Chapter 1

Blocks: a Detailed Analysis

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 is 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. 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.

In this chapter we will discuss some basic block behavior such as the notion of a static environment defined at block compile-time. To understand blocks, we will describe how the execution of a program happens. Then we will present some deeper issues. But let us first recall some basics.

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.
```

Other messages. Blocks are instances of class BlockClosure. The list below presents 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 is evaluated once and its result returned for any subsequent evaluations. A typical usage is the following one:

Method 1.1: *Example for resources caching using once*

```
myResourceMethod
  ^ [expression] once
```

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 Variables and Blocks

A block can have its own temporaries variables, such variables are initialized during each block execution and are local to the block. Now the question we want to make clear is what is happening when a block refers to other (non local) variables. Later, for the curious, we will present how local variables are implemented and stored.

In Smalltalk, private variables (such as self, instance variables, method 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. Traditionally, the context in which a block is defined is named the *block home context*.

In essence a context represents information about the current execution step (what is the context from which the current one is executed, what is the next byte code to be executed, what is the value of the temporary variables...) a context is an activation record representing a smalltalk execution stack element. This is important and we will come back later to this concept.

Some little experiences

Let's experiment a bit to understand how variables are bound in a block. Define a class named Bexp (for BlockExperience) and the following methods:

```
Object subclass: #Bexp
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'BlockExperiment'
```

Stéf ►*I should rename textVariable: by setVariableAndExecuteBlock:* ◄

Experience one: Variable lookup. A variable is looked up in the block definition context.

```
Bexp>>testVariable: aBlock
   | t |
   t := nil.
   aBlock value

Bexp>>testVariable
   | t |
   t := 42.
   self testVariable: [ t logCr ]
```

Execute Bexp new testVariable. Executing the testVariable message prints 42 in the Transcript. The value of the temporary variable t defined in method testVariable is the one used rather than the one of t defined inside testVariable:. Even if the block is executed during the execution of the method testVariable:.

The variable t is not looked up in the context of the executing method testVariable: but in the context of the method testVariable which defined the block.

Let's look at it in detail. Figure 1.1 shows the execution of the expression Bexp new testVariable. During the execution of method testVariable, a variable t is defined and it is assigned 42. Then a block is created and this block refers to the method activation context - which holds temporaries variables (Step 1). The method testVariable: defines its own local variable t with the same name than the one in the block. However, this is not this variable that is used when the block is executed. While executing the method testVariable: the block is executed, during the execution of the expression t logCr the non local variable t is looked up in the home context of the block *i.e.*, the method context that created the block and not the context of the currently executed method.

Experience two: Changing a variable value. A block can change the value of a non local variable. Now the method testVariable2 shows that a non local variable value can be changed during the execution of a block. Executing Bexp new testVariable2 prints 33, since 33 is the last value of the variable t.

```
Bexp>>testVariable2
   | t |
   t := 42.
   self testVariable: [ t := 33. t logCr ]
```

Experience Three: Accessing a shared non local variable. Note that two blocks can share a non local variable. Let us define a new method testVariable3 as follows:

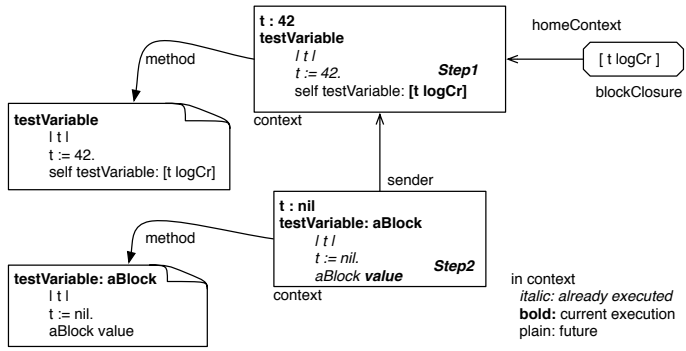


Figure 1.1: Non local variable are looked in the method activation context where the block was created and not executed.

