Fun with Interpreters in Pharo

Stéphane Ducasse and Guillermo Polito

March 12, 2024

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

Contents

Illustrations

CHAPTER 1

The Language: ObjVlisp

We are going to write a programming language interpreter. And this raises the questions: what language are we going to interpret? A language that is too-complex would take ages to implement, the book would be too long. Think of implementing any programming language you know: how many features does it have? Complex languages have a lot of accidental complexity (think C, C++, Java).

In this book we have made a choice: ObjVlisp. ObjVLisp is small enough to be implementable in a couple of afternoons. And it is large enough to be interesting, as we will show you next.

1.1 ObjVlisp: a language kernel

ObjVlisp is an object-oriented programming language originally published in 1986 when the foundation of object-oriented programming was still emerging. Actually, ObjVlisp is not a typical programming language, but a language kernel. ObjVlisp is not a programming language because it does not have an associated syntax: we will attach one to it later, don't worry. ObjVlisp is a language kernel because it defines the core concepts of the programming language, and their associated semantics.

A bit of History:

ObjVlisp was inspired from the kernel of Smalltalk-78. The IBM SOM-DSOM kernel is similar to ObjVLisp while implemented in C++. ObjVlisp is a subset of the reflective kernel of CLOS (Common Lisp Object System) since CLOS reifies instance variables, generic functions, and method combination.

Why ObjVlisp?

ObjVlisp has the following properties that make it interesting to study and learn:

- It unifies the concepts of class and instance (there is only one data structure to represent all objects, classes included),
- It is composed of only two classes `Class` and `Object` (it relies on existing elements such as booleans, arrays, and string of the underlying implementation language),
- It raises the question of meta-circular infinite regressions (a class is an instance of another class that is an instance of yet another class, etc.) and how to resolve it,
- It requires consideration of allocation, class and object initialization, message passing as well as the bootstrap process,
- It can be implemented in less than 30 methods in Pharo.

Another important point: this kernel is self-described. This does not mean that we are going to *write* ObjVlisp in itself (we cannot because ObjVLisp does not have a syntax). What this means is that some of its concepts are recursive: for example, classes are objects in ObjVlisp, which means that classes are instances of classes. See? We will start by explaining some aspects, but since everything is linked, you may have to read the chapter twice to fully get it.

1.2 ObjVLisp's six postulates

The original ObjVlisp kernel is defined by six postulates. Some of them look a bit dated by modern standards, and the 6th postulate is simply wrong as we will explain later (a solution is simple to design and implement). Here are the six postulates as stated in the original paper for the sake of historical perspective.

1. An object represents a piece of knowledge and a set of capabilities.
2. The only protocol to activate an object is message passing: a message specifies which procedure to apply (denoted by its name, the selector) and its arguments.
3. Every object belongs to a class that specifies its data (attributes called fields) and its behavior (procedures called methods). Objects will be dynamically generated from this model; they are called instances of the class. Following Plato, all instances of a class have same structure and shape, but differ through the values of their common instance variables.

4. A class is also an object, instantiated by another class, called its metaclass. Consequently (P3), to each class is associated a metaclass which describes its behavior as an object. The initial primitive metaclass is the class Class, built as its own instance.
5. A class can be defined as a subclass of one (or many) other class(es). This subclassing mechanism allows sharing of instance variables and methods, and is called inheritance. The class Object represents the most common behavior shared by all objects.
6. If the instance variables owned by an object define a local environment, there are also class variables defining a global environment shared by all the instances of a same class. These class variables are defined at the metaclass level according to the following equation: class variable [an-object] = instance variable [an-object's class].

1.3 ObjVLisp Model Overview

Contrary to a real uniform language kernel, ObjVlisp does not consider arrays, booleans, strings, numbers or any other elementary objects as part of the kernel as this is the case in a real bootstrap such as the one of Pharo. ObjVLisp's kernel focuses on understanding the core relationships between classes and objects.

Figure 1-1 shows the two core classes of the kernel:

- `Object` is the root of the inheritance graph and is an instance of `Class`.
- `Class` is the first class and root of the instantiation tree and instance of itself as we will see later.

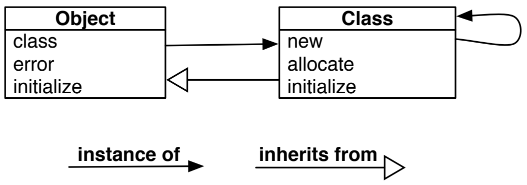


Figure 1-1 The ObjVlisp kernel: a minimal class-based kernel.

Now imagine we wanted to implement a class `Workstation` (which means *a really old computer*, you can look it up online ;)). Figure 1-2 shows how that could take place in this abstract model. The class `Workstation` is an instance of the class `Class` since it is a class. It inherits from `Object`, meaning that it has the default behavior that objects exhibit.

Moreover, we can do something really cool in ObjVlisp: we can change how classes work using *metaclasses*. The class `WithSingleton` is an instance of the class `Class` (as any other class) but differently from our `Workstation` it inherits from `Class`. This means that `WithSingleton` will *behave* like a class! This means that this class we just created is not just a normal but a metaclass: its instances are classes. Metaclasses change the behavior of classes. Could you guess by the name how the instances of `WithSingleton` are supposed to behave?

Finally, suppose we implement a class named `SpecialWorkstation` as an instance of the class `WithSingleton` and inheriting from `Workstation`. This structure illustrates the different roles of instantiation and inheritance. Our `SpecialWorkstation` will behave as a class `WithSingleton`, and it's instances will behave as defined by `SpecialWorkstation` and its superclass `Workstation`.

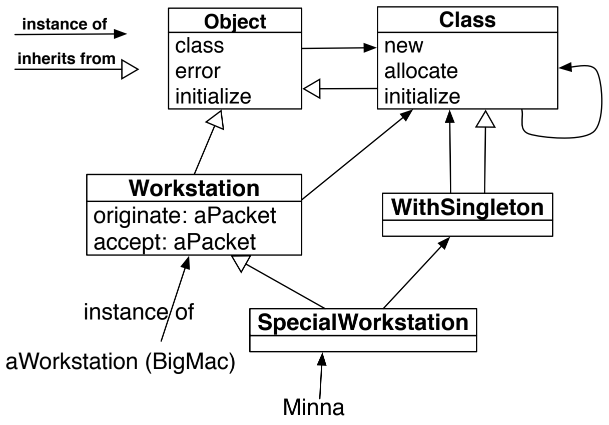


Figure 1-2 The kernel with specialized metaclasses.

The two diagrams 1-1 and 1-2 will be explained step by step throughout this chapter.

Note: We will see later that understanding such an architecture is important to understand message passing and how methods get executed. Message passing always looks up methods in the class of the receiver of the message and then follows the inheritance chain (See Figure 1-3) thus following first the instantiation link, then the inheritance link! Figure 1-3 illustrates two main cases:

- When we send a message to `BigMac` or `Minna`, the corresponding method is looked up in their corresponding classes `Workstation` or `SpecialWorkstation` and follows the inheritance link up to `Object`.
- When we send a messsage to the classes `Workstation` or `Special-`

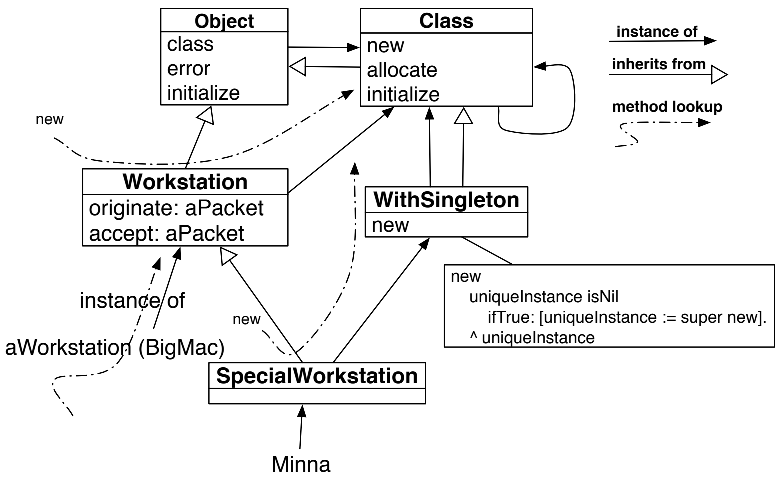


Figure 1-3 Understanding metaclasses using message passing.

`Workstation`, the corresponding method is looked up in their class, the class `Class` and up to `Object`.

1.4 Modelling Instances and Classes

In this kernel, classes and objects are linked by the instantiation link, as shown by Figure 1-4:

- Terminal instances are obviously objects: a workstation named `mac1` is an instance of the class `Workstation`, a point `10@20` is instance of the class `Point`.
- Classes are also objects (instances) of other classes: the class `Workstation` is an instance of the class `Class`, the class `Point` is an instance of the class `Class`.

Things to think about: What could be the class of the class `Class` and why?

A class defines an ordered sequence of instance variables definitions. Each variable definition is just the name of the variable. All instances of a class share the same variable definitions. However, each instance will have its own specific value for each variable definition. For example, Figure 1-5 shows that instances of `Workstation` have two values: a name and a next node.

In our diagrams, we represent terminal instances as rounded rectangles. Inside each rectangle, there is a list of the values of its instance variables. Since classes are objects, *when we want to stress that classes are objects* we will later use a different graphical convention.

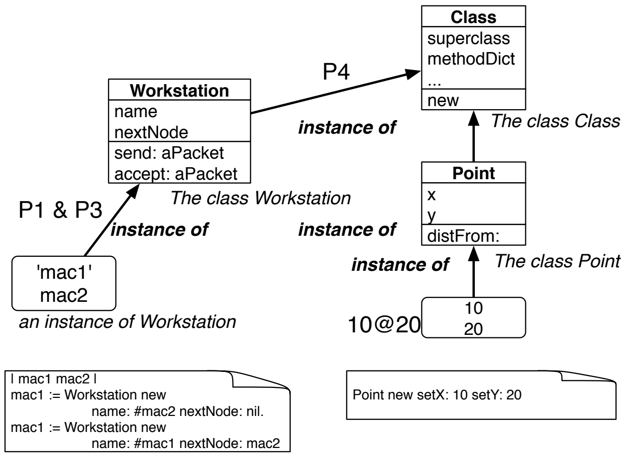


Figure 1-4 Chain of instantiation: classes are objects, too.

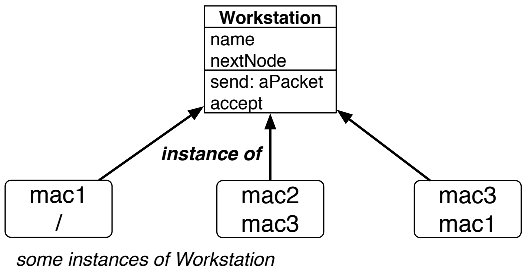


Figure 1-5 Instances of `Workstation` have two values: their names and their next node.

Notice also that an object has a reference to its class. As we will see when we discuss inheritance later on, every object possesses an instance variable class (inherited from `Object`) that references to its class. We will not add that extra instance variable in the diagrams, because it is redundant with the arrow.

Things to think about:

- How is this model different so far from languages such as Pharo and Java? And what about Python and Javascript? Do these languages have classes too?
- What about the instance variables? How are they declared? Are they fixed or can be dynamically added? What about types?
- A bit more complex: Are classes objects too in those languages? Do

they have metaclasses?

1.5 A Brief Introduction to Messages and Methods

So far we have defined the structural part of our language kernel: the entities that make part of it and the relationships between themselves. But let's remember that we define a programming language so we can automate computations with it: we want to execute some code!

This kernel presents a single operation to perform computation: message passing. Message passing is the act of one object (the sender) to send a message to another object (the receiver). When an object receives a message, it must search for a method to execute, a mechanism we will call *method lookup*. In `ObjVlisp`, the *method lookup* will be as in most traditional object-oriented programming languages: when an object receives a message, it will search for the method in the class hierarchy starting from its class.

The `ObjVlisp` kernel represents how methods are stored and looked up as follows. Methods belong to a class and are stored into a dictionary that associates method names (the selectors) with the method bodies containing the code to execute. Since methods are stored in a class, the method dictionary should be described in the metaclass. Therefore, the method dictionary of a class is the *value* of the instance variable `methodDict` defined on the metaclass `Class`. Each class will have its own method dictionary.

`ObjVlisp` does not specify how methods are represented, we will choose and attach a representation in the following chapters.

1.6 Conclusion

We presented a small kernel composed of two classes: `Object`, the root of the inheritance tree and `Class`, the first metaclass root of the instantiation tree. We briefly revisited the ideas behind message passing. In the next chapter we propose to you how to implement such a kernel.

Further readings

The kernel presented in this chapter is a kernel with explicit metaclasses and as such it is not a panacea. Indeed it results in problems with metaclass composition as explained in Bouraqadi et al.'s excellent article or .

CHAPTER 2

Representing Code with Abstract Syntax Trees

To execute actual code on top of our ObjVlisp kernel, we need code to execute. And since ObjVlisp does not force a way to represent code, we need to choose one oneselves.

Probably the simplest way to represent code we can think about are strings. That is actually what we write in editors: strings. However, strings are not the easiest to *manipulate* when we want to execute code. Instead of manipulating strings, we are going to transform strings into a more practical data structure using a parser. However, in this book we are not interested in getting into the problems of parsing (lots of books do a very fine job on that already). We will use an already existing parser and we will borrow a syntax to not define our own: Pharo's parser and Pharo's syntax.

Now that we have decided what syntax we will have at the surface of our language, we need to choose a data structure to represent our code. One fancy way to represent code is using abstract syntax trees, or in short, ASTs. An abstract syntax tree is a tree data structure that represents a program from the syntax point of view. In other words, each node in the tree represents an element that is written in a program such as variables, assignments, strings, and message sends. To illustrate it, consider the piece of Pharo code below that assigns into a variable named `variable` the result of sending the `,` message to a `'constant'` string, with `self message` as argument.

```
variable := 'constant' , self message
```

This chapter presents ASTs by looking at the existing AST implementation in Pharo, the RBAST. RBAST is the AST implementation used currently (Pharo 11.0) by many tools in Pharo's tool-chain, such as the compiler, the syntaxhighlighter, the auto-completion, the code quality engine and the refactoring engine. As so, it's an interesting piece of engineering, and we will find it provides most of what we will need for our journey to have fun with interpreters.

In the following chapter we will study (or re-study, for those who already know it) the Visitor design pattern. To be usable by the many tools named before, RBASTs implement a visitor interface. Tools performing complex operations on ASTs may then define visitor classes with their algorithms. As we will see in the chapters after this one, one such tool is an interpreter, thus mastering ASTs and visitors is essential.

2.1 Pharo Abstract Syntax Trees

An abstract syntax tree is a tree data structure that represents a program from the syntax point of view. In the tree, nodes represent the syntactic elements of the program. The edges in the tree represent how those nodes are related. To make it concrete, Figure 2-1 shows the AST that represents the code of our previous example: `variable := 'constant' , self message`. As we will see later, each node in the tree is represented by an object, and different kind of nodes will be instances of diferent classes, forming a composite.

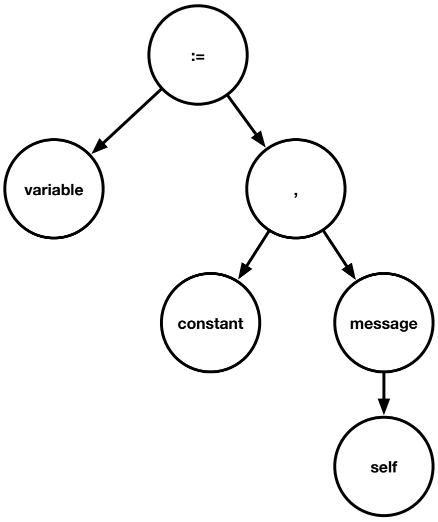


Figure 2-1 AST representing the code of `variable := 'constant' , self message`.

The Pharo standard distribution comes with a pretty complete AST implementation that is used by many tools. To get our hands over an AST, we could build it ourselves manually, or as we will do in this chapter, we ask a parser to parse some text and build an AST for us. Fortunately, Pharo also includes a parser that does exactly this: the `RBParser`. The `RBParser` class implements a parser for Pharo code. It has two main modes of working: parsing expressions and parsing methods.

For the Purists: abstract vs concrete trees

People tend to make the distinction between abstract and concrete syntax trees. The difference is the following: an abstract syntax tree does not contain information about syntactic elements per se. For example an abstract syntax does not contain information about parentheses since the structure of the tree itself reflects it. This is similar for variable definition delimiters (pipes) or statement delimieters (periods) in Pharo. A concrete tree on the other hand keeps such information because tools may need it. From that perspective, the Pharo AST is in between both. The tree structure contains no information about the concrete elements of the syntax, but these informations are remembered by the nodes so the source code can be rebuilt as similar as the original code as possible. However, we make a bit of language abuse and we refer to them as ASTs.

2.2 Parsing Expressions

Expressions are constructs that can be evaluated to a value. For example, the program `17 max: 42` is the message-send `max:` to receiver 17 with argument 42, and can be evaluated to the value 42 (since it is bigger than 17).

```
| expression |
expression := RBParser parseExpression: '17 max: 42'.
expression receiver formattedCode
>>> 17

expression selector
>>> #max

expression arguments first formattedCode
>>> 42
```

Expressions are a natural instances of the composite pattern, where expressions can be combined to build more complex expressions. In the following example, the expression `17 max: 42` is used as the receiver of another message expression, the message `asString` with no arguments.

```
| expression |
expression := RBParser parseExpression: '(17 max: 42) asString'.
expression receiver formattedCode
>>> (17 max: 42)
```

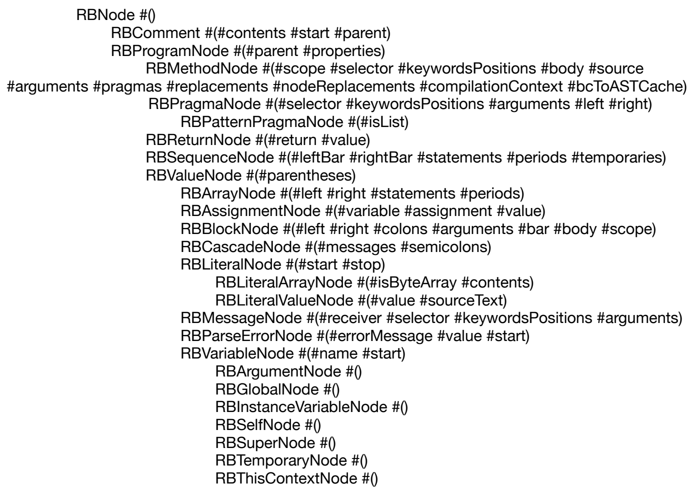
```
expression selector
>>> #asString

expression arguments
>>> #()
```

Of course, message sends are not the only kind of expressions we have in Pharo. Another kind of expression that appeared already in the examples above are literal objects such as numbers.

```
| expression |
expression := RBParser parseExpression: '17'.
expression formattedCode
>>> 17
```

Pharo is a simple language, the number of different nodes that can compose the method ASTs is structured in a class hiearchy. Figure 2-2 shows the node inheritance hierarchy of Pharo rendered as a textual tree.



```
RBNode #()
    RBComment #(#contents #start #parent)
    RBProgramNode #(#parent #properties)
        RBMethodNode #(#scope #selector #keywordsPositions #body #source
#arguments #pragmas #replacements #nodeReplacements #compilationContext #bcToASTCache)
        RBPragmaNode #(#selector #keywordsPositions #arguments #left #right)
            RBPatternPragmaNode #(#isList)
        RBReturnNode #(#return #value)
        RBSequenceNode #(#leftBar #rightBar #statements #periods #temporaries)
        RBValueNode #(#parentheses)
            RBArrayNode #(#left #right #statements #periods)
            RBAssignmentNode #(#variable #assignment #value)
            RBBlockNode #(#left #right #colons #arguments #bar #body #scope)
            RBCascadeNode #(#messages #semicolons)
            RBLiteralNode #(#start #stop)
                RBLiteralArrayNode #(#isByteArray #contents)
                RBLiteralValueNode #(#value #sourceText)
            RBMessageNode #(#receiver #selector #keywordsPositions #arguments)
            RBParseErrorNode #(#errorMessage #value #start)
            RBVariableNode #(#name #start)
                RBArgumentNode #()
                RBGlobalNode #()
                RBInstanceVariableNode #()
                RBSelfNode #()
                RBSuperNode #()
                RBTemporaryNode #()
                RBThisContextNode #()
```

Figure 2-2 Overview of the method node hierarchy (TODO: remove RBPattern-PragmaNode - Add RBSelectorNode). Indentation implies inheritance.

2.3 Literal Nodes

Literal nodes represent literal objects. A literal object is an object that is not created by sending the `new` message to a class. Instead, the developer writes directly in the source code the value of that object, and the object is created automatically from it (could be at parse time, at compile time, or at runtime, depending on the implementation). Literal objects in Pharo are strings, symbols, numbers, characters, booleans (`true` and `false`), `nil` and literal arrays (`#( )`).

Literal nodes in Pharo are instances of the `RBLiteralValueNode`, and understand the message `value` which returns the value of the object. In other words, literal objects in Pharo are resolved at parse time. Notice that the `value` message does not return a string representation of the literal object, but the literal object itself.

