Chapter 1

Block and Dynamic Behavior of Smalltalk-Runtime

with the participation of:
Jean-Baptiste Arnaud

Blocks (lexical closures) are a powerful and essential feature of Smalltalk. Without them it would be difficult to have a so small and compact syntax. The use of blocks in Smalltalk is the key to get conditional and loops not hardcoded in the language syntax but just simple messages – Simple messages having blocks as arguments. This is why we can say that blocks work extremely well with the message passing syntax of Smalltalk.

In addition blocks are effective to improve the readability, reusability and efficiency of code. However the dynamic runtime semantics of Smalltalk is often not well documented. Blocks in presence of return statements behave like an escaping mechanism and while this can lead to ugly code when use to its extreme, it is important to understand it.

In this chapter we will discuss some basic block behavior such as the notion of a static environment defined at block compiled time. Then we will present some deeper issues. But let us first recall some basics.

We presented blocks in Pharo by Example for normal users. Here we just will focus on deeper aspects and their run time behavior. Note that their escaping behavior is shown here to describe the big picture

1.1 Basics

What is a block? Historically, it's a Lambda expression, or an anonymous function. A block is a piece of code whose execution is frozen and kicked in using a specific protocol. Blocks are defined by square brackets.

If you execute and print the result of the following block you will not get 3 but a block.

```
[ 1 + 2 ]
```

A block is evaluated by sending the value message to it. More precisely blocks can be executed using value (when no argument is mandatory), value: (when one argument), value:value:, value:value:value: and valueWithArguments: anArray...). These messages are the basic and historical API for block execution. They were presented in Pharo by Example.

```
[ 1 + 2 ] value
      ⟶  3

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

Some handy extensions

Pharo includes some handy messages such as cull: and friends to support the execution of blocks even in presence of more values than necessary. This allows us to write blocks more concisely when we are not necessarily interested in all the available arguments. cull: fills the same need as valueWith[Possible/Enough]Args:, but does not require creating an Array with the arguments, and will raise an error if the receiver has more arguments than provided rather than pass nil in the extraneous ones. Hence, from where the block is provided, they look almost the same, but where the block is executed, the code is usually cleaner.

Here are some examples of cull: and valueWithPossibleArgs: usages.

```
[ 1 + 2 ] cull: 5
      ⟶  3
[ :x | 1 + 2 + x ] cull: 5
      ⟶  8
[ 1 + 2 ] cull: 5 cull: 6
      ⟶  3
[ :x | 1 + 2 + x ] cull: 5
      ⟶  8
[ :x | 1 + 2 + x ] cull: 5 cull: 3
      ⟶  8
```

```
[ :x :y | 1 + y + x ] cull: 5 cull: 2
      ⟶  8
[ :x :y | 1 + y + x ] cull: 5
   raises an error mentioning that the block requires two arguments.
[ :x :y | 1 + y + x ] valueWithPossibleArgs: #(5)
   leads to an error because nil is passed as arguments.
```

The message once is another extension that caches the results and interned it until the receiver is uncached. A typical usage is the following one:

Method 1.1: *Example for resources caching using once*

