Pharo Chosen Pieces

S. Ducasse

March 13, 2024

Layout and typography based on the sbabook LaTeX class by Damien Pollet.

Contents

Illustrations

1

Blocks: a Detailed Analysis

This chapter is based on the chapter of Deep into Pharo written by S. Ducasse with the participation of C. Béra.

Lexically-scoped block closures, blocks in short, are a powerful and essential feature of Pharo. Without them it would be difficult to have such a small and compact syntax. The use of blocks is key to get conditionals and loops as library messages and not hardcoded in the language syntax. This is why we can say that blocks work extremely well with the message passing syntax of Pharo.

In addition blocks are effective to improve the readability, reusability and efficiency of code. The precise dynamic runtime semantics of blocks, however, is not well documented. For example, 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 you will learn about the central notion of context (objects that represent point in program execution) and the capture of variables at block creation time. You will learn how block returns can change program flow. Finally to understand blocks, we describe how programs execute and in particular we present contexts, also called activation records, which represent a given execution state. We will show how contexts are used during the block execution. This chapter complements the one on exceptions (see Chapter **??**). In the Pharo by Example book, we presented how to write and use blocks. On the contrary, this chapter focuses on deep aspects and their runtime behavior.

Listing 1-1 Block definition

```
[ 1 + 2 ]
>>> [ 1 + 2 ]
```

Listing 1-2 Block Evaluation

```
[ 1 + 2 ] value
>>> 3

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

1.1 Basics

What is a block? A block is a lambda expression that captures (or closes over) its environment at creation-time. We will see later what it means exactly. For now, imagine a block as an anonymous function or method. A block is a piece of code whose execution is frozen and can be kicked in using messages. Blocks are defined by square brackets.

If you execute and print the result of the following code, you will not get 3, but a block. Indeed, you did not ask for the block value, but just for the block itself, and you got it.

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

1.2 Some handy extensions

Beyond the `value` messages, Pharo includes some handy messages such as `cull:` and friends to support the execution of blocks even in the presence of more values than necessary. `cull:` will raise an error if the receiver requires more arguments than provided.

The `valueWithPossibleArgs:` message is similar to `cull:` but takes an array of parameters to pass to a block as argument. If the block requires more arguments than provided, `valueWithPossibleArgs:` will fill them with `nil`.

```
[ 1 + 2 ] cull: 5
>>> 3
[ 1 + 2 ] cull: 5 cull: 6
>>> 3
[ :x | 2 + x ] cull: 5
>>> 7
```

```
[ :x | 2 + x ] cull: 5 cull: 3
>>> 7
[ :x :y | 1 + x + y ] cull: 5 cull: 2
>>> 8
[ :x :y | 1 + x + y ] cull: 5
>>> error because the block needs 2 arguments.
[ :x :y | 1 + x + y ] valueWithPossibleArgs: #(5)
>>> error because 'y' is nil and '+' does not accept nil as a
    parameter.
```

1.3   Other messages

Some messages are useful to profile execution:

- bench. Return how many times the receiver block can be executed in 5 seconds.

- durationToRun. Answer the duration (instance of class 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 Chapter **??**.

- ensure: terminationBlock. Execute the termination block after evaluating the receiver, regardless of whether the receiver's evaluation completes.

- ifCurtailed: onErrorBlock. Execute the receiver, and, if the evaluation does not complete, execute the error block. If evaluation of the receiver finishes normally, the error block is not executed.

- on: exception do: catchBlock. Execute the receiver. If an exception exception is raised, execute the catch block.

- on: exception fork: catchBlock. Execute the receiver. If an exception exception is raised, fork a new process, which will handle the error. The original process will continue running as if the 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. The context stack, starting from the context which sent this message to the receiver and up to the top of the stack will be transferred to the forked process, with the catch block on top. Eventually, the catch block will be executed in the forked process.

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 evaluating the receiver.

- `forkAt: aPriority`. Create and schedule a Process evaluating the receiver at the given priority. Answer the newly created process.

- `newProcess`. Answer a Process evaluating the receiver. The process is not scheduled.

1.4 Variables and blocks

A block can have its own temporary variables. Such variables are initialized during each block execution and are local to the block. We will see later how such variables are kept. Now the question we want to make clear is what is happening when a block refers to other (non-local) variables. A block will close over the external variables it uses. It means that even if the block is executed later in an environment that does not lexically contain the variables used by a block, the block will still have access to the variables during its execution. Later, we will present how local variables are implemented and stored using contexts.

In Pharo, private variables (such as self, instance variables, method temporaries and arguments) are lexically scoped: an expression in a method can access to the variables visible from that method, but the same expression put in another method or class cannot access the same variables because they are not in the scope of the expression (i.e., visible from the expression).

At runtime, the variables that a block can access, 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. It means that a block, when executed somewhere else can access variables that were in its scope (visible to the block) when the block was *created*. Traditionally, the context in which a block is defined is named the *block home context.*

The block home context represents a particular point of execution (since this is a program execution that created the block in the first place), therefore this notion of block home context is represented by an object that represents program execution: a *context* object in Pharo. In essence, a context (called stack frame or activation record in other languages) represents information about the current execution step such as the context from which the current one is executed, the next instruction (byte code) to be executed, and temporary variable values. A context is a Pharo execution stack element. This is important and we will come back later to this concept. A block is created inside a context (an object that represents a point in the execution) and it can access variables that were accessible at creation time.

1.5 **Some little experiments**

Let's experiment a bit to understand how variables are bound in a block. Define a class named BExp (for BlockExperiment) as a subclass of TestCase so that we can define quickly some tests to make sure that our results are always correct.