From now on we will omit the declaration of temporaries in the code snippets for the sake of space.

```
integerExpression := RBParser parseExpression: '17'.
integerExpression value
>>> 17
```

```
trueExpression := RBParser parseExpression: 'true'.
trueExpression value
>>> true
```

```
"Remember, strings need to be escaped"
stringExpression := RBParser parseExpression: '''a string'''.
stringExpression value
>>> 'a string'
```

A special case of literals are literal arrays, which have their own node: `RBLiteralArrayNode`. Literal array nodes understand the message `value` as any other literal, returning the literal array instance. However, it allows us to access the sub collection of literals using the message `contents`.

```
arrayExpression := RBParser parseExpression: '#(1 2 3)'.
arrayExpression value
>>> #(1 2 3)

arrayExpression contents first
>>> RBLiteralValueNode(1)
```

In addition to messages and literals, Pharo programs can contain variables.

The Variable Node, Self and Super Nodes

Variable nodes in the AST tree are used when variables are used or assigned to. Variables are instances of `RBVariableNode` and know their `name`.

```
variableExpression := RBParser parseExpression: 'aVariable'.
variableExpression name
>>> 'aVariable'
```

Variable nodes are used to equally denote temporary, argument, instance, class or global variables. That is because at parse-time, the parser cannot differentiate when a variable is of one kind or another. This is especially true when we talk about instance, class and global variables, because the context to distinguish them has not been made available. Instead of complexifying the parser with this kind of information, the Pharo toolchain does it in a pipelined fashion, leaving the tools using the AST decide on how to proceed. The parser generates a simple AST, later tools annotate the AST with semantic information from a context if required. An example of this kind of treatment is the compiler, which requires such contextual information to produce the correct final code.

For the matter of this book, we will not consider nor use semantic analysis, and we will stick with normal `RBVariableNode` objects. The only exception to this are `self`, `super` and `thisContext` special variables. Special variables are variables that are recognised at parse-time, and generating special nodes `RBSelfNode`, `RBSuperNode` and `RBThisContextNode` for them. These special nodes inherit from `RBVariableNode` and work as normal variable nodes for the purposes of this book.

2.4 Assignment Nodes

Assignment nodes in the AST represent assignment expressions using the `:=` operator. In Pharo, following Smalltalk design, assignments are expressions: their value is the value of the variable after the assignment. This allows to chain assignments. We will see in the next chapter, when implementing an evaluator, why this is important.

An assignment node is an instance of `RBAssignmentNode`. If we send it the `variable` message, it answers the variable it assigns to. The message `value` returns the expression at the right of the assignment.

```
assignmentExpression := RBParser parseExpression: 'var := #( 1 2 )
    size'.
assignmentExpression variable
>>> RBVariableNode(var)
```

```
assignmentExpression value
>>> RBMessageNode(#(1 2) size)
```

2.5 Message Nodes

Message nodes are the core of Pharo programs, and they are the most common composed expression nodes we find in the AST. Messages are instances of `RBMessageNode` and they have a receiver, a selector and a collection of arguments, obtained through the `receiver`, `selector` and `arguments` messages. We say that message nodes are composed expressions because the `receiver` and `arguments` of a message are expressions in themselves, which can be as simple as literals or variables, or other composed messages too.

```
messageExpression := RBParser parseExpression: '17 max: 42'.
messageExpression receiver
>>> RBLiteralValueNode(17)
```

Note that `arguments` is a normal collection of expressions - in the sense that there is not special node class to represent such a sequence.

```
messageExpression arguments
>>> an OrderedCollection(RBLiteralValueNode(42))
```

And that the message selector returns also just a symbol.

```
messageExpression selector
>>> #max:
```

A note on message nodes and precedence

For those readers that already master the syntax of Pharo, you remember that there exist three kind of messages: unary, binary and keyword messages. Besides their number of parameters, the Pharo syntax accords an order of precedence between them too, i.e., unary messages get to be evaluated before binary messages, which get to be evaluated before keyword messages. Only parentheses override this precedence. Precedence of messages in ASTs is resolved at parse-time. In other words, the output of `RBParser` is an AST respecting the precedence rules of Pharo.

Let's consider a couple of examples illustrating this, illustrated in Figure 2-3. If we feed the `RBParser` with the expression below, it will create a `RBMessageNode` as we already know it. The root of that message node is the `keyword:` message, and its first argument is the `argument + 42 unaryMessage` subexpression. That subexpression is in turn another message node with the `+` binary selector, whose first argument is the `42 unaryMessage` subexpression.

```
variable keyword: argument + 42 unaryMessage
```

Now, let's change the expression adding extra parenthesis as in:

```
variable keyword: (argument + 42) unaryMessage
```

The resulting AST completely changed! The root is still the `keyword:` message, but now its first argument is the `unaryMessage` sent to a (now in parenthesis) `(argument + 42)` receiver.

Finally, if we modify the parenthesis again to wrap the keyword message, the root of the resulting AST has changed too. It is now the `+` binary message.

```
(variable keyword: argument) + 42 unaryMessage
```

`RBParser` is a nice tool to play with Pharo expressions and master precedence!

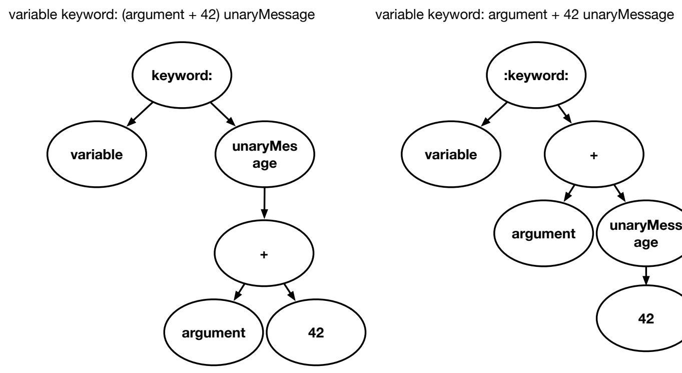

variable keyword: (argument + 42) unaryMessage
keyword:
variable
unaryMessage
+
argument
42
variable keyword: argument + 42 unaryMessage
:keyword:
variable
+
argument
unaryMessage
42

Figure 2-3 Different precedence results in different ASTs.

2.6 Cascade Nodes

Cascade nodes represent cascaded message expressions, i.e., messages sent to the same receiver. Cascaded messages are messages separated by semi-colons (;) such as in the following example.

```
OrderedCollection new
    add: 17;
    add: 42;
    yourself
```

This cascade is, in practical terms, equivalent to a sequence of messages to the same receiver:

```
t := OrderedCollection new.
t add: 17.
t add: 42.
t yourself
```

However, in contrast with the sequence above, cascades are expressions: their value is the value of the last message in the cascade.

A cascade node is an instance of `RBCascadeNode`. A cascade node understands the `receiver` message, returning the receiver of the cascade. It also understands the `messages` message, returning a collection with the messages in the cascade. Note that the messages inside the cascade node are normal `RBMessageNode` and have a receiver too. They indeed share the same receiver than the cascade. In the following chapters we will have to be careful when manipulating cascade nodes, to avoid to wrongly manipulate twice the same receiver.

```
cascadeExpression := RBParser parseExpression: 'var msg1; msg2'.
cascadeExpression receiver
>>> RBVariableNode(var)

cascadeExpression messages
>>> an OrderedCollection(RBMessageNode(var msg1) RBMessageNode(var
    msg2))
```

2.7 Dynamic Literal Array Nodes

Pharo has dynamic literal arrays. A dynamic literal array differs from a literal array in that its elements are calculated at runtime instead of at parse time. To delay the execution of the elements in the dynamic array, a dynamic array node contains expressions, separated by dots.

```
{ 1 + 1 . self message . anObject doSomethingWith: anArgument + 3 }
```

Dynamic literal arrays nodes are instances of `RBArrayNode`. To access the expressions inside a dynamic array node, they understand the message `children`

```
arrayNode := RBParser parseExpression: '{
  1 + 1 .
  self message .
  anObject doSomethingWith: anArgument + 3 }'.

arrayNode children.
>>> an OrderedCollection(
  RBMessageNode((1 + 1))
  RBMessageNode(self message)
  RBMessageNode((anObject doSomethingWith: anArgument + 3)))
```

2.8 Method and Block Nodes

Now that we have studied the basic nodes representing expressions, we can build up methods from them. Methods are represented as instances of `RBMethodNode` and need to be parsed with a variant of the parser we have used so far, a method parser. The `RBParser` class fulfills the role of a method parser when we use the message `parseMethod:` instead of `parseExpression:`. For example, the following piece of code returns a `RBMethodNode` instance for a method named `myMethod`.

```
methodNode := RBParser parseMethod: 'myMethod
  1+1.
  self'
```

A method node differs from the expression nodes that we have seen before by the fact that method nodes can only be roots in the AST tree. Method nodes cannot be children of other nodes. This differs from other programming languages in block in which method definitions are indeed expressions or statements that can be nested. In Pharo method definitions are not statements: like class definitions, they are top level elements. This is why Pharo is not a block structure language, even if it has closures (named blocks) that can be nested, passed as arguments or stored.

Method nodes have a name or selector, accessed through the `selector` message, a list of arguments, accessed through the `arguments` message, and as we will see in the next section they also contain a body with the list of statements in the method.

```
methodNode selector
>>> #myMethod
```

2.9 Sequence Nodes

Method nodes have a body, represented as a `RBSequenceNode`. A sequence node is a sequence of instructions or statements. All expressions are statements, including all nodes we have already seen such as literals, variables, arrays, assignments and message sends. We will introduce later two more kind of nodes that can be included as part of a sequence node: block nodes and return nodes. Block nodes are expressions that are syntactically and thus structurally similar to methods. Return nodes, representing the return instruction ^, are statement nodes but not expression nodes, i.e., they can only be children of sequence nodes.

If we take the previous example, we can access the sequence node body of our method with the `body` message.

```
methodNode := RBParser parseMethod: 'myMethod
  1+1.
  self'.

methodNode body
>>> RBSequenceNode(1 + 1. self)
```

And we can access and iterate the instructions in the sequence by asking it its `statements`.

```
methodNode body statements.
>>> an OrderedCollection(RBMessageNode(1 + 1) RBSelfNode(self))
```

Besides the instructions, sequence nodes also are the ones defining temporary variables. Consider for example the following method defining a temporary.

```
myMethod
  | temporary |
  1+1.
  self'
```

In an AST, temporary variables are defined as part of the sequence node and not the method node. This is because temporary variables can be defined inside a block node, as we will see later. We can access the temporary variables of a sequence node by asking it for its `temporaries`.

```
methodNode := RBParser parseMethod: 'myMethod
  | temporary |
  1+1.
  self'.
methodNode body temporaries.
>>> an OrderedCollection(RBVariableNode(temporary))
```

2.10 Return Nodes

AST return nodes represent the instructions that are syntactically identified by the caret character ^. Return nodes, instances of `RBReturnNode` are not expression nodes, i.e., they can only be found as a direct child of sequence nodes. Return nodes represent the fact of returning a value, and that value is an expression, which we is accessible through the `value` message.

```
methodNode := RBParser parseMethod: 'myMethod
  1+1.
  ^ self'.

returnNode := methodNode body statements last.
>>>RBReturnNode(^ self)

returnNode value.
>>>RBSelfNode(self)
```

Note that as in Pharo return statements are not mandatory in a method, they are not mandatory in the AST either. Indeed, we can have method ASTs without return nodes. In those cases, the semantics of Pharo specifies that `self` is implicitly returned. It is interesting to note that the AST does not contain semantics but only syntax: we will add semantics to the AST when we evaluate it in a subsequent chapter. In Pharo this is the compiler that ensures that a method always return self when return statements are absent in some execution paths.

Also, as we said before, return nodes are not expressions, meaning that we cannot write any of the following:

```
x := ^ 5
```

```
{ 1 . ^ 4 }
```

Block Nodes

Block nodes represent block closure expressions. A block closure is an object syntactically delimited by square brackets `[]` that contains statements and can be evaluated using the `value` message and its variants. The block node is the syntactic counter-part of a block closure: it is the expression that, when evaluated, will create the block object.

Block nodes work by most means like method nodes: they have a list of arguments and a sequence node as body containing temporaries and statements. They differentiate from methods in two aspects: first, they do not have a selector, second, they are expressions (and thus can be parsed with `parseExpression:`). They can be stored in variables, passed as message arguments and returned by messages.

```
blockNode := RBParser parseExpression: '[ :arg | | temp | 1 + 1.
    temp ]'.
blockNode arguments
>>>an OrderedCollection(RBVariableNode(arg))

blockNode body temporaries
>>>an OrderedCollection(RBVariableNode(temp))

blockNode body statements
>>>an OrderedCollection(RBMessageNode(1 + 1) RBVariableNode(temp))
```

2.11 Basic ASTs Manipulations

We have already covered all of Pharo AST nodes, and how to access the information in them. Those knowing ASTs for other languages, would have noticed that we have indeed few nodes. This is because in Pharo, control-flow statements such as conditionals or loops are expressed as messages, so no special case for them is required in the syntax. Because of this, Pharo's syntax fits in a post-card.

In this section we will explore some core-messages of Pharo's AST, that allow common manipulation for all nodes: iterating the nodes, storing meta-data and testing methods. Most of these manipulations are rather primitive and simple. In the next Chapter we will see how the visitor pattern in conjunction with ASTs empowers us, and gives us the possibility to build more complex applications such as concrete and abstract evaluators as we will see in the next chapters.

Iterating over an AST

ASTs are indeed trees, and we can traverse them as any other tree. RBASTs provide several protocols for accessing and iterating any AST node in a generic way.

- `aNode children`: returns a collection with the direct children of the node.
- `aNode allChildren`: returns a collection with all recursive children found from the node.
- `aNode nodesDo: aBlock`: iterates over allChildren applying `aBlock` on each of them.
- `aNode parent`: returns the direct parent of the node.
- `aNode methodNode`: returns the method node that is the root of the tree. For consistency, expressions nodes parsed using `parseExpression:` are contained within a method node too.

Storing Properties

Some manipulations require storing meta-data associated to AST nodes. Pharo ASTs provide a set of messages for storing arbitrary properties inside a node. Properties stored in a node are indexed by a key, following the API of Pharo dictionaries.

- `aNode propertyAt: aKey put: anObject`: inserts `anObject` at `aKey`, overriding existing values at `aKey`.
- `aNode hasProperty: aKey`: returns a boolean indicting if the node contains a property indexed by `aKey`.
- `aNode propertyAt: aKey`: returns the value associated with `aKey`. If `aKey` is not found, fails with an exception.
- `aNode propertyAt: aKey ifAbsent: aBlock`: returns the value associated with `aKey`. If `aKey` is not found, evaluates the block and returns its value.
- `aNode propertyAt: aKey ifAbsentPut: aBlock`: returns the value associated with `aKey`. If `aKey` is not found, evaluates the block, inserts the value of the block at `aKey` and returns the value.
- `aNode propertyAt: aKey ifPresent: aPresentBlock ifAbsent: anAbsentBlock`: Searches for the value associated with `aKey`. If `aKey` is found, evaluates `aPresentBlock` with its value. If `aKey` is not found, evaluates the block and returns its value.
- `aNode removeProperty: aKey`: removes the property at `aKey`. If `aKey` is not found, fails with an exception.
- `aNode removeProperty: aKey ifAbsent: aBlock`: removes the property at `aKey`. If `aKey` is not found, evaluates the block and returns its value.

Testing Methods

ASTs provide a testing protocol that can be useful for small applications and writing unit tests. All ASTs answer the messages `isXXX` with a boolean `true` or `false`. A first set of methods allow us to ask a node if it is of a specified type:

- `isLiteralNode`
- `isLiteralArray`
- `isVariable`
- `isAssignment`
- `isMessage`
- `isCascade`
- `isDynamicArray`
- `isMethod`
- `isSequence`
- `isReturn`

And we can also ask a node if it is an expression node or not:

- `isValue`

2.12 Exercises

Draw the AST of the following code, indicating what kind of node is each. You can help yourself by parsing and inspecting the expressions in Pharo.

Exercises on expressions

1. Draw the AST of expression `true`.
2. Draw the AST of expression `17`.
3. Draw the AST of expression `#( 1 2 true )`.
4. Draw the AST of expression `self yourself`.
5. Draw the AST of expression `a := b := 7`.
6. Draw the AST of expression `a + #( 1 2 3 )`.
7. Draw the AST of expression `a keyword: 'message'`.
8. Draw the AST of expression `(a max: 1) min: 17`.
9. Draw the AST of expression `a max: (1 min: 17)`.

10. Draw the AST of expression `a max: 1 min: 17`.
11. Draw the AST of expression `a asParser + b asParser parse: 'some-text' , somethingElse .`
12. Draw the AST of expression `(a asParser + b asParser) parse: ('sometext' , somethingElse) .`
13. Draw the AST of expression `(a asParser + b asParser parse: 'sometext') , somethingElse .`
14. Draw the AST of expression `((a asParser + b) asParser parse: 'sometext') , somethingElse .`

Exercises on Blocks

1. Draw the AST of block `[ 1 ]`.
2. Draw the AST of block `[ :a ]`.
3. Draw the AST of block `[ :a | a ]`.
4. Draw the AST of block `[ :a | a + b ]`.
5. Draw the AST of block `[ :a | a + b . 7 ]`.
6. Draw the AST of block `[ :a | [ b ] . 7 ]`.
7. Draw the AST of block `[ :a | | temp | [ ^ b ] . ^ 7 ]`.

Exercises on Methods

1. Draw the AST of method

```
someMethod
  "this is just a comment, ignored by the parser"
```

1. Draw the AST of method

```
unaryMethod
  self
```

1. Draw the AST of method

```
unaryMethod
  ^ self
```

1. Draw the AST of method

```
+ argument
  argument > 0 ifTrue: [ ^ argument ].
  ^ self
```

1. Draw the AST of method

```
+
  strange indentation
```

1. Draw the AST of method

```
foo: arg1 bar: arg2
  | temp |
  temp := arg1 bar: arg2.
  ^ self foo: temp
```

Exercises on Invalid Code

Explain why the following code snippets (and thus their ASTs) are invalid:

1. Explain why this expression is invalid `(a + 1) := b`.
2. Explain why this expression is invalid `a + ^ 81`.
3. Explain why this expression is invalid `a + ^ 81`.

Exercises on Control Flow

As we have seen so far, there is no special syntax for control flow statements (i.e., conditionals, loops...). Instead, Pharo uses normal message-sends for them (`ifTrue:`, `ifFalse:`, `whileTrue:` ...). This makes the ASTs simpler, and also turns control flow statements into control flow expressions.

1. Give an example of an expression using a conditional and its corresponding AST
2. Give an example of an expression using a loop and its corresponding AST
3. What do control flow expressions return in Pharo?

2.13 Conclusion

In this chapter we have studied AST, short for abstract syntax trees, an object-oriented representation of the syntactic structure of programs. We have also seen an implementation of them: the RB ASTs. RB provides a parser for Pharo methods and expressions that transforms a string into a tree representing the program. We have seen how we can manipulate those ASTs. Any other nodes follow a similar principal. You should have now the basis to understand the concept of ASTs and we can move on to the next chapter.

CHAPTER 3

Manipulating ASTs with the Visitor Pattern

In the previous Chapter we have seen how to create and manipulate AST nodes. The `RBParser` class implements a parser of expressions and methods that returns AST nodes for the text given as argument. With the AST manipulation methods we have seen before, we can already write queries on an AST. For example, counting the number of message-sends in an AST is as simple as the following loop.

```
count := 0.
aNode nodesDo: [ :n |
  n isMessage
    ifTrue: [ count := count + 1 ] ].
count
```

However, more complex manipulations do require more than an iteration and a conditional. When a different operation is required for each kind of node in the AST, potentially with special cases depending on how nodes are composed, one object-oriented alternative is to implement it using the Visitor design pattern.

In this section we start reviewing the visitor pattern, and we then apply it for a single task: searching a string inside the tree.

3.1 The Visitor Pattern

The Visitor pattern is one of the original design patterns from Gamma et al. , . The main purpose of the Visitor pattern is to externalize an operation from a data structure. For example, let's consider a file system implemented with the composite pattern, where nodes can be files or directories. This composite forms a tree, where file nodes are leaf nodes and directory nodes are non-leaf nodes.

```
Object subclass: #FileNode
  instanceVariableNames: 'size'
  package: 'VisitorExample'
```

```
Object subclass: #DirectoryNode
  instanceVariableNames: 'children'
  package: 'VisitorExample'
```

Using this tree, we can take advantage of the Composite pattern and the polymorphism between both nodes to calculate the total size of a node implementing a polymorphic `size` method in each class.

```
FileNode >> size [
  ^ size
]

DirectoryNode >> size [
  ^ children sum: [ :each | each size ]
]
```

Now, let's consider the users of the file system library want to extend it with their own operations. If the users have access to the classes, they may extend them just by adding methods to them. However, chances are users do not have access to the library classes. One way to open the library classes is to implement the visitor **protocol**: each library class will implement a generic `acceptVisitor: aVisitor` method that will perform a re-dispatch on the argument giving information about the receiver. For example, when a `FileNode` receives the `acceptVisitor:` message, it will send the argument the message `visitFileNode:`, identifying itself as a file node.

```
FileNode >> acceptVisitor: aVisitor [
  ^ aVisitor visitFileNode: self
]

DirectoryNode >> acceptVisitor: aVisitor [
  ^ aVisitor visitDirectoryNode: self
]
```

In this way, we can re-implement a `SizeVisitor` that calculates the total size of a node in the file system as follows. When a size visitor visits a file, it asks the file for its size. When it visits a directory, it must iterate the children and sum the sizes. However, it cannot directly ask the `size` of the children, because only `FileNode` instances do understand it but directories do not. Because of this, we need to make a recursive call and re-ask the child node to accept the visitor. Then each node will again dispath on the size visitor.

```
SizeVisitor >> visitFileNode: aFileNode [
  ^ aFileNode size
]

SizeVisitor >> visitDirectoryNode: aDirectoryNode [
  ^ aDirectoryNode children sum: [ :each | each acceptVisitor: self ]
]
```

As the file system library forms trees that we can manipulate with a visitor, ASTs do so too. RBASTs are already extensible through visitors: its nodes implement `acceptVisitor:` methods on each of the nodes. This means we can introduce operations on the AST that were not foreseen by the original developers. In such cases, it is up to us to implement a visitor object with the correct `visitXXX:` methods.

