Chapter 1

Blocks: a Detailed Analysis

with the participation of:
Clément Bera *(bera.clement@gmail.com)*

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 fine 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.

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.

```
[ 1 + 2 ]
    ⟶  [ 1 + 2 ]
```

A block is evaluated by sending the value message to it. More precisely, blocks can be evaluated 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 evaluation. They were presented in the Pharo by Example book.

```
[ 1 + 2 ] value
    ⟶  3

[ :x | x + 2 ] value: 5
    ⟶  7
```

Some handy extensions

Beyond the value messages, Pharo includes some handy messages such as cull: and friends to support the evaluation 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.
```

Other messages. Some messages are useful to profile evaluation (more information in the Chapter **??**):

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

durationToRun. Answer the duration (instance of class Duration) taken to evaluate the receiver block.

timeToRun. Answer the number of milliseconds taken to evaluate this block.

Some messages are related to error handling (as explained in the Chapter **??**).

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

ifCurtailed: onErrorBlock. Evaluate the receiver, and, if the evaluation does not complete, evaluate the error block. If evaluation of the receiver finishes normally, the error block is not evaluated.

on: exception do: catchBlock. Evaluate the receiver. If an exception exception is raised, evaluate the catch block.

on: exception fork: catchBlock. Evaluate 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 evaluated 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.

1.2 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 evaluated. It means that a block, when evaluated 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 evaluation step such as the context from which the current one is executed, the next 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).

Some little experiments

Let's experiment a bit to understand how variables are bound in a block. Define a class named Bexp (for BlockExperiment):

```
Object subclass: #Bexp
   instanceVariableNames: ''
```

```
  classVariableNames: ''
  poolDictionaries: ''
  category: 'BlockExperiment'
```

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 traceCr] and one that defines a new variable with the same name and executes a block defined elsewhere.

```
Bexp>>setVariableAndDefineBlock
  | t |
  t := 42.
  self evaluateBlock: [ t traceCr ]

Bexp>>evaluateBlock: aBlock
  | t |
  t := nil.
  aBlock value

Bexp new setVariableAndDefineBlock
  ⟶  42
```

Executing the Bexp new setVariableAndDefineBlock expression prints 42 in the Transcript (message traceCr). The value of the temporary variable t defined in the setVariableAndDefineBlock method is the one used rather than the one defined inside the method evaluateBlock: even if the block is evaluated 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 evaluateBlock:).

Let's look at it in detail. Figure 1.1 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 block refers to the method activation context - which holds temporary variables (Step 1).
- The method evaluateBlock: 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 evaluated. While executing the method evaluateBlock: the block is evaluated (Step 2), during the execution of the expression t traceCr 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.

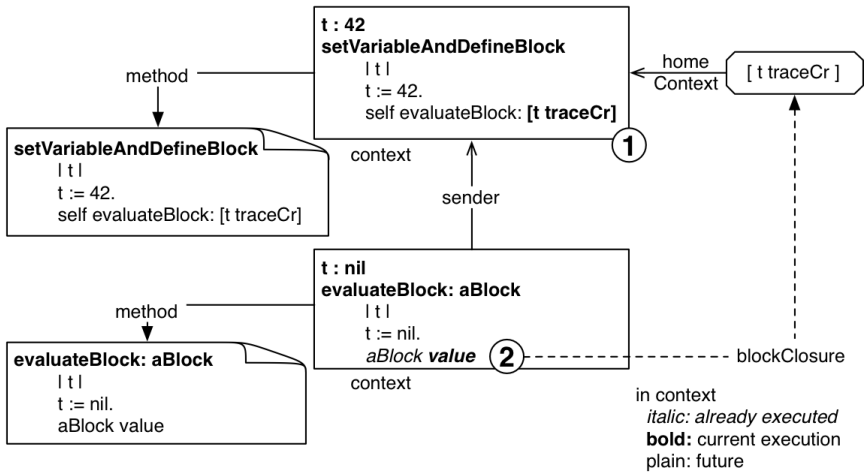

Figure 1.1: Non-local variables are looked up the method activation context where the block was *created* and not where it is *evaluated*.

> 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.

Experiment 2: Changing a variable value. Let's continue our experiments. The method setVariableAndDefineBlock2 shows that a non-local variable value can be changed during the evaluation of a block. Executing Bexp new setVariableAndDefineBlock2 prints 33, since 33 is the last value of the variable t.