```
TestCase subclass: #BExp
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'BlockExperiment'
```

1.6 **Experiment 1: Variable lookup.**

A variable is looked up in the block *definition* context. We define two methods: one that defines a variable t and sets it to 42 and a block [t] and one that defines a new variable with the same name and executes a block defined elsewhere.

```
BExp >> setVariableAndDefineBlock
  | t |
  t := 42.
  ^ self executeBlock: [ t ]
```

```
BExp >> executeBlock: aBlock
  | t |
  t := 33.
  ^ aBlock value
```

```
BExp new setVariableAndDefineBlock
>>> 42
```

The following test passes. Executing the BExp new setVariableAndDefineBlock expression returns 42 .

```
BExp >> testSetVariableAndDefineBlock
  self assert: self setVariableAndDefineBlock equals: 42
```

The value of the temporary variable t defined in the setVariableAndDefineBlock method is the one used rather than the one defined inside the method executeBlock: even if the block is executed during the execution of this method. The variable t is looked up in the context of the block creation (context created during the execution of the method setVariableAndDefineBlock and not in the context of the block evaluation (method executeBlock:).

Let's look at it in detail. Figure 1-3 shows the execution of the expression BExp new setVariableAndDefineBlock.

- During the execution of method setVariableAndDefineBlock, a variable t is defined and it is assigned 42. Then a block is created and this

5

block refers to the method activation context - which holds temporary variables.

- The method `executeBlock:` defines its own local variable `t` with the same name than the one in the block. This is not this variable, however, that is used when the block is executed. While executing the method `executeBlock:` the block is executed, during the execution of the expression `t` 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.
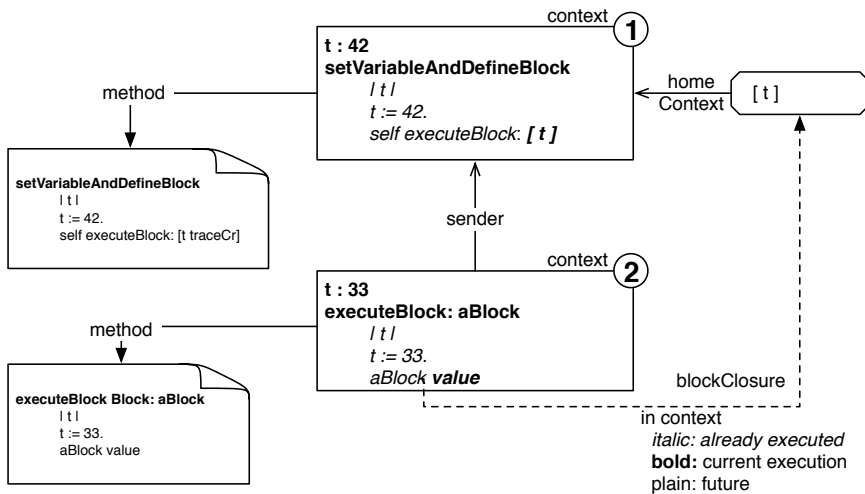
Figure 1-3 Non-local variables are looked up in the method activation context where the block was *created* and not where it is *executed*.

Non-local variables are looked up in the *home context* of the block (i.e., the method context that *created* the block) and not the context executing the block.

1.7   **Experiment 2: Changing a variable value**

Let's continue our experiments. The method `setVariableAndChanging-VariableBlock` shows that a non-local variable value can be changed during the evaluation of a block. Executing `BExp new setVariableAndChanging-VariableBlock` returns 2008, since 2008 is the last value of the variable `t`.

```
BExp >> setVariableAndChangingVariableBlock
  | t |
  t := 42.
  ^ self executeBlock: [ t := 2008. t ]
```

The test verifies this behavior.

```
BExp >> testSetVariableAndChangingVariableBlock
  self assert: self setVariableAndChangingVariableBlock equals: 2008
```

1.8   Experiment 3: Accessing a shared non-local variable.

Different blocks can share a non-local variable and they can modify the value
of this variable at different moments. To see this, let us define a new method
accessingSharedVariables as follows:

```
BExp >> accessingSharedVariables
  | t |
  ^ String streamContents: [ :st |
    t := 42.
    self executeBlock: [ st print: t. st cr. t := 99. st print: t.
    st cr ].
    self executeBlock: [ st print: t. st cr. t := 66. st print: t.
    st cr ].
    self executeBlock: [ st print: t. st cr. ]
  ]
```

The following test shows the results:

```
BExp >> testAccessingSharedVariables
  self assert: self accessingSharedVariables equals: '42
  99
  99
  66
  66
  '
```

BExp new accessingSharedVariables will print 42, 99, 99, 66 and 66. Here
the two blocks [st print: t. st cr. t := 99. st print: t. st cr
] and [st print: t. st cr. t := 66. st print: t. st cr] ac-
cess the same variable t and can modify it. During the first execution of
the method executeBlock: its current value 42 is printed, then the value
is changed and printed. A similar situation occurs with the second call. This
example shows that blocks share the location where variables are stored and
also that a block does not copy the value of a captured variable. It just refers
to the location of the variables and several blocks can refer to the same loca-
tion.

1.9   Experiment 4: Variable lookup is done at execution time.

The following example shows that the value of the variable is looked up at
runtime and not copied during the block creation. First add the instance

variable `block` to the class `BExp`.