3.2 Introducing AST visitors: measuring the depth of the tree

To introduce how to implement an AST visitor on RBASTs, let's implement a visitor that returns the max depth of the tree. That is, a tree with a single node has a depth of 1. A node with children has a depth of 1 + the maximum depth amongst all its children. Let's call that visitor `DepthCalculatorVisitor`.

```
Object subclass: #DepthCalculatorVisitor
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'VisitorExample'
```

Pharo's AST nodes implement already the visitor pattern. They have an `acceptVisitor:` method that will dispatch to the visitor with corresponding visit methods.

This means we can already use our visitor but we will have to define some methods else it will break on a visit.

Visiting message nodes

Let's start by calculating the depth of the expression `1+1`. This expression is made of a message node, and two literal nodes.

```
expression := RBParser parseExpression: '1+1'.
expression acceptVisitor: DepthCalculatorVisitor new.
>>> Exception! DepthCalculatorVisitor does not understand
    visitMessageNode:
```

If we execute the example above, we get a debugger because `DepthCalculatorVisitor` does not understand `visitMessageNode:`. We can then proceed to introduce that method in the debugger using the button **create** or by creating it in the browser. We can implement the visit method as follows, by iterating the children to calculate the maximum depth amongst the children, and then adding 1 to it.

```
DepthCalculatorVisitor >> visitMessageNode: aRBMessageNode [
  ^ 1 + (aRBMessageNode children
      inject: 0
      into: [ :max :node | max max: (node acceptVisitor: self) ])
]
```

Visiting literal nodes

As soon as we restart the example, it will stop again with an exception again, but this time because our visitor does not know how to visit literal nodes. We know that literal nodes have no children, so we can implement the visit method as just returning one.

```
DepthCalculatorVisitor >> visitLiteralValueNode: aRBLiteralValueNode
    [
  ^ 1
]
```

Calculating the depth of a method

A method node contains a set of statements. Statements are either expressions or return statements. The example that follows parses a method with two statements whose maximum depth is 3. The first statement, as we have seen above, has a depth of 2. The second statement, however, has depth of three, because the receiver of the + message is a message itself. The final depth of the method is then 5: 1 for the method node, 1 for the sequence node, and 3 for the statements.

```
method := RBParser parseMethod: 'method
  1+1.
  self factorial + 2'.
method acceptVisitor: DepthCalculatorVisitor new.
>>> Exception! DepthCalculatorVisitor does not understand
    visitMethodNode:
```

To calculate the above, we need to implement three other visiting methods: `visitMethodNode:`, `visitSequenceNode:` and `visitSelfNode:`. Since for the first two kind of nodes we have to iterate over all children in the same way, let's implement these similarly to our `visitMessageNode:`. Self nodes are variables, so they are leafs in our tree, and can be implemented as similarly to literals.

```
DepthCalculatorVisitor >> visitMethodNode: aRBMethodNode [
  ^ 1 + (aRBMethodNode children
      inject: 0
      into: [ :max :node | max max: (node acceptVisitor: self) ])
]
```

```
DepthCalculatorVisitor >> visitSequenceNode: aRBSequenceNode [
  ^ 1 + (aRBSequenceNode children
      inject: 0
      into: [ :max :node | max max: (node acceptVisitor: self) ])
]
```

```
DepthCalculatorVisitor >> visitSelfNode: aSelfNode [
  ^ 1
]
```

3.3 Refactoring the implementation

This simple AST visitor does not actually require different implementation for each of its nodes. We have seen above that we can differentiate the nodes between two kinds: leaf nodes that do not have children, and internal nodes that have children. A first refactoring to avoid the repeated code in our solution may extract the repeated methods into a common ones: `visitNodeWithChildren:` and `visitLeafNode:`.

```
DepthCalculatorVisitor >> visitNodeWithChildren: aNode [
  ^ 1 + (aNode children
      inject: 0
      into: [ :max :node | max max: (node acceptVisitor: self) ])
]
```

```
DepthCalculatorVisitor >> visitMessageNode: aRBMessageNode [
  ^ self visitNodeWithChildren: aRBMessageNode
]
[[[
DepthCalculatorVisitor >> visitMethodNode: aRBMethodNode [
  ^ self visitNodeWithChildren: aRBMethodNode
]
```

```
DepthCalculatorVisitor >> visitSequenceNode: aRBSequenceNode [
  ^ self visitNodeWithChildren: aRBSequenceNode
]
```

```
DepthCalculatorVisitor >> visitLeafNode: aSelfNode [
  ^ 1
]
```

```
DepthCalculatorVisitor >> visitSelfNode: aSelfNode [
  ^ self visitLeafNode: aSelfNode
]
```

```
DepthCalculatorVisitor >> visitLiteralValueNode: aRBLiteralValueNode
    [
  ^ self visitLeafNode: aRBLiteralValueNode
]
```

3.4 Second refactoring

As a second step, we can refactor further by taking into account a simple intuition: leaf nodes do never have children. This means that `aNode children` always yields an empty collection for leaf nodes, and thus the result of the following expression is alwaysa program that zero:

```
(aNode children
  inject: 0
  into: [ :max :node | max max: (node acceptVisitor: self) ])
```

In other words, we can reuse the implementation of `visitNodeWithChildren:` for both nodes with and without children, to get rid of the duplicated `1+`.

Let's then rename the method `visitNodeWithChildren:` into `visitNode:` and make all visit methods delegate to it. This will allow us also to remove the, now unused, `visitLeafNode:`.

```
DepthCalculatorVisitor >> visitNode: aNode [
  ^ 1 + (aNode children
      inject: 0
      into: [ :max :node | max max: (node acceptVisitor: self) ])
]
```

```
DepthCalculatorVisitor >> visitMessageNode: aRBMessageNode [
  ^ self visitNode: aRBMessageNode
]
```

```
DepthCalculatorVisitor >> visitMethodNode: aRBMethodNode [
  ^ self visitNode: aRBMethodNode
]
```

```
DepthCalculatorVisitor >> visitSequenceNode: aRBSequenceNode [
  ^ self visitNode: aRBSequenceNode
]
```

```
DepthCalculatorVisitor >> visitSelfNode: aSelfNode [
  ^ self visitNode: aSelfNode
]
```

```
DepthCalculatorVisitor >> visitLiteralValueNode: aRBLiteralValueNode
    [
  ^ self visitNode: aRBLiteralValueNode
]
```

3.5 Refactoring: A common Visitor superclass

If we take a look at our visitor above, we see a common structure has appeared. We have a lot of little visit methods per kind of node where we could do specific per-node treatments. For those nodes that do not do anything specific, with that node, we treat them as a more generic node with a more generic visit method. Our generic visit methods could then be moved to a common superclass named `BaseASTVisitor` defining the common structure, but making a single empty hook for the `visitNode:` method.

```
Object subclass: #BaseASTVisitor
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'VisitorExample'
```

```
BaseASTVisitor >> visitNode: aNode [
  "Do nothing by default. I'm meant to be overridden by subclasses"
]
```

```
BaseASTVisitor >> visitMessageNode: aRBMessageNode [
  ^ self visitNode: aRBMessageNode
]
```

```
BaseASTVisitor >> visitMethodNode: aRBMethodNode [
  ^ self visitNode: aRBMethodNode
]
```

```
BaseASTVisitor >> visitSequenceNode: aRBSequenceNode [
  ^ self visitNode: aRBSequenceNode
]
```

```
BaseASTVisitor >> visitSelfNode: aSelfNode [
  ^ self visitNode: aSelfNode
]
```

```
BaseASTVisitor >> visitLiteralValueNode: aRBLiteralValueNode [
  ^ self visitNode: aRBLiteralValueNode
]
```

And our `DepthCalculatorVisitor` is then redefined as a subclass of it:

```
BaseASTVisitor subclass: #DepthCalculatorVisitor
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'VisitorExample'
```

```
DepthCalculatorVisitor >> visitNode: aNode [
  ^ 1 + (aNode children
      inject: 0
      into: [ :max :node | max max: (node acceptVisitor: self) ])
]
```

A more elaborate visitor could provide many more hooks. For example, in our example above we could have differentiated `RBSelf` nodes from `RBVariableNodes`, defining the following.

```
BaseASTVisitor >> visitSelfNode: aRBSelfNode [
  ^ self visitVariableNode: aRBSelfNode
]
```

```
BaseASTVisitor >> visitVariableNode: aRBVariableNode [
  ^ self visitNode: aRBVariableNode
]
```

Fortunately for us, Pharo's ASTs already provide `RBProgramNodeVisitor` a base class for our visitors, with many hooks to override in our specific subclasses.

3.6 Searching the AST for a Token

Calculating the depth of an AST is a pretty naïve example for a visitor because we do not need special treatment per node. It is however a nice example to introduce the concepts, learn some common patterns, and it furthermore forced us to do some refactorings and understanding a complex visitor structure. Moreover, it was a good introduction for the `RBProgramNodeVisitor` class.

In this section we will implement a visitor that does require different treatment per node: a node search. Our node search will look for a node in the tree that contains a token matching a string. For the purposes of this example, we will keep it scoped to a **begins with** search, and will return all nodes it finds, in a depth-first in-order traversal. We leave as an exercise for the reader implementing variants such as fuzzy string search, traversing the AST in different order, and being able to provide a stream-like API to get only the next matching node on demand.

Let's then start to define a new visitor class `SearchVisitor`, subclass of `RBProgramNodeVisitor`. This class will have an instance variable to keep the token we are looking for. Notice that we need to keep the token as part of the state of the visitor: the visitor API implemented by Pharo's ASTs do not support additional arguments to pass around some extra state. This means that this state needs to be kept in the visitor.

```
RBProgramNodeVisitor subclass: #SearchVisitor
  instanceVariableNames: 'token'
  classVariableNames: ''
  package: 'VisitorExample'

SearchVisitor >> token: aToken [
  token := aToken
]
```

The main idea of our visitor is that it will return a collection with all matching nodes. If no matching nodes are found, an empty collection is returned.

Searching in variables nodes

Let's then start implementing the visit methods for variable nodes.`RBProgramNodeVisito` will already treat special variables as variable nodes, so a single visit method is enough for all four kind of nodes. A variable node matches the search if its name begins with the searched token.

```
SearchVisitor >> visitVariableNode: aNode [
  ^ (aNode name beginsWith: token)
      ifTrue: [ { aNode } ]
      ifFalse: [ #() ]
]
```

Searching in message nodes

Message nodes will match a search if their selector begins with the searched token. In addition, to follow the specification children of the message need to be iterated in depth first in-order. This means the receiver should be iterated first, then the message node itself, finally the arguments.

```
SearchVisitor >> visitMessageNode: aNode [
  ^ (aNode receiver acceptVisitor: self),
    ((aNode selector beginsWith: token)
      ifTrue: [ { aNode } ]
      ifFalse: [ #() ]),
    (aNode arguments gather: [ :each | each acceptVisitor: self ])
]
```

Searching in literal nodes

Literal nodes contain literal objects such as strings, but also booleans or numbers. To search in them, we need to transform such values as string and then perform the search within that string.

```
SearchVisitor >> visitLiteralNode: aNode [
  ^ (aNode value asString beginsWith: token)
      ifTrue: [ { aNode } ]
      ifFalse: [ #() ]
]
```

The rest of the nodes

The rest of the nodes do not contain strings to search in them. Instead the contain children we need to search. We can then provide a common implementation for them by simply redefining the `visitNode:` method.

```
SearchVisitor >> visitNode: aNode [
  ^ aNode children gather: [ :each | each acceptVisitor: self ]
]
```

Another design would be to store the collection holding the rest inside the visitor and to avoid the temporary copies. We let you refactor your code to implement it.

3.7 Exercises

Exercises on the Visitor Pattern

1. Implement mathematical expressions as a tree, to for example model expressions like `1 + 8 / 3`, and two operations on them using the composite pattern: (a) calculate their final value, and (b) print the tree in pre-order. For example, the result of evaluating the previous expression is 3, and printing it in pre-order yields the string `'/ + 1 8 3'`.
2. Re-implement the code above using a visitor pattern.
3. Add a new kind of node to our expressions: raised to. Implement it in both the composite and visitor implementations.
4. About the difference between a composite and a visitor. What happens to each implementation if we want to add a new operation? And what happens when we want to add a new kind of node?

Exercises on the AST Visitors

1. Implement an AST lineariser, that returns an ordered collection of all the nodes in the AST (similar to the pre-order exercise above).
2. Extend your AST lineariser to handle different linearisation orders: breadth-first, depth-first pre-order, depth-first post-order.
3. Extend the Node search exercise in the chapter to have alternative search orders. E.g., bottom-up, look not only if the strings begin with them.
4. Extend the Node search exercise in the chapter to work as a stream: asking `next` repeatedly will yield the next occurrence in the tree, or `nil` if we arrived to the end of the tree. You can use the linearisations you implemented above.

3.8 Conclusion

In this chapter we have reviewed the visitor design pattern first on a simple example, then on ASTs. The visitor design pattern allows us to extend tree-like structures with operations without modifying the original implementation. The tree-like structure, in our case the AST, needs only to implement an accept-visit protocol. RB ASTs implement such a protocol and some handy base visitor classes.

Finally, we implemented two visitors for ASTs: a depth calculator and a node searcher. The depth calculator is a visitor that does not require special manipulation per-node, but sees all nodes through a common view. The search visitor has a common case for most nodes, and then implements special search conditions for messages, literals and variables.

In the following chapters we will use the visitor pattern to implement AST interpreters: a program that specifies how to evaluate an AST. A normal evaluator interpreter yields the result of executing the AST. However, we will see that abstract interpreters will evaluate an AST in an abstract way, useful for code analysis.

CHAPTER 4

Representing Objects and Memory

4.1 Class as an object

Here is the minimal information that a class should have:

- A list of instance variables to describe the values that the instances will hold,
- A method dictionary to hold methods,
- A superclass to look up inherited methods.

This minimal state is similar to that of Pharo: the Pharo `Behavior` class has a format (compact description of instance variables), a method dictionary, and a superclass link.

In `ObjVLisp`, we have a name to identify the class. As an instance factory, the metaclass Class possesses four instance variables that describe a class:

- name, the class name,
- superclass, its superclass (we limit to single inheritance),
- i-v, the list of its instance variables, and
- methodDict, a method dictionary.

Since a class is an object, a class has the instance variable `class` inherited from `Object` that refers to its class as any object.

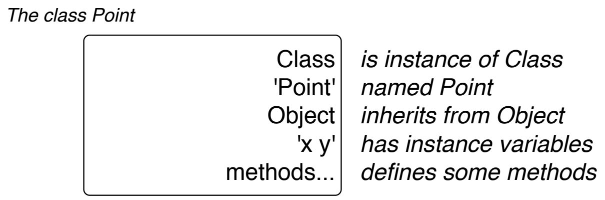


Figure 4-1 `Point` class as an object.

Example: class Point

Figure 4-1 shows the class `Point` as an instance with all the values taken in its instance variables. This graphical representation is the same to the one we used for terminal objects before, except that each variable is now annotated with its meaning. The values we show here are those declared by the programmer just before class initialization and inheritance take place.

- It is an instance of class `Class`: indeed this is a class.
- It is named `'Point'`.
- It inherits from class `Object`.
- It has two instance variables: `x` and `y`. After inheritance it will be three instance variables: `class`, `x`, and `y`.
- It has a method dictionary.

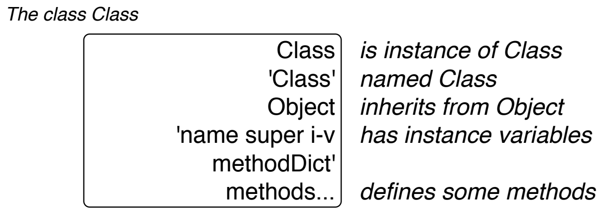


Figure 4-2 `Class` as an object.

Example: class Class

Figure 4-2 describes the class `Class` itself. Indeed it is also an object.

- It is an instance of class `Class`: indeed this is a class.
- It is named `'Class'`.
- It inherits from class `Object`

- It has four locally defined instance variables: `name`, `superclass`, `i-v`, and `methodDict`.
- It has a method dictionary.

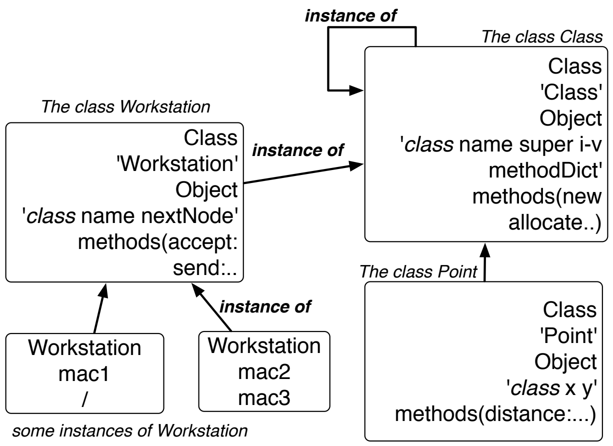


Figure 4-3 Through the prism of objects.

Everything is an object

Figure 4-3 describes a typical situation of terminal instances, class and metaclasses when viewed from an object perspective. We see three levels of instances: terminal objects (`mac1` and `mac2` which are instances of `Workstation`), class objects (`Workstation` and `Point` which are instances of `Class`) and the metaclass (`Class` which is instance of itself).

4.2 Object creation

Now we are ready to understand the creation of objects. In this model there is only one way to create instances: we should send the message `new` to the class with a specification of the instance variable values as argument.

4.3 Creation of instances of the class Point

The following examples show several point instantiations. What we see is that the model inherits from the Lisp tradition of passing arguments using keys and values, and that the order of arguments is not important.

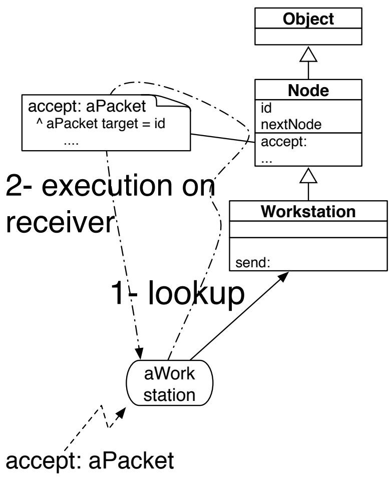


Figure 4-4 Sending a message is two-step process: method lookup and execution.

```
Point new :x 24 :y 6
>>> aPoint (24 6)
Point new :y 6 :x 24
>>> aPoint (24 6)
```

When there is no value specified, the value of an instance variable is initialized to nil. CLOS provides the notion of default values for instance variable initialization. It can be added to ObjVlisp as an exercise and does not bring conceptual difficulties.

```
Point new
>>> aPoint (nil nil)
```

When the same argument is passed multiple times, then the implementation takes the first occurrence.

```
Point new :y 10 :y 15
>>> aPoint (nil 10)
```

We should not worry too much about such details: The point is that we can pass multiple arguments with a tag to identify them.

4.4 Creation of the class Point instance of Class

Since the class `Point` is an instance of the class `Class`, to create it, we should send the message `new` to the class as follows:

```
Class new
   :name 'Point'
   :super 'Object'
   :ivs #(x y)
>>> aClass
```

What is interesting to see here is that we use exactly the same way to create an instance of the class `Point` as the class itself. Note that the possibility to have the same way to create objects or classes is also due to the fact that the arguments are specified using a list of pairs.

An implementation could have two different messages to create instances and classes. As soon as the same `new`, `allocate`, or `initialize` methods are involved, the essence of the object creation is similar and uniform.

Instance creation: Role of the metaclass

The following diagram (Figure 4-5) shows that despite what one might expect, when we create a terminal instance the metaclass `Class` is involved in the process. Indeed, we send the message `new` to the class, to resolve this message, the system will look for the method in the class of the receiver (here `Workstation`) which is the metaclass `Class`. The method `new` is found in the metaclass and applied to the receiver, the class `Workstation`. Its effect is to create an instance of the class `Workstation`.