```
myResourceMethod
  ↑ [expression] once
```

The table below lists some of the messages available on the class BlockClosure whose blocks are instance of.

silentlyValue	Execute the receiver but avoiding progress bar notifications to show up.
once	Answer and remember the receiver value, answering exactly the same object in any further sends of once or value. The expression will be evaluated once and its result returned for any subsequent evaluations.

Some messages are useful to profile execution (more information on Chapter **??**:

bench	Returns how many times the receiver can get executed in 5 seconds.
durationToRun	Answer the duration taken to execute the receiver block.
timeToRun	Answer the number of milliseconds taken to execute this block.

Some messages are related to error handling as explain in the Exception Chapter **??**.

ensure: aBlock	Execute a termination block after evaluating the receiver, regardless of whether the receiver's evaluation completes.
ifCurtailed: aBlock	Evaluate the receiver with an abnormal termination action. Evaluate aBlock only if execution is unwound during execution of the receiver. If execution of the receiver finishes normally do not evaluate aBlock.
on: exception do: handlerAction	Evaluate the receiver in the scope of an exception handler.
on: exception fork: handlerAction Jannik ►*need a better display. Maybe a paragraph for each and an example*◄	Activate the receiver. In case of exception, fork a new process, which will handle an error. An original process will continue running as if receiver evaluation finished and answered nil, *i.e.*, an expression like: [self error: 'some error'] on: Error fork: [:ex | 123] will always answer nil for original process, not 123. The context stack, starting from context which sent this message to receiver and up to the top of the stack will be transferred to forked process, with handlerAction on top. When the forked process is resuming, it will enter the handlerAction).

Some messages are related to process scheduling. We list the most important ones. Since this Chapter is not about concurrent programming in Pharo we will not go deep into them.

fork	ECreate and schedule a Process running the code in the receiver.
forkAt: aPriority	Create and schedule a Process running the code in the receiver at the given priority. Answer the newly created process.
newProcess	Answer a Process running the code in the receiver. The process is not scheduled.

1.2 Variables and Blocks

In Smalltalk, private variables (such as self, instance variables, temporaries and arguments) are lexically scoped. These variables are bound in the context in which the block that contains them is defined, rather than the context in which the block is executed. We call the context (set of bindings) in which a block is defined, the *block home context*.

Let's have fun and experiment a bit to understand. Define a class named BExp (for BlockExperience) and the following methods:

```
BExp>>testScope
   | t |
   t := 42.
   self testBlock: [self crLog: t printString]

BExp>>testBlock: aBlock
   | t |
   t := nil.
   aBlock value
```

Execute BExp new testScope. Executing the testScope message will print 42 in the Transcript. What you see is that the value of the temporary variable t defined in method testScope is the one used and that t inside [self crLog: t printString] is not looked up in the context of the executing method testBlock: but in the context of the testScope the method defining the block.

```
BExp>>testScope2
   | t |
   t := 42.
   self testBlock: [t := 33.
                    self crLog: t printString]

BExp>>testBlock: aBlock
   | t |
   t := nil.
   aBlock value
```

This experience shows that a block is not only an anonymous method but one with an execution context or environment. In this environment temporary variables are bound with the values they hold when the block is defined. Naturally we can expect that method arguments are also bound and also self and instance variables of the class in which the method defining a block is. Let's illustrate these points now.

Jannik ►*I have a display bug here*◄

For method arguments.

```
BExp>>testScopeArg: arg
   "self new testScopeArg: 'foo'"

   self testScopeArgValue: [self crLog: arg ; cr]

BExp>>testScopeArgValue: aBlock
   | arg |
   arg := 'zork'.
   aBlock value
```

Now executing self new testScopeArg: 'foo' prints foo even if in the method testScopeArgValue: the temporary arg is redefined.

self binding. For binding of self, we can simply define a new class and a couple of methods. Add the instance variable x to the class BExp and define the initialize method as follows:

```
Object subclass: #BExp
   instanceVariableNames: 'x'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'BlockExperiment'
```

```
BExp>>initialize
   x := 666.
```

Define another class named BExp2 (subclass of BExp but inheritance is orthogonal to what we want to show).

```
BExp2>>initialize
   super initialize.
   x := 69.

BExp2>>testScopeSelf: aBlock
   aBlock value
```

Then define the methods that will invoke methods defined in BExp2.

```
BExp>>testScopeSelf
   "self new testScopeSelf"
   self testScopeSelf: [self crLog: self printString ; logCr: x]

BExp>>testScopeSelf: aBlock
   BEXp2 new testScopeSelf: aBlock
```

Now when we execute BExp new testScopeSelf and we see that a BExp666 gets printed, showing that a block captures self too.

An example of sharing. Variables referred to by a block continue to be accessible and shared with other expressions. Let us take some examples.

```
BExp>>foo
   | a |
   [ a := 0 ] value.
   ↑ a
```

Here what you see is that the value is shared between the method body and the block. Inside the method body we can access the variable whose value was set by the block execution. Both the method and block bodies access to the temporary variable a.

Now imagine that we define the method foo as follows:

```
BExp>>foo
    | a |
    a := 0.
    ↑ {[ a := 2] . [a]}
```

The method foo defines a temporary variable a. It sets the value to a to zero and returns an array whose first element is a block setting the value to 2 and second element just returns the value of the temporary variable.

