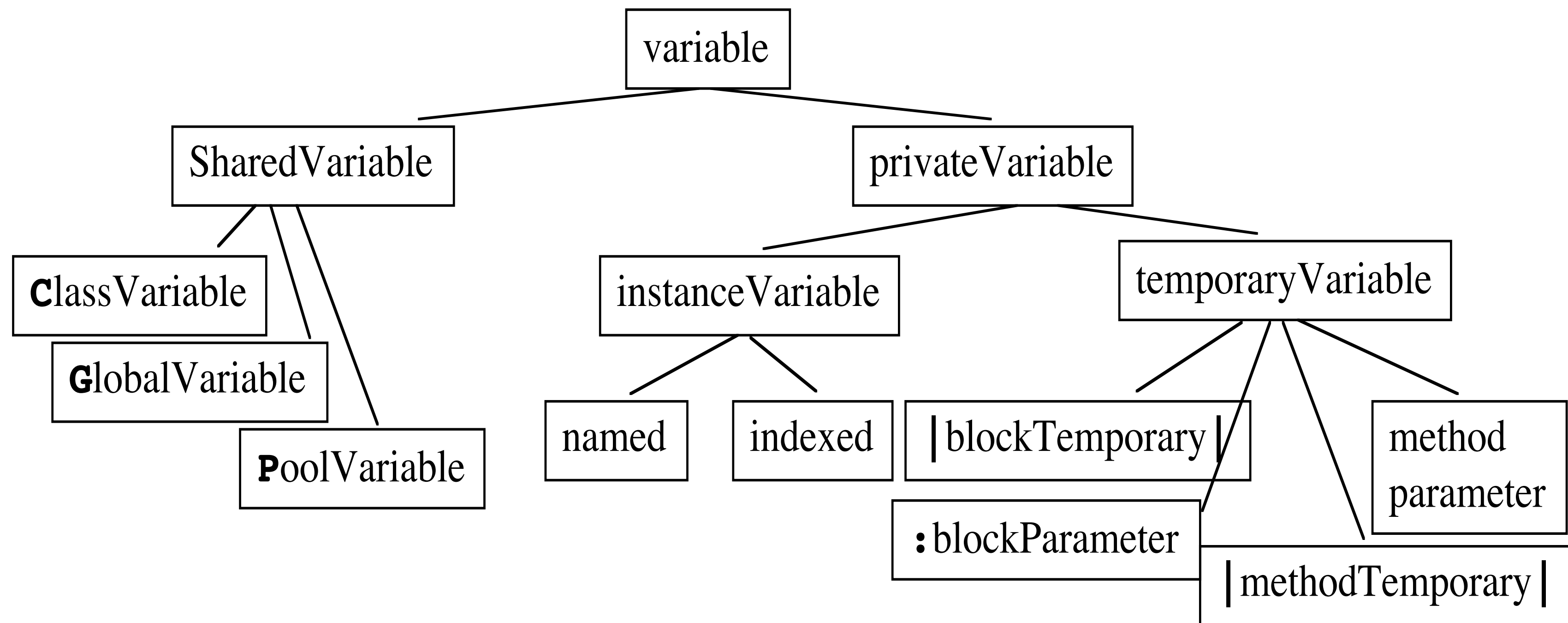
```
ar
  ^ #(100@100 200@200) copy
```

Symbols vs. Strings

- Symbols are used as method selectors, unique keys for dictionaries
- A symbol is a read-only object, strings are mutable objects
- A symbol is unique, strings are not
  #calvin == #calvin PrIt-> true
  'calvin' == 'calvin' PrIt-> false
  #calvin, #zeBest PrIt-> 'calvinzeBest'
- Symbols are good candidates for identity based dictionaries (IdentityDictionary)
- Hint: Comparing strings is slower then comparing symbols by a factor of 5 to 10. However, converting a string to a symbol is more than 100 times more expensive.