```
Object subclass: #BExp
  instanceVariableNames: 'block'
  classVariableNames: ''
  package: '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 in the home context of the block.
Executing `BExp new variableLookupIsDoneAtExecution` return '69' as
confirmed by the tests.

```
BExp >> variableLookupIsDoneAtExecution

  ^ String streamContents: [ :st |
    | t |
    t := 42.
    block := [ st print: t ].
    t := 69.
    self executeBlock: block ]
```

```
BExp >> testVariableLookupIsDoneAtExecution
  self assert: self variableLookupIsDoneAtExecution equals: '69'
```

1.10   **Experiment 5: For method arguments**

We can naturally expect that method arguments referred by a block are
bound in the context of the defining method. Let's illustrate this point. De-
fine the following methods.

```
BExp >> testArg5: arg
  ^ String streamContents: [ :st |
    block := [ st << arg ].
    self executeBlockAndIgnoreArgument: 'zork']
```

```
BExp >> executeBlockAndIgnoreArgument: arg
  ^ block value
```

```
BExp >> testArg5
  self assert: (self testArg5: 'foo') equals: 'foo'
```

Executing `BExp new testArg: 'foo'` returns 'foo' even if in the method
`executeBlockAndIgnoreArgument:` the temporary arg is redefined. The
block execution looks for the value of arg is its definition context which
is the one where `testArg5: ' arg` is bound to 'foo' due to the execution of
method `testArg5`.

1.11   **Experiment 6: self binding**

Now we may wonder if `self` is also captured in a block. It should be but let us test it. To test we need another class. Let's 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: 'block x'
  classVariableNames: ''
  package: 'BlockExperiment'
```

```
BExp >> initialize
  x := 123.
```

Define another class named `BExp2`.

```
Object subclass: #BExp2
  instanceVariableNames: 'x'
  classVariableNames: ''
  package: 'BlockExperiment'
```

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

```
BExp2 >> executeBlockInAnotherInstance6: aBlock
  ^ aBlock value
```

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

```
BExp >> selfIsCapturedToo
  ^ String streamContents: [ :st |
      self executeBlockInAnotherInstance6: [ st print: self ; print:
    x ] ]
```

```
BExp >> executeBlockInAnotherInstance6: aBlock
  ^ NBexp2 new executeBlockInAnotherInstance6: aBlock
```

Finally we write a test showing that this is indeed the instance of `BExp` that is bound to `self` even if it is executed in `BExp2`.

```
BExp >> testSelfIsCapturedToo
  self assert: self selfIsCapturedToo equals:
    'NBexp>>#testSelfIsCapturedToo123'
```

Now when we execute `BExp new selfIsCapturedToo`, we get `NBexp>>#test-SelfIsCapturedToo` printed 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. The printing is `NBexp>>#testSelfIsCapturedToo` because `BExp` is subclass of `TestCase`.

Conclusion.

We show that blocks capture variables that are reached from the *context in which the block was defined* and not where there are executed. Blocks keep references to variable locations that can be shared between multiple blocks.

1.12 About temporary variables

Before looking at how block temporaries are managed we propose a little review of temporary variable aspects in Pharo.

Temporaries cannot shadow other temporaries. The following expression is not allowed by the compiler because the second `tmp` definition is illegal.

```
[
   | tmp |
   tmp := 2.
   [
      | tmp |
      tmp := 3 ]]
```

Similarly a temporary cannot shadow a block or a method parameter. Therefore the following expression is not allowed.

```
[ :tmp |
   | tmp |
   tmp := 2]
```

The following method is not valid either.

```
with: arg
   | arg |
   ^ arg
```

Finally parameters are read only. Therefore the following method is not valid.

```
with: arg
   arg := 42
```

As well as the following expression

```
[ :tmp |
   tmp := 2
   ]
```

We can say that inside a method the names of parameters, temporaries and instance variables are flattened and use a single namespace.

1.13 **Block-local variables**

As we saw previously a block is a lexical closure that is connected to the place where it is defined. In the following, we will illustrate this connection by showing that block local variables are allocated in the execution context linked to their creation. We will show the difference when a temporary variable is local to a block or to a method.

Block allocation

The following method `blockLocalTemp` stores in a collection a block referencing a temporary for each index of the iteration. It is interesting since it shows that each single block will have its own home context and allocation of the temporary variable `temp`.

```
BExp >> blockLocalTemp
  | collection |
  collection := OrderedCollection new.
  #(1 2 3) do: [ :index |
    | temp |
    temp := index.
    collection add: [ temp ] ].
  ^ collection collect: [ :each | each value ]
```

Let's comment this method:

- It creates a collection to store the results of the loop.

- It defines a loop that assigns the current index (a block argument) in the temporary variable `temp` defined in the loop block.

- Then its stores a block that accesses this variable in the collection collecting results.

- After the loop ends, it executex each of the stored blocks of the collection and return the collection of values.

Executing this method results in a collection with 1, 2 and 3.

```
BExp >> testBlockLocalTemp
  self assert: self blockLocalTemp asArray equals: #(1 2 3)
```

This result shows that each block in the collection refers to a different `temp` variable. This is due to the fact that a different execution context is created for each block creation (at each loop step) and that the block [`temp`] is stored in this context (see Figure 1-4).

Method allocation

Now let us create a new method that is the same as `blockLocalTemp` except that the variable `temp` is a method variable instead of a block variable.
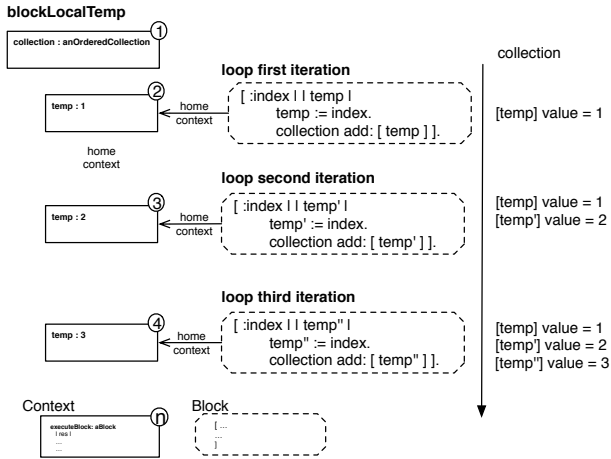
Figure 1-4 `blockLocalTemp` execution: each block created during a loop step has its own home context.