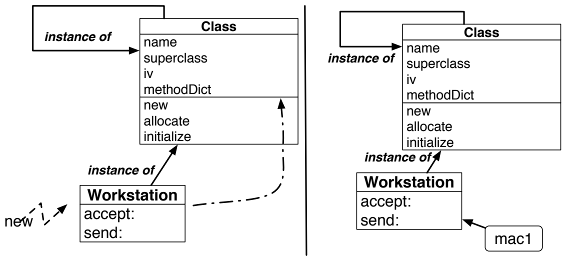


Figure 4-5 Metaclass role during instance creation: Applying plain message resolution.

The same happens when creating a class. Figure 4-6 shows the process. We send a message, now this time, to the class `Class`. The system makes no exception and to resolve the message, it looks for the method in the class of the receiver. The class of the receiver is itself, so the method `new` found in `Class` is applied to `Class` (the receiver of the message), and a new class is created.

new = allocate and initialize

Creating an instance is the composition of two actions: a memory allocation `allocate` message and an object initialisation message `initialize`.

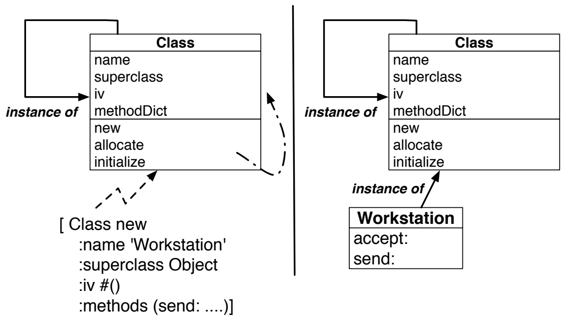


Figure 4-6 Metaclass role during class creation: Applying plain message resolution - the self instantiation link is followed.

In Pharo syntax it means:

```
aClass new: args = (aClass allocate) initialize: args
```

We should see the following:

- The message `new` is a message sent to a class. The method `new` is a class method.
- The message `allocate` is a message sent to a class. The method `allocate` is a class method.
- The message `initialize:` will be executed on any newly created instance. If it is sent to a class, a class `initialize:` method will be involved. If it is sent to a terminal object, an instance `initialize:` method will be executed (defined in `Object`).

Object allocation: the message allocate

Allocating an object means allocating enough space to the object state but there's more: instances should be marked with their class name or id. There is an invariant in this model and in general in object-oriented programming models. Every single object must have an identifier to its class, else the system will break when trying to resolve a message.

Object allocation should return a newly created instance with:

- empty instance variables (pointing to nil for example);
- an identifier to its class.

In our model, the marking of an object as instance of a class is performed by setting the value of the instance variable `class` inherited from `Object`. In Pharo this information is not recorded as an instance variable but encoded in the internal object representation in the virtual machine.

The `allocate` method is defined on the metaclass `Class`. Here are some examples of allocation.

```
Point allocate
>>> #(Point nil nil)
```

A point allocation allocates three slots: one for the class and two for x and y values.

```
Class allocate
>>>#(Class nil nil nil nil nil)
```

The allocation for an object representing a class allocates six slots: one for class and one for each of the class instance variables: name, super, iv, keywords, and methodDict.

Object initialization

Object initialization is the process of passing arguments as key/value pairs and assigning the value(s) to the corresponding instance variable(s).

This is illustrated in the following snippet. An instance of class `Point` is created and the key/value pairs (:y 6) and (:x 24) are specified. The instance is created and it received the `initialize:` message with the key/value pairs. The `initialize:` method is responsible for setting the corresponding variables in the receiver.

```
Point new :y 6  :x 24
>>> #(Point nil nil) initialize: (:y 6 :x 24)]
>>> #(Point 24 6)
```

When an object is initialized as a terminal instance, two actions are performed:

- First we should get the values specified during the creation, i.e., get that the y value is 6 and the x value is 24,
- Second we should assign the values to the corresponding instance variables of the created object.

Class initialization

During its initialization a class should perform several steps:

- First as with any initialization it should get the arguments and assign them to their corresponding instance variables. This is basically implemented by invoking the `initialize` method of `Object` via a super call, since `Object` is the superclass of `Class`.
- Second the inheritance of instance variables should be performed. Before this step the class `iv` instance variable just contains the instance

variables that are locally defined. After this step the instance variable `iv` will contain all the instance variables inherited and local. In particular this is where the `class` instance variable inherited from `Object` is added to the instance variables list of the subclass of `Object`.

- Third the class should be declared as a class pool or namespace so that programmers can access it via its name.

4.5 The Class class

Now we get a better understanding of what is the class `Class`:

- It is the initial metaclass and initial class.
- It defines the behavior of all the metaclasses.
- It defines the behavior of all the classes.

In particular, metaclasses define three messages related to instance creation.

- The `new` message, which creates an initialized instance of the class. It allocates the instance using the class message `allocate` and then initializes it by sending the message `initialize:` to this instance.
- The `allocate` message. Like message `new`, it is a class message. It allocates the structure for the newly created object.
- Finally the message `initialize:`. This message has two definitions, one on `Object` and one on `Class`.

There is a difference between the method `initialize:` executed on any instance creation and the class `initialize:` method only executed when the created instance is a class.

- The first one is a method defined on the class of the object and potentially inherited from `Object`. This `initialize:` method just extracts the values corresponding to each instance variable from the argument list and sets them in the corresponding instance variables.
- The class `initialize:` method is executed when a new instance representing a class is executed. The message `initialize:` is sent to the newly created object but its specialization for classes will be found during method lookup and it will be executed. Usually this method invokes the default ones, because the class parameter should be extracted from the argument list and set in their corresponding instance variables. But in addition, instance variable inheritance and class declaration in the class namespace is performed.

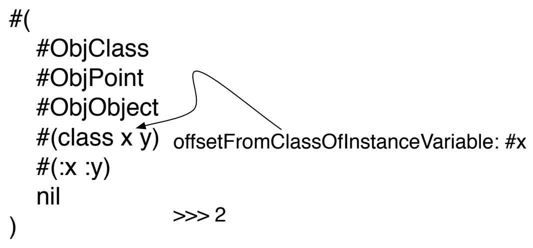


Figure 4-7 Instance variable offset asked to the class.

4.6 Accessing object instance variable values

A first simple method.

The following test illustrates the behavior of the message `offsetFromClassOfInstanceVariable:`

```
ObjTest >> testIVOffset
   "(self  selector: #testIVOffset) run"

   self assert: ((pointClass offsetFromClassOfInstanceVariable: #x)
    = 2).
   self assert: ((pointClass offsetFromClassOfInstanceVariable:
    #lulu) = 0)
```

Your job.

In the protocol `'iv management'` define a method called `offsetFromClassOfInstanceVariable: aSymbol` that returns the offset of the instance variable represented by the symbol given in the parameter. It returns `0` if the variable is not defined. Look at the tests `#testIVOffset` of the class `ObjTest`.

Hints: Use the Pharo method `indexOf:`. Pay attention that such a primitive is applied to an objClass as shown in the test.

Make sure that you execute the test method: `testIVOffset`

A second simple method.

The following test illustrates the expected behavior

```
ObjTest >> testIVOffsetAndValue
   "(self  selector: #testIVOffsetAndValue) run"
```

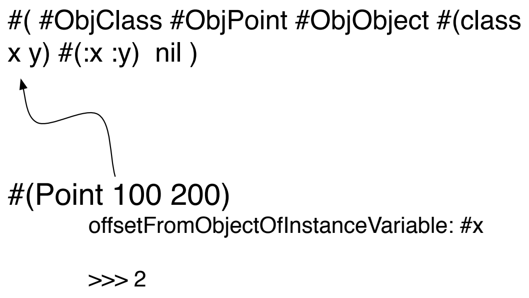

#(#ObjClass #ObjPoint #ObjObject #(class x y) #(:x :y) nil)
#(Point 100 200)
offsetFromObjectOfInstanceVariable: #x
>>> 2

Figure 4-8 Instance variable offset asked to the instance itself.

```
self assert: ((aPoint offsetFromObjectOfInstanceVariable: #x) =
  2).
self assert: ((aPoint valueOfInstanceVariable: #x) = 10)
```

Your job.

Using the previous method, define in the protocol `'iv management'`:

1. the method `offsetFromObjectOfInstanceVariable: aSymbol` that returns the offset of the instance variable. Note that this time the method is applied to an objInstance presenting an instance and not a class (as shown in Figure 4-8).
2. the method `valueOfInstanceVariable: aSymbol` that returns the value of this instance variable in the given object as shown in the test below.

Note that for the method `offsetFromObjectOfInstanceVariable:` you can check that the instance variable exists in the class of the object and else raise an error using the Pharo method `error:`.

Make sure that you execute the test method: `testIVOffsetAndValue` and it passes.

4.7 Object allocation and initialization

The creation of an object is the composition of two elementary operations: its *allocation* and its *initialization*. We now define the primitives that allow us to allocate and initialize an object. Remember that:

- allocation is a class method that returns a nearly empty structure, nearly empty because the instance represented by the structure should at least know its class, and

- initialization is an instance method that given a newly allocated instance and a list of initialization arguments fill the instance.

Instance allocation

As shown in the class `ObjTest`, if the class `ObjPoint` has two instance variables: `ObjPoint allocateAnInstance` returns `#(#ObjPoint nil nil)`.

```
ObjTest >> testAllocate
   "(self  selector: #testAllocate) run"
   | newInstance |
   newInstance := pointClass allocateAnInstance.
   self assert: (newInstance at: 1) = #ObjPoint.
   self assert: (newInstance size) = 3.
   self assert: (newInstance at: 2) isNil.
   self assert: (newInstance at: 3) isNil.
   self assert: (newInstance objClass = pointClass)
```

Your job.

In the protocol `'instance allocation'` implement the primitive called `allocateAnInstance` that sent to an *objClass* returns a new instance whose instance variable values are nil and whose objClassId represents the objClass.

Make sure that you execute the test method: `testAllocate`

CHAPTER 5

Implementing an Evaluator

An evaluator is a kind of interpreter that executes a program. For example a Pharo evaluator is an interpreter that takes as input a Pharo program and executes each one of its statements, finally returning the result of the execution. In this chapter and the following ones we will implement a Pharo evaluator as an AST interpreter, using the Visitor pattern we have seen before, meaning that the input of our evaluator will be AST nodes of a program to evaluate.

For presentation purposes, we will develop the evaluator in several stages, each in a different chapter. First, in this chapter, we will show how to implement a structural evaluator, i.e., an evaluator that reads and writes the structures of objects, starting the presentation from constant values. Later chapters will incrementally add support for other language features that deserve a chapter for themselves such as messages and blocks.

This chapter is presented in a somehow-relaxed TDD (test driven development) style. For each new feature we first define the scenario we want to cover. Since we are developing an evaluator, each scenario will be some code to execute and an expected result. We then define a test for the scenario and we make it pass. Before going to the next scenario, we do some refactorings to enhance the quality of our code.

During this process we will define and refine an AST visitor. Note that we will write the visitor from scratch but we will reuse the node of the Pharo AST and their functionalities.

5.1 Setting Up the Stage

To start writing our Pharo evaluator in TDD style, we will start by creating out test class `CHInterpreterTest`. Our class names are prefixed with CH because we named the package of the interpreter Champollion.

```
TestCase subclass: #CHInterpreterTest
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Champollion-Tests'
```

This class will, until a change is imperative, host all our test methods.

Preparing the Scenarios

Our scenarios, made out of classes and methods to be interpreted, need to be written somewhere too. We could store those scenarios in strings in our test class, that we will then need to parse and give to our interpreter as input. However, for simplicity, and because in this book we do not want to center ourselves in parsing issues, we will write our scenarios as normal Pharo code, in normal Pharo classes. This solution is simple enough and versatile to support more complex situations in the future.

We will host our first scenarios as methods in a new class named `CHInterpretable`.

```
Object subclass: #CHInterpretable
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Champollion-Test'
```

5.2 Evaluating Literals: Integers

Testing our evaluator requires that we test and assert some observable behavior. In Pharo there are two main observable behaviors: either side-effects through assignments or results of methods through return statements. We have chosen to use return statements, to introduce variables and assignments later. To start, our first scenario is a method returning an integer, as in the code below:

```
CHInterpretable >> returnInteger [
  ^ 5
]
```

Executing such a method should return an integer with value `5`.

Writing a Red Test

Our first test implements what our scenario defined above: executing our method should return 5. This first test specifies not only part of the behaviour of our interpreter, but also helps us in defining the part of its API: we want our interpreter to be able to start executing from some method's AST. Below we define a first test for it: `testReturnInteger`.

```
CHInterpreterTest >> testReturnInteger [
  | ast result |
  ast := (CHInterpretable >> #returnInteger) parseTree.
  result := self interpreter execute: ast.
  self assert: result equals: 5
]
```

This first test is worth one comment: since our evaluator is an AST interpreter, it requires an AST as input. In other words, we need to get the AST of the `returnInteger` method. Instead of invoking the parser to get an AST from source code, we will use Pharo's reflective API to get the AST of an already existing method.

5.3 Making the test pass: a First Literal Evaluator

Executing our first test fails first because our test does not understand `interpreter`, meaning we need to implement a method for it in our test class. We implement it as a factory method in our test class, returning a new instance of `CHInterpreter`, and we define the class `CHInterpreter` as follows.

```
CHInterpreterTest >> interpreter [
  ^ CHInterpreter new
]
```

```
Object subclass: #CHInterpreter
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Champollion-Core'
```

The class `CHInterpreter` is the main entry point for our evaluator, and it will implement a visitor pattern over the Pharo method ASTs. Note that it does not inherit from the default Pharo AST Visitor. The Pharo AST visitor already implements generic versions of the `visitXXX:` methods that will do nothing instead of failing. Not inheriting from it allows us to make it clear when something is not yet implemented: we will get problems such as does not understand exceptions that we will be able to implement them step by step in the debugger. We nevertheless follow the same API as the default AST visitor and we use the nodes' `accept:` visiting methods.

At this point, re-executing the test fails with a new error: our `CHInterpreter` instance does not understand the message `execute:`. We implement `execute:` to call the visitor main entry point, i.e., the method `visitNode:`.

```
CHInterpreter >> execute: anAST [
  ^ self visitNode: anAST
]
```

```
CHInterpreter >> visitNode: aNode [
  ^ aNode acceptVisitor: self
]
```

Since we evaluate a method AST, when we reexecute the test, the execution halts because of the missing `visitMethodNode:`. A first implementation for this method simply continues the visit on the body of the method.

```
CHInterpreter >> visitMethodNode: aMethodNode [
  ^ self visitNode: aMethodNode body
]
```

Execution then arrives to a missing `visitSequenceNode:`. Indeed, the body of a method is a sequence node containing a list of temporary variable definitions and a list of statements. Since our scenario has only a single statement with no temporary variables, a first version of `visitSequenceNode:` ignores temporary declarations and handles all the statements paying attention that the last statement value should be returned. So we visit all the statements except the last one, and we then visit the last one and return its result.

```
CHInterpreter >> visitSequenceNode: aSequenceNode [
  "Visit all but the last statement without caring about the result"

  aSequenceNode statements allButLast
    do: [ :each | self visitNode: each ].
  ^ self visitNode: aSequenceNode statements last
]
```

Then the visitor visits the return node, for which we define the `visitReturnNode:` method. This method simply visits the contents of the return node (invoking recursively the visitor) and returns the obtained value. At the point, the value is not yet covered by the visitor.

```
CHInterpreter >> visitReturnNode: aReturnNode [
  ^ self visitNode: aReturnNode value
]
```

Finally, the contents of the return node, the integer `5` is represented as a literal value node. To handle this node, we define the method `visitLiteralValueNode:`. The implementation just returns the value of the node, which is the integer we were looking for.

```
CHInterpreter >> visitLiteralValueNode: aLiteralValueNode [
  ^ aLiteralValueNode value
]
```

Our first test is now green and we are ready to continue our journey.

5.4 Evaluating Literals: Floats

For completeness, let's implement support for literal floats. Since we already have integer constants working, let's consider next a method returning a float literal. We can see such scenario in the code below:

```
CHInterpretable >> returnFloat [
  ^ 3.14
]
```

Executing such method should return `3.14`.

Writing a Test

Testing this case is straight forward, we should test that evaluating our method should return 3.14. We already defined that our interpreter understands the `execute:` message, so this test can follow the implementation of our previous test.

```
CHInterpreterTest >> testReturnFloat [
  | ast result |
  ast := (CHInterpretable >> #returnFloat) parseTree.
  result := self interpreter execute: ast.
  self assert: result equals: 3.14
]
```

Two discussions come from writing this test. First, this test is already green, because the case of floating point constants and integer constants exercise the same code, so nothing is to be done on this side. Second, some would argue that this test is somehow repeating code from the previous scenario: we will take care of this during our refactoring step.

5.5 Refactor: Improving the Test Infrastructure

Since we will write many tests with similar structure during this book, it comes handy to share some logic between them. The two tests we wrote so far show a good candidate of logic to share as repeated code we can extract.

The method `executeSelector:` extracts some common logic that will make our tests easier to read and understand: it obtains the AST of a method from its selector, evaluates it, and returns the value of the execution.

```
CHInterpreterTest >> executeSelector: aSymbol [
  | ast |
  ast := (CHInterpretable >> aSymbol) parseTree.
  ^ self interpreter execute: ast
]
```

And we can now proceed to rewrite our first two tests as follows:

```
CHInterpreterTest >> testReturnInteger [
  self
    assert: (self executeSelector: #returnInteger)
    equals: 5
]
```

```
CHInterpreterTest >> testReturnFloat [
  self
    assert: (self executeSelector: #returnFloat)
    equals: 3.14
]
```

We are ready to efficiently write tests for the other constants.

5.6 Evaluating booleans

Boolean literals are the objects `false` and `true`, typically used for conditionals and control flow statements. In the previous sections we implemented support for numbers, now we introduce support for returning boolean values as follows:

```
CHInterpretable >> returnBoolean [
  ^ false
]
```

Evaluating such a method should return `false`. We define a test for our boolean scenario. Note that here we do not use `deny:`, because we want to make the result explicit for the reader of the test.

```
CHInterpreterTest >> testReturnBoolean [
  "Do not use deny: to make explicit that we expect the value false"
  self
    assert: (self executeSelector: #returnBoolean)
    equals: false
]
```

If everything went ok, this test will be automatically green, without the need for implementing anything. This is because booleans are represented in the AST with literal value nodes, which we have already implemented.

5.7 Evaluating Literals: Arrays

Now that we support simple literals such as booleans and numbers, let's introduce literal arrays. Literal arrays are arrays that are defined inline in methods with all their elements being other literals. Literals refer to the fact that such objects are created during the parsing of the method code and now by sending a message. For this scenario, let’s define two different test scenarios: an empty literal array and a literal array that has elements.

```
CHInterpretable >> returnEmptyLiteralArray [
  ^ #()
]

CHInterpretable >> returnRecursiveLiteralArray [
  ^ #(true 1 #('ahah'))
]
```

These two methods should return the respective arrays.

Writing a Red Test

Writing tests to cover these two scenarios is again straight forward:

```
CHInterpreterTest >> testReturnEmptyLiteralArray [
  self
    assert: (self executeSelector: #returnEmptyLiteralArray)
    equals: #()
]

CHInterpreterTest >> testReturnRecursiveLiteralArray [
  self
    assert: (self executeSelector: #returnRecursiveLiteralArray)
    equals: #(true 1 #('ahah'))
]
```

Making the test pass: visiting literal array nodes

We have to implement the method `visitLiteralArrayNode:` to visit literal arrays. Literal arrays contain an array of literal nodes, representing the elements inside the literal array. To make our tests pass, we need to evaluate a literal array node to an array where each element is the value of its corresponding literal node. Moreover, literal arrays are recursive structures: an array can contain other arrays. In other words, we should handle the visit of literal arrays recursively. Here we return the values returned by the interpretation of the elements.

```
CHInterpreter >> visitLiteralArrayNode: aLiteralArrayNode [
  ^ aLiteralArrayNode contents
      collect: [ :literalNode | self visitNode: literalNode ]
      as: Array
]
```

This makes our tests pass, and so far there is nothing else to refactor or clean. Up until now we did not consider any form of variable and we should handle them.

5.8 Conclusion

In this chapter we have used the visitor pattern over AST nodes to implement a first version of a structural evaluator. This evaluator covers the basic literals: integers, floats, booleans and arrays. Although we have not talked about it explicitly, we also implemented a first version of the visit of statements and return nodes.

We leave for the reader the exercise of extending this prototype with support for dynamic arrays (e.g., `{ self expression. 1+1 }`).

CHAPTER 6

Variables and Name Resolution

In the previous Chapter we wrote a first version of an evaluator managing literals: integers, floats, arrays and booleans. This chapter introduces in our evaluator the capability of evaluating variables. Variables are, using a very general definition, storage location associated with a name. For example, a global variable in Pharo is stored in a global dictionary of key-value associations, where the association's key is the name of the variable and its value is the value of the variable respectively. Instance variables, on their side, are storage locations inside objects. Instance variables in Pharo have each a location in the object, specified by an index, and that index corresponds to the index of the variable's name in its class' instance variable list.

In this chapter we will study how variables are resolved. Since basically variable nodes contain only their name, this involves a "name resolution" step: finding where the variable is located, and access it accordingly following the lexical scoping (or static scoping) rules. Name resolution using lexical scoping discovers variables using the nesting hierarchy of our code, where each nesting level represents a scope. For example, classes define a scope with variables visible only by their instances; the global scope defines variables that are visible everywhere in the program, and classes are nested within the global scope.