Variables

- Maintains a reference to an object
- Dynamically typed and can reference different types of objects
- Shared (starting with uppercase) or local (starting with lowercase)



Temporary Variables

- To hold temporary values during evaluation (method execution or sequence of instructions)
- Can be accessed by the expressions composing the method body.
  - |mac1 pc node1 printer mac2 packet|

Temporary Variables Good Style

- Avoid using the same name for a temporary variable and a method argument, an instance variable or another temporary variable or block temporary. Your code will be more portable. Do not write:

```
aClass>>printOn: aStream
|aStream|

...
```

- Instead, write:

```
aClass>>printOn: aStream
|anotherStream|

...
```

- Hint: Avoid using the same temporary variable for referencing two different objects

Assignments

- An Assignment is not done by message passing. It is one of the few syntactic elements of Smalltalk.

```
variable := aValue
three := 3 raisedTo: 1
variable1 := variable2 := aValue
```

- Avoid using var := var2 := var3
- To not try to know in which order the expressions is evaluated. You will write good code

Variables Pointing to the Same Object

- In Smalltalk, objects are manipulated via implicit pointers: everything is a pointer. Take care when different variables point to the same object:

```
p1 := p2 := 0@100
p1 x: 100
p1 PrIt-> 100@100
p2 PrIt-> 100@100
```

Method Arguments

- Can be accessed by the expressions composing the method.
- Exist during the execution of the defining method.
- Method Name Example:

```
accept: aPacket
```

- In C++ or Java:

```
void Printer::accept(aPacket Packet)
```

Arguments are read-only

- Method arguments cannot change their value within the method body.
- Invalid Example, assuming contents is an instance variable:

```
MyClass>>contents: aString
  aString := aString, 'From Lpr'.
```

- Valid Example

```
MyClass>>contents: aString
  | addressee |
  addressee := aString , 'From Lpr'
```

Instance Variables

- Private to a particular instance (not to all the instances of a class like in C++).
- Can be accessed by all the methods of the defining class and its subclasses.
- Has the same lifetime as the object.
- Declaration

```
	Object subclass: #Node
		instanceVariableNames: 'name nextNode '
		...
```

Instance Variables

- Scope: all the methods of the class

```
Node>>setName: aSymbol nextNode: aNode
	name := aSymbol.
	nextNode := aNode
```

- But preferably accessed using accessor methods

```
Node>>name
	^name
```

Six Pseudo-Variables

- Smalltalk expressions make references to true, false, nil, self, super thisContext, but cannot change their values. They are hardwired into the compiler.
- nil

  nothing, the value for the uninitialized variables.

  Unique instance of the class UndefinedObject

Six Pseudo-Variables

- true
  unique instance of the class True
- false
  unique instance of the class False

- Hint: Don't use False instead of false. false is the boolean value, False the class representing it. So, the first produces an error, the second not:

```
False ifFalse: [Transcript show: 'False']

false ifFalse: [Transcript show: 'False']
```

self, super, and thisContext

- Only make sense in a method body
- **self** refers to the receiver of a message.
- **super**
  refers also to the receiver of the message but its semantics affects the lookup of the method. It starts the lookup in the superclass of the class of the method containing the super.
- **thisContext**
  refers to the instance of MethodContext that represents the context of a method (receiver, sender, method, pc, stack). Specific to VisualWorks and to Squeak

self and super examples

```
PrinterServer>>accept: thePacket
  "If the packet is addressed to me, print it.
  Otherwise behave normally."

  (thePacket isAddressedTo: self)
     ifTrue: [self print: thePacket]
     ifFalse: [super accept: thePacket]
```

Global Variables

- Always Capitalized (convention)
  MyGlobalPi := 3.1415
- If it is unknown, Smalltalk will ask you if you want to create a new global
  Smalltalk at: #MyGlobalPi put: 3.14
  MyGlobalPi PrIt-> 3.14
  Smalltalk at: #MyGlobalPi PrIt-> 3.14
- Stored in the default environment: Smalltalk in Squeak, VW has namespaces

- Design Hints: Accessible from everywhere, but it is not a good idea to use them

Global Variables

- To remove a global variable:

  Smalltalk removeKey: #MyGlobal

- Some predefined global variables:

  Smalltalk (classes + globals)

  Undeclared (aPoolDictionary of undeclared variables accessible from the compiler)

  Transcript (System transcript)

  ScheduledControllers (window controllers)

  Processor (a ProcessScheduler list of all the process)

Objects and Messages

Objects and Messages

- Objects communicate by sending message
- Objects react to messages by executing methods

```
Turtle new go: 30 + 50
```

A message is composed of:

a receiver, always evaluated (Turtle new)

a selector, never evaluated #go:

and a list possibly empty of arguments that are all evaluated (30 + 50)

The receiver is linked with self in a method body.

