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 such a small and compact syntax. The use of blocks in Smalltalk is the key to get conditionals and loops not hardcoded in the language syntax but just 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 are often not well documented. Blocks in the presence of return statements behave like an escaping mechanism and while this can lead to ugly code when used 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 compile-time. Then we will present some deeper issues. But let us first recall some basics.

We already presented blocks in Pharo by Example. This presentation was simple and showing simply how to define blocks and to use them. Here we just will focus on deeper aspects and their run time behavior.

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 specific messages. 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 ]
      ⟶   [ 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 until the receiver is uncached. A typical usage is the following one:

Method 1.1: *Example for resources caching using once*

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

Blocks are instances of class BlockClosure. The table below lists some of the messages available on this class.

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	Return how many times the receiver can get executed in 5 seconds.
durationToRun	Answer the duration (instance of 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 explained 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: aBlock	Evaluate the receiver. If an exception exception is raised, executes the block aBlock.
on: exception fork: aBlock	Execute the receiver. In case of exception, fork a new process, which will handle the error. The 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 to the original process, not 123. The context stack, starting from context which sent this message to the receiver and up to the top of the stack will be transferred to the forked process, with handlerAction on top. When the forked process will resume, it will enter the block aBlock).

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	Create 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 Method Execution

Before going into the details of block handling, we will have a look at the way methods are executed. Imagine that we have a simple method

```
BEXp>>first: arg
   | temp |
   temp := arg * 2.
   ↑ temp
```

We can easily imagine that when such method is invoked multiple times with different arguments, we need a way to keep the value of the argument arg and the temporary variable temp. In addition, the instruction or program counter (the index saying what is the next instruction) can hold different values depending on the execution state and location. Therefore, there is a need to represent such information. Literature calls it a context. A Smalltalk interpreter needs five information to represent its current execution state:

1. The CompiledMethod whose bytecodes are being executed.
2. The location of the next bytecode to be executed in that Compiled-Method. This is the interpreter's instruction pointer.
3. The receiver and arguments of the message that invoked the Compiled-Method.
4. Any temporary variables needed by the CompiledMethod.
5. A stack.

In Pharo, the class MethodContext represents such execution information. Instances of MethodContext hold information about a specific execution point and we can obtain them using the pseudo-variable thisContext.

Let us look at an example. Modify the method as follow and execute it using BEXp new first: 33. You will get the inspector shown in Figure 1.1.

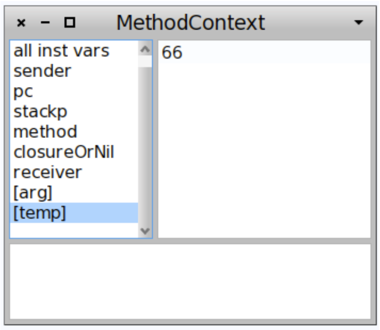
```
| temp |
temp := arg * 2.
thisContext copy inspect.
↑ temp
```

Note that we copy the current context obtained using thisContext because the Virtual Machine reuses contexts to avoid their creation when not necessary and it nilles out some values such as the temp value.

Let us have a look at some value of the current context:

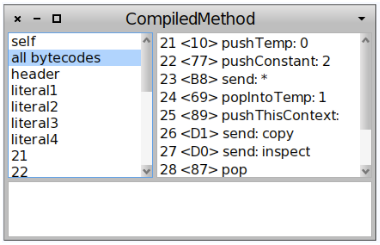
- sender points to the previous context that led to the creation of the current context. Here when you executed the expression, a context was created and this context is the sender of the current one.
- pc holds the value of the last executed instruction. Here its value is 27. To see which instruction it is, double click on the method instance variable and select the all bytecodes field, you should obtain the situation depicted in Figure 1.2.

```
| temp |
temp := arg * 2.
```



```
all inst vars      66
sender
pc
stackp
method
closureOrNil
receiver
[arg]
[temp]
```

Figure 1.1: A method context where we can access the value of the temporary variable temp at that given point of execution.



```
self             21 <10> pushTemp: 0
all bytecodes    22 <77> pushConstant: 2
header           23 <B8> send: *
literal1         24 <69> popIntoTemp: 1
literal2         25 <89> pushThisContext:
literal3         26 <D1> send: copy
literal4         27 <D0> send: inspect
21               28 <87> pop
22
```

Figure 1.2: The last instruction executed was the message send inspect.