```
Bexp>>setVariableAndDefineBlock2
   | t |
   t := 42.
   self evaluateBlock: [ t := 33. t traceCr ]

Bexp new setVariableAndDefineBlock2
      ⟶   33
```

Experiment 3: Accessing a shared non-local variable. Two 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 setVariableAndDefineBlock3 as follows:

```
Bexp>>setVariableAndDefineBlock3
   | t |
   t := 42.
   self evaluateBlock: [ t traceCr. t := 33. t traceCr ].
   self evaluateBlock: [ t traceCr. t := 66. t traceCr ].
   self evaluateBlock: [ t traceCr ]
```

```
Bexp new setVariableAndDefineBlock3
      ⟶  42
      ⟶  33
      ⟶  33
      ⟶  66
      ⟶  66
```

Bexp new setVariableAndDefineBlock3 will print 42, 33, 33, 66 and 66. Here the two blocks [t := 33. t traceCr] and [t := 66. t traceCr] access the same variable t and can modify it. During the first execution of the method evaluateBlock: 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 location.

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: ''
   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 evaluated. And this is the last value (69) that is effectively printed because it is looked up at execution-time. Executing Bexp new setVariableAndDefineBlock4 prints 69.

```
Bexp>>setVariableAndDefineBlock4
   | t |
   t := 42.
   block := [ t traceCr: t ].
   t := 69.
   self evaluateBlock: block
```

```
Bexp new setVariableAndDefineBlock4
    ⟶  69.
```

Experiment 5: For method arguments. We can expect that method arguments are bound in the context of the defining method. Let's illustrate this point now. Define the following methods.

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

Bexp>>testArg: arg
   block := [arg traceCr].
   self evaluateBlockAndIgnoreArgument: 'zork'.

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

Now executing `Bexp new testArg: 'foo'` prints `'foo'` even if in the method `evaluateBlockAndIgnoreArgument:` the temporary `arg` is redefined. In fact each method invocation has its own values for the arguments.

Experiment 6: self binding. Now we may wonder if self is also captured. 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: ''
   poolDictionaries: ''
   category: 'BlockExperiment'

Bexp>>initialize
   super initialize.
   x := 123.
```

Define another class named `Bexp2`.

```
Object subclass: #Bexp2
   instanceVariableNames: 'x'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'BlockExperiment'

Bexp2>>initialize
   super initialize.
```

```
  x := 69.

Bexp2>>evaluateBlock: aBlock
  aBlock value
```

Then define the methods that will invoke methods defined in Bexp2.

```
Bexp>>evaluateBlock: aBlock
  Bexp2 new evaluateBlock: aBlock

Bexp>>evaluateBlock
  self evaluateBlock: [self crTrace ; traceCr: x]
```

```
Bexp new evaluateBlock
    ⟶    a Bexp123  "and not a Bexp269"
```

Now when we execute Bexp new evaluateBlock, we 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.

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.

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 link to their creation. We will show the difference when a variable is local to a block or to a method (see Figure 1.2).

Block allocation. Implement the following method blockLocalTemp.

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

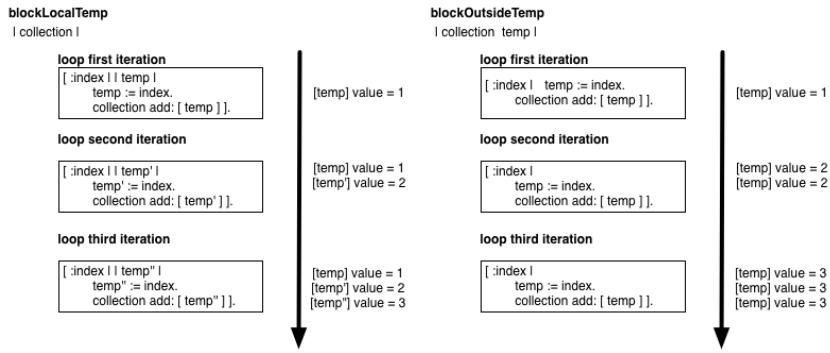

Figure 1.2: blockLocalTemp execution (Left) - blockOutsideTemp execution (Right)