```
Bexp>>testVariable3
   | t |
   t := 42.
   self testVariable: [ t logCr. t := 33. t logCr ].
   self testVariable: [ t logCr. t := 66. t logCr ].
   self testVariable: [ t logCr ]
```

Bexp new testVariable3 will print 42, 33, 33, 66 and 66. Here the two blocks [t := 33. t logCr] and [t := 66. t logCr] access the variable t and can modify it. During the first execution of the method testVariable: its current value 42 is printed, then the value is changed and printed. Similar situation occurs in the following block. This example shows that blocks shares the same context where variables are stored.

Experience Four: Variable lookup is done at execution time. The following example shows that the value of the variable is looked up at runtime. First add the instance variable block to the class Bexp.

```
Object subclass: #Bexp
   instanceVariableNames: 'block'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'BlockExperiment'
```

Here the initial value of the variable t is 42. The block is created and stored into the instance variable block but the value to t is changed to 69 before the block is executed. And this is the last value (69) that is effectively printed because it is looked up at execution-time. Executing Bexp new testVariable4 prints 69.

```
Bexp>>testVariable4
   | t |
   t := 42.
   block := [ t logCr: t ].
   t := 69.
   self testVariable: block
```

Bexp new testVariable4 prints 69.

Experience Five: For method arguments. Naturally we can expect that method arguments are bound in the context of defining method. Let's illustrate this point now. Define the following methods and add the instance variable block to the class.

```
Bexp>>testArg
   self testArg: 'foo'.

Bexp>>testArg: arg
   block := [arg crLog].
   self testArg2: 'zork'.

Bexp>>testArg2: arg
   block value.
```

Now executing Bexp new testArg: 'foo' prints 'foo' even if in the method testArg: the temporary arg is redefined.

Experience Six: 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
   super 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). Stéf ►*would be better to have executeBlock instead of testScopeSelf:*◄

```
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
  Bexp2 new testScopeSelf: aBlock "Here we ask another class to execute the block"

Bexp>>testScopeSelf
  self testScopeSelf: [self crLog ; traceCr: x]
```

Now when we execute Bexp new testScopeSelf, you get a Bexp123 printed in the Transcript, showing that a block captures self too, since an instance of Bexp2 executed the block but the printed object (self) is the original Bexp instance that was accessible at the block creation time.

1.3 Variables can outlive their defining method

Non block local variables referred to by a block continue to be accessible and shared with other expressions even if the method execution terminated. We say that the variables outlive the method execution that defined them. Let us take some examples.

Method-Block Sharing. 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 define the method twoBlockArrays as follows:

```
Bexp>>twoBlockArrays
  | a |
  a := 0.
  ^ {[ a := 2] . [a]}
```

The method twoBlockArrays 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 of a to 2 and second element is a block just returning the value of the temporary variable a.

Now we store the array returned by twoBlockArrays and access and execute the blocks stored in the array. This is what the following code snippet is doing.

```
| res |
res := Bexp new twoBlockArrays.
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 twoBlockArrays.
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 twoBlockArrays 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 method execution.

At the implementation, you can see from this example, that while temporary variable are somehow stored in an activation context, it is a bit more subtle than that. The block implementation needs to keep referenced variables in a structure that is not the execution stack but lives on 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 returning block. Let us start illustrating some basic points first.

Basics on return

By default the returned value of a method is the receiver of the message *i.e.*, self. A return expression (the expression starting with the character ^) allows one to return a different value than the receiver of the message. In addition, when a return statement is defined in a method, the execution of a return statement exits the currently executed method and returns to its caller. This cancels the expressions following the return statement.

