

Introduction to Blocks

Damien Cassou, Stéphane Ducasse and Luc Fabresse

W2S06



<http://www.pharo.org>



Blocks

Blocks are:

- kind of anonymous methods
 - also called (lexical) closures
- used everywhere in Pharo
 - loops, conditionals, iterators, ...
 - GUI frameworks, DSLs, ...
 - at the heart of the system
- just introduced in Java 8.0



Block Definition

A block is **defined** by []

[expressions. ...]



Block Definition Does not Execute Code

- Executing code may signal an Error

```
(1/0)  
-> Error
```

- But, no error when **defining** a block
 - a block definition does not execute its body
 - a block definition freezes its body computation

```
[1/0]  
> [1/0]
```

```
[1/0].  
1+2  
> 3
```

Executing a Block

Executing a block is done **explicitly** through value

```
[ 2 + 6 ] value  
> 8
```

```
[ 1 / 0 ] value  
> Error
```

A Block with 1 Argument

A block can take arguments (just like methods)

```
[ :x | x + 2 ]
```

- [] delimits the block
- :x is a block argument
- x + 2 is the block body

```
[ :x | x + 2 ] value: 5  
> 7
```

- value: 5 executes the block with 5 as argument
 - x is 5 during the block execution

Block Execution Value

Block execution returns the value of the last expression

```
[ :x |  
  x + 33.  
  x + 2 ] value: 5  
> 7
```

Blocks can be Stored

- A block can be stored in a variable
- A block can be evaluated multiple times

```
| add2 |  
add2 := [ :x | x + 2 ].
```

```
add2 value: 5.  
> 7
```

```
add2 value: 33  
> 35
```



Defining a Block with 2 Arguments

Example:

```
[ :x:y | x + y ]
```

`:x :y` are block arguments

How to execute a block with two arguments?

```
[ :x:y | x + y ] ??? 5 7  
> 12
```

Executing a Block with 2 Arguments

```
[ :x:y | x + y ] value: 5 value: 7  
> 12
```

- value: 5 value: 7 evaluates the block with 5 and 7
 - x is 5 and y is 7 during the block evaluation

A Block with Temporary Variables

Blocks can define temp. variables (just like methods)

```
Collection>>affect: anObject when: aBoolean  
  self do: [ :index | | args |  
    args := ....  
    aBoolean  
    ifTrue: [ anObject do: args ]  
    ifFalse: [ anObject doDifferently: args ] ].
```

- | args | defines a temporary variable named args
- args exists only during block evaluation



Returning from a Block Returns from the Method

When a return [^] is executed in a block, computation exits the method defining the block

```
Integer>>factorial
```

```
"Answer the factorial of the receiver."
```

```
self = 0 if True: [ ^ 1 ].
```

```
self > 0 if True: [ ^ self * (self - 1) factorial ].
```

```
self error: 'Not valid for negative integers'
```

```
0 factorial
```

```
>1
```

```
42 factorial
```

```
>1405006117752879898543142606244511569936384000000000
```

A Design Advice

- Use blocks with 2 or 3 arguments maximum
- Define a class instead of a block for more arguments
- A block encapsulates only 1 computation
 - it cannot define more facets (e.g., printing)



Summary on Blocks

```
[ :variable1 :variable2 ... |  
  | tmp |  
  expression1.  
  ... variable1 ...  
] value: ... value: ...
```

- Kind of anonymous method
- Technically lexical closures
- Can be stored in variables and method arguments
- Basis of conditionals, loops and iterators (see companion lectures)
- Further readings: <http://deepintopharo.org>

A course by



and



in collaboration with



Inria 2020

Except where otherwise noted, this work is licensed under CC BY-NC-ND 3.0 France

<https://creativecommons.org/licenses/by-nc-nd/3.0/fr/>