Let's comment the code: we create a loop that stores the current index (an block argument) in a temporary variable temp created in the loop. We then store a block that accesses this variable in a collection. After the loop, we execute each accessing block and return the collection of values. If we execute this method, we get a collection with 1, 2 and 3. This result shows that each block in the collection refers to a different temp variable. This is due to the fact that an execution context is created for each block creation (at each loop step) and that the block [temp] is stored in this context.

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.

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

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 blockOutsideTemp context leading to the fact that temp is shared by the blocks.

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

Method-Block Sharing. 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.

```
Bexp>>foo
   | a |
   [ a := 0 ] value.
   ^ a

Bexp new foo
      ⟶    0
```

When we execute Bexp new foo, 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 evaluation. 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 evaluate the blocks stored in the array. This is what the following code snippet is doing.

```
| res |
res := Bexp new twoBlockArray.
res second value.    ⟶    0
res first value.
res second value.    ⟶    2
```

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

```
| res |
res := Bexp new twoBlockArray.
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 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.

1.4 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 or store into instance variables. A block with an 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, the execution of a return statement exits the currently executed method and returns to its caller. This ignores the expressions following the return statement.

Experiment 7: Return's Exiting Behavior. Define the following method. Executing Bexp new testExplicitReturn prints 'one' and 'two' but it will not print

not printed, since the method testExplicitReturn will have returned before.

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

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

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 |
                  self traceCr: each printString.
                  each = 3
                     ifTrue: [^ 3]].
   ^ 42

Bexp new jumpingOut
    ⟶    3
```

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. 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.

Understanding return

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 (defineBlock) creates an escaping block, one (arg:) evaluates this block and one (evaluatingBlock:) that executes the block. Note that to stress the escaping behavior of a return we defined evaluatingBlock: so that it endlessly loops after evaluating 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 evaluateBlock: aBlock.
    self traceCr: 'arg end'.
    ^ res