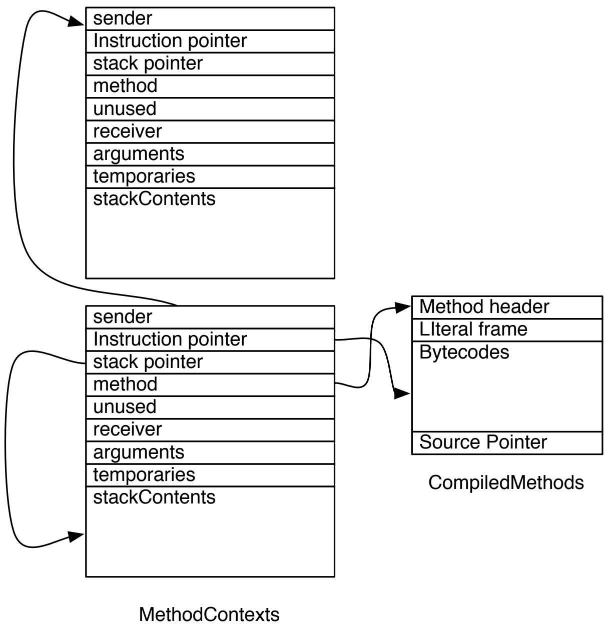
```
    thisContext copy inspect.
    temp := arg * 3.
    thisContext inspect.
    self halt.
    ↑ temp
```

Why without the self halt in the inspector temp is nil? Context recycling? Why can it see the difference between the first and the second thisContext?

Therefore, there is a need to represent the specific execution state of a compiled method: the argument, temporary variable, next instruction to execute.

```
BEXp>>first: arg

    | temp |
    temp := arg *2.
    thisContext inspect.
    self second: temp.
```



```
BEXp>>second: arg2

   self halt.
   ↑ arg2
```

In the inspector, you can access the temporary variable of method first: using the expression self tempNamedAt: 'arg'

Contexts

Contexts are objects representing a given execution state also called activation record, like a C stack represents execution of a C program. Contexts maintain the program counter and stack pointer, holding pointers to the sending context, the method for which this is appropriate, etc. MethodContext represents an executing method, it points back to the context from which it was activated, holds onto its receiver and compiled method.

To send a message to a receiver, the VM has to:

1. find the class of the receiver using the receiver object's header.

2. lookup the method in the class methodDictionary.

3. if the method is not found, repeat this lookup in superclasses. When no class in the superclass chain can understand the message, send the message doesNotUnderstand: to the receiver so that the error can be handled in a manner appropriate to that object.
4. extract the appropriate CompiledMethod from the MethodDictionary where the message was found and then
   (a) check for a primitive associated with the method by reading the method header
   (b) if there is a primitive, execute it.
   (c) if it completes successfully, return the result object directly to the message sender.
   (d) otherwise, continue as if there was no primitive called.
5. Create a new activation record. Set up the program counter, stack pointer, home contexts, then copy the arguments and receiver from the message sending context's stack to the new stack.
6. Activate that new context and start executing the instructions in the new method.

Handling Temporary Variables

Stéf ►*to rewrite*◄ Temporary Variables. Temporary variables are created for a particular execution of a CompiledMethod and cease to exist when the execu- tion is complete. The CompiledMethod indicates to the interpreter how many temporary variables will be required. The arguments of the invoking message and the values of the temporary variables are stored together in the temporary frame. The arguments are stored first and the temporary variable values immediately after. They are accessed by the same type of bytecode (whose comments refer to a temporary frame lo- cation). Since merge: takes a single argument, its two temporary vari- ables use the second and third locations in the temporary frame. The compiler enforces the fact that the values of the argument names can- not be changed by never issuing a store bytecode referring to the part of the temporary frame inhabited by the arguments.