```
BExp >> blockOutsideTemp
  | collection temp |
  collection := OrderedCollection new.
  #(1 2 3) do: [ :index |
    temp := index.
    collection add: [ temp ] ].
  ^ collection collect: [ :each | each value ]

BExp >> testBlockOutsideTemp
  self assert: self blockOutsideTemp asArray equals: #(3 3 3)
```

When we execute the method `blockOutsideTemp`, we now get a collection with 3, 3 and 3. This result shows that each block in the collection now refers to a single variable `temp` allocated in the method `blockOutsideTemp` context. This leads to the fact that `temp` is shared by the blocks (see Figure 1-5).

1.14 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 variables *outlive* the method execution that defined them. Let's look at some examples.

We start with a simple example showing that a variable is shared between a method and a block (as in the previous experiments in fact). Define the following method `foo` which defines a temporary variable a.

Figure 1-5 `blockOutsideTemp` execution: one context is shared by all the blocks created in the loop step.

```
BExp >> sharing
  | a |
  [ a := 0 ] value.
  ^ a

BExp new sharing
>>> 0
```

When we execute BExp new sharing, we get 0 and not nil. What you see here 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 the same temporary variable a.

Let's make it slightly more complicated. Define the method twoBlockArray as follows:

```
BExp >> twoBlockArray
  | a |
  a := 0.
  ^ {[ a := 2] . [a]}
```

The method twoBlockArray 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 twoBlockArray and execute the blocks stored in the array. This is what the following test is doing.

```
BExp >> testTwoBlockArray
  | array |
  array := self twoBlockArray.
  self assert: array second value equals: 0.

  array first value.
  self assert: array second value equals: 2
```

Let us step back and look at an important point. In the previous test when the expressions `array second value` and `array first value` are executed, the method `twoBlockArray` 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 a new value.

This experiment shows that the variables referred to by a block may live longer than the method which created the block that refers to them. We say that the variables outlive the execution of their defining method.

You can see from this example that while temporary variables are somehow stored in an activation context, the implementation is a bit more subtle than that. The block implementation needs to keep referenced variables in a structure that is not in the execution stack but lives on the heap. The compiler performs some analysis and when it detects that a variable may outlive its creation context, it allocates the variables in a structure that is not allocated on the execution stack.

1.15 Returning from inside a block

In this section we explain why it is not a good idea to have return statements inside a block (such as `[^ 33]`) that you pass to other methods (except conditional) or store into instance variables. A block with an explicit return statement is called a *non-local returning block.*

Let us start illustrating some really basic points first. 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, the execution of a return statement exits the currently executed method and returns to its caller. This ignores the expressions following the return statement.

1.16 Experiment 7: Return's exiting behavior

Define the following method. Executing `BExp new withExplicitReturn` returnn 'one' and 'two' but it will not print `not printed`, since the method `withExplicitReturn` will have returned before.

```
BExp >> withExplicitReturn
  ^ String streamContents: [:str |
    str nextPutAll: 'one'; cr.
    0 isZero ifTrue: [ str nextPutAll: 'two'. ^ str contents ].
    str nextPutAll: 'not printed']
```

```
BExp >> testWithExplicitReturn
  self assert: self withExplicitReturn equals: 'one
  two'
```

Note that the return expression should be the last statement of a block body.

1.17 Escaping behavior of non-local return

A return expression behaves also like an escaping mechanism since the execution flow will directly jump out to the current invoking method. Let us define a new method `jumpingOut` as follows to illustrate this behavior.

```
BExp >> jumpingOut
  #(1 2 3 4) do: [:each |
        each = 3
          ifTrue: [^ 3]].
  ^ 42
```

```
BExp >> testJumpingOut
  self assert: self jumpingOut equals: 3
```

The expression `BExp new jumpingOut` will return 3 and not 42. The expression `^ 42` will never be reached. The expression `[ ^3 ]` could be deeply nested, its execution jumps out all the levels and returns to the method caller. Some old code (predating introduction of exceptions) passes non-local returning blocks around leading to complex flows and difficult to maintain code. We strongly suggest not using this style because it leads to complex code and bugs. In subsequent sections we will carefully look at where a return is actually returning. A return in a method returns a value to the sender of the method and stop executing the method containing the return. A non-local return does the same even if the block is executed by another method.

1.18 Understanding return

Now to see that a return is really escaping the current execution, let us build a slightly more complex call flow.

First we add and initialize an instance variable named: `stream` and some helper methods to access the stream.