Define the following method. Executing Bexp new testExplicitReturn prints 'one' and 'two' but it will not print not printed, since the method textExplicitReturn will have returned before.

```
Bexp>>testExplicitReturn
   self traceCr: 'one'.
   0 isZero ifTrue: [ self traceCr: 'two'. ^ self].
   self traceCr: 'not printed'
```

Escaping behavior of non-local return

A return expression behaves also like an escape mechanism since the execution flow will jump out to the current invoking method and not just one level up. For example, the following expression Bexp new jumpingOut will return 3 and not 42. ^ 42 will never be reached. The expression [^3] could be deeply nested, its execution jumps out all the levels and return to the method caller. This is why it is important to avoid to use this style and use exceptions 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. In subsequent section we will carefully look at where a return is actually returning.

```
Bexp>>jumpingOut
   #(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 an escaping block defineBlock and one executes this block (arg:). Pay attention that to stress the escaping behavior of a return we defined executingBlock: so that it endlessly loops after executing its argument.

```
Bexp>>start
   | res |
   self traceCr: 'start start'.
```

```
    res := self defineBlock.
    self traceCr: 'start 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>>arg: aBlock
    | res |
    self traceCr: 'arg start'.
    res := self executeBlock: aBlock.
    self traceCr: 'arg end'.
    ^ res
```

```
Bexp>>executeBlock: aBlock
    | res |
    self traceCr: 'executeBlock start'.
    res := self executeBlock: aBlock value.
    self traceCr: 'executeBlock loops so should never print that one'.
    ^  res
```

Executing `Bexp new start` prints the following (we added indentation to stress the calling flow).

```
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 the method `executeBlock:`. The execution of the block returns to the sender of the block home context. Indeed the block should return and it will return to the method context of its home context. Here the home context of the block is the execution context of the method `defineBlock` . The sender of this execution was created by the execution of the method `start`, therefore the return execution returns to the execution point of the start execution context.

When the return statement of the block is executed in method executeBlock: , the execution discards the pending computation and returns to the method execution point that created the home context of the block. The block is defined in method defineBlock, the home context of the block is the activation context that represents the execution of the method defineMethod. This is why the rest of the computation in the block itself, the defineMethod method as well as arg: are discarded.

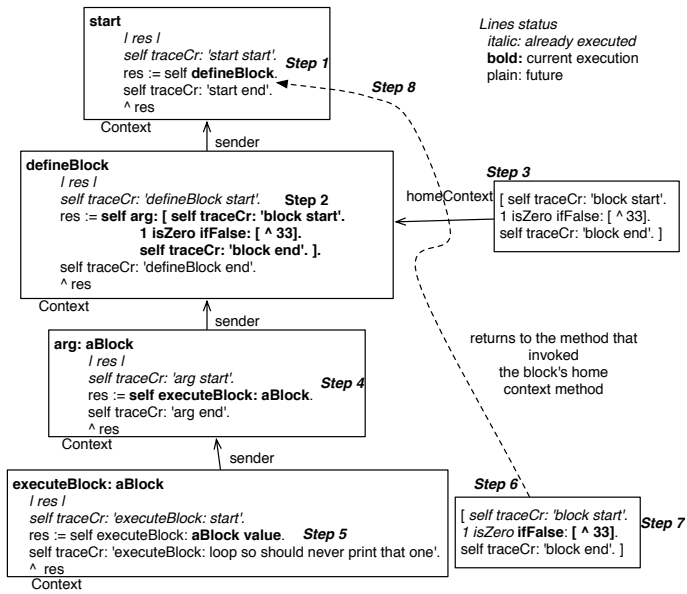


Figure 1.2: A block with non local return execution returns to the method execution that activated the block home context.

As shown by Figure 1.2, [^33] will return to the sender of its home context. [^33] home context is the context that represents the execution of the method defineBlock, therefore it will return its result to the method start execution.

Step 1 represents the execution up to the invocation of the method defineBlock: 'start start' is printed.

Step 2 the execution up to the block creation, which is done in Step 3: 'defineBlock' is printed. The home context of the block is the defineContext method execution context.