When a message is sent, all five parts of the in-terpreter's state may have to be changed in order to execute a different CompiledMethod in response to this new message. The interpreter's old statemust be remembered because the bytecodes after the send must be executed after the value of the message is returned. The interpreter saves its state in objects called contexts. There will be many contexts in the system at any one time. The context that represents the current state of the interpreter is called the active context. When a send bytecode in the active context's CompiledMethod requires a new CompiledMethod to be executed, the active context becomes sus- pended and a new context is created and made active. The suspended context retains the state associated with the original CompiledMethod until that context becomes active again. A context must remember the context that it suspended so that the suspended context can be resumed when a result is returned. The suspended context is called the new con- text's sender.

1.3 Variables and Blocks

In Smalltalk, private variables (such as self, instance variables, temporaries and arguments) are lexically scoped. These variables are bound (get a value associated to them) 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 (an activation record representing a smalltalk execution stack element) 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:

Experience one.

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

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

Execute BExp new testScope. Executing the testScope message will print 42 in the Transcript. The value of the temporary variable t defined in method testScope is the one used rather than the one of t defined inside testBlock:. The variable t is not looked up in the context of the executing method testBlock: but in the context of the method testScope which defined the block.

Stéf ►*would be nice to have a picture*◄

Experience two.

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

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

Executing BExp new testScope2 prints 33. 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.

If we redefine testScope2 as follows:

```
testScope2
   |t|
   t := 42.
   self testBlock: [t := 33. self traceCr: t printString].
   self testBlock: [t := 66. self traceCr: t printString].
   self testBlock: [ self traceCr: t printString]
```

We will get 33, 66 and 66 printed.

For method arguments. Naturally we can expect that method arguments as well as self and instance variables are bound in the context of defining method. Let's illustrate these points now.

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

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

   self testScopeArgValue: [self traceCr: arg ; cr]
```

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

self binding. For binding of self, we 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 := 123.
```

Define another class named BExp2 (it can be a subclass of BExp since 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: aBlock
   "Here we ask another class to execute the block"
   BEXp2 new testScopeSelf: aBlock

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

Now when we execute BExp new testScopeSelf. You get a BExp123 printed, showing that a block captures self too, since an instance of BExp2 executed the block but the printed object was the original BExp instance.

An example of sharing. Variables referred to by a block continue to be accessible and shared with other expressions. Let us take some examples. Define the following method foo which defines a temporary variable a.

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

When we execute BExp new foo, we get 0 and not nil. 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 of a to zero and returns an array whose first element is a block setting the value to 2 and second element is a block just returning the value of the temporary variable.

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

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

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

Let us step back and look at an important point. In the previous code snippet when the expressions 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 which created the block that refers to them. We say that the variables outlive their defining context.

The block implementation needs to keep referenced variables 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.

1.4 Returning from inside a block

It is really not a good idea to have return statement in a block such as [↑ 33] that you pass or store into instance variables and we will explain why in this section. A block with explicit return statement is called a non-local return block.

Basics on Return

A return statement allows one to return a different value than the receiver of the message.

```
BExp>>testExplicitReturn
```

```
  "self new testExplicitReturn"

  self traceCr: 'one'.
  0 isZero ifTrue: [ self traceCr: 'two'. ↑ self].
  self traceCr: 'not printed'
```

Executing BExp new testExplicitReturn will print one and two. But not not printed.

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

```
[ self traceCr: 'two'.
 ↑ self.
 self traceCr: 'not printed' ]

     ⇝    End of block expected −>
```

Escape with Non Local Return

A return expression behaves also like an escape mechanism 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 and returns *from* the method that defines it. This is why it is important to avoid to use this style and use Exception when such behavior is needed. Some old code in Pharo predates Exception introduction and returning blocks are passed around leading to complex flow and difficult to maintain code.

```
BExp>>foo

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

Now to see that a return is really escaping the current execution. Let us build a slightly more complex call flow. We define four methods among which one creates and escaping block defineBlock and one execute this block (arg2:).