```
| res |
res := BExp new foo.
res second value.
    returns 0.
res first
res second
    returns 2.
```

You can also define the code as follows and open a transcript to see the results.

```
res := BExp new foo.
res second value crLog.
res first value.
res second value crLog.
```

Notice that when the expression res second value and res first value are executed, the method foo has already finished its execution - as such it is not on the execution stack anymore. Still the temporary variable a can be accessed and set to new value. It means that the variables referred to by a block may live longer than the methods that created the block that refers to them. We said that the variables outlive their defining context.

There the block implementation will have to keep the variable in a structure that is not linked to the execution stack but lives in the heap. We will go in more details in a following section.

1.3 Returning from inside a block

It is not a really good idea to have return statement in a block that you pass or or that store into instance variables and we will explain why in this section.

Basics on Return

A return statement allows one to return a different value than the receiver of the message. Now a return expression behaves also like an escape mecha-

nism since the execution flow will jump out to the current caller and not just one level up. For example, the following code will return 3 and 42 will never be reached. The expression [↑3] could be deeply nested, its execution jumps out all the levels.

```
BExp>>foo

   #(1 2 3 4) do: [:each | self crLog: each printString.
                   each = 3
                      ifTrue: [↑ 3]].
   ↑ 42
```

Now to see that a return is really escaping the current execution. We define

```
Foo>>start
   | res |
   self logCr: 'start start'.
   res := self defineBlock.
   self logCr: 'start end'.
   ↑ res
```

```
Foo>>defineBlock
   | res |
   self logCr: 'defineBlock start'.
   res := self arg: [ self logCr: 'block start'.
                      1 isZero ifFalse: [ ^ 33 ].
                      self logCr: 'block end'. ].
   self logCr: 'defineBlock end'.
   ↑ res
```

```
Foo>>arg: aBlock
   | res |
   self logCr: 'arg start'.
   res := self arg2: aBlock.
   self logCr: 'arg end'.
   ↑ res
```

```
Foo>>arg2: aBlock
   | res |
   self logCr: 'arg2 start'.
   res := self arg2: aBlock value.
   self logCr: 'arg2 end'.
   ↑ res
```

In Pharo ↑ should be the last statement of a block body. You should get a compile error if you type and compile the following expression.

Different blocks

Jannik ►*need to be improved and code explained*◄

We can classify blocks based on their usage or not of return statement.

Simple block. [:x :y| x*x. x+y] returns the value of the last statement to the method that sends it the message value. Here the first expression is useless.

Continuation blocks. [:x :y| x*x. ↑ x + y] returns the value to the method that activated its homeContext. As a block is always evaluated in its homeContext, it is possible to attempt to return from a method which has already returned using other return. This runtime error condition is trapped by the VM.

```
Object>>returnBlock
   "self new returnBlock value -> error"

   ↑[↑self]

Object new returnBlock
-> Exception
```

```
|b|
b:= [:x| Transcript show: x. x].
b value: a. b value: b.
b:= [:x| Transcript show: x. ↑x].
b value: a. b value: b.
```

Continuation blocks cannot be executed several times!

```
Test>>testScope
      |t|
       t := 15.
      self testBlock: [Transcript show: "--",t printString, "--".
      ↑35 ].
   ↑ 15

Test>>testBlock:aBlock
      |t|
      t := 50.
      aBlock value.
      self halt.
```

```
Test new testBlock
print: *15* and not halt.
return: 35
```

```
|val|
val := [:exit |
      |goSoon|
      goSoon := Dialog confirm: 'Exit now?'.
      goSoon ifTrue: [exit value: 'Bye'].
      Transcript show: 'Not exiting'.
      'last value'] myValueWithExit.
Transcript show: val.
val
yes -> print Bye and return  Bye
no -> print Not Exiting 2 and return 2
```