```
BExp >> initialize
  x := 123.
  stream := '' writeStream
```

```
BExp >> traceCr: aString
  stream nextPutAll: aString; cr
```

```
BExp >> contents
  ^ stream contents
```

We define four methods among which one (defineBlock) creates an escaping block, one (arg:) executes this block and one (executeBlock:) that executes the block. Note that to stress the escaping behavior of a return we defined executeBlock: 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; contents prints the following (indentation added 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 an escaping block returns to the *sender of its home context sender* (i.e., the context that invoked the method that created the block). In our example, the home context of `[^33]` is the method `defineBlock` so its execution will return to `start`.

When the return statement of the block is executed in the method `execute-Block:`, 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 the method `defineBlock`. The home context of the block is the activation context that represents the definition of the method `defineBlock`. Therefore the return expression returns to the `start` method execution just after the `defineBlock` execution. This is why the pending executions of `arg:` and `executeBlock:` are discarded and why we see the execution of the method `start` end.

Non local return `[^ ...]` returns to the sender of the block home context, i.e., to the method execution point that called the one that created the block. As shown by Figure 1-6, `[^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`.

- Step a represents the execution up to the invocation of the method `defineBlock`. The trace `'start start'` is printed.

- Step c represents the execution up to the block creation, which is done in Step b. `'defineBlock start'` is printed. The home context of the block is the `defineBlock` method execution context.

- Step d represents the execution up to the invocation of the method `executeBlock:`. `'arg start'` is printed.

- Step e represents the execution up to the block evaluation. `'execute-Block start'` is printed.

- Step f represents the execution of the block up to the condition: `'block start'` is printed.

- Step g represents the execution up to the return statement.

- Step h 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'` gets printed.

start
    l res l
    self traceCr: 'start start'.
    res := self **defineBlock**.
    self traceCr: 'start end'.
    ^ res

(a) (1) (h)

sender

defineBlock
    l res l
    self traceCr: 'defineBlock start'.
    res := **self arg: [self traceCr: 'block start'.**
                1 isZero ifFalse: [^ 33].
                self traceCr: 'block end'.].
    self traceCr: 'defineBlock end'.
    ^ res

(b) (2) (c)

homeContext

[self traceCr: 'block
start'.
1 isZero ifFalse: [^ 33].
self traceCr: 'block end'.]

returns to the method that
invoked the block's home
context method

sender

arg: aBlock
    l res l
    self traceCr: 'arg start'.
    res := **self executeBlock: aBlock**.
    self traceCr: 'arg end'.
    ^ res

(d) (3)

sender

executeBlock: aBlock
    l res l
    self traceCr: 'executeBlock: start'.
    res := self executeBlock: **aBlock value**.
    self traceCr: 'executeBlock: should never get here'.
    ^ res

(e) (4) (f)

[*self traceCr: 'block*
start'.
1 isZero **ifFalse**: **[^ 33]**.
self traceCr: 'block end'.]

(g)

Code line legend:
    italic: already executed
    bold: current execution
    plain: future

Context

executeBlock: aBlock
  l res l
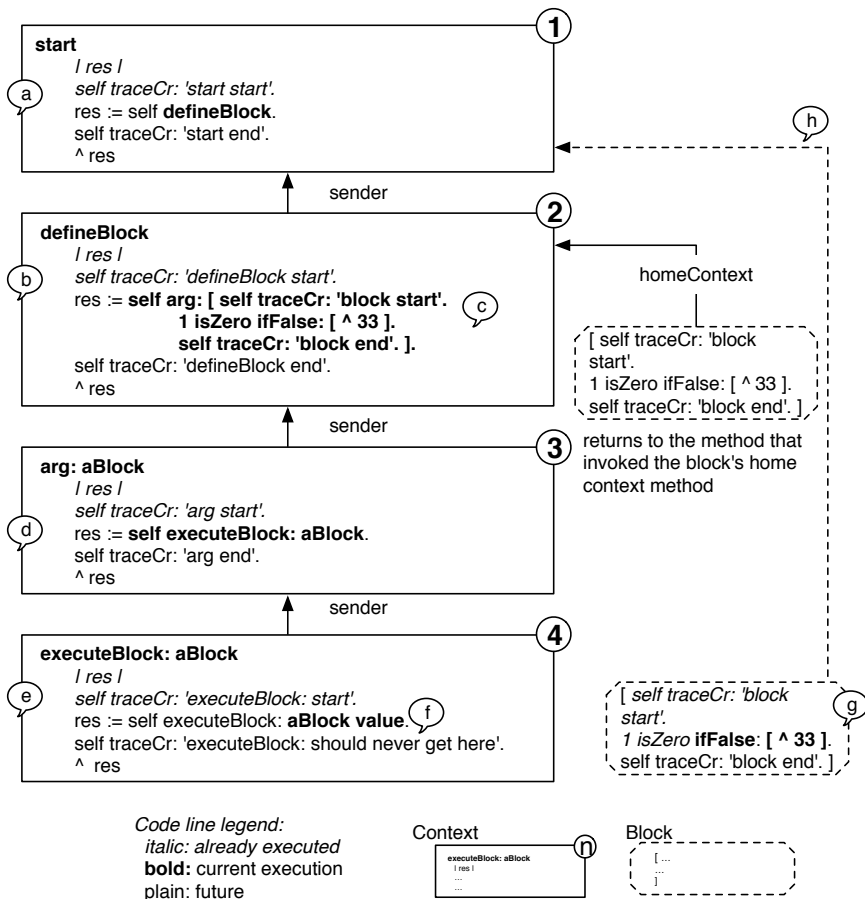 ...
  ...

(n)

Block

[...
...
]

Figure 1-6 A block with non-local return execution returns to the method execution that activated the block home context. Dashed bevelled rectangles represent the same block at different execution points.

1.19   **Accessing context information**

To manually verify and find the home context of a block 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 evaluation.

```
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`.

1.20   **Couple more examples**

The previous example shows that the method `start` was fully executed. The following examples show that escaping blocks jump to sender of their home contexts. We define `valuePassingEscapingBlock` on the class `BlockClosure` as follows.

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

We define the following method.

```
BExp >> testValueWithExitBreak
  | val |
  [ :break |
     1 to: 10 do: [ :i |
              val := i.
              val = 4 ifTrue: [ break value ] ] ]
    valuePassingEscapingBlock.
  Transcript show: 'Passed here!'.
  self assert: val equals: 4
```

This method defines a block whose argument `break` is executed as soon as the step 4 of a loop is reached. We make sure that its value is 4. The test

passes: the loop has been stopped, the 'Passed here!' is printed to the transcript and the assert is validated.