Step 4 represents the execution up to the invocation of method executeBlock:. 'defineBlock start' is printed.

Step 5 represents the execution up to the block execution. 'executeBlock: start' is printed.

Step 6 represents the execution of the block up to the condition: 'block start' is printed.

Step 7 represents the execution up to the return statement.

Step 8 represents the execution of the return statement. It returns to the sender of the block home context, *i.e.*, just after the invocation of the method defineBlock in the method start. The execution continues and 'start end' is printed.

Accessing information. To verify manually and find the home context of a closure we can do the following: add the expression `thisContext home inspect` in the block of the method `defineBlock`. We can also add the expression `thisContext closure home inspect` which accesses the closure via the current execution context and gets its home context. Note that in both cases, even if the block is executed during the execution of the method `executeBlock:`, the home context of the block is the method `defineBlock`. Note that such expressions will be executed during the block execution.

```
Bexp>>defineBlock
   | res |
   self traceCr: 'defineBlock start'.
   res := self arg: [ thisContext home inspect. self traceCr: 'block start'.
                      1 isZero ifFalse: [ ^ 33 ].
                      self traceCr: 'block end'. ].
   self traceCr: 'defineBlock end'.
   ^ res
```

To verify where the execution will end you can use the expression `thisContext home sender copy inspect.` which returns a method context pointing to the assignment in the method `start`.

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. We define `valuePassingEscapingBlock` on the class BlockClosure as follows.

```
BlockClosure>>valuePassingEscapingBlock
    self value: [ ^nil ]
```

Then we define a simple assert: method that raises an error if its argument is false.

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

We define the following method: it defines a block whose argument is executed as soon as we reach the step 4 of a loop. Then a value is printed and we checked that the value of variable val is correctly 4.

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

Executing `Bexp new testValueWithExitBreak` performs without raising error and print 4 to the Transcript: the loops has been stopped, the value printed and the assert validated.

If you change `valuePassingEscapingBlock` by `value: [^ nil]` in the method above. You will not get the trace because the execution of the method `testValueWithExitBreak` will exit when the block is executed. Here `valuePassingEscapingBlock` is not equivalent to `value: [^nil]` because it changes the home context of the escaping block `[ ^ nil ]`. With `valuePassingEscapingBlock` the home context of the block `[^ nil]` is not the method `testValueWithExitContinue` but `valuePassingEscapingBlock`. Therefore when executed the escaping block will change the execution flow to the `valuePassingEscapingBlock` message in the method `testValueWithExitBreak` (similarly to the previous example where the flow came back just after the invocation of the `defineBlock` message). Put a self halt before the assert: to convince you. In one case, you will reach the halt while in the other not.

Non-local return blocks. As a block is always evaluated in its home context, it is possible to attempt to return from a method execution which has already returned using other return. This runtime error condition is trapped by the VM.

```
Bexp>>returnBlock
   ^ [ ^ self ]

Bexp new returnBlock value
-> Exception
```

When we execute returnBlock, the method return the block to its caller (here the top level execution). Then when executing the block, it should return to its home context sender but the method execution already terminated, therefore this is an error.

1.5 Opening the Trunk

The Virtual Machine represents execution state as context objects, one per method or block currently executed (or activated). In Pharo, method and block activations are represented by MethodContext instances. We will explain contexts as well as method execution to finish by block closure execution.