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

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

```
start start
  defineBlock start
    arg start
      evaluateBlock 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 `evaluateBlock:`. The evaluation of the block returns to the *block home context sender* (*i.e.*, the context that invoked the method creating the block).

When the return statement of the block is executed in the method `evaluateBlock:`, 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`.

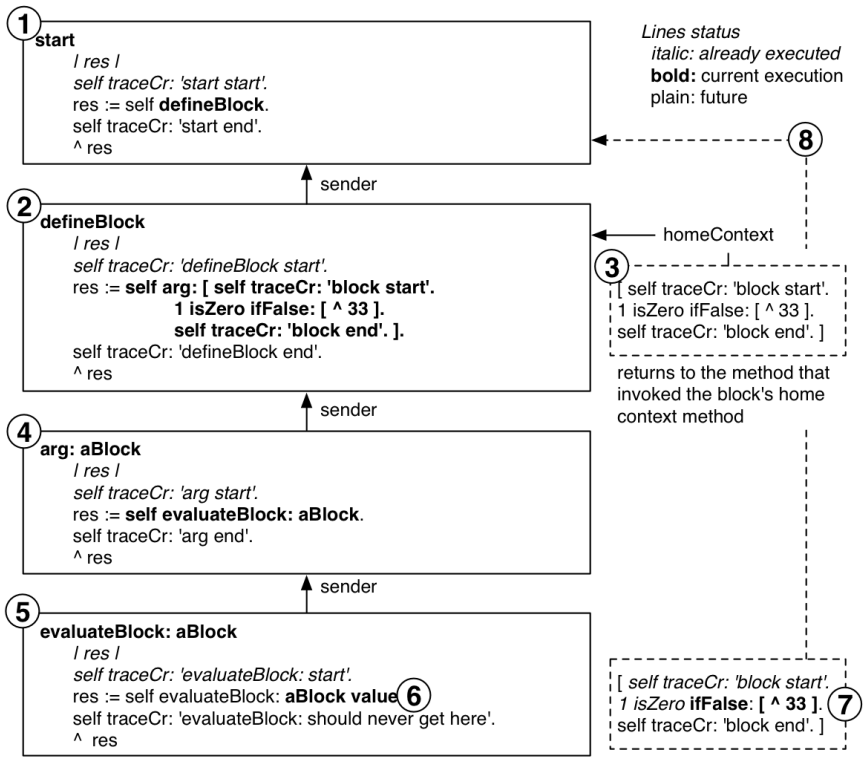

Figure 1.3: A block with non-local return execution returns to the method execution that activated the block home context. Frames represent contexts and dashed frames represent the same block at different execution points.

Therefore the return expression returns to the start method execution just after the defineBlock execution. This is why the pending executions of arg: and evaluateBlock: are discarded and why we see the execution of the method start end.

As shown by Figure 1.3, [^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 1 represents the execution up to the invocation of the method defineBlock. The trace 'start start' is printed.
- Step 3 represents the execution up to the block creation, which is done in Step 2. 'defineBlock start' is printed. The home context of the block is the defineBlock method execution context.

- Step 4 represents the execution up to the invocation of the method evaluateBlock:. 'arg start' is printed.
- Step 5 represents the execution up to the block evaluation. 'evaluateBlock 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' gets printed.

> 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.

> 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.

Accessing 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 evaluated during the execution of the method evaluateBlock:, 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.

Couple more examples. The following examples show that escaping blocks jump to sender of their home contexts. 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.

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

This method defines a block whose argument break is evaluated as soon as the step 4 of a loop is reached. A variable val is then printed and we make sure that its value is 4. Executing Bexp new testValueWithExitBreak performs without raising an error and prints 4 to the Transcript: the loop has been stopped, the value has been printed, and the assert has been validated.

If you change the valuePassingEscapingBlock message sent by value: [^ nil] in the testValueWithExitBreak method above, you will not get the trace because the execution of the method testValueWithExitBreak will exit when the block is evaluated. In this case, calling valuePassingEscapingBlock is not equivalent to calling value: [^nil] because the home context of the escaping block [^ nil] is different. With the original valuePassingEscapingBlock, the home context of the block [^ nil] is valuePassingEscapingBlock and not the method testValueWithExitContinue itself. Therefore when evaluated, 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 another case 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. 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.