Three Kinds of Messages

Unary Messages

```
2.4 inspect
macNode name
```

- Binary Messages

```
1 + 2 -> 3
(1 + 2) * (2 + 3) PrIt-> 15
3 * 5 PrIt-> 15
```

- Keyword Messages

```
6 gcd: 24 PrIt-> 6
pcNode nextNode: node2
Turtle new go: 30 color: Color blue
```

Unary Messages

aReceiver aSelector

- node3 nextNode -> printerNode
- node3 name -> #node3
- 1 class PrIt-> SmallInteger
- false not PrIt-> true
- Date today PrIt-> Date today September 19, 1997
- Time now PrIt-> 1:22:20 pm
- Double pi PrIt-> 3.1415926535898d

Binary Messages

aReceiver aSelector anArgument

- Used for arithmetic, comparison and logical operations
- One or two characters taken from:
  + - / \ * ~ < > = @ % | & ! ? ,

```
1 + 2
2 >= 3
100@100
'the', 'best'
```

- Restriction:
  second character is never $-

Simplicity has a Price

no mathematical precedence so take care

3 + 2 * 10 -> 50

3 + (2 * 10) -> 23

(1/3) + (2/3) and not

1/3 + 2/3

Keyword Messages

receiver keyword1: argument1 keyword2: argument2

1 between: 0 and: 5

dict at: #blop put: 8+3

In C-like languages it would be:

receiver.keyword1keyword2...(argument1 type1, argument2, type2) : return-type

Keyword Messages

```
Workstation withName: #Mac2
mac nextNode: node1
Packet send:  'This packet travelled around to' to:
  #lw100
1@1 setX: 3
#(1 2 3) at: 2 put: 25
1 to: 10 -> (1 to: 10) anInterval
Browser newOnClass: Point
Interval from:1 to: 20 PrIt-> (1 to: 20)
12 between: 10 and: 20 PrIt-> true
x > 0 ifTrue:['positive'] ifFalse:['negative']
```

Composition Rules

- Unary-Msg > Binary-Msg > Keywords-Msg
- at same level, from the left to the right

```
2 + 3 squared -> 11
2 raisedTo: 3 + 2 -> 32
#(1 2 3) at: 1+1 put: 10 + 2 * 3 ->  #(1 36 3)

2 raisedTo: 3 + 2 <=> (2 raisedTo: (3+2)) -> 32
```

Composition Rules

(Msg) > Unary-Msg > Binary-Msg > Keywords-Msg

```
69 class inspect
(0@0 extent: 100@100) bottomRight
```

Hints for Keyword Msg Composition

Use () when two keyword-based messages occur within a single expression, otherwise the precedence order is fine.

```
x isNil ifTrue: [...]
```

isNil is an unary message, so it is evaluated prior to **ifTrue:**

```
x includes: 3 ifTrue: [...]
```

is read as the message includes:ifTrue:

```
(x includes: 3) ifTrue: [...]
```

We use () to disambiguate them

Sequence

```
message1 .
message2 .
message3
```

. is a separator, not a terminator

```
| macNode pcNode node1 printerNode |
macNode := Workstation withName: #mac.
Transcript cr.
Transcript show: 1 printString.
Transcript cr.
Transcript show: 2 printString
```

For Lazy: the cascade

```
receiver
    selector1;
    selector2; ...
```

To send multiple messages to the same object

```
Transcript show: 1 printString.
Transcript show: cr
```

is equivalent to:

```
Transcript show: 1 printString ; cr
```

Let's be Precise!

The semantics of the cascade is to send all the messages in the cascade to the receiver of the FIRST message involved in the cascade.

Workstation new name: #mac ; nextNode: aNode

Where the msg name: is sent to the newly created instance of workstation and the msg nextNode: too.

Let’s be Precise!

(OrderedCollection with: 1) add: 25; add: 35

In the example the FIRST message involved in the cascade is the first add: msg and not #with:. So all the messages are sent to the result of the parenthesised expression, the newly created instance of anOrderedCollection

One Problem

```
(OrderedCollection with: 1)
        add: 25;
        add: 35
```

PrIt-> 35

One problem: the expression returns 35 and not the collection object.

