Introduction to Blocks

Stéphane Ducasse and Damien Cassou

http://stephane.ducasse.free.fr/ stephane.ducasse@inria.fr

Objectives

- Called closures or lexical closures in other languages.
- Just introduced in Java 8.0.
- Really important and are at the heart of Pharo.
- Used for loops, conditionals and iterators.
- You can define your own control flow.
- Used in UI development.
- Really powerful concept

- As a first approximation, blocks are kind of anonymous methods

Block Syntax

- a block is delimited by `[ ]`

```
[ expressions.... ]
```

A Block

- Executing `(1 / 0)` raises an error.

```
(1 / 0)
—> Error
```

- Executing `[ 1 / 0 ]` does not raise an error because the block body is not executed.

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

- If we do not ask a block to be executed, nothing happens.

A Block is Freezing Computation

- A block is not executed.
- A block blocks execution: its body is not executed.

```
[ 2 + 6 ]
> [ 2 + 6 ]
```

Another view

- Turns a program into ’data’

```
1
> 1

’abc’
> ’abc’

[ 2 + 6 ]
> [ 2 + 6 ]
```

Executing a Block

To execute a block we should ask **explicitly** its execution using the message `value`

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

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

A Block with one argument

- A bloc can take arguments (the same way a method can)

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

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

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

- `value:` is a message that executes a block passing a value, here 5 as argument. `x` will have the value 5.

Block execution value

- Execution returns the value of the last expression

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

Blocks can be stored

- We can store a block in variable
- A block can be executed multiple times

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

b value: 5
> 7

b value: 33
> 35
```

Blocks are used to express conditions

```
max: anObject
"Answer the receiver or the argument, whichever has the greater anObject."

self > anObject
ifTrue: [^ self ]
ifFalse: [^ anObject ]
```

Yes this is a message `ifTrue:ifFalse:` sent to a Boolean

Blocks are used to express loops

- Some simple loops
- Printing 10 dots

```
10 timesRepeat: [ File stdout << '.' ]
> ...........
```

Blocks are used to express loops

```
1 to: 10 do: [:i | File stdout << i ]
> 12345678910
```

Blocks are used to express loops

- a traditional `for` loop for i=1,100, i++

```
1 to: 100 by: 3 do: [:i | File stdout << i ]
> 147101316192225283134374043464952555861646770737679828588919497100
```

Blocks are used to express loops

- Basis for iterators

```
#(2 4 5 −4 3 −2) collect: [ :each | each abs ]
> #(2 4 5 4 3 2)
```

Full Syntax

```
[ :blockArg1 :blocArg2 |
  | localVariable |
  expression1.
  expression2.

  expressionn ]
```

A Design Advice

- Do not use blocks with too many arguments (3 max).
- Use object instead of block if you should pass more arguments.
- A block is only one single computation it cannot embed more facets (printing, testing)

Return in a bloc, return from the method

- When a block containing a return is executed, computation exits the method that defined the block.

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

    self = 0 ifTrue: [ ^ 1 ].
    self > 0 ifTrue: [ ^ self * (self − 1) factorial ].
    self error: 'Not valid for negative integers'
```

More precisely

- When a block containing a return is executed, computation returns from the method that defined the block.
- Since blocks can be passed around, from methods to methods, blocks behaves as an exception mechanism.
- Do not overuse this mechanim, better use Exception
- Always think twice when you put a return in a block

Exercises

- Guess how to execute a block taking two arguments

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

- Read the BlockClosure class
- Propose a non recursive definition of factorial

Other examples

```
[ 2 + 3 + 4 + 5 ] value
> 14
[ :x | x + 3 + 4 + 5 ] value: 2
> 14
[ :x :y | x + y + 4 + 5] value: 2 value: 3
> 14
```

Yes ifTrue:ifFalse: is a message!

```
Weather isRaining
    ifTrue: [ self takeMyUmbrella ]
    ifFalse: [ self takeMySunglasses ]
```

- Conceptually `ifTrue:ifFalse:` is a message sent to an object: a boolean!
- `ifTrue:ifFalse:` is in fact radically optimized by the compiler.
- Implement another one such as `siAlors:sinon:` and try it at home.

Implementing ifTrue:ifFalse:

- Do you see the pattern?
- Remember that a closure blocks execution and that the message `value` launches the execution of a frozen code.
- Propose an implementation

Implementing ifTrue:ifFalse:

- Let us the receiver decides!

```
True>> ifTrue: aTrueBlock ifFalse: aFalseBlock
    ^ aTrueBlock value
```

```
False>> ifTrue: aTrueBlock ifFalse: aFalseBlock
    ^ aFalseBlock value
```

Implementation Note

- Note that the Virtual Machine shortcuts calls to Boolean such as condition for speed reason.
- But you can implement your own conditional methods and debug to see that sending a message is dispatching to the right object.

Summary

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

- Approximately similar to anonymous method
- Technically lexical closures
- Can be passed as arguments to methods, stored in instance variables
- Basis of conditionals
- Basis of iterators (See following lecture)
- Further readings: `http://deepintopharo.org`