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

1.5 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 CompiledMethod. 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.

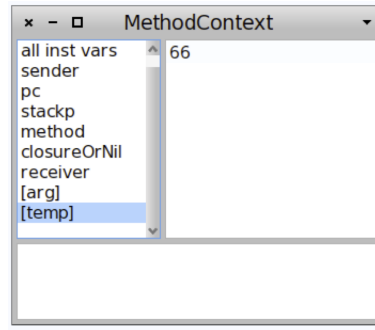

Figure 1.4: A method context where we can access the value of the temporary variable temp at that given point of execution.

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 1.4. 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 instance variable and select the all bytecodes field, you should get the situation depicted in Figure 1.5, 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 of the arguments by sending the arguments message and the value of a particular temporary variable by sending tempNamed:.

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.

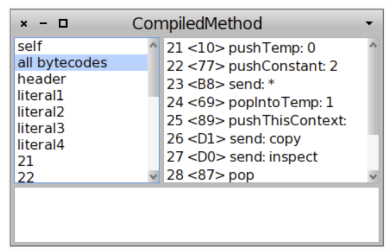

Figure 1.5: The pc variable holds 27 because the last (bytecode) instruction executed was the message send inspect.

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.

1.6 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.

Sending 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 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.

3. When an appropriate method is found:

   (a) check for a primitive associated with the method by reading the method header;

   (b) if there is a primitive, execute it;

   (c) if the primitive completes successfully, return the result object to the message sender;

   (d) when there is no primitive or the primitive fails, continue to the next step.

4. 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.

5. 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 1.6 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.

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

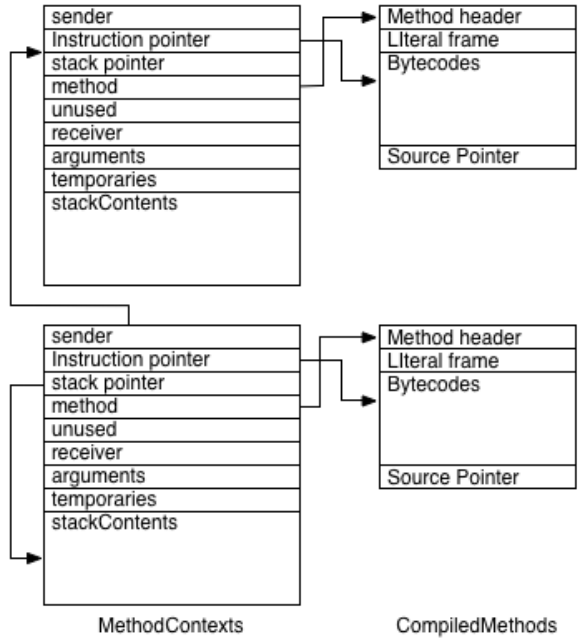

Figure 1.6: Relations between contexts and compiled methods .

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 1.7. 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 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 evaluated outside their home context and still refer to remote variables. Hence all home contexts might outlive the method activation.

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

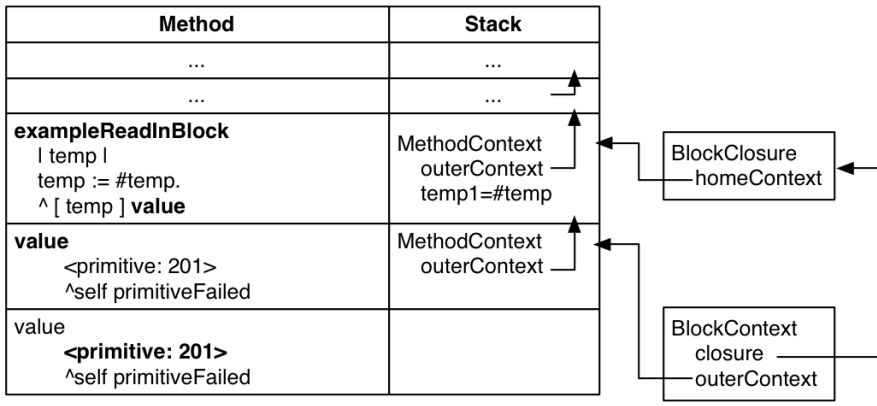

Figure 1.7: A first understanding of closures.

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

1.7 Chapter 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

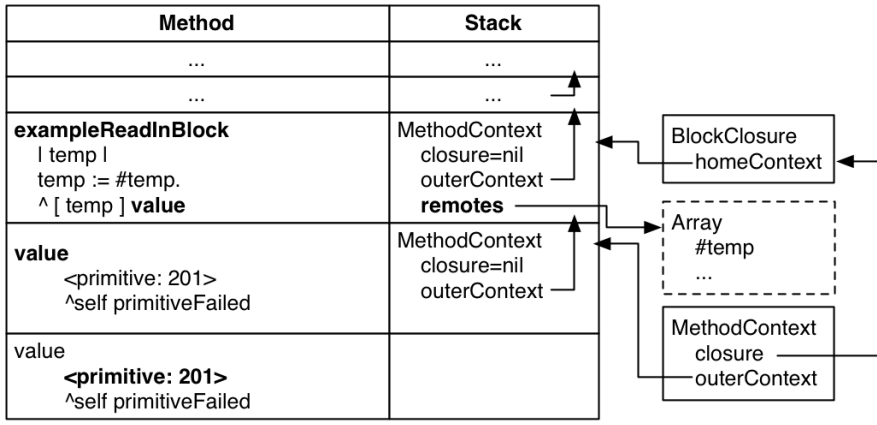

Figure 1.8: How the VM stores remote variables so that they continue to leave when a method returns.

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.

We thank Eliot Miranda for the clarifications.