Outline
- Literals: numbers, strings, arrays.....
- Variable, assignments, returns
- Pseudo-variables
- Message Expressions
- Block expressions
- Conditional and Loops

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

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, 2.467, 0.75d
  - (1/3) + (2/3) -> 1
- 1000 factorial / 999 factorial -> 1000
- 2/3 + 1 -> (5/3)

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

Strings
- Strings:
  - #mac asString -> 'mac'
  - 12 printString -> '12'
  - 'This packet travelled around 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 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' -> true
  - #calvin == 'calvin' -> false
  - #calvin, #zeBest. -> '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.

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 print it or do it, etc.
Temporary Variable 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. Four code will be more portable. Do not write:

  aClass>>printOn:aStream |aStream|

  ...|anotherStream|

  ! ! ! ...

- Instead, write:

  aClass>>printOn:aStream |anotherStream|

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

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 |
```

Arrays

- Heterogenous

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

- An array of objects:

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

Syntax Summary

- Arrays

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

- An array of strings:

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

Roadmap

- Literals: numbers, strings, arrays
  - Variable, assignments, returns
  - Pseudo-variables
  - Message Expressions
  - Block expressions
  - Conditional and loops

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

Temporary Variable 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. Four code will be more portable. Do not write:

  aClass>>printOn:aStream |aStream|

  ...|anotherStream|

- Instead, write:

  aClass>>printOn:aStream |anotherStream|

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

Syntax Summary

- Arrays

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

- An array of objects:

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

- An array of strings:

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

- Literals: numbers, strings, arrays
  - Variable, assignments, returns
  - Pseudo-variables
  - Message Expressions
  - Block expressions
  - Conditional and loops

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 |
```

Arrays

- Heterogenous

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

- An array of symbols:

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

- When one prints it it shows

```
#(mac node1 pc node2 node3 lpr)
```

- An array of strings:

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

- Byte Array

```
#[1 2 255]
```

- Integer

```
1,2,3
```

- Float

```
1.5, 6.03e-34, 4.2e7
```

- Boolean

```
true, false
```

- Pointer

```
10@120
```

Roadmap

- Literals: numbers, strings, arrays
  - Variable, assignments, returns
  - Pseudo-variables
  - Message Expressions
  - Block expressions
  - Conditional and loops

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

Syntax Summary

- Arrays

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

- An array of objects:

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

- An array of strings:

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

- Literals: numbers, strings, arrays
  - Variable, assignments, returns
  - Pseudo-variables
  - Message Expressions
  - Block expressions
  - Conditional and loops

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 |
```

Arrays

- Heterogenous

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

- An array of symbols:

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

- When one prints it it shows

```
#(mac node1 pc node2 node3 lpr)
```

- An array of strings:

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

- Byte Array