Now let us change the message `valuePassingEscapingBlock` by `value: [^ nil]` in the `testValueWithExitBreak` method above, you will not get the halt because the execution of the method `testValueWithExitBreak` will exit when the block is executed. In this case, calling `valuePassingEscapingBlock` is not equivalent to calling `value: [^nil]` because the home context of the escaping block [^ nil] is different.

```
BExp >> testValueWithExitBreak2
  | val |
  [ :break |
      1 to: 10 do: [ :i |
                val := i.
                val = 4 ifTrue: [ break value ] ] ] value: [^nil].
  self halt.
  self assert: val equals: 4
```

With the original `valuePassingEscapingBlock`, the home context of the block `[^ nil]` is `valuePassingEscapingBlock` and not the method `testValueWithExitContinue` itself. 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). The `self halt` before the `assert:` will not be executed.

1.21 **Non-local return blocks**

As a block is always executed in its home context, it is possible to attempt to return from a method execution which has already returned. This runtime error condition is trapped by the VM.

```
BExp >> returnBlock
  ^ [ ^ self ]

BExp new returnBlock value ~~> Exception
```

When we execute `returnBlock`, the method returns the block to its caller (here the top level execution). When evaluating the block, because the method defining it has already terminated and because the block is containing a return expression that should normally return to the sender of the block home context, an error is signaled.

Blocks with non-local expressions ([^ . . .]) return to the sender of the block home context (the context representing the execution led to the block creation).

1.22   **Contexts: representing method execution**

We saw that blocks refer to the home context when looking for variables. So now we will look at contexts. Contexts represent program execution. The Pharo execution engine represents its current execution state with the following information:

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 program pointer;

3. the receiver and arguments of the message that invoked the `CompiledMethod`;

4. any temporary variable needed by the `CompiledMethod`;

5. a call stack.

In Pharo, the class `MethodContext` represents this execution information. A `MethodContext` instance holds information about a specific execution point. The pseudo-variable `thisContext` gives access to the current execution point.

1.23   **Interacting with Contexts**

Let us look at an example. Define the following method and execute it using `BExp new first: 33`.

```
BExp>>first: arg
   | temp |
   temp := arg * 2.
   thisContext copy inspect.
   ^ temp
```

You will get the inspector shown in Figure 1-7. Note that we copy the current context obtained using `thisContext` because the Virtual Machine limits memory consumption by reusing contexts.

`MethodContext` does not only represent activation context of method execution but also the ones for blocks. Let us have a look at some values of the current context:

- `sender` points to the previous context that led to the creation of the current one. Here when you executed the expression, a context was created and this context is the sender of the current one.

- `method` points to the currently executing method.

- `pc` holds a reference to the latest executed instruction. Here its value is 27. To see which instruction is referred to, double click on the `method`
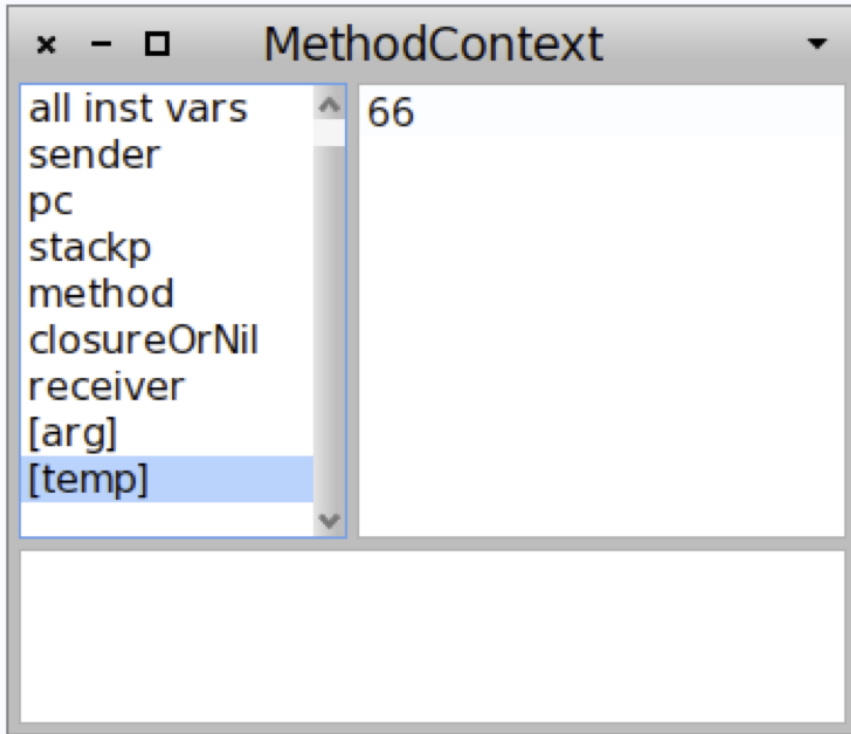
Figure 1-7 A method context where we can access the value of the temporary variable temp at that given point of execution.

instance variable and select the all bytecodes field, you should get the situation depicted in Figure 1-8, which shows that the next instruction to be executed is pop (instruction 28).

- stackp defines the depth of the stack of variables in the context. In most cases, its value is the number of stored temporary variables (including arguments). But in certain cases, for example during a message send, the depth of the stack is increased: the receiver is pushed, then the arguments, lastly the message send is executed and the depth of the stack goes back to its previous value.

- closureOrNil holds a reference to the currently executing closure or nil.

- receiver is the message receiver.