```
BlockClosure>>myValueWithExit
        self value: [:arg| ↑arg ].
    ↑ '2'
BlockClosure>>myValueWithExit
↑ self value: [:arg | ↑ arg]
```

1.4 Lexical Closure

Jannik ►*english form must be verified*◄

Lexical closure is a concept introduced by SCHEME in 70s. Scheme uses lambda expression which is basically an anonymous function (such the block). But using anonymous function implies to connect it to the current execution context. Jannik ►*please a verb*◄That why the lexical closure is important because it define when variables of block are bound to the execution context Jannik ►*redo this sentence*◄. The variable is depending of the scope where it's Jannik ►*no reduction in the text*◄ define. Let's illustrate that :

```
blockLocalTemp
   | collection |
      collection := OrderedCollection new.
      1 to: 3 do: [ :index || temp |
         temp := index.
         collection add: [ temp ] ].
      ↑collection collect: [:each | each value].
```

Let's Jannik ►*too much let's*◄ comment the code, we create a loop the store the arg value, in a temporary variable created in the loop (then local) and change it in the loop. We store a block containing the simply temp read access in a collection. And after the loop, we evaluate each block and return the collection of value. If we evaluate this method that will return #(1 2 3). What's happen? At each loop we create a variable existing locally and bind

it to a block. Then at the end evaluate block, we evaluate each block with this contextual *temp*. Jannik ►*should be redone*◄

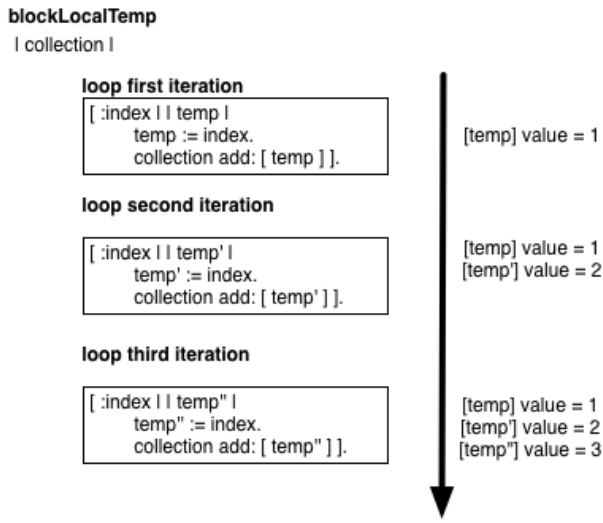

Figure 1.1: blockLocalTemp Execution

Now see another case :

```
blockOutsideTemp
      | collection temp |
      collection := OrderedCollection new.
      1 to: 3 do: [ :index |
         temp := index.
         collection add: [ temp ] ].
      ↑collection collect: [:each | each value].
```

Same case except the *temp*, variable will be declare in the upper scope. Then what will happen? Here the temp at each loop is the **same** shared variable bind. So when we collect the evaluation of the block at the end we will collect #(3 3 3).

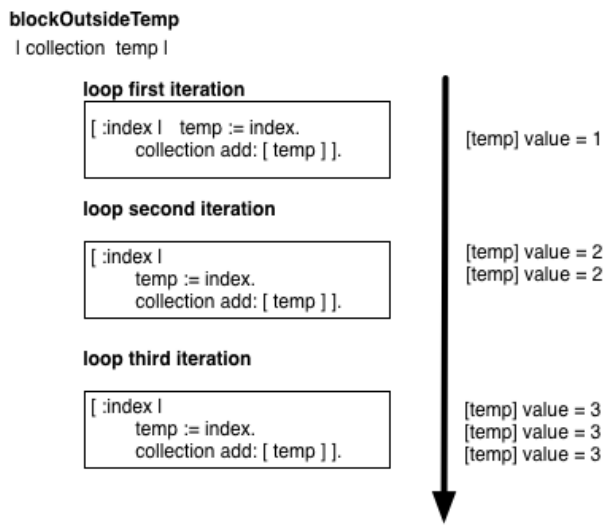

Figure 1.2: blockOutsideTemp Execution

When we look at the following Scheme expression and evaluate it you get 4. Indeed a binding is created which associates the variable index to the value 0. Then y a lambda expression is defined and it returns the variable index (its value). Then within this context another expression is evaluated which starts with a begin statement: first the value of the variable index is set to 4. Second the lambda expression is evaluated. It returns then the value of the