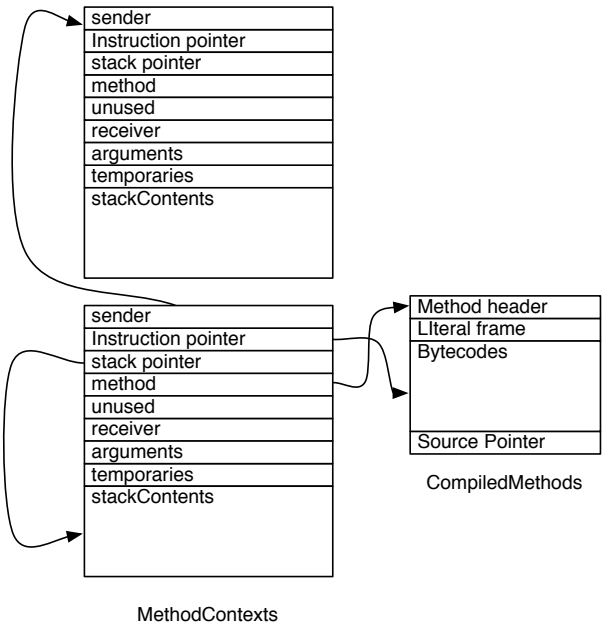
Sending a Message

Let us look at what happens when we send a message.

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. 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.
3. Extract the appropriate compiled method from the method dictionary 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 and pass to step 4.
4. 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.
5. Activate that new context and start executing the instructions in the new method.

When a message is sent, computation state may have to be changed to execute a different compiled method in response to this new message. The execution old state must be remembered because the instructions after the message send must be executed after the value of the message is returned. The execution 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 execution is called the active context.

When a message send happens in the active context, the active context becomes suspended and a new context is created and made active. The suspended context retains the state associated with the original compiled method 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 context's sender.

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. A MethodContext represents an executing method, it points back to the context from which it was activated, holds onto its receiver and compiled method.

Stéf ►*until here*◄

1.6 Context: Representing Method Execution

Imagine that we have a simple method as the following one.

```
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 the following 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 CompiledMethod. This is the interpreter's instruction pointer.
3. The receiver and arguments of the message that invoked the CompiledMethod.
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 1.3.

```
  | 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.

MethodContext does not only represent activation context of method execution but also the ones for block closures as we will see later. 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.

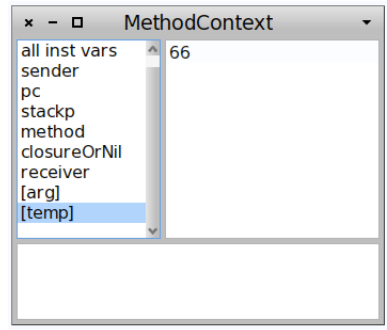


Figure 1.3: A method context where we can access the value of the temporary variable temp at that given point of execution.

- 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 1.5, which shows that the next instruction to be executed is pop Stéf ►*is it true?*◄
- stackp defines the number of stored temporary variables and
- method points to the currently executed method.
- closureOrNil holds a reference to the closure currently executed or nil.
- receiver is the message receiver.

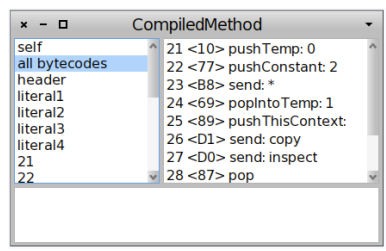


Figure 1.4: The last instruction executed was the message send inspect.

Using the following definition you can get two inspectors open on a copy of the context at the execution point.

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

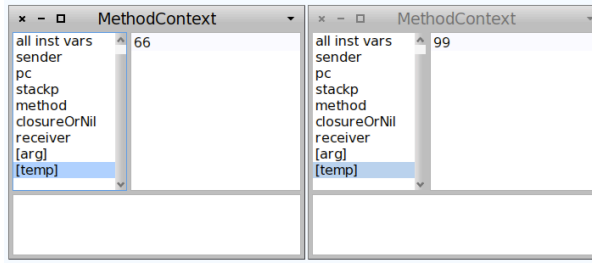


Figure 1.5: Two contexts at different execution points.

```
thisContext copy inspect.
temp := arg * 3.
thisContext copy inspect.
^ temp
```

Studying Contexts

Methods arguments and temporaries are stored in contexts. Contexts have a variable part that store them.