The class MethodContext and its superclasses define many methods to get information about a particular context. For example, you can get the values
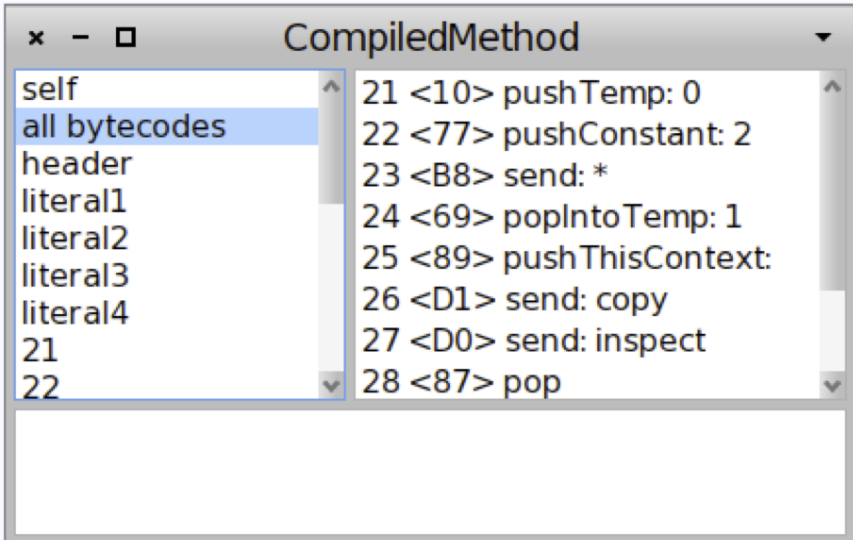
Figure 1-8 The pc variable holds 27 because the last (bytecode) instruction executed was the message send `inspect`.

of the arguments by sending the `arguments` message and the value of a particular temporary variable by sending `tempNamed:`.

1.24 Block nesting and contexts

Now let us look at the case of block nesting and its impact on home contexts. In fact, a block points to a context when it was created: it is its *outer context.* Now depending on the situation the outer context of a block can be its home context or not. This is not complex: Each block is created inside some context. This is the block's outer context. The outer context is the direct context in which a block was created. The home context is the one at the method level. If the block is not nested then the outer context is also the block home context.

If the block is nested inside another block execution, then the outer context refers to that block execution context, and the block execution's outerContext is the home context. There are as many outer context steps as there are nesting levels.

Let's look at the following example. When you execute, just press Ok to the dialogs popping up.

```
| homeContext b1 |
homeContext := thisContext.
b1 := [| b2 |
        self assert: thisContext closure == b1.
        self assert: b1 outerContext == homeContext.
      self assert: b1 home = homeContext.
        b2 := [self assert: thisContext closure == b2.
                self assert: b2 outerContext closure outerContext
    == homeContext].
          self assert: b2 home = homeContext.
        b2 value].
b1 value
```

- First we set in `homeContext`, the context before the block creation. `homeContext` is the home context of the blocks `b1` and `b2` because they are defined during this execution.

- `thisContext closure b1` shows that the context inside the execution of the block `b1` has a pointer to `b1`. The outer context of `b1` is `homeContext`. Nothing new because `b1` is defined during the execution starting after the assignment. The home context of `b1` is the same as its outer context.

- Inside `b2` execution, the current context points to `b2` itself since it is a closure. The outer context of the closure in which `b2` is defined i.e., `b1` points to `homeContext`. Finally the home context of `b2` is `homeContext`. This last point shows that all the nested blocks have a separate outer context, but they share the same home context.

1.25 Message execution

The Virtual Machine represents execution state as context objects, one per method or block currently executed (the word *activated* is also used). In Pharo, method and block executions are represented by `MethodContext` instances. In the rest of this chapter we survey contexts, method execution, and block closure execution.

1.26 Sending a message

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

- Find the class of the receiver using the receiver object's header.

- Lookup the method in the class method dictionary. If the method is not found, repeat this lookup in each superclass. When no class in the superclass chain can understand the message, the VM sends the message `doesNotUnderstand:` to the receiver so that the error can be handled in a manner appropriate to that object.

- When an appropriate method is found:
1. check for a primitive associated with the method by reading the method header;
2. if there is a primitive, execute it;
3. if the primitive completes successfully, return the result object to the message sender;
4. when there is no primitive or the primitive fails, continue to the next step.
   - Create a new context. 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.
   - Activate that new context and start executing the instructions in the new method.

The execution state before the message send must be remembered because the instructions after the message send must be executed when the message returns. State is saved using contexts. There will be many contexts in the system at any 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 is suspended and a new context is created and activated. 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. Figure 1-9 represents the relations between compiled methods and context. The method points to the currently executed method.

The program counter points to the last instruction of the compiled method. Sender points to the context that was previously active.

1.27   Sketch of implementation

Temporaries and arguments for blocks are handled the same way as in methods. Arguments are passed on the stack and temporaries are held in the corresponding context. Nevertheless, a block can access more variables than a method: a block can refer to arguments and temporaries from the enclosing method. As we have seen before, blocks can be passed around freely and activated at any time. In all cases, the block can access and modify the variables from the method it was defined in.