```
#[1 2 255]
```

- Integer

```
1,2,3
```

- Float

```
1.5, 6.03e-34, 4.2e7
```

- Boolean

```
true, false
```

- Pointer

```
10@120
```

Roadmap

- Literals: numbers, strings, arrays
  - Variable, assignments, returns
  - Pseudo-variables
  - Message Expressions
  - Block expressions
  - Conditional and loops

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

Syntax Summary

- Arrays

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

- An array of objects:

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

- An array of strings:

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

- Literals: numbers, strings, arrays
  - Variable, assignments, returns
  - Pseudo-variables
  - Message Expressions
  - Block expressions
  - Conditional and loops

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 |
```
**Method Arguments**
- Can be accessed by the expressions composing the method.
- Exist during the execution of the defining method.
- Method Name Example:
  ```Smalltalk
  accept: aPacket
  ```
- In C++ or Java:
  ```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:
  ```Smalltalk
  MyClass>>contents: aString
  aString := aString, 'From Lp'.
  ```
- Valid Example
  ```Smalltalk
  MyClass>>contents: aString
  | addressee |
  | addressee := aString, 'From Lp'
  ```

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

**Six Pseudo-Variables**
- Smalltalk expressions can contain 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

**Method Return**
- Use ^ expression to return the value of expression from a method
  ```Smalltalk
  Rectangle>>area
  ^ width * height
  ```
- By default self is returned

**Global Variables**
- Always Capitalized (convention)
  ```Smalltalk
  MyGlobalPi := 3.1415
  ```
- If it is unknown, Smalltalk will ask you if you want to create a new global
  ```Smalltalk
  MyGlobalPi := 3.1415
  ```
- Design Hints: Accessible from everywhere, but it is not a good idea to use them

**Roadmap**
- Literals: numbers, strings, arrays
- Variable, assignments, returns
- Pseudo-variables
- Message Expressions
- Block expressions
- Conditional and Loops
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

Objects and Messages

- Objects communicate by sending message
- Objects react to messages by executing methods
- Bot new: 30 + 50
- A message is composed of:
  - a receiver, always evaluated (Bot new)
  - a selector, never evaluated if it's a constant.
  - 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) -> 15
  - 3 * 5 -> 15
- Keyword Messages
  - 6 gcd: 24 -> 6
  - goNode nextNode: node2
  - Turtle new go: 30 color: Color blue

Unary Messages

- aReceiver aSelector
- node3 nextNode -> printerNode
- node3 name -> fnode3
- 1 class -> SmallInteger
- false not -> true
- Date today -> Date today September 19, 1997
- Time now -> 1:22:20 pm
- Double pi -> 3.1415926535898d

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
dct at: 'hi' skip: 3
- In C-like languages it would be:
  - receiver keyword1:keyword2...(argument1 type1, argument2, type3) : return-type
Conditional: messages to booleans

- aBoolean ifTrue: aTrueBlock ifFalse: aFalseBlock
- aBoolean ifTrue: aTrueBlock ifFalse: aFalseBlock

 Hint: Take care — true is the boolean value and True is the class of true, its unique instance!

Boolean Messages

- Logical Comparisons: &, |, xor, not
  aBooleanExpr comparison aBooleanExpr
  - (1 isZero) & false
  - Date today isRaining not
  - Uniform, but optimized and inlined (macro expansion at compile time)
- aBooleanExpression or: orBlock
  orBlock will only be evaluated if aBooleanExpression is false
  and: [1 error: 'crazy']
  -> false and not an error

Blocks - Continued

- 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

Blocks Continued

- Variables / variables 2
  blockTemporary1 blockTemporary2
- expression 1
  ...variable 1 ...

- 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...

Weather isRaining

ifTrue: [self takeMyUmbrella]
ifFalse: [self takeMySunglasses]

ifTrue ifFalse is sent to an object: a boolean!

Evaluation of a block

[...]
!! or value: ! (for one arg)
!! or value:value:!           (for two ... :z | x + y + z + 5] value: 2 value: 3 value: 4 value: 5

Yes ifTrue: is sent to a boolean

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...

Conditional and Loops

- Literal: numbers, strings, arrays...
- Variable, assignments, return...
- Pseudo-variables
- Message Expressions!
- Block expressions
- Conditional and Loops

Blocks

- 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...

- The value of a block is the value of its last statement, except if there is an explicit return ^

Conditional and Loops
**Some Basic Loops**

- `aBlockTest whileTrue
  aBlockTest whileFalse
  aBlockTest whileTrue: aBlockBody
  aBlockTest whileFalse: aBlockBody
  aBlockTest whileTrue: aBlockBody
  aBlockTest whileFalse: aBlockBody
  aBlockTest whileTrue: aBlockBody
  aBlockTest whileFalse: aBlockBody
  aBlockTest whileTrue: aBlockBody
  aBlockTest whileFalse: aBlockBody
  10 timesRepeat: [ Transcript show: 'hello' ; cr ]`

**Opening the Box**

Iterators are messages sent to collection objects
Collection is responsible of its traversal
SequenceableCollection>>do: aBlock
"Evaluate aBlock with each of the receiver's elements as the argument."
1 to: self size do: [:i | aBlock value: (self at: i)]

**Choose your Camp (II)**

- You could also write:

  ```smalltalk
  (2 -3 4 -35 4 -11) collect: [:each | each abs ]
  ```

- Really important: Contrary to the first solution, the second solution works well for indexable collections and also for sets.

```
for i in range(10):
    Transcript show: (i ** 2).printString; cr.
```

```
for i in range(10):
    Transcript show: (i ** 3).printString; cr.
```

**For the Curious...**

```
BlockClosure>>whileTrue: aBlock
   ^ self value ifTrue: [ aBlock value. 
         self whileTrue: aBlock ]
BlockClosure>>whileTrue
   ^ [ self value ] whileTrue: []
```

**Iteration Abstraction: do:/collect:**

```
aCollection do: aOneParameterBlock
aCollection collect: aOnePredicateBlock
aCollection detect: aOneParameterPredicateBlock
aCollection reject: aOnePredicateBlock
```

**Select:/Reject:/Detect:**

```smalltalk
aCollection select: aPredicateBlock
aCollection reject: aPredicateBlock
aCollection detect: aOneParameterPredicateBlock
ifNone: aNoneBlock
```

```
(15 10 19 68) do: [:i | Transcript show: (i ** 2) printString; cr.]
```

```
(15 10 19 68) collect: [:i | i odd] ifNone: [1]
```

```
(12 10 19 68) detect: [:i | i odd] ifNone: [1].
```

**Inject:Into:**

```
aCollection inject: aStartValue into: aBinaryBlock
```

```
<table>
<thead>
<tr>
<th>acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

```
(1 2 3 4 5) do: [:element | acc := acc + element].
acc.
```

Do not use it if the resulting code is not crystal clear!
Other Collection Methods

- aCollection includes: anElement
- aCollection size
- aCollection isEmpty
- aCollection contains: aBooleanBlock

(1 2 3 4 5) includes: 4 -> true
(1 2 3 4 5) size -> 5
(1 2 3 4 5) isEmpty -> false
(1 2 3 4 5) contains: [:each | each isOdd] -> true

What we saw

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

What we saw

- Three kinds of messages
  - Unary: Node new
  - Binary: 1 + 2, 3@4
  - Keywords: aTomagoshi eat: ... 2
- Anonymous method
- Passed as method argument:
  - Anonymous method
  - Passed as method argument:
    - Functions
      - $f(x)$ = $x^2 + 3$, $f(2)$
      - $f(x) = x^2 + 3$ for $x$ in $2$
    - Anonymous method
    - Passed as method argument:
      - Function
        - $\text{factorial}(n) = 1$
        - $2 \text{ to } self \{ i | \text{temp} = \text{temp} \times i \}$