Files and Streams

work := FileSystem disk workingDirectory.
stream := (work / 'Foo.txt') writeStream.
stream nextPutAll: 'Hello World'.
stream close.
stream := (work / 'Foo.txt') readStream.
stream contents.         \=> 'Hello World'
stream close.

Pharo: a Live Programming Environment

Pharo comes with an integrated development environment. Pharo is a live programming environment: you can modify your objects and your code while your program is executing. All Pharo tools are implemented in Pharo:

- a code browser with refactorings;
- a debugger, a workspace, and inspectors;
- the compiler itself and much, much more.

Code can be inspected and evaluated directly in the image, using simple key combinations and menus (open the contextual menu on any selected text to see available options).

The 5 Panes Pharo Code Browser

- The packages pane shows all the packages of the system.
- The classes pane shows the class hierarchy of the selected package; the class side checkbox allows for getting the methods of the metaclass.
- The protocols pane groups the methods of the selected class to ease navigation. When a protocol name starts with a *, methods of this protocol belong to a different package (e.g., the *Fuel protocol groups methods that belong to the Fuel package);
- The methods pane lists the methods of the selected protocol; icons are clickable and trigger special actions;
- The source code pane shows the source code of the selected method.

Defining a class

To add a class or edit a class, edit the proposed template! The following expression defines the class Counter as a subclass of Object. It defines two instance variables count and initialValue inside the package MyCounter.

    Object subclass: #Counter
        instanceVariableNames: 'count initialValue'
        classVariableNames: ''
        package: 'MyCounter'

The method initialize is automatically invoked when a new instance is created by sending the message new to the class i.e., Counter new.

    Counter >> initialize
        super initialize.
        count := 0.

    Counter >> initialize is a notation to indicate that the following text is the content of the method initialize in the class Counter.

Methods

Methods are public and virtual. They are always looked up in the class of the receiver. By default a method returns self. Class methods follow the same dynamic lookup as instance methods.

    Integer >> factorial
        "Answer the factorial of the receiver."
        self = 0 ifTrue: [^ 1].
        self > 0 ifTrue: [^ self * (self - 1) factorial].

Unit testing

A test must be implemented in a method whose name starts with test and in a class that inherits from TestCase.

    OrderedCollectionTest >> testAdd
        | added |
        added := collection add: 'foo'.
        self assert: (collection includes: 'foo').

A simple, uniform and powerful model

Pharo has a simple dynamically-typed object model:
Message Sending

When we send a message to an object (the receiver), the corresponding method is selected and executed, and the method answers an object. Message syntax mimics natural languages, with a subject, a verb, and complements.

```
| foo bar | declaration of two temporary variables
| var := expr | assignment
| expr1. expr2 | period - statement separator
| [:p | expr] | code block with a parameter
| unary? | method annotation
| <key: 'any' wrd:#lit | with any literal arguments
| ~ expr | caret - return/answer a result from a method
```

### Three Types of Messages: Unary, Binary, and Keyword

A unary message has no arguments.

```
Array new.    ~> anArray
#(4 2 1) size. ~> 3
```

new is an unary message sent to classes (classes are objects).

A binary message takes only one argument and is named by one or more symbol characters from +, -, *, =, <, >, ...

```
3 + 4    ~> 7
'H ello', 'W orld'    ~> 'Hello World'
```

The + message is sent to the object 3 with 4 as argument. The string 'Hello' receives the message , (comma) with 'World' as the argument.

A keyword message can take one or more arguments that are inserted in the message name.

```
'PharO' allButFirst: 2. ~> 'aro'
[:x | x + 2] value: 7 ~> 9
3 to: 10 by: 2. ~> (3 to: 10 by: 2)
```

The second line executes a block. The third example sends to: by: to 3, with arguments 10 and 2; this returns an interval containing 3, 5, 7, and 9.

### Message Precedence

Parentheses > unary > binary > keyword, and finally from left to right.

```
(15 between: 1 and: 2+4*3) not ~> false
```

Messages + and * are sent first, then between: and: is sent, and not. The rule suffers no exception: operators are just binary messages with no notion of mathematical precedence. 2+4*3 reads left-to-right and gives 18, not 14!

### Cascade: Sending Multiple Messages to the Same Object

Using ; (a cascade) multiple messages are sent to the result of the same expression. Here ; arrives after add: 1, so messages add: 2 and add: 3 are sent to add: 1's receiver: a collection.

```
OrderedCollection new
add: 1;
add: 2;
add: 3.
```

The whole message cascade value is the value of the last message sent (here 3). To return the receiver of the message cascade instead (i.e., the collection), send yourself as the last message of the cascade.

### Blocks

Blocks are objects containing code that is executed on demand. They are the basis for control structures: conditionals & loops.

```
2 = 2
ifTrue: [ Error signal: 'Help' ].
```

Send the message ifTrue: to the boolean true (computed from 2 = 2) with a block as argument. Because the boolean is true, the block is executed and an exception is signaled.

```
#('Hello World' $!)
do: [:e | Transcript show: e ]
```

Send the message do: to an array. This executes the block once for each element, passing it via the e parameter. As a result, Hello World! is printed.

### Common Constructs

#### Conditionals

<table>
<thead>
<tr>
<th>condition</th>
<th>if (condition)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifTrue: [ action ]</td>
<td>{ action(); }</td>
</tr>
<tr>
<td>ifFalse: [ anotherAction ]</td>
<td>else { anotherAction(); }</td>
</tr>
</tbody>
</table>

```
[ condition ] whileTrue: [ action(); anotherAction(); ]
```

<table>
<thead>
<tr>
<th>Loops/Iterators</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 to: 11 do: [:i</td>
<td>Transcript show: i ]; for(int i=1; i&lt;11; i++)</td>
</tr>
<tr>
<td>names := #('A', 'B', 'C'); names do: [:each</td>
<td>Transcript show: each];</td>
</tr>
</tbody>
</table>

Collections start at 1. aCol at: i accesses element at i and aCol at: i put: value sets element at i to value.

#### Collections

```
#(4 2 1) at: 3 ~> 1
#(4 2 1) copy at: 3 put: 6 ~> #(4 2 6)
(4 . 2 . 1) at: 3 put: 6 ~> #(4 2 6)
```

```
(ArrAy new: 2) add: 4; add: 2; yourself ~> #((4 2)
Set new add: 4; add: 4; yourself ~> aSet(4)
```

Dictionary new
```
at: #a put: 'Alpha'; yourself ~> a Dictionary(#a-
>`Alpha`)```