Let us analyze a bit…

```
OrderedCollection>>add: newObject
  "Include newObject as one of the receiver's elements.  Answer
  newObject."
  ^self addLast: newObject

OrderedCollection>>addLast: newObject
  "Add newObject to the end of the receiver.  Answer
  newObject."
  lastIndex = self basicSize ifTrue: [self makeRoomAtLast].
  lastIndex := lastIndex + 1.
  self basicAt: lastIndex put: newObject.
  ^newObject
```

Yourself: Accessing the Receiver of a Cascade

- Use yourself
- yourself returns the receiver of the cascade.

```
(OrderedCollection with: 1)
        add: 25;
        add: 35 ;
        yourself
-> OrderedCollection(1 25 35)
```

Really got it?

yourself returns the receiver of the cascade:

Here the receiver of the cascade is a newly created instance anOrderedCollection and not the class OrderedCollection. The self in the yourself method is linked to this instance

```
(OrderedCollection with: 1) add: 25; add: 35 ; yourself
      anOrderedCollection(1) = self
```

- So what is the code of yourself?

```
Object>>yourself
    ^ self
```

Blocks

- A deferred sequence of actions
- The return value is the result of the last expression of the block
- Similar to Lisp Lambda-Expressions, C functions, anonymous functions or procedures
- Delimited by [ ]

Block Example

fct(x) = x ^ 2 + x

fct (2) = 6

fct (20) = 420

```
|fct|
fct:= [:x | x * x + x].
fct value: 2 PrIt-> 6
fct value: 20 PrIt-> 420
fct PrIt-> aBlockClosure
```

Other Blocks

```
[ :variable1 :variable2 |
    | blockTemporary1 blockTemporary2 |
    expression1.
    ...variable1 ...      ]
```

- Two blocks without arguments and temporary variables

```
    PrinterServer>>accept: thePacket
        (thePacket isAddressedTo: self)
        ifTrue: [self print: thePacket]
        ifFalse: [super accept: thePacket]
```

Block Evaluation

```
[....] value
        or value:                   (for one arg)
        or value:value:             (for two args)
        or value:value:value:            …
        or valueWithArguments: anArray

[2 + 3 + 4 + 5] value
[:x | x + 3 + 4 + 5 ] value: 2
[:x :y | x + y + 4 + 5] value: 2 value: 3
[:x :y :z | x + y + z + 5] value: 2 value: 3 value: 4
[:x :y :z :w | x + y + z + w] value: 2 value: 3 value: 4
  value: 5
```

Block

- The value of a block is the value of its last statement, except if there is an explicit return ^
- Blocks are first class objects.
- They are created, passed as argument, stored into variables...

Blocks - Continued

```
|index bloc |
index := 0.
bloc := [index := index +1].
index := 3.
bloc value -> 4
```

```
Integer>>factorial
  "Answer the factorial of the receiver. Fail if the receiver is
  less than 0."
        | tmp |
        ....
        tmp := 1.
        2 to: self do: [:i | tmp := tmp * i].
        ^tmp
```

Blocks - Continued

- For performance reasons, avoid referring to variables outside a block.
- Or using ^ inside blocks

A Word about Primitives

- For optimization, if a primitive fails, the code following is executed.

```
Integer>>@ y
	"Answer a new Point whose x value is the receiver
	and whose y value is the argument."

	<primitive: 18>
	^Point x: self y: y
```

At the End of the Smalltalk World

We need some operations that are not defined as methods on objects but direct calls on the underlying implementation language (C, Assembler,....)

```
== anObject
  "Answer true if the receiver and the argument are the same
  object (have the same object pointer) and false otherwise.  Do
  not redefine the message == in any other class!  No Lookup."

  <primitive: 110>
  self primitiveFailed
```

+ - < >* / = == bitShift:\\ bitAnd: bitOr: >= <= at: at:put: new new:

What we saw

- Numbers (integer, real, float…), Character $a, String 'abc', Symbols (unique Strings) #jkk, Arrays (potentially not homogenous) #(a #(1 2 3), Array with: 2+3 <=> {2+3}
- Variables:
  - Lowercase => private
    - Instance variables (visible in by all methods), method arguments (read-only), local variable |a|
  - Uppercase => global
- Pseudo Var: true, false, nil, self, super
  - self = **always** represents the msg receiver
  - nil = undefined value

What we saw

- Three kinds of messages
  - Unary: Node new
  - Binary: 1 + 2, 3@4
  - Keywords: aTomagoshi eat: #cooky furiously: true
- (Msg) > unary > binary > keywords
- Same Level from left to right
- Block
  - Functions

    fct(x)= x*x+3, fct(2).

    fct :=[:x| x * x + 3]. fct value: 2
  - Anonymous method
  - Passed as method argument:

```
factorial
  tmp:= 1.
  2 to: self do: [:i| tmp := tmp * i]
```