```
Bexp>>executeBlock: aBlock
  | res |
  self traceCr: 'executeBlock start'.
  res := self executeBlock: aBlock value.
```

```
   self traceCr: 'executeBlock loop so should never print that one'.
   ↑ res
```

```
Bexp>>arg: aBlock
   | res |
   self traceCr: 'arg start'.
   res := self executeBlock: aBlock.
   self traceCr: 'arg end'.
   ↑ res
```

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

```
BExp>>start
   | res |
   self traceCr: 'start start'.
   res := self defineBlock.
   self traceCr: 'start end'.
   ↑ res
```

Executing BExp new start prints:

```
start start
defineBlock start
arg start
executeBlock start
block start
start end
```

What we see is that the calling method start is fully executed. The method defineBlock is not completely executed. Indeed its escaping block [↑33] is executed two calls away in executeBlock:. The execution of the block returns to the method that created the home context of the block. Here when the block is executed in method executeBlock:, the execution discards the pending computation and returns from the method that defines the block, here defineBlock. This is why the rest of the computation in the block itself, the defineMethod method as well as arg: are discarded.

As shown by Figure 1.3, [↑3] will return to the method that activated its home context unclear what does it mean activated. [↑33] home context is the context that represents the execution of the method defineBlock, therefore it will return its result to the method start.

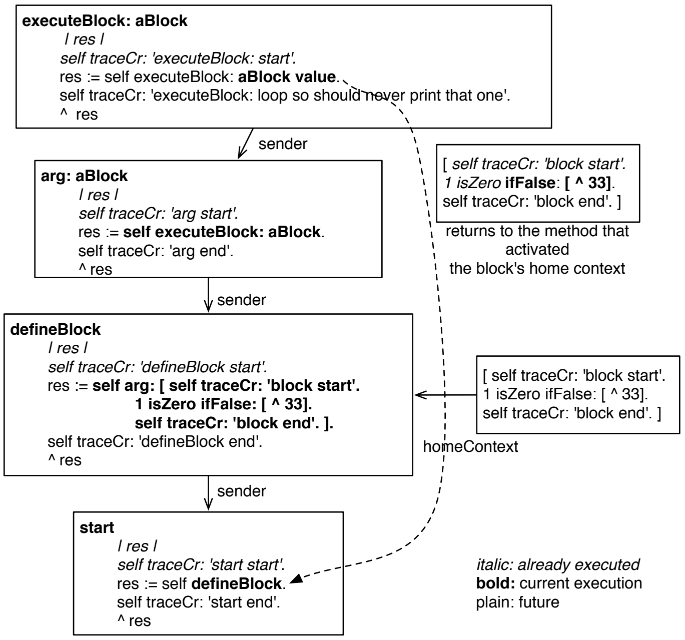

executeBlock: aBlock
| res |
self traceCr: 'executeBlock: start'.
res := self executeBlock: aBlock value.
self traceCr: 'executeBlock: loop so should never print that one'.
^ res
sender
arg: aBlock
| res |
self traceCr: 'arg start'.
res := self executeBlock: aBlock.
self traceCr: 'arg end'.
^ res
sender
defineBlock
| res |
self traceCr: 'defineBlock start'.
res := self arg: [self traceCr: 'block start'.
1 isZero ifFalse: [^ 33].
self traceCr: 'block end'.].
self traceCr: 'defineBlock end'.
^ res
sender
start
| res |
self traceCr: 'start start'.
res := self defineBlock.
self traceCr: 'start end'.
^ res
[self traceCr: 'block start'.
1 isZero ifFalse: [^ 33].
self traceCr: 'block end'.]
returns to the method that activated the block's home context
[self traceCr: 'block start'.
1 isZero ifFalse: [^ 33].
self traceCr: 'block end'.]
homeContext
italic: already executed
bold: current execution
plain: future

Figure 1.3: A block with non local return execution returns to the method that activated the block home context.

Couple more examples. The following examples shows that escaping blocks jumped to their home context but do not unwind the stack after this point. For example the previous example shows that the method start was fully executed. valueWithExit is defined on the class BlockClosure as follows.

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

```
BExp>>assert: aBoolean
   aBoolean ifFalse: [Error signal]

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

The method testValueWithExitBreak shows that the block break is created and activated once and as effect it cancels the rest of the computation.