When inspecting the following method, stackp holds 3 to represent that there is one argument and two temporary variables.

```
Bexp>>twoTempsOneArg: arg
   "self new twoTempsOneArg: 33"
   | temp1 temp2 |
   temp1 := temp2 := arg.
   thisContext inspect.
   ^ temp1
```

```
ContextPart>>arguments
   "returns the arguments of a message invocation"

   | arguments numargs |
   numargs :=  self method numArgs.
   arguments := Array new: numargs.
   1 to: numargs do: [:i | arguments at: i put: (self tempAt: i) ].
   ^ arguments
```

Invoking Another Method

```
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'

1.7 Closures

```
#(1 2 3) inject: 0 into: [:sum :each | sum + each]
```

```
inject: thisValue into: binaryBlock
   "Accumulate a running value associated with evaluating the argument,
   binaryBlock, with the current value of the argument, thisValue, and the
   receiver as block arguments. For instance, to sum the numeric elements of a
   collection, aCollection inject: 0 into: [:subTotal :next | subTotal + next]."
   | nextValue |
   nextValue := thisValue.
   self do: [:each | nextValue := binaryBlock value: nextValue value: each].   ^ nextValue
```

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. This 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*.

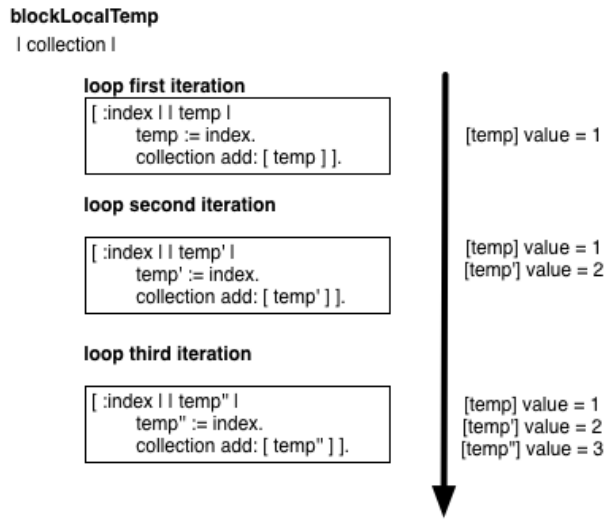


Figure 1.6: blockLocalTemp Execution

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).

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

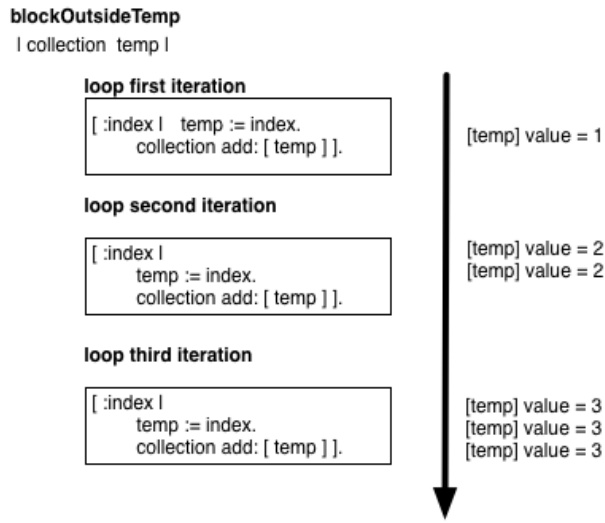


Figure 1.7: blockOutsideTemp Execution

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.

1.8 Notes from eliot blog

- way to control execution -> block closures - way to store arg and temp -> method context - a way of accessing locals in enclosing block or method activations but to implement access to locals in enclosing activations without access through those activations.

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 execution is complete. The CompiledMethod indicates to the interpreter how many temporary variables will be required. The arguments of the in- voking message and the values of the temporary variables are stored together in the temporary frame (method context???). The arguments are stored first and the temporary variable values immediately after.