Let us consider the example shown in Figure 1-10. The `temp` variable used in the block of the `exampleReadInBlock` method is non-local or remote variable.`temp` is initialized and changed in the method body and later on read in
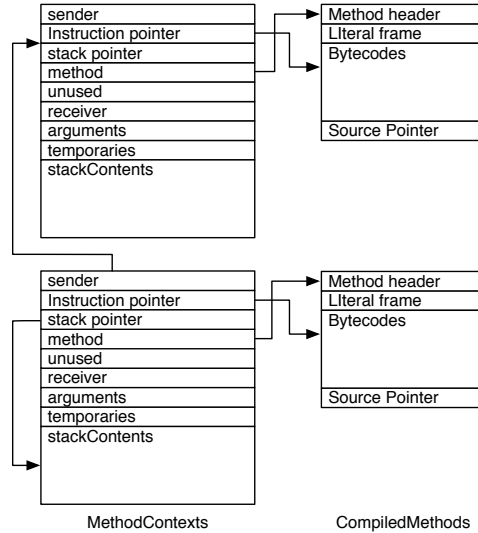
sender
Instruction pointer
stack pointer
method
unused
receiver
arguments
temporaries
stackContents

Method header
LIteral frame
Bytecodes
Source Pointer

sender
Instruction pointer
stack pointer
method
unused
receiver
arguments
temporaries
stackContents

Method header
LIteral frame
Bytecodes
Source Pointer

MethodContexts   CompiledMethods

Figure 1-9 Relations between contexts and compiled methods.

the block. The actual value of the variable is not stored in the block context but in the defining method context, also known as home context. In a typical implementation the home context of a block is accessed through its closure. This approach works well if all objects are first-class objects, including the method and block context. Blocks can be executed outside their home context and still refer to remote variables. Hence all home contexts might outlive the method activation.
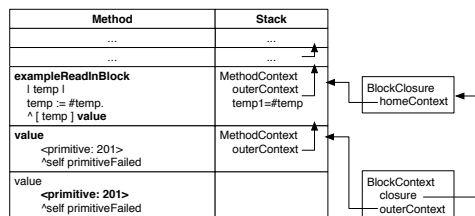
Method	Stack
...	...
...	...
exampleReadInBlock	MethodContext
I temp I	outerContext
temp := #temp.	temp1=#temp
^ [temp] **value**	
value	MethodContext
<primitive: 201>	outerContext
^self primitiveFailed	
value	
<primitive: 201>	
^self primitiveFailed	

BlockClosure / homeContext

BlockContext closure / outerContext

Figure 1-10 A first understanding of closures.

1.28 Implementation

The previously mentioned approach for block contexts has disadvantages from a low-level point of view. If method and block contexts are normal objects that means they have to be garbage collected at some point. Combined

with the typical coding practice of using small methods that call many other objects, Pharo can generate a lot of contexts.

The most efficient way to deal with method contexts is to not create them at all. At the VM level, this is done by using real stack frames. Method contexts can be easily mapped to stack frames: whenever we call a method we create a new frame, whenever we return from a method we delete the current frame. In that matter Pharo is not very different from C. This means whenever we return from a method the method context (stack frame) is immediately removed. Hence no high-level garbage collection is needed. Nevertheless, using the stack gets much more complicated when we have to support blocks.

As mentioned before, method contexts that are used as home contexts might outlive their activation. If method contexts work as we explained up to now we would have to check each time for home contexts if a stack frame is removed. This comes with a big performance penalty. Hence the next step in using a stack for contexts is to make sure method contexts can be safely removed when we return from a method.

The Figure 1-11 shows how non-local variables are no longer directly stored in the home context, but in a separate remote array which is heap allocated.
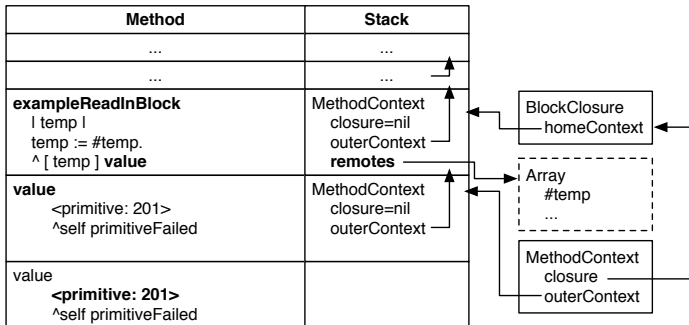

Figure 1-11   How the VM stores remote variables so that they continue to leave when a method returns.

1.29   **Full block implementation**

The full block term refers to the new implementation of blocks made my C. Béra and E. Miranda to avoid

```
CompilationContext bytecodeBackend: EncoderForSistaV1.
CompilationContext optionFullBlockClosure: true.
NBexp recompile: #blockLocalTemp.
(NBexp >> #blockLocalTemp) inspect
```

```
blockLocalTemp
  | collection |
  collection := OrderedCollection new.
  #(1 2 3) do: [ :index |
    | temp |
    temp := index.
    collection add: [ temp ] ].
  ^ collection collect: [ :each | each value ]
```

```
65 <10> pushLit: OrderedCollection
66 <7C> send: new
67 <D0> popIntoTemp: 0
68 <21> pushConstant: #(1 2 3)
69 <40> pushTemp: 0
70 <F9 02 01> fullClosure:compiledBlockNumCopied: 1
73 <7B> send: do:
74 <D8> pop
75 <40> pushTemp: 0
76 <F9 03 00> fullClosure:compiledBlockNumCopied: 0
79 <94> send: collect:
80 <5C> returnTop
```

1.30 **Conclusion**

In this chapter we learned how to use *blocks*, also called*lexical closures*, and how they are implemented. We saw that we can use a block even if the method defining it has returned. A block can access its own variables and also *non local* variables: instance variables, temporaries and arguments of the defining method.

We also saw how blocks can terminate a method and return a value to the sender. We say that these blocks are *non-local returning blocks* and that some care has to be taken to avoid errors: a block can not terminate a method that has already returned.

Finally, we show what contexts are and how they play an important role with block creation and execution. We show what the thisContext pseudo variable is and how to use it to get information about the executing context and potentially change it.