The following method testValueWithExitContinue shows that just the computation left from the block activation is skipped (here val := val + 1. last := i, when i = 4) and this is why the value of val is 9 and not 10. In this example, a new block is created

```
BExp>>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.
```

Pay attention here valueWithExit is not equivalent to value: [↑nil] because it changes the home context of the block. In the first case the homeContext of the block is not the method testValueWithExitContinue while in the second it is. Put a self halt before the assert: to convince you. In one case, you will reach the halt while in the other not.

Stéf ►*up until here*◄

Different blocks

Stéf ►*should say something about that.*◄

Non-local return 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
```

Stéf ►*restarted here*◄ Block temporary variables are local to the blocks. For example

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

You get #(1 2) because t is a block local variable. Its value is initialized during each block execution.

Stéf ►*until here*◄

1.5 Blocks and Contexts

The Virtual Machine represents the state of execution as context objects, one per method or block currently executed (or activated). In Pharo method and block activations are represented by MethodContext. Contexts are first-class activation record.

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

We call the home context, the activation record or 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.

1.6 Block Scope Optimization

1.7 Chapter conclusion

We want to thank Ben Coman for his english corrections.

1.8 To be treated

```
|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.
```

Non local returning 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]
```

1.9 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 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*◄

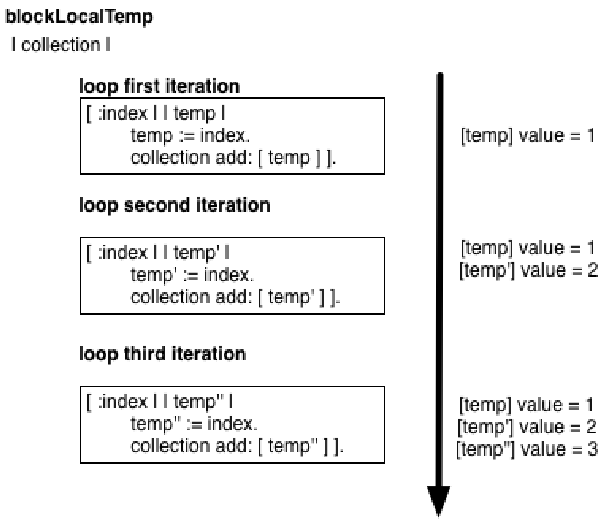


Figure 1.4: blockLocalTemp Execution

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).

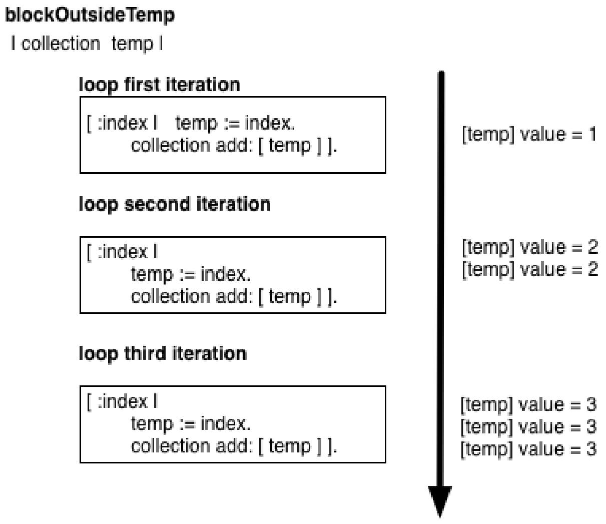


Figure 1.5: blockOutsideTemp Execution

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.