In this chapter we will implement name resolution by modelling the program scopes. Scope objects will act as adapters for the real storage locations of variables, giving us a polymorphic way to read and write variables. Moreover, scopes will be organized in a chain of scopes, that will model the lexical organization of the program.

6.1 Starting up with `self` and `super`

To start working with variables, we begin with a method returning `self` as the one below. There is now a big difference between this scenario and the previous ones about literals. Previous scenarios did always evaluate to the same value regardless how or when they were executed. In this case, however, evaluating this method will return a different object if the receiver of the message changes.

```
CHInterpretable >> returnSelf [
  ^ self
]
```

Writing a Red Test

To properly test how `self` behaves we need to specify a receiver object, and then assert that the returned object is that same object with an identity check. For this purpose we use as receiver a new object instance of `Object`, which implement equality as identity, guaranteeing that the object will be not only equals, but also the same. We use an explicit identity check in our test to convey our intention.

```
CHInterpreterTest >> testReturnSelf [
  | receiver |
  receiver := Object new.
  "Convey our intention of checking identity by using an explicit
    identity check"
  self assert: (self
      executeSelector: #returnSelf
      withReceiver: receiver)
          == receiver
]
```

Introducing Support to Specify a Receiver

In this section we implement a first version of an execution context to introduce receiver objects. This first support will get us up and running to implement initial variable support, but it will run short to properly implement message sends. This set up will be extended in later chapters to support to message sends and block temporaries.

This initial support requires that the evaluator gets access to the object that is the receiver in the current method execution. For now we add an instance variable in our evaluator called `receiver`.

```
Object subclass: #CHInterpreter
  instanceVariableNames: 'receiver'
  classVariableNames: ''
  package: 'Champollion-Core'
```

We define an accessor so that we can easily adapt when we will introduce a better way representation later. This instance variable is hidden behind this accessor, allowing us to change later on the implementation without breaking users.

```
CHInterpreter >> receiver [
  ^ receiver
]
```

Then we replace the existing implementation of `execute:` in `CHInterpreter` by `execute:withReceiver:` which allows us to specify the receiver.

```
CHInterpreter >> execute: anAST withReceiver: anObject [
  receiver := anObject.
  ^ self visitNode: anAST
]
```

And finally, since that change break all our tests, we will create a new helper method in our tests `executeSelector:withReceiver:` and then redefine the method `executeSelector:` to use this new method with a default receiver.

```
CHInterpreterTest >> executeSelector: aSymbol [
  ^ self executeSelector: aSymbol withReceiver: nil
]

CHInterpreterTest >> executeSelector: aSymbol withReceiver:
    aReceiver [
  | ast |
  ast := (CHInterpretable >> aSymbol) parseTree.
  ^ self interpreter execute: ast withReceiver: aReceiver
]
```

Now we have refactored our code, introduced the possibility to specify a message receiver, and still keep all our previous tests passing. Our new test is still red, but we are now ready to make it pass.

Making the test pass: visiting self nodes

The interpretation of `self` is done in the method `visitSelfNode:`. The Pharo parser resolves `self` and `super` nodes statically and creates special nodes for them, avoiding the need for name resolution. Implementing it is simple, it just returns the value of the receiver stored in the interpreter. Note that this method does not access the `receiver` variable directly, but uses instead the accessor, leaving us the possibility of redefining that method without breaking the visit.

```
CHInterpreter >> visitSelfNode: aNode [
  ^ self receiver
]
```

Introducing super

Following the same logic as for `self`, we improve our evaluator to support `super`. We start by defining a method using `super` and its companion test.

```
CHInterpretable >> returnSuper [
  ^ super
]
```

```
CHInterpreterTest >> testReturnSuper [
  | receiver |
  receiver := Object new.
  "Convey our intention of checking identity by using an explicit
    identity check"
  self assert: (self
      executeSelector: #returnSuper
      withReceiver: receiver)
          == receiver
]
```

What the interpretation of `super` shows is that this variable is also the receiver of the message. Contrary to a common and wrong belief, `super` is not the superclass or an instance of the superclass. It simply is the receiver:

```
CHInterpreter >> visitSuperNode: aNode [
  ^ self receiver
]
```

6.2 Introducing Lexical Scopes

Variables in Pharo are of different kinds: instance variables represent variable stored in the evaluated message receiver, temporaries and arguments are variables visible only within a particular method evaluation, shared variables and global variables are variables that are in the global scope and independent of the receiver.

All these ”contexts” definining variables are also named scopes, because they define the reach of variable definitions. At any point during program execution, we can organize scopes in a hierarchy of scopes following a parent relationship. When a method evaluates, the current scope is the method scope with all the arguments and temporary variables defined in the method. The parent of a method scope is the instance scope that defines all the variables for an instance. The parent of the instance scope is the global scope that defines all the global variables.

Notice that in this scope organization we are not explicitly talking about classes. The instance scope takes care of that resolving all instance variables in the hierarchy of the receiver object. A class scope could be added later to resolve class variables, and class pools.

The basic structure of our lexical scope implementation introduces the method `scopeDefining:` in our interpreter, that returns the scope defining the given name. The method `scopeDefining:` forwards the search to the current scope, obtained through `currentScope`. Finally, we will extend our interpreter with `visitVariableNode:` and `visitAssignmentNode:` to handle variable reads and writes. Reads (and writes) obtain the scope for the read (written) variable and delegate the read (write) to it. This means our scope objects need to define methods `scopeDefining:` and `read:` and `write:withValue:`.

```
CHInterpreter >> scopeDefining: aName [
  ^ self currentScope scopeDefining: aName
]

CHInterpreter >> currentScope [
  ^ ...
]

CHInterpreter >> visitVariableNode: aVariableNode [
  ^ (self scopeDefining: aVariableNode name) read: aVariableNode name
]

CHInterpreter >> visitAssignmentNode: anAssignmentNode [
  | value |
  value := self visitNode: anAssignmentNode value.
  (self scopeDefining: anAssignmentNode variable name)
    write: anAssignmentNode variable name
    withValue: value.
  ^ value
]
```

6.3 Evaluating Variables: Instance Variable Reads

The first step is to support instance variable reads. Since such variables are evaluated in the context of the receiver object, we need a receiver object that has instance variables. We already had such a class: `CHInterpretable`. We then add an instance variable and a getter and setter for it to be able to control the values in that instance variable.

```
Object subclass: #CHInterpretable
  instanceVariableNames: 'x'
  classVariableNames: ''
  package: 'Champollion-Tests'

CHInterpretable >> returnX [
  ^ x
]
```

```
CHInterpretable >> x: anInteger [
  x := anInteger
]
```

To test the correct evaluation of the instance variable read, we check that the getter returns the value in the instance variable, which we can previously set.

```
CHInterpreterTest >> testReturnInstanceVariableRead [
  | receiver |
  receiver := CHInterpretable new.
  receiver x: 100.
  self
    assert: (self executeSelector: #returnX withReceiver: receiver)
    equals: 100
]
```

Our test is failing so we are ready to make it work.

Creating an Instance Scope

To make our tests go green, we need to implement instance scopes and the `CHInterpreter>>currentScope` method. We will model instance scopes with a new class. This class will know the receiver object, extract the list of instance variables from it, and know how to read and write from/to it.

```
Object subclass: #CHInstanceScope
  instanceVariableNames: 'receiver'
  classVariableNames: ''
  package: 'Champollion-Core'

CHInstanceScope >> receiver: anObject [
    receiver := anObject
]

CHInstanceScope >> scopeDefining: aString [
  (self definedVariables includes: aString)
    ifTrue: [ ^ self ].

  ^ self parentScope scopeDefining: aString
]

CHInstanceScope >> definedVariables [
  ^ receiver class allInstVarNames
]

CHInstanceScope >> read: aString [
  ^ receiver instVarNamed: aString
]
```

We then can define our `currentScope` method as follows.

```
CHInterpreter >> currentScope [
    ^ CHInstanceScope new
      receiver: self receiver;
      yourself
]
```

All our tests should now pass!

6.4 Refactor: Improving our Tests **setUp**

Let's simplify how receivers are managed in tests. Instead of having to explicitly create and manage a receiver object, we add the receiver as an instance variable to the `CHInterpreterTest`.

```
TestCase subclass: #CHInterpreterTest
  instanceVariableNames: 'interpreter receiver'
  classVariableNames: ''
  package: 'Champollion-Tests'
```

We can then redefine the `setUp` method to create a new instance of `CHInterpretable` and assign it to the variable `receiver`.

```
CHInterpreterTest >> setUp [
  super setUp.
  receiver := CHInterpretable new
]
```

And now we use this new instance variable in `executeSelector:` as the default receiver instead of `nil`.

```
CHInterpreterTest >> executeSelector: aSymbol [
  ^ self executeSelector: aSymbol withReceiver: receiver
]
```

And finally we rewrite all our test methods using an explicit receiver:

```
CHInterpreterTest >> testReturnSelf [
  "Convey our intention of checking identity by using an explicit
    identity check"
  self assert: (self
    executeSelector: #returnSelf
    withReceiver: receiver)
        == receiver
]

CHInterpreterTest >> testReturnSuper [
  "Convey our intention of checking identity by using an explicit
    identity check"
  self assert: (self
    executeSelector: #returnSelf
    withReceiver: receiver)
        == receiver
]

CHInterpreterTest >> testReturnInstanceVariableRead [
  receiver x: 100.
  self
    assert: (self executeSelector: #returnX)
    equals: 100
]
```

6.5 Instance Variable Writes

Now that we have support for instance variable reads, we keep on going with instance variable writes. In our scenario, a method writes a literal integer into an instance variable x and then returns the value of the assignment. This scenario has two observable behaviors that we will be tested separately. First, such an assignment should be observable from the outside by reading that variable. Second, an assignment is an expression whose value is the assigned value, thus the return value should also be the assigned value.

```
CHInterpretable >> store100IntoX [
  ^ x := 100
]
```

To test this behavior, we evaluate the method above and then we validate that effectively the instance variable was mutated. To make sure the value was modified, we set an initial value to the variable before the evaluation. After the evaluation we should not keep that value. The case of the return is similar to our previous tests.

```
CHInterpreterTest >> testStore100IntoX [
  receiver x: 17.
  self executeSelector: #store100IntoX.
  self assert: receiver x equals: 100
]

CHInterpreterTest >> testAssignmentReturnsAssignedValue [
  self
    assert: (self executeSelector: #store100IntoX)
    equals: 100
]
```

And finally, we implement a getter for the x variable. This getter is used to extract the value in the tests.

```
CHInterpretable >> x [
  ^ x
]
```

To make this tests pass, let's first see in detail our method visitAssignmentNode.

```
CHInterpreter >> visitAssignmentNode: anAssignmentNode [
  | value |
  value := self visitNode: anAssignmentNode value.
  (self scopeDefining: anAssignmentNode variable name)
    write: anAssignmentNode variable name
    withValue: value.
  ^ value
]
```

Evaluating an assignment node with the form `variable := expression` requires that we evaluate the `expression` of the assignment, also called the right hand side of the assignment, and then set that value to the left-side variable. Notice that the variable we are assigning into does not need to be evaluated/visited: the evaluation of a variable is equivalent to reading its value. And in the left part of an assignment, we do not care about the value of the variable, but about the location of the variable. Our visit method then looks as follows: we recursively visit the right-side of the assignment (got through the `value` accessor), we set that value to the variable by delegating to the scope, and we finally return the stored value. To set the value to the variable, we implement `write:withValue:` in our instance scope.

```
CHInstanceScope >> write: aString withValue: anInteger [
  receiver instVarNamed: aString put: anInteger
]
```

Now the tests should pass.

6.6 Evaluating Variables: Global Reads

We finish this chapter with the reading of global variables, which cover two cases: proper global variables, and the access to classes. To better control our testing environment, we decided to not use the Pharo environment by default. Instead, the interpreter will know its global scope in an instance variable and lookup globals in it using a simple API, making it possible to use the system's global environment instead if we wanted to. We also leave outside of the scope of this chapter the writing to global variables. The case of global variable writing is similar to the instance variable assignment.

Our first testing scenario, similar to the previous ones, is as follows: we define a method `returnGlobal` reads and returns the global named `Global`. When defining such method in Pharo, the IDE will ask what `Global` is, because such a variable does not exist: select to create it as a global variable. Such a detail is only necessary to keep Pharo itself happy and it does not have any impact in our implementation. Remember that we are writing our own Pharo implementation in the interpreter, and we will decide what to do with `Global` ourselves.

```
CHInterpretable >> returnGlobal [
  ^ Global
]
```

We define a test which specifies that the interpreter' environment has a binding whose key is `#Global` and value is a new object.

```
CHInterpreterTest >> testReturnGlobal [
  | globalObject |
  globalObject := Object new.
  interpreter globalEnvironmentAt: #Global put: globalObject.
  self assert: (self executeSelector: #returnGlobal) equals:
    globalObject
]
```

We introduce a new global scope class `CHGlobalScope`, and an instance variable named `globalScope` in the class `CHInterpreter` initialized to a global scope.

```
Object subclass: #CHGlobalScope
  instanceVariableNames: 'globalsDictionary'
  classVariableNames: ''
  category: 'Champollion-Core'

CHGlobalScope >> initialize [
  super initialize.
  globalsDictionary := Dictionary new
]

CHGlobalScope >> globalsDictionary: anObject [
  globalsDictionary := anObject
]

CHGlobalScope >> at: aKey ifAbsent: aBlock [
  ^ globalsDictionary at: aKey ifAbsent: aBlock
]

Object subclass: #CHInterpreter
  instanceVariableNames: 'globalScope'
  classVariableNames: ''
  category: 'Champollion-Core'

CHInterpreter >> initialize [
  super initialize.
  globalScope := CHGlobalScope new
]

CHInterpreter >> globalEnvironment [
  ^ globalScope
]
```

We define the method `globalEnvironmentAt:put:` to easily add new globals from our test.

```
CHInterpreter >> globalEnvironmentAt: aSymbol put: anObject [
  globalScope at: aSymbol put: anObject
]

CHGlobalScope >> at: aKey put: aValue [
  globalsDictionary at: aKey put: aValue
]
```

We decided that trying to access a global that is not defined will raise an error in the interpreter and halt the interpretation. An alternative design would return a default value instead such as `nil`. A more complex design would have been to throw an exception in the interpreted program. For the moment, halting the interpreter with an error suffices for our job.

```
CHInterpreter >> visitGlobalVariableNode: aRBGlobalNode [
  ^ self globalEnvironment
      at: aRBGlobalNode name
      ifAbsent: [ self error: aRBGlobalNode name, ' not found' ]
]
```

Finally, we need to chain the global scope to the instance scope, and define a `scopeDefining:` and a `read:` method in our global scope.

```
CHInterpreter >> currentScope [
  ^ CHInstanceScope new
    receiver: self receiver;
    parentScope: globalScope;
    yourself
]

CHInstanceScope >> parentScope: anObject [
  parentScope := anObject
]

CHInstanceScope >> parentScope [
  ^ parentScope
]

CHGlobalScope >> scopeDefining: aString [
  "I'm the root scope..."
  ^ self
]

CHGlobalScope >> read: aString [
  ^ globalsDictionary at: aString
]
```

6.7 Conclusion

In this chapter we introduced support for variables and name resolution,to support instance variables and global variables. This first evaluator serves as basis for implementing the execution of the main operations in Pharo: messages. The following chapters will start by decomposing message resolution into method-lookup and apply operations, introduce the execution stack, the management of temporary variables and argument, and 'super' message-sends.

We further invite the reader to explore changing the language semantics by modifying this simple evaluator: How could we implement read-only objects? Log all variable reads and writes?

CHAPTER 7

Implementing Message Sends: Calling Infrastructure

In the previous chapter we focused on structural evaluation: reading literal objects and reading and writing values from objects and globals. However, the key abstraction in object-oriented programming and in Pharo in particular is message-sending. The work we did in the previous chapter is nevertheless important to set up the stage: we have a better taste of the visitor pattern, we started a first testing infrastructure, and eventually message-sends need to carry out some work by using literal objects or reading and writing variables.

Message-sends deserve a chapter on their own because they introduce many different concerns. On the one hand, each message-send is resolved in two steps: first the method-lookup searches in the receiver's hierarchy the method to be executed, and second that method is applied on the receiver (i.e., it is evaluated with self bound to the receiver). On the other hand, each method application needs to set up an execution context to store the receiver, arguments and temporary variables for that specific method execution. These execution contexts form the execution stack or call-stack. Sending a message pushes a new context in the call-stack, returning from a method pops a context from the call-stack. This is mechanics that we will cover in this chapter, so that in the following chapter we can implement logic and support late-binding.

7.1 Introduction to Stack Management

The way we managed the receiver so far is overly simplistic. Indeed, each time a program will send a message to another object, we should change the receiver and when a method execution ends, we should restore the previous receiver. Moreover, the same happens with method arguments and temporaries. Therefore to introduce the notion of message-send we need a stack. And each element in the stack needs to capture all the execution state require to come back to it later on when a message-send will return. Each element in the call-stack is usually named a stack frame, an activation record, or in Pharo's terminology a context. For the rest of this book we will refer to them as frames, for shortness, and to distinguish them from the reified contexts from Pharo.

A first step to introduce stack management without breaking all our previous tests is to replace the single `receiver` instance variable with a stack that will be initialized when the evaluator is created. The top of the stack will represent the current execution, and thus we will take the current receiver at each moment from the stack top. Moreover, each time we tell our interpreter to execute something we need to initialize our stack with a single frame.

As an example, let us consider Figure 7-1, which presents a call-stack with two methods. The first method in the stack (at its bottom) is method `foo`. Method `foo` calls method `bar` and thus it follows it in the stack. The current method executing is the one in the top of the stack. When a method returns, we can restore all the state of the previous method just by **popping** the top of the stack.

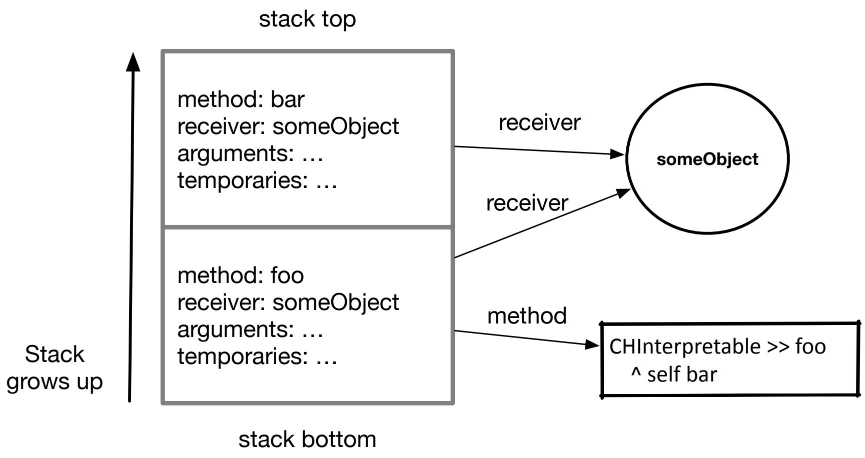


Figure 7-1 A call-stack with two frames, executing the method foo which calls bar.

Listing 7-2 Replace the receiver instance variable by a stack

```
Object subclass: #CHInterpreter
  instanceVariableNames: 'stack'
  classVariableNames: ''
  package: 'Champollion-Core'

CHInterpreter >> initialize [
  super initialize.
  stack := CTStack new.
]
```

With this new schema, we can now rewrite the access to the receiver to just access the value of `#self` of the top frame.

```
CHInterpreter >> receiver [
  ^ self topFrame receiver
]

CHInterpreter >> topFrame [
  ^ stack top
]
```

The final step is to set up a frame when the execution starts, which happened so far in our method `execute:withReceiver:`. We extend the `execute:withReceiver:` to create a new frame and define the receiver as `#self` in the top frames before start the evaluation.

```
CHInterpreter >> execute: anAST withReceiver: anObject [
  self pushNewFrame.
  self topFrame receiver: anObject.
  ^ self visitNode: anAST
]
```

The last piece in the puzzle is the method `pushNewFrame`, which creates a new frame and pushes it on the top of the stack. Since methods define a scope with their temporary variables and arguments, we represent frames using a new kind of scope: a method scope. The method scope will for now store the current receiver, and later its parent scope, and a set of key-value pairs representing the variables defined in the current method execution: the arguments and temporaries.

```
CHInterpreter >> pushNewFrame [
  | newTop |
  newTop := CHMethodScope new.
  stack push: newTop.
  ^ newTop
]

Object subclass: #CHMethodScope
```

```
  instanceVariableNames: 'receiver variables'
  classVariableNames: ''
  package: 'Champollion-Core'

CHMethodScope >> receiver: aCHInterpretable [
  receiver := aCHInterpretable
]

CHMethodScope >> receiver [
  ^ receiver
]
```

This refactor kept all the test green, and opened the path to introduce message-sends. As the reader may have observed, this stack can only grow. We will take care of popping frames from the stack later when we revisit method returns.

7.2 Evaluating a First Message Send

Let's start as usual by defining a new method exhibiting the scenario we want to work on. In this case, we want to start by extending our evaluator to correctly evaluate return values of message sends.

Our scenario method `sendMessageReturnX` does a self message-send and returns the value returned by this message send. On the one hand, we want that in our scenario the receiver of both messages is the same. On the other hand, we want that the message send is correctly evaluated to the return value of the activated method.

```
CHInterpretable >> sendMessageReturnX [
  ^ self returnX
]
```

Notice that our method `sendMessageReturnX` and the implementation of the message it sends `returnX` live in the same class. This means that in this first scenario we can concentrate on the stack management and return value of the message sends, without caring too much about the details of the method lookup algorithm. For this first version we will define a simple and incomplete yet useful method lookup algorithm. SD-Guille I do not get the following paragraph - I could not really understand and I could not fix it. In our first test we want to ensure that in a `self` message-send, the receiver of both the called and callee methods is the same. One way to do that is with a side-effect: if we write one instance variable in one method with some value `x` (say and we access that value from the other method, we should get the same value `x` (say). This will show that the object represented by `self` is the same and that we did not push for example `nil`.

```
CHInterpreterTest >> testSelfSend [
  receiver x: 100.
  self
    assert: (self executeSelector: #sendMessageReturnX)
    equals: 100
]
```

To make this test green, we need to implement the method `visitMessageNode:`. Evaluating a message node requires that we recursively evaluate the receiver node, which may be a literal node or a complex expression such as another message-send. From such an evaluation we obtain the actual receiver object. Starting from the receiver, we will lookup the method with the same selector as the message-send. In our first implementation we will just fetch the desired method's AST from the receiver's class. Finally, we can activate this method with the receiver using `execute:withReceiver:` the activation will push a new frame to the call-stack with the given receiver, evaluate the method, and eventually return with a value.

```
CHInterpreter >> visitMessageNode: aMessageNode [
  | newReceiver method |
  newReceiver := self visitNode: aMessageNode receiver.
  method := (newReceiver class compiledMethodAt: aMessageNode
    selector) ast.
  ^ self execute: method withReceiver: newReceiver
]
```

7.3 Balancing the Stack

We mentioned earlier that when the execution of a method is finished and the execution returns to its caller method, its frame should be also discarded from the stack. The current implementation clearly does not do it. Indeed, we also said that our initial implementation of the stack only grows: it is clear by reading our code that we never pop frames from the stack.

To solve this issue, let us write a test showing the problem first. The idea of this test is that upon return, the frame of the caller method should be restored and with its receiver. If we make that the caller and callee methods have different receiver instances, then this test can be expressed by calling some other method, ignore its value and then return something that depends only on the receiver. In other words, this test will fail if calling a method on some other object modifies the caller!

The following code snippet shows an scenario that fulfills these requirements: it sets an instance variable with some value, sends a message to an object other than `self` and upon its return it accesses its instance variable again before returning it. Assuming the collaborator object does not modify `self`, then the result of evaluating this message should be that 1000 is returned.

```
CHInterpretable >> setXAndMessage [
    x := 1000.
    collaborator returnX.
    ^ x
]
```

Our test `testBalancingStack` executes the message `setXAndMessage` that should return 1000.

```
CHInterpreterTest >> testBalancingStack [
  self
    assert: (self executeSelector: #setXAndMessage)
    equals: 1000
]
```

We then finish our setup by extending `CHInterpretable` to support delegating to a collaborator object. We add a `collaborator` instance variable to the class `CHInterpretable` with its companion accessors. This way we will be able to test that the correct object is set and passed around in the example.

```
Object subclass: #CHInterpretable
  instanceVariableNames: 'x collaborator'
  classVariableNames: ''
  package: 'Champollion-Core'

CHInterpretable >> collaborator [
  ^ collaborator
]

CHInterpretable >> collaborator: anObject [
  collaborator := anObject
]
```

And in the `setUp` method we pass a collaborator to our initial receiver.

```
CHInterpreterTest >> setUp [
  super setUp.
  receiver := CHInterpretable new.
  receiver collaborator: CHInterpretable new
]
```

Making the test pass

Executing this test breaks because the access to the instance variable `x` returns nil, showing the limits of our current implementation. This is due to the fact that evaluating message send `returnX` creates a new frame with the collaborator as receiver, and since that frame is not popped from of the stack, when the method returns, the access to the `x` instance variable accesses the one of the uninitialized collaborator instead of the caller object.

To solve this problem, we should pop the frame when a method activation finishes. This way the stack is balanced. This is what the new implementation of `executeMethod:withReceiver:` does.

```
CHInterpreter >> executeMethod: anAST withReceiver: anObject [
  self pushNewFrame.
  self topFrame receiver: anObject.
  result := self visitNode: anAST.
  self popFrame.
  ^ result
]

CHInterpreter >> popFrame [
  stack pop
]
```

7.4 Ensuring the receiver is correctly set: an Extra Test

Our previous tests did ensure that messages return the correct value, activate the correct methods, and that the stack grows and shrinks. However, we did not ensure yet that the receiver changes correctly on a message send, and since we do not lose any opportunity to strenghten our trust in our implementation with a new test, let's write a test for it.

The scenario, illustrated in `changeCollaboratorX` will ask the collaborator to `store100IntoX`, implemented previosly. In this scenario, we must ensure that the state of the receiver and the collaborator are indeed separate and that changing the collaborator will not affect the initial receiver's state.

```
CHInterpretable >> changeCollaboratorX [
  collaborator store100IntoX
]
```

Our test for this scenario is as follows. If we give some value to the receiver and collaborator, executing our method should change the collaborator but not the initial receiver.

```
CHInterpreterTest >>
    testInstanceVariableStoreInMethodActivationDoesNotChangeSender [
  receiver x: 200.
  collaborator x: 300.

  "changeCollaboratorX will replace collaborator's x but not the
    receiver's"
  self executeSelector: #changeCollaboratorX.

  self assert: receiver x equals: 200.
  self assert: collaborator x equals: 100
]
```

To make our test run, we will store as a convenience the collaborator object in an instance variable of the test too.

```
TestCase subclass: #CHInterpreterTest
  instanceVariableNames: 'receiver collaborator'
  classVariableNames: ''
  package: 'Champollion-Tests'

CHInterpreterTest >> setUp [
  super setUp.
  receiver := CHInterpretable new.
  collaborator := CHInterpretable new.
  receiver collaborator: collaborator
]
```

This test passes, meaning that our implementation already covered correctly this case. We are ready to continue our journey in message-sends.

7.5 Supporting Message Arguments

So far we have worked only with unary messages. Unary messages have no arguments, so the number of programs we can express with them only is rather limited. The next step towards having a full-blown interpreter is to support message arguments, which will open us the door to support binary and keyword messages. From the evaluator point of view, as well as from the AST point of view, we will not distinguish between unary, binary and keyword messages. The parser already takes care about distinguishing them and handling their precedence. Indeed, message nodes in the AST are the same for all kind of messages, they have a selector and a collection of argument nodes. Precedence is then modelled as relationships between the AST nodes.

In addition of simply passing the arguments, from an evaluator point of view, we need to care about evaluation order too. This is particularly important because Pharo is an imperative language where messages can trigger side effects. Evaluating two messages in one order may not have the same result as evaluating them in a different order. Arguments in Pharo are evaluated eagerly after evaluating the receiver expression, but before evaluating the message, from left to right. Once all expressions are evaluated, the resulting objects are send as part of the message-send.

Initial Argument Support

To implement some initial support for arguments, our first scenario is to simply send a message with an argument. For our scenario we already count with one message with an argument: the `x:` setter. We can then define a method `changeCollaboratorWithArgument` which uses it.

```
CHInterpretable >> changeCollaboratorWithArgument [
  collaborator x: 500
]
```

In the test, we verify that the method evaluation effectively modifies the collaborator object as written in `changeCollaboratorWithArgument`, and not the initial receiver object.

```
CHInterpreterTest >> testArgumentAccess [

  receiver x: 200.
  collaborator x: 300.

  self executeSelector: #changeCollaboratorWithArgument.

  self assert: receiver x equals: 200.
  self assert: collaborator x equals: 500
]
```

Since we have not implemented any support for arguments yet, this test should fail.

Implementing argument support requires two main changes:

- On the caller side, we need to evaluate the arguments in the context of the caller method and then store those values in the new frame.
- On the callee side, when an argument access is evaluated, those accesses will not re-evaluate the expressions in the caller. Instead, argument access will just read the variables pre-stored in the current frame.

Let's start with the second step, the callee side, and since all variable reads are concentrated on the scope lookup, we need to add the method scope in the scope chain, and define a `read:` method for it.

```
CHInterpreter >> execute: anAST withReceiver: anObject [
  | result |
  self pushNewFrame.

  "Set up the scope chain"
  self topFrame parentScope: (CHInstanceScope new
    receiver: anObject;
    parentScope: globalScope;
    yourself);
  yourself.

  self topFrame receiver: anObject.
  result := self visitNode: anAST.
  self popFrame.
  ^ result
]
CHInterpreter >> currentScope [
  ^ self topFrame
]

CHMethodScope >> scopeDefining: aString [
  (variables includesKey: aString)
    ifTrue: [ ^ self ].

  ^ self parentScope scopeDefining: aString
]

CHMethodScope >> read: aString [
  ^ variables at: aString
]
```

Then we need to update `visitMessageNode:` to compute the arguments by doing a recursive evaluation, and then use those values during the new method activation.

```
CHInterpreter >> visitMessageNode: aMessageNode [
  | newReceiver method args |
  args := aMessageNode arguments collect: [ :each | self visitNode:
    each ].
  newReceiver := self visitNode: aMessageNode receiver.
  method := (newReceiver class compiledMethodAt: aMessageNode
    selector) ast.
  ^ self executeMethod: method withReceiver: newReceiver
    andArguments: args
]
```

To include arguments in the method activation, let's add a new `arguments` parameter to our method `execute:withReceiver:` to get `execute:withReceiver:withArguments:`.

In addition to adding the receiver to the new frame representing the execution, we add a binding for each parameter (called unfornately arguments in Pharo AST) with their corresponding value in the argument collection. We use the message `with:do:` to iterate both the parameter list and actual arguments as pairs.

```
CHInterpreter >> execute: anAST withReceiver: anObject andArguments:
    aCollection [
  | result |
  self pushNewFrame.

  "Set up the scope chain"
  self topFrame parentScope: (CHInstanceScope new
    receiver: anObject;
    parentScope: globalScope;
    yourself);
  yourself.

  self topFrame receiver: anObject.
  anAST arguments
    with: aCollection
    do: [ :arg :value | self topFrame at: arg name put: value ].
  result := self visitNode: anAST.
  self popFrame.
  ^ result
]
```

Instead of just removing the old `executeMethod:withReceiver:` method, we redefine it calling the new one with a default empty collection of arguments. This method was used by our tests and is part of our public API, so keeping it will avoid migrating extra code and an empty collection of arguments seems like a sensible and practical default value.

```
CHInterpreter >> executeMethod: anAST withReceiver: anObject [
  ^ self
    executeMethod: anAST
    withReceiver: anObject
    andArguments: #()
]
```

Our tests should all pass now.

7.6 Refactoring the Terrain

Let's now refactor a bit the existing code to clean it up and expose some existing but hidden functionality. Let us extract the code that accesses `self` and the frame parameters into two other methods that make more intention revealing that we are accessing values in the current frame.

```
CHInterpreter >> tempAt: aSymbol put: anInteger [
  self topFrame at: aSymbol put: anInteger
]
```

```
CHInterpreter >> execute: anAST withReceiver: anObject andArguments:
     aCollection [
  | result |
  self pushNewFrame.

  "Set up the scope chain"
  self topFrame parentScope: (CHInstanceScope new
    receiver: anObject;
    parentScope: globalScope;
    yourself);
  yourself.

  self topFrame receiver: anObject.
  anAST arguments
    with: aCollection
    do: [ :arg :value | self tempAt: arg name put: value ].
  result := self visitNode: anAST.
  self popFrame.
  ^ result
]
```

7.7 Handling Temporaries

Temporary variables, or local variables, are variables that live within the scope of a method's **execution**. Memory for such variables is allocated when a method is activated, and released when the method returns. Because of this property, temporary variables are also called automatic variables in languages like C.

The usual way to implement such temporary variables is to allocate them in the method execution's frame. This way, when the method returns, the frame is popped and all the values allocated in temporaries are discarded and can be reclaimed. In other words, we will manage temporaries the same way as we manage arguments.

Our first scenario introducing temporaries will verify the default value of temporaries. Indeed when temporaries are allocated in Pharo, the execution engine (in this case our evaluator) should make sure these variables are correctly initialized to a default value, in this case `nil`.

Notice that temporaries cannot be observed from outside the execution of a method unless we halt the evaluation of a method in the middle of the evaluation. Since our testing approach is more like a black-box approach, we need to make our scenarios visible from the outside somehow. Because of these reasons, our tests will rely on returns again, as we did before with literal objects.

```
CHInterpretable >> returnUnassignedTemp [
  | temp |
  ^ temp
]
```

The companion test verifies that the value of a uninitialized temporary is `nil`.

```
CHInterpreterTest >> testUnassignedTempHasNilValue [
  self
    assert: (self executeSelector: #returnUnassignedTemp)
    equals: nil
]
```

The current subset of Pharo that we interpret does not contain blocks and their local/temporary variables (We will implement blocks and more complex lexical scopes in a subsequent chapter). Therefore the temporary variable management we need to implement so far is rather simple.

To make our test pass, we modify the `execute:withReceiver:andArguments:` method to define the temporaries needed with `nil` as value.

```
CHInterpreter >> executeMethod: anAST withReceiver: anObject
    andArguments: aCollection [
  | result thisFrame |
  self pushNewFrame.
  self topFrame parentScope: (CHInstanceScope new
    receiver: anObject;
    parentScope: globalScope;
    yourself);
  yourself.

  self topFrame receiver: anObject.
  anAST arguments with: aCollection do: [ :arg :value | self tempAt:
    arg name put: value ].
  anAST temporaryNames do: [ :tempName | self tempAt: tempName name
    put: nil ].
  result := self visitNode: anAST body.
  self popFrame.
  ^ result
]
```

The tests should be pass.

7.8 Implementing Temporary Variable Writes

Finally we test that writes to temporary variables are working too. We define our scenario method `writeTemporaryVariable`, which defines a temporary variable, assigns to it and returns it. An optimizing compiler for this code would be smart enough to do constant propagation of the literal integer and then realize that the temporary is dead code and remove it, leaving us with a method body looking like `^ 100`. However, since the parser does not do such optimizations by itself, we are sure that the AST we get contains both the temporary definition, the assignment, and the temporary return.

```
CHInterpretable >> writeTemporaryVariable [
  | temp |
  temp := 100.
  ^ temp
]
```

Its companion test checks that evaluating this method does effectively return 100, meaning that the temporary variable write succeeded, and that `temp` means the same variable in the assignment and in the access.

```
CHInterpreterTest >> testWriteTemporaryVariable [
  self
    assert: (self executeSelector: #writeTemporaryVariable)
    equals: 100
]
```

Since temporary variable name resolution is already managed by our method scopes, we just need to implement `write:withValue:` in it to make all our tests pass.

```
  CHMethodScope >> write: aString withValue: aValue [
    variables at: aString put: aValue
]
```

7.9 Evaluation Order

The last thing we need to make sure is that arguments are evaluated in the correct order. The evaluation order in Pharo goes as follows: before evaluating a message, the receiver and all arguments are evaluated. The receiver is evaluated before the arguments. Arguments are evaluated in left-to-right order.

Testing the evaluation order in a black-box fashion as we were doing so far is rather challenging with our current evaluator. Indeed, our evaluator does not yet handle arithmetics, allocations nor other kind of primitive, so we are not able to easily count! A simple approach to test is to make a counter out of Peano Axiomshttps://en.wikipedia.org/wiki/Peano_axioms. The main idea is to implement numbers as nested sets, where the empty set is the zero, the set that contains the zero is one, the set that contains a one is a two, and so on. The only support we need for this is to extend our literal support for dynamic array literals. The code illustrating the idea follows.

```
CHInterpretable >> initialize [
  super initialize.
  current := { "empty" }.
]
```

```
CHInterpretable >> next [
  | result |
  "Implement a stream as an increment in terms of Peano axioms.
  See https://en.wikipedia.org/wiki/Peano_axioms"
  result := current.
  "We increment the counter"
  current := { current }.
  "We return the previous result"
  ^ result
]
```

```
CHInterpreterTests >> peanoToInt: aPeanoNumber [
  "Helper method to transform a peano number to a normal Pharo
    integer"
  ^ aPeanoNumber
```

```
    ifEmpty: [ 0 ]
    ifNonEmpty: [ 1 + (self peanoToInt: aPeanoNumber first) ]
]
```

Using this support, we can express our evaluation order scenario and test as follows. We will add a new instance variable to `CHInterpretable` to store its evaluation order. Then, we are going to send a message with many arguments, evaluating for each argument `self next`. The message receiving the arguments will receive as argument three generated peano values, that we will return as dynamic literal array. If evaluation order is right, the evaluation order of the receiver should be 0, the evaluation of the first argument should be 1, and so on.

```
Object subclass: #CHInterpretable
  instanceVariableNames: 'x collaborator evaluationOrder'
  classVariableNames: ''
  package: 'Champollion-Core'
```

```
CHInterpretable >> evaluationOrder [
  ^ evaluationOrder
]
```

```
CHInterpretable >> evaluateReceiver [
  evaluationOrder := self next.
  ^ self
]
```

```
CHInterpretable >> returnEvaluationOrder [
  ^ self evaluateReceiver
      messageArg1: self next
      arg2: self next
      arg3: self next
]
```

```
CHInterpretable >> messageArg1: arg1 arg2: arg2 arg3: arg3 [
  ^ {arg1 . arg2 . arg3}
]

CHInterpreterTests >> testEvaluationOrder [
  | argumentEvaluationOrder |
  argumentEvaluationOrder := self executeSelector:
    #returnEvaluationOrder.

  self assert: (self peanoToInt: receiver evaluationOrder) equals: 0.
  self
    assert: (argumentEvaluationOrder collect:
    [ :peano | self peanoToInt: peano])
    equals: #(1 2 3)
]
```

To make this test green we need to implement previously some new support in our interpreter: writing to temporary variables and dynamic literal arrays.

```
CHInterpreter >> visitArrayNode: aRBArrayNode [
  ^ aRBArrayNode statements
    collect: [ :e | self visitNode: e ]
    as: Array
]
```

At this point our test will fail because the evaluation order is wrong! The receiver was evaluated 4th, after all arguments. This is solved by changing the order of evaluation in `visitMessageNode:`.

```
CHInterpreter >> visitMessageNode: aMessageNode [
  | newReceiver method args |
  newReceiver := self visitNode: aMessageNode receiver.
  args := aMessageNode arguments collect: [ :each | self visitNode:
    each ].
  method := (newReceiver class compiledMethodAt: aMessageNode
    selector) ast.
  ^ self executeMethod: method withReceiver: newReceiver
    andArguments: args
]
```

7.10 About Name Conflict Resolution

Inside the scope of a method, statements have access to parameters, temporaries, instance and global variables. A name conflict appears when two variables that should be visible in a method share the same name. In a conflict scenario the language developer needs to device a resolution strategy for these problems, to avoid ambiguities.

For example, consider a method `m:` that has an argument named `integer` and defines a temporary variable also named `integer`. How should values of that name be resolved? How are assignments resolved? A conflict resolution strategy provides a set of deterministic rules to answer these questions and let developers understand what their program do in a non-ambiguous way.

A first simple strategy to avoid conflicts is preventing them at construction time. That is, the language should not allow developers to define variables if they generate a name conflict. For example, a method should not be able to define a temporary variable with the same name as an instance variable of its class. Usually these validations are done once at compile time, and programs that do not follow this rules are rejected.

Another strategy to solve this problem is to allow shadowing. That is, we give each variable in our program a priority, and then the actual variable to read or write is looked-up using this priority system. Typically priorities in these schemas are modelled as lexical scopes. Lexical scoping divides a program in a hierarchy of scopes. Each scope defines variables and all but the top level scope have a parent scope. For example, the top level scope defines global variables, the class scope defines the instance variables, the method scope defines the parameters and temporaries. In this way, variable visibility can be defined in terms of a scope: the variables visible in a scope are those defined in the scope or in the parents of the scope. Moreover, scoping also gives a conflict resolution strategy: variables defined closer to the current scope in the scope hierarchy have more priority than those defined higher in the scope hierarchy.

7.11 About Return

As a reader you may wonder why we did not do anything for return expression and this is an interesting question. Up to now interpreting a return is just return the value of the interpretation of the return expression. In fact up until now a method execution has a single path of execution: it means that the complete method body should be executed and that we did not introduce different condition control flow. This is when we will introduce block closure and conditional control flow that we will have to revisit the interpretation of return.

7.12 Conclusion

Supporting message sends and in particular method execution is the core of the computation in an object-oriented language and this is what this chapter covered.

Implementing messages implied modelling the call-stack and keeping it balanced on method returns. We have seen that a call-stack is made up of frames, each frame representing the activation of a method: it stores the method, receiver, arguments, and temporaries of the method that is executing. When a message takes place, receiver and arguments are evluated in order from left to right, a new frame is created and all values are stored in the frame.

CHAPTER 8

Late Binding and Method Lookup

Method lookup deserves a chapter on its own: it represents the core internal logic of late-binding and it the first part of sending a message. The method-lookup algorithm needs to support normal message-sends as well as 'super' message-sends. We will implement method lookup for message send to an object. Then we will present message send to super and we will finish by looking at how to support error (message `doesNotUnderstand:`).

8.1 Method Lookup Introduction

So far we have concentrated on method evaluation and put aside method lookup. Our current solution fetches methods from the class of the receiver, without supporting inheritance. In this section we address this problem and implement a proper method lookup algorithm.

To implement and test the method lookup, we will extend our scenario classes with a class hierarchy. We introduce two superclasses above `CHInterpretable`: `CHInterpretableSecondSuperclass` and its subclass `CHInterpretableSuperclass`. With this setup we will be able to test all interesting situations, even the ones leading to infinite loops if our method lookup is wrongly implemented.

```
Object subclass: #CHInterpretableSecondSuperclass
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Champollion-Core'
```

```
CHInterpretableSecondSuperclass subclass: #CHInterpretableSuperclass
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Champollion-Core'
```

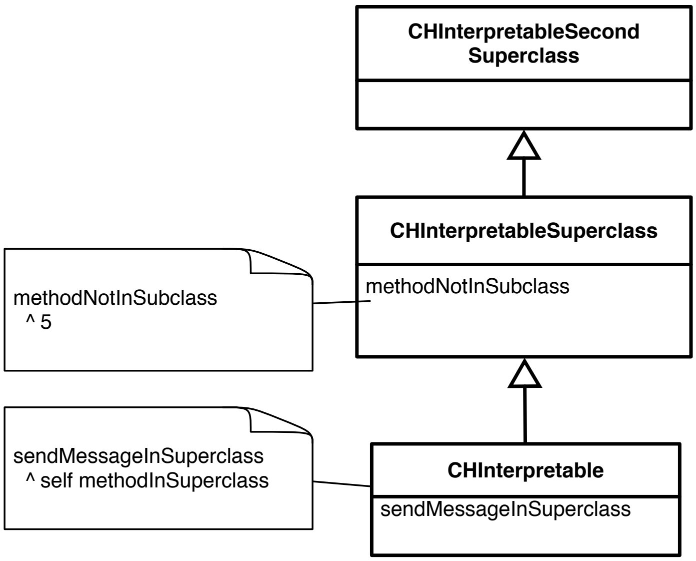


Figure 8-1 A simple hierarchy for self-send lookup testing.

```
CHInterpretableSuperclass subclass: #CHInterpretable
  instanceVariableNames: 'x collaborator evaluationOrder current'
  classVariableNames: ''
  package: 'Champollion-Core'
```

Our first scenario for method lookup will check that sending a message climbs up the inheritance tree when a method is not found in the receiver's class class. In the code below, we define a method in `CHInterpretable` that does a `self` message whose method is implemented in its `CHInterpretableSuperclass` superclass. Executing the first method should send the message, find the superclass method, and evaluate it.

```
CHInterpretableSuperclass >> methodInSuperclass [
  ^ 5
]

CHInterpretable >> sendMessageInSuperclass [
  ^ self methodInSuperclass
]

CHInterpreterTest >> testLookupMessageInSuperclass [
  self assert: (self executeSelector: #sendMessageInSuperclass)
    equals: 5
]
```

8.2 Implement a First Lookup

The test should fail with our evaluator as is, because the evaluation of the message send will not find the method in the receiver's class. A first step towards implementing the lookup is to refactor the method `visitMessageNode:` and extract the wrong code into a `lookup:fromClass:` method.

```
CHInterpreter >> visitMessageNode: aMessageNode [
  | newReceiver method args |
  newReceiver := self visitNode: aMessageNode receiver.
  args := aMessageNode arguments collect: [ :each | self visitNode:
    each ].
  method := self lookup: aMessageNode selector fromClass:
    newReceiver class.
  ^ self execute: method withReceiver: newReceiver andArguments: args
]

CHInterpreter >> lookup: aSelector fromClass: aClass [
  ^ (aClass compiledMethodAt: aMessageNode selector) ast.
]
```

The method `lookup:fromClass:` is now the place to implement the method lookup algorithm:

- if the current class defines the method returns the corresponding AST;
- if the current class does not define the method and we are not on the top of the hierarchy, we recursively lookup in the class' superclass;
- else when we are on top of the hierarchy and the `lookup:fromClass:` returns nil to indicate that no method was found.

The method `lookup:fromClass:` does not raise an error because this way the `visitMessageNode:` method will be able to send the `doesNotUnderstand:` message to the receiver, as we will see later in this chapter.

```
CHInterpreter >> lookup: aSymbol fromClass: aClass [
  "Return the AST of a method or nil if none is found"

  "If the class defines a method for the selector, return the AST
    corresponding to the method"
  (aClass includesSelector: aSymbol)
    ifTrue: [ ^ (aClass compiledMethodAt: aSymbol) ast ].

  "Otherwise lookup recursively in the superclass.
  If we reach the end of the hierarchy return nil"
  ^ aClass superclass
    ifNil: [ nil ]
    ifNotNil: [ self lookup: aSymbol fromClass: aClass superclass ]
]
```

We should call the method `lookup:fromClass:` from the `visitMessageNode:` and our test will pass.

8.3 The Case of Super

Many people gets confused by the semantics of super. The super variable has two different roles in the execution of an object-oriented language. When the super variable is read, its value is the receiver of the message as we saw it in the first chapter, it has the same value as self.

The second role of the super variable is to alter the method lookup when super is used as the receiver of the message send. When super is used as the receiver of a message send, the method lookup does not start at the class of the receiver, but at the class where the method is installed instead, allowing it to go up higher and higher in the hierarchy.

We define a method doesSuperLookupFromSuperclass below. It is not really good since it uses super while it is not needed. The handling of overridden messages will present better tests.

```
CHInterpretableSuperclass >> isInSuperclass [
  ^ true
]

CHInterpretable >> isInSuperclass [
  ^ false
]

CHInterpretable >> doesSuperLookupFromSuperclass [
  ^ super isInSuperclass
]
```

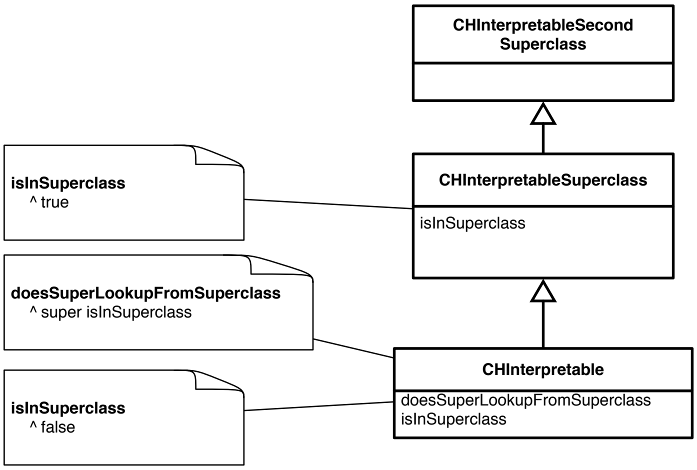

CHInterpretableSecond Superclass
CHInterpretableSuperclass
isInSuperclass
CHInterpretable
doesSuperLookupFromSuperclass
isInSuperclass
isInSuperclass
^ true
doesSuperLookupFromSuperclass
^ super isInSuperclass
isInSuperclass
^ false

Figure 8-2 A simple hierarchy for super-send lookup testing.

Once these methods defined, we can now test that the isInSuperclass message activates the method in the superclass, returning true.

```
CHInterpreterTest >> testLookupSuperMessage [
  self assert: (self executeSelector: #doesSuperLookupFromSuperclass)
]
```

The super variable changes the method lookup described previously. When the receiver is super, the lookup does not start from the class of the receiver, but from the superclass of the class defining the method of the current frame. This implies that we need a way to access the method that is being currently executed, and the class where it is defined.

We can again store this information in the current frame during the method's activation. We add it for now as a fake temporary variable in the frame, with the name ___method. By prefixing the variable's name with ___ we make it less probable this fake variable creates a conflict with a real variable. If we would have just named it e.g., method, any method with a normal normal temporary called method would be broken.

```
CHInterpreter >> executeMethod: anAST withReceiver: anObject
    andArguments: aCollection [
  | result |
  self pushNewFrame.
  self tempAt: #___method put: anAST.
  self tempAt: #self put: anObject.
  anAST arguments with: aCollection do: [ :arg :value | self tempAt:
    arg name put: value ].
  result := self visitNode: anAST body.
  self popFrame.
  ^ result
]
```

We also define a convenience accessor method currentMethod, to get the current method stored in the current frame. In the future, if we want to change this implementation, we will have less places to change if we hide the access to the method behind an accessor.

```
CHInterpretable >> currentMethod [
  ^ self tempAt: #___method
]
```

Note that using the current frame to store the current method will work, even if we have several messages in sequence. When a message is sent a new frame is pushed with a new method, and upon return the frame is popped along with its method. So the top frame in the stack will be always contain the method it executes. Finally, we redefine the visitMessageNode: method to change class where to start looking for the method.

```
CHInterpreterTest >> visitMessageNode: aMessageNode [

  | newReceiver method args lookupClass pragma |
  newReceiver := self visitNode: aMessageNode receiver.
  args := aMessageNode arguments collect: [ :each | self visitNode:
    each ].

  lookupClass := aMessageNode receiver isSuper
    ifTrue: [ self currentMethod methodClass superclass ]
    ifFalse: [ newReceiver class ].
  method := self lookup: aMessageNode selector fromClass:
    lookupClass.
  ^ self executeMethod: method withReceiver: newReceiver
    andArguments: args
]
```

8.4 Overridden Messsages

We have made sure that sending a message to `super` starts looking methods in the superclass of the class defining the method. Now we would like to make sure that the lookup works even in presence of overridden methods.

Let’s define the method `overriddenMethod` in a superclass returning a value, and in a subclass just doing a super send with the same selector.

```
CHInterpretableSuperClass >> overriddenMethod [
  ^ 5
]

CHInterpretable >> overriddenMethod [
  ^ super overriddenMethod
]
```

If our implementation is correct, sending the `overriddenMethod` message to our test receiver should return `5`. If it is not, the test should fail, or worse, loop infinitely.

Then we check that our test returns the correct value. If the test loops infinitely the test will timeout.

```
CHInterpreterTest >> testLookupRedefinedMethod [
  self assert: (self executeSelector: #overriddenMethod) equals: 5
]
```

If our previous implementation was correct, this test should pass.

8.5 Checking Correct Semantics

To ensure that the method lookup is correctly implemented, especially in the presence of `super` messages, we need to verify an extra condition. Lot of material wrongly defines that `super` messages look up methods starting from the superclass of the class of the receiver. This definition, illustrated in the code snippet below, is incorrect: it only works when the inheritance depth is limited to two classes, a class and its superclass. In other cases, this definition will create an infinite loop.

```
CHInterpreter >> visitMessageNode: aMessageNode [
  | newReceiver method args lookupClass |
  newReceiver := self visitNode: aMessageNode receiver.
  args := aMessageNode arguments collect: [ :each | self visitNode:
    each ].

  lookupClass := aMessageNode receiver isSuper
    ifTrue: [ newReceiver class superclass ]
    ifFalse: [ newReceiver class ].
  method :=  self lookup: aMessageNode selector fromClass:
    lookupClass.

  ^ self executeMethod: method withReceiver: newReceiver
    andArguments: args
]
```

A scenario showing such a problem is shown in Figure 8-3. In this scenario, our inheritance depth is of three classes and we create two methods with the same selector. In the highest class, the method returns a value. In the middle class, the first method is overridden doing a super send.

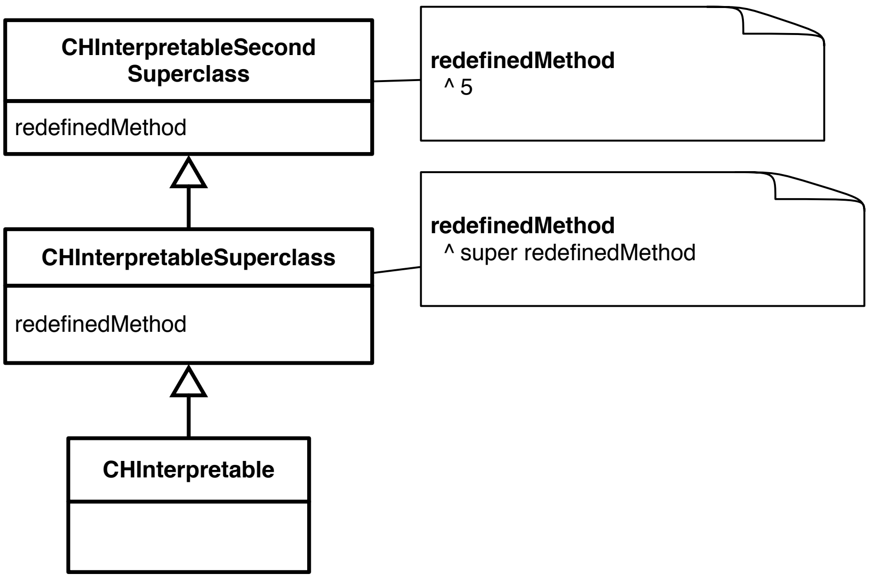


Figure 8-3 A simple situation making wrongly defined super loop endlessly: sending the message `redefinedMethod` to an instance of the class `CHInterpretable` loops forever.

```
CHInterpretableSecondSuperClass >> redefinedMethod [
  ^ 5
]

CHInterpretableSuperClass >> redefinedMethod [
  ^ super redefinedMethod
]
```

To finish our scenario, we create an instance of the lower subclass in the hierarchy, and we send it a message with the offending selector.

```
CHInterpreterTest >> testLookupSuperMessageNotInReceiverSuperclass [
  self assert: (self executeSelector: #redefinedMethod) equals: 5
]
```

With the incorrect semantics, our test will start by activating `CHInterpretableSuperclass>>#redefinedMethod`. When the interpreter finds the super send, it will start the lookup from the superclass of the receiver's class: `CHInterpretableSuperclass`. Starting the lookup from this class will again find and activate `CHInterpretableSuperclass>>#redefinedMethod`, which will lead to activating the same method over and over again...

Coming back to our previous correct definition, it works properly, and makes our test pass:

```
CHInterpreterTest >> visitMessageNode: aMessageNode [

  | newReceiver method args lookupClass pragma |
  newReceiver := self visitNode: aMessageNode receiver.
  args := aMessageNode arguments collect: [ :each | self visitNode:
    each ].

  lookupClass := aMessageNode receiver isSuper
    ifTrue: [ self currentMethod methodClass superclass ]
    ifFalse: [ newReceiver class ].
  method := self lookup: aMessageNode selector fromClass:
    lookupClass.
  ^ self executeMethod: method withReceiver: newReceiver
    andArguments: args
]
```

8.6 Does not understand and Message Reification

To finish this chapter we will implement support for the `doesNotUnderstand:` feature. In Pharo, when an object receives a message for which the lookup does not find a corresponding method, it sends instead the `doesNotUnderstand:` message to that object, with the "original message" as argument. The same mechanism exists in many other languages. This original message is not only the selector but it comprises the arguments too. The interpreter should take selector and arguments to create an object representation of the message. We say the interpreter reifies the message.

About Reification.

Reification is the process of making concrete something that was not. In the case of the interpreter of a programming language, many of the operations of the language are implicit and hidden in the interpreter execution. For example, the implementation of message-sends and assignments are hidden to the developer in the sense that the developer cannot manipulate assignments for example to count the number of time an assignment has been used during program execution. While information hiding in interpreters is important to make languages safe and sound, the language has no way to manipulate those abstractions. Reifications enter in the game to enable those manipulations: interpreter concepts are concretized as objects in the interpreted language, they are "lifted-up" from the interpreter level to the application.

Reifications are a powerful concept that allow us to manipulate implementation concerns from the language itself. In this case, the does not understand mechanism allows us to intercept the failing message-lookup algorithm and to implement in our program a strategy to handle the error. There exist in Pharo many different reifications such as classes and methods. In the scope of interpreters, we will see in the chapters that follow other kind of reification: context objects representing execution frames.

A word is to be said about the performance implications of reifications. Reifications add levels of indirection in the execution. In addition it allocates objects and this adds a significant overhead in the interpretation, and increases the pressure in the garbage collector. Production interpreters try to minimize this cost to delay reifications as much as possible, and avoid them when they are not necessary. This is what we will do with message reifications: we will create them when a method-lookup effectively fails and not before, penalizing only the execution of does not understand messages.

8.7 Implementing doesNotUnderstand:

To implement the does not understand feature, let's start by setting up our testing scenario: a method sending a not understood messageIDoNotUnderstandWithArg1:withArg:2 message. This message should be looked-up and not found, so the interpreter should send a doesNotUnderstand: message to the same receiver with the message reification. For the message reification, we are going to follow Pharo's behaviour and expect an instance of Message that should have the selector and an array with all the arguments.

```
CHInterpretable >> doesNotUnderstand: aMessage [
  ^ aMessage
]

CHInterpretable >> sendMessageNotUnderstood [
  ^ self messageIDoNotUnderstandWithArg1: 17 withArg2: 27
]
```

```
CHInterpreterTest >> testDoesNotUnderstandReifiesMessageWithSelector
    [
  self
    assert: (self executeSelector: #sendMessageNotUnderstood)
    selector
    equals: #messageIDoNotUnderstandWithArg1:withArg2:
]

CHInterpreterTest >>
    testDoesNotUnderstandReifiesMessageWithArguments [
  self
    assert: (self executeSelector: #sendMessageNotUnderstood)
    arguments
    equals: #( 17 27 )
]
```

These two tests will fail in the interpreter, because the method lookup will return nil, which will fail during method activation. To fix it, we need to handle this problem and send the doesNotUnderstand: message, as we said before.

```
CHInterpreter >> visitMessageNode: aMessageNode [

  | newReceiver method args lookupClass |
  newReceiver := self visitNode: aMessageNode receiver.
  args := aMessageNode arguments collect: [ :each |
    self visitNode: each ].

  lookupClass := aMessageNode receiver isSuperVariable
            ifTrue: [ self currentMethod methodClass superclass ]
            ifFalse: [ newReceiver class ].
  method := self lookup: aMessageNode selector fromClass:
    lookupClass.
  method ifNil: [ | doesNotUnderstandMethod messageReification |
    "Handle does not understand:
     - lookup the #doesNotUnderstand: selector
     - reify the message
     - activate"
    doesNotUnderstandMethod := self lookup: #doesNotUnderstand:
     fromClass: lookupClass.
    messageReification := Message
      selector: aMessageNode selector
      arguments: args asArray.
    ^ self
      execute: doesNotUnderstandMethod
      withReceiver: newReceiver
      andArguments: { messageReification } ].

  ^ self execute: method withReceiver: newReceiver andArguments: args
]
```

Note that reifying does not understand requires that our interpreter knows two new things about our language: what selector is used for `#doesNotUnderstand:`, and what class is used to reify `Message`. In this case we are implementing a Pharo evaluator that runs in the same environment as the evaluated program: they share the same memory, classes, global variables. Because of this we make use of the existing selector and classes in Pharo. In contrast, implementing an evaluator that runs on a different environment than the evaluated program (e.g., a Pharo evaluator implemented in C), such dependencies need to be made explicit through a clear language-interpreter interface. This is for this reason that the Pharo virtual machine needs to know the selector of the message to be sent in case of message not understood.

8.8 Refactoring Time

As a final step, we refactor our `visitMessageNode:` to avoid repeating some code. We extract the method activation and send, separating it from the decision of the class to start the lookup.

```
CHInterpreter >> send: aSelector receiver: newReceiver
    lookupFromClass: lookupClass arguments: arguments [
  "Lookup a selector from a class, and activate the method.
  Handle does not undertand case and message reification on demand
    if lookup fails."

  | method |
  method := self lookup: aSelector fromClass: lookupClass.
  method ifNil: [ | messageReification |
    "Handle does not understand:
     - lookup the #doesNotUnderstand: selector
     - reify the message
     - activate"
    messageReification := Message
      selector: aSelector
      arguments: arguments.
    ^ self
      send: #doesNotUnderstand:
      newReceiver: receiver
      lookupFromClass: lookupClass
      arguments: { messageReification } ].
  ^ self execute: method withReceiver: newReceiver andArguments:
    arguments
]
```

And then make use of it in `visitMessageNode:`.

```
CHInterpreter >> visitMessageNode: aMessageNode [

  | newReceiver args lookupClass |
  newReceiver := self visitNode: aMessageNode receiver.
  args := aMessageNode arguments collect: [ :each |
        self visitNode: each ].

  lookupClass := aMessageNode receiver isSuperVariable
          ifTrue: [ self currentMethod methodClass superclass ]
          ifFalse: [ newReceiver class ].
  ^ self
    send: aMessageNode selector
    receiver: newReceiver
    lookupFromClass: lookupClass
    arguments: args asArray
]
```

8.9 Conclusion

In this chapter, we extended our interpreter to evaluate messages. Messages are the arguably the most important part of our interpreter, as operations in object-oriented languages are expressed in terms of them. It is also the most complex part that we have implemented so far.

At the end of this chapter we have seen the method lookup algorithm to resolve what method to execute given a receiver and a selector. We have also seen the particularities of `self` and `super` sends. Finally we have shown how the `doesNotUnderstand:` feature is implemented, by handling the lookup error, and we introduced the concept of reification to concretize and lift-up the failing message from our evaluator to the language.

CHAPTER 9

Primitive Operations

Our interpreter does not handle yet any essential behavior such as basic number operations. This prevents us to evaluate complex programs. This chapter introduces the concept of primitive methods in Pharo. We call primitive behavior the behaviour that needs to be implemented in the interpreter or evaluator because it cannot be purely expressed in the programming language, Pharo in this case. Let's consider for example the operation of adding up two numbers (+). We cannot express in a pure and concise way a normal method executing an addition. Along with arithmetics, other examples of primitive operations are object allocation and reflective object access. Such primitive behavior is expressed as special methods, namely **primitive methods** in Pharo, whose behavior is defined in the interpreter.

Differently from languages such as Java or C, that express arithmetics as special operators that are compiled/interpreted differently, Pharo maintains the message-send metaphore for primitive behavior. Indeed, in Pharo + is a message, which triggers a look-up and a method activation. This separation makes redefining operators as simple as implementing a method with the selector + in our own class, without the need for special syntax for it.

In addition of essential behavior, primitive behavior is often used to implement performance-critical code in a much more efficient way, since primitives are implemented in the implementation language and do not suffer the interpretation overhead.

In this chapter we will study how primitive methods work, and how they should be properly implemented, including the evaluation of their fallback code (i.e., what happens when a primitive fails). We will then visit some of the essential primitives we need to make work to execute complex examples.

9.1 Primitives in Pharo

In Pharo the design of primitives is split in three different parts: *messages*, *primitive methods* (the Pharo method that is annotated), and the *primitive* itself which is provided by the interpreter (in our case this is a method that implements the primitive behavior. In the case of a Virtual Machine it would be a C function).

Primitives invoked as Messages. The first thing to note is that in Pharo programs primitive behavior is invoked through standard message sends. Indeed sending the message `1 + 2` is handled as a message, but the method + on `Integer` is a primitive (during Pharo execution it calls primitive functions transparently from the developer).

This management of primitives as messages allows developers to define operators such as + as non-primitives on their own classes, by just implementing methods with the corresponding selectors and without any additional syntax.

With some terminology abuse we could think of this as ”operator redefinition”, although it is no such, it is just a standard method re/definition. This operator redefinition feature is useful for example when creating internal domain specific languages (DSLs), or to have polymorphism between integers and floats.

This is why in Pharo it is possible to define a new method + on any class as follows:

```
MyClass >> + anArgument [
  "Redefine +"
  ...
]
```

Primitive annotation. To define primitive behavior, Pharo relies on special methods called *primitive methods*: Primitive methods are normal Pharo methods with a `primitive` annotation. This annotation identifies that the method is special.

For example, let us consider the method `SmallInteger>>+` method below:

```
SmallInteger >> + aNumber [
  <primitive: 1>
  ^ super + aNumber
]
```

This method looks like a normal method with selector +, and with a normal method body doing `^super + aNumber`. The only difference between this method and a normal one is that this method also has an annotation, or pragma, indicating that it is the primitive number 1.

The body if the method is normally not executed. In its place the primitive 1 is executed. The method body is only executed if the primitive failed.

Interpreter primitives. Before diving into how *primitive methods* are executed, let us introduce the third component in place: the *interpreter primitives*. A primitive is a piece of code (another method) defined in the interpreter that will be executed when a primitive method is executed.

To make parallel between our interpreter and the Pharo virtual machine, the virtual machine is executing a C-function is executed when a primitive method is executed.

The interpreter defines a set of supported primitives with unique ids. In our case, for example, the primitive with id 1 implements the behavior that adds up two integers.

When a primitive method is activated, it first looks-up what *primitive* to execute based on its primitive id number, and executes it. The primitive performs some validations if required, executes the corresponding behavior, and returns either with a success if everything went ok, or a failure if there was a problem. On success, the method execution returns with the value computed by the primitive. On failure, the body of the method is executed instead. Because of this, the body of a primitive method is also named the ”fall-back code”.

Note that in Pharo some primitives called essential such as the object allocation cannot be executed from Pharo. For such primitive, implementors added a method body to describe what the primitive is doing if it could be written in Pharo.

9.2 Infrastructure for Primitives

To implement primitives in our evaluator we only need to change how methods are activated. Indeed, as we have seen above, the method lookup nor other special nodes are required for the execution of primitives, and the AST already supports pragma nodes, from which we need to extract the method's primitive id.

We will extend the method evalution in a simple way: During the activation of a primitive method, we need to look for the primitive to execute, and check for failures. Therefore we need to map primitive ids to primitive methods.

We implement such a mapping using a table with the form `<id, evaluator_selector>`. The `primitives` instance variable is initialized to a dictionary as follows:

```
CHInterpreter >> initialize [
  super initialize.
  stack := Stack new.
  primitives := Dictionary new.
  self initializePrimitiveTable.
]
```

Then we define the method `initializePrimitiveTable` to initialize the mapping between the primitive id and the Pharo method to be executed.

```
CHInterpreter >> initializePrimitiveTable [
  primitives at: 1 put: #primitiveSmallIntegerAdd
]
```

Let's start by setting up our testing scenario: adding up two numbers. We make sure to work with small enough numbers in this test, to not care about primitive failures yet. Doing `1 + 5` the primitive should always be a success and return `6`.

```
CHInterpretable >> smallintAdd [
  ^ 1 + 5
]

CHInterpreterTests >> testSmallIntAddPrimitive [
  self
    assert: (self executeSelector: #smallintAdd)
    equals: 6
]
```

9.3 Primitives Implementation

In our first iteration we will not care about optimizing our evaluator, for which we had already and we will have tons of opportunities. To have a simple implementation to work on, we execute the primitive after the method's frame creation, in the `visitMethodNode:` method. This way the primitive has a simple way to access the receiver and the arguments by reading the frame. We leave primitive failure management for our second iteration.

Upon primitive method execution, we extract the primitive id from the pragma, get the selector of that id from the table, and use `perform:` method on the interpreter with that selector to execute the primitive.

```
CHInterpreter >> executePrimitiveMethod: anAST [
  | primitiveNumber |
  primitiveNumber := anAST pragmas
    detect: [ :each | each isPrimitive ]
    ifFound: [ :aPragmaPrimitive | aPragmaPrimitive arguments first
     value ]
    ifNone: [ self error: 'Not a primitive method' ].
  ^ self perform: (primitives at: primitiveNumber)
]
```

We also need to take care of sending the receiver and arguments of the message to the primitive, so it can manipulate them.

```
CHInterpreter >> visitMethodNode: aMethodNode [
  aMethodNode isPrimitive ifTrue: [
    "Do not handle primitive failures for now"
    ^ self executePrimitiveMethod: aMethodNode ].
  ^ self visitNode: aMethodNode body
]
```

We define the primitive `primitiveSmallIntegerAdd` as follows:

```
CHInterpreter >> primitiveSmallIntegerAdd [
  | receiver argument |
  receiver := self receiver.
  argument := self argumentAt: 1.
  ^ receiver + argument
]

CHInterpreter >> argumentAt: anInteger [
  ^ self tempAt: (self currentMethod arguments at: anInteger) name
]
```

9.4 Primitive Failures and Fallback Code

Let’s now consider what should happen when a primitive fails. For example, following Pharo’s specification, primitive 1 fails when the receiver or the argument are not small integers, or whenever their sum overflows and does not fit into a small integer anymore.

To produce one of such failing cases, we can implement primitive 1 in our `CHInterpretable` class, which should fail because the receiver should be a small integer. When it fails, the fallback code should execute.

We define two methods `failingPrimitive` and `callingFailingPrimitive` to support the test of failing primitive.

```
CHInterpretable >> failingPrimitive [
  <primitive: 1>
  ^ 'failure'
]

CHInterpretable >> callingFailingPrimitive [
  ^ self failingPrimitive
]
```

```
CHInterpreterTests >> testFailingPrimitive [
  self
    assert: (self executeSelector: #callingFailingPrimitive)
    equals: 'failure'
]
```

To add primitive failures in a clean way, we introduce them as exceptions. We define a new subclass of `Exception` named `CHPrimitiveFail`

In the primitive `primitiveSmallIntegerAdd`, if we detect a failure condition, we raise a `CH-PrimitiveFail` error. Note that this the primitive is incomplete since we should also test that the argument and the result is small integer as shown in the following sections.

```
CHInterpreter >> primitiveSmallIntegerAdd [
  | receiver argument |
  receiver := self receiver.
  receiver class = SmallInteger
    ifFalse: [ CHPrimitiveFail signal ].
  argument := self argumentAt: 1.
  ^ receiver + argument
]
```

We then need to modify the way we evaluate methods to handle `CHPrimitiveFail` exceptions and continue evaluating the body.

```
CHInterpreter >>visitMethodNode: aMethodNode [
  [ aMethodNode isPrimitive ifTrue: [
    "Do not handle primitive failures for now"
    ^ self executePrimitiveMethod: aMethodNode ]]
    on: CHPrimitiveFail do: [ :err |
      "Nothing, just continue with the method body" ].

  ^ self visitNode: aMethodNode body
]
```

With these changes, everything should work fine now.

9.5 Typical Primitive Failure Cases

For primitives to work properly, and for Pharo to be a safe language, primitives should properly do a series of checks. This is particularly important when the interpreter fully controls all other aspects of the language, such as the memory. In such cases, primitives, as well as the other parts of the evaluator, have full power over our objects, potentially producing memory corruptions.

Among the basic checks that primitives should do, they should not only verify that arguments are of the primitive's expected type, as we have shown above. In addition a general check is that the primitive was called with the right number of arguments. This check is particularly important because developers may wrongly define primitives such as we did before, where we have defined a unary method while the primitive was expecting one argument. If we don't properly check the arguments trying to access it could cause an interpreter failure, while the proper behaviour should be to just fail the primitive and let the fallback code carry on the execution.

In the following sections, we implement a series of essential primitives taking care of typical failure cases. With such primitives, it will be possible to execute a large range of Pharo programs.

9.6 Essential Primitives: Arithmetic

The basic arithmetic primitives are small integer addition, substraction, multiplication, and division. They all require a small integer receiver, a small integer argument, and that the result is also a small integer. Division in addition fails in case the argument is `0`. The following code snippet illustrates integer addition and division. For space reason, we do not include substraction and multiplication, their implementation is similarly to the one of addition.

```
CHInterpreter >> initializePrimitiveTable [
  ...
  primitives at: 1    put: #primitiveSmallIntegerAdd.
  primitives at: 2    put: #primitiveSmallIntegerMinus.
  primitives at: 9    put: #primitiveSmallIntegerMultiply.
  primitives at: 10   put: #primitiveSmallIntegerDivide.
  ...
]
```

The addition primitive now checks that the receiver, argument and result are small integers.

```
CHInterpreter >> primitiveSmallIntegerAdd [
  | receiver argument result |
  self numberOfArguments < 1
    ifTrue: [ CHPrimitiveFail signal ].

  receiver := self receiver.
  receiver class = SmallInteger
    ifFalse: [ CHPrimitiveFail signal ].

  argument := self argumentAt: 1.
  argument class = SmallInteger
    ifFalse: [ CHPrimitiveFail signal ].

  result := receiver + argument.
  result class = SmallInteger
    ifFalse: [ CHPrimitiveFail signal ].
  ^ result
]
```

```
CHInterpreter >> primitiveSmallIntegerDivide [
  | receiver argument result |
  self numberOfArguments < 1
    ifTrue: [ CHPrimitiveFail signal ].

  receiver := self receiver.
  receiver class = SmallInteger
    ifFalse: [ CHPrimitiveFail signal ].
  argument := self argumentAt: 1.
  (argument class = SmallInteger
    and: [ argument ~= 0 ])
      ifFalse: [ CHPrimitiveFail signal ].

  result := receiver / argument.
  result class = SmallInteger
    ifFalse: [ CHPrimitiveFail signal ].
  ^ result
]
```

9.7 Essential Primitives: Comparison

Comparison primitives span in two different sets. The first set contains the primitives implementing number comparisons such as less than or greater or equals than. The second set contains the primitives for object identity comparison: identity equals to and identity not equals to. All number comparisons all require a small integer receiver, a small integer argument. Identity comparisons only require that the primitive receives an argument to compare to. The following code snippet illustrates both kind of methods with small integer less than and object idenity equality.

```
CHInterpreter >> initializePrimitiveTable [
  ...
  primitives at: 3  put: #primitiveSmallIntegerLessThan.
  primitives at: 4  put: #primitiveSmallIntegerGreaterThan.
  primitives at: 5  put: #primitiveSmallIntegerLessOrEqualsThan.
  primitives at: 6  put: #primitiveSmallIntegerGreaterOrEqualsThan.

  primitives at: 7  put: #primitiveSmallIntegerEqualsThan.
  primitives at: 8  put: #primitiveSmallIntegerNotEqualsThan.

  primitives at: 110  put: #primitiveIdentical.
  primitives at: 111  put: #primitiveNotIdentical.
  ...
]
```

```
CHInterpreter >> primitiveSmallIntegerLessThan [
  | receiver argument result |
  self numberOfArguments < 1
    ifTrue: [ CHPrimitiveFail signal ].

  receiver := self receiver.
  receiver class = SmallInteger
    ifFalse: [ CHPrimitiveFail signal ].

  argument := self argumentAt: 1.
  argument class = SmallInteger
    ifFalse: [ CHPrimitiveFail signal ].

  ^ receiver < argument
]
```

```
CHInterpreter >> primitiveIdentical [
  self numberOfArguments < 1
    ifTrue: [ CHPrimitiveFail signal ].

  ^ self receiver == (self argumentAt: 1)
]
```

9.8 Essential Primitives: Array Manipulation

So far our interpreter is able to manipulate only objects with instance variables, but not arrays or their variants e.g., strings. Arrays are special objects whose state is accessed with primitives, usually in methods named `at:` and `at:put:` and `size`. Array access primitives check that the receiver is of the right kind and that the index arguments are integers within bounds of the array. The following code snippet illustrates `Array` access primitives for general Arrays, and Strings.

```
CHInterpreter >> initializePrimitiveTable [
  ...
  primitives at: 60   put: #primitiveAt.
  primitives at: 61   put: #primitiveAtPut.
  primitives at: 62   put: #primitiveSize.
  primitives at: 63   put: #primitiveStringAt.
  primitives at: 64   put: #primitiveStringAtPut.
  ...
]
```

The primitive `primitiveSize` verifies that the receiver is an object supporting the notion of size.

```
CHInterpreter >> primitiveSize [
  self receiver class classLayout isVariable
    ifFalse: [ CHPrimitiveFail signal ].

  ^ self receiver basicSize
]
```

The primitive `primitiveAt` verifies that the receiver is an object supporting the notion of size and in addition that the index is an integer in the range of the size of the receiver.

```
CHInterpreter >> primitiveAt [
  self numberOfArguments < 1
    ifTrue: [ CHPrimitiveFail signal ].

  self receiver class classLayout isVariable
    ifFalse: [ CHPrimitiveFail signal ].

  ((self argumentAt: 1) isKindOf: SmallInteger)
    ifFalse: [ CHPrimitiveFail signal ].

  "Bounds check"
  self receiver size < (self argumentAt: 1)
```

```
    ifTrue: [ CHPrimitiveFail signal ].

  ^ self receiver basicAt: (self argumentAt: 1)
]
```

The primitive `primitiveStringAt` verifies that the receiver is from a class whose elements are bytes.

```
CHInterpreter >> primitiveStringAt [
  self numberOfArguments < 1
    ifTrue: [ CHPrimitiveFail signal ].

  self receiver class classLayout isBytes
    ifFalse: [ CHPrimitiveFail signal ].

  ((self argumentAt: 1) isKindOf: SmallInteger)
    ifFalse: [ CHPrimitiveFail signal ].
  "Bounds check"

  self receiver size < (self argumentAt: 1)
    ifTrue: [ CHPrimitiveFail signal ].

  ^ self receiver at: (self argumentAt: 1)
]
```

9.9 Essential Primitives: Object Allocation

Object allocation is implemented by primitives `new` and `new:`. The method `new` allocates a new object from a fixed-slot class. The method `new:` allocates a new object from a variable-slot class such as `Array`, using the number of slots specified as argument.

Both these primitives validate that the receiver are classes of the specified kinds. In addition `new:` does check that there is an argument, it is a small integer.

```
CHInterpreter >> initializePrimitiveTable [
  ...
  primitives at: 70   put: #primitiveBasicNew.
  primitives at: 71   put: #primitiveBasicNewVariable.
  ...
]
```

```
CHInterpreter >> primitiveBasicNew [
  self receiver isClass
    ifFalse: [ CHPrimitiveFail signal ].
  ^ self receiver basicNew
]
```

```
CHInterpreter >> primitiveBasicNewVariable [
  self numberOfArguments < 1
    ifTrue: [ CHPrimitiveFail signal ].

  self receiver isClass
```

```
    ifFalse: [ CHPrimitiveFail signal ].
  self receiver class classLayout isVariable
    ifFalse: [ CHPrimitiveFail signal ].

  ((self argumentAt: 1) isKindOf: SmallInteger)
    ifFalse: [ CHPrimitiveFail signal ].

  ^ self receiver basicNew: (self argumentAt: 1)
]
```

9.10 Conclusion

This chapter presented primitive behavior, implementing behaviour that cannot be purely expressed in the evaluated language. Primitive behaviour is accessed through *primitive methods*, which are methods marked with a `primitive:` pragma. When a primitive method executes, it first executes the primitive behavior associated with the primitive id. If it fails, the body of the method is executed as in non-primitive methods.

We have then discussed about primitive failures and verifications, and presented a short list of essential primitives that are required to execute more interesting Pharo programs.

CHAPTER 10

Block Closures and Control Flow Statements

In this chapter we will extend our evaluator to manage block closures. Block closures, also named lexical closures, or just blocks in Pharo, are an important concept in most modern programming languages, including Pharo. A lexical closure is an anonymous functions that captures its definition environment.

Closures allow developers to abstract general algorithms from their particular details. For example, a sorting algorithm can be separated from its sorting criteria by making the sorting criteria a block closure passed as argument to it. This allows developers to have the sorting algorithm defined and tested in a single place, and being able to reuse it with multiple criterion in different contexts.

In Pharo, blocks are *lexical* closures i.e., basically functions without a name that capture the environment in which they are defined. Lexical closures are at the center of the Pharo language, because Pharo leverages closures to define its *control-flow* instructions: conditionals, iterations, and early returns. This means that implementing block closures is enough to support all kind of control flow statements in Pharo. Moreover, Pharo libraries make usage of block closures to define library-specific control flow instructions, such as the `do:` and `select:` messages understood by collections. Pharo developers often use closures in the Domain Specific languages that they design. Developers are also encouraged to define their own control flow statements, to hide implementation details of their libraries from their users.

This chapter starts by explaining what blocks are and how they are evaluated. Block evaluation, being a core part of the language definition, is a service that is requested to the evaluator/interpreter through a primitive. We then dive into the lexical capture feature of blocks: when a block closure is created, it captures its defining context, namely its enclosing context (i.e., the visible variables that the block can see). This makes blocks able to read and write not only its own temporary variables but also all the variables accessible to its enclosing context and to maintain such a link even when passed around. Finally, we implement non-local returns: return instructions that return to the block *definition context* instead of the current one. Non-local returns are really important in Pharo since they are used to express early returns (the fact that the execution of a method can be stopped at a given point) a frequent language feature similar to `break` statements in other languages. Without non-local return it would difficult to quit the current execution.

10.1 Representing a Block Closure

When a block expression is executed `[ 1+2 ]`, the instructions inside the block definition are not executed. Instead, a block object is created, containing those instructions. The execution of those instructions is delayed until we send the message `value` to the block object.

This means that from the evaluator point of view, the evaluation of the closure will be different from the evaluation of its execution. Evaluating a block node will return a block object, and the method `value` will require a primitive to request the interpreter the block's execution. This means that we need a way to represent a closure object in our evaluator, and that closure should store the code it is supposed to evaluate later when receiving the `value` message.

Let us define the class `CHBlock` to represent a block. It has an instance variable `code` to hold the block's AST, instance of the `RBBlockNode` class. Notice that we do not use the existing `Block-Closure` class from Pharo, since this class is tied up with the Pharo bytecode. For the sake of simplicity, we will not reconciliate bytecode and AST implementations, meaning that we need our own AST-based block implementation.

```
Object subclass: #CHBlock
  instanceVariableNames: 'code'
  classVariableNames: ''
  package: 'Champollion-Core'
```

```
CHBlock >> code: aRBBlockNode [
  code := aRBBlockNode
]
```

```
CHBlock >> code [
  ^ code
]
```

did we explain before that a method with no return returns self? I think we did not, we should add it! :)

10.2 Blocks Return their last Expression

Differently from the execution of a method that implicitly returns `self` when it has no explicit return statement, a block without return statement implicitly returns the result of its last expression.

Let us write a testing scenario for this case: evaluating the following block should return `5` as it is its last expression.

```
CHInterpretable >> returnBlockValue [
  ^ [ 1 . 5 ] value
]
```

```
CHInterpreterTest >> testBlockValueIsLastStatementValue [
  self assert: (self executeSelector: #returnBlockValue) equals: 5
]
```

When the interpreter encounters a block node, it creates a block object for it. We define the method `visitBlockNode:` as follows:

```
CHInterpreter >> visitBlockNode: aRBBlockNode [
  