```
(let* ((index 0)
      (y (lambda () index)))
  (begin
```

```
  (set index 4)(y)))
```

```
(let ((index 0))
 (let ((y (lambda () index)))
  (begin
   (set index 4)(y))))
```

```
((lambda (index)
  ((lambda () (begin
          (set index 4)index))))0)
```

What you see is that the lambda expression is sharing the binding (index 0) with expression (begin...) therefore when this binding is modify from the body of the begin expression, the lambda expression sees its impact and this is why it returns 4 and not 0 because.

1.5 To sort

I have a method that takes OBCommands and returns Actions Jannik ►*what is OBCommands and Actions? In the code I see GLMAction*◄:

```
actionsFrom: aCollectionOfOBCommandClasses on: aTarget for: aRequestor
  | command |
  ↑ aCollectionOfOBCommandClasses collect: [ :each |
    command := each on: aTarget for: aRequestor.
    GLMAction new
      icon: command icon;
      title: command label;
      action: [:presentation | command execute ];
      yourself
    ]
```

These actions have a block that will be executed at a later time. The problem here was that the command in the action block was always pointing to the same command object, even at each point the command variable was populated correctly. Jannik ►*I do not see the problem, should be more explicit*◄

However, when the command is defined inside the block, everything works as expected.

```
actionsFrom: aCollectionOfOBCommandClasses on: aTarget for: aRequestor
  ↑ aCollectionOfOBCommandClasses collect: [ :each |
    | command |
    command := each on: aTarget for: aRequestor.
    GLMAction new
```

```
        icon: command icon;
        title: command label;
        action: [:presentation | command execute ];
        yourself
    ]
```

The semantics change in various ways. The trivial example that everyone knows is this **Jannik** ►*everyone knows ??? are you sure ? You are saying to reader that if he does not know these 5 lines, he is an idiot !*◄**:**

```
factorial := [ :n |
  n > 1
    ifTrue: [ n * (factorial value: n - 1) ]
    ifFalse: [ 1 ] ].
factorial value: 10.
```

Without closures you get an error, with closures you get the expected result. **Jannik** ►*you affirms that but as a reader, I do not know why.*◄

Another significant change is the existence of local variables in blocks. Without closures blocks don't have local variables, with closures they do: **Jannik** ►*do not understand this sentence*◄

```
b := [ :p |
  | t |
  t ifNil: [ t := p ] ].
{ b value: 1. b value: 2 }
```

In a non-closure image you get #(1 1) as result, because t is not a block local variable even if it looks like one. In a closure image you get #(1 2) because t is a block local variable. **Jannik** ►*same as before, I see the result, but I do not understand why*◄

Jannik ►*what is this source code ?*◄

```
testValueWithExitBreak

  | val |
  [ :break |
    1 to: 10 do: [ :i |
      val := i.
      i = 4 ifTrue: [break value].
    ]
  ] valueWithExit.
  self assert: val = 4.



testValueWithExitContinue
```

```
  | val last |
  val := 0.
  1 to: 10 do: [ :i |
    [ :continue |
      i = 4 ifTrue: [continue value].
      val := val + 1.
      last := i
    ] valueWithExit.
  ].
  self assert: val = 9.
  self assert: last = 10.

BlockClosure>>valueWithExit
    self value: [ ↑nil ]
```

1.6 Blocks and Contexts

Jannik ►*have to be written*◄

VM represents the state of execution as Context objects for method MethodContext for block BlockContext

aContext contains a reference to the context from which it is invoked, the receiver arguments, temporaries in the Context

We call home context the context in which a block is defined

Arguments, temporaries, instance variables are lexically scoped in Smalltalk These variables are bound in the context in which the block is defined and not in the context in which the block is evaluated

1.7 Block Scope Optimization

Jannik ►*have to be written*◄

1.8 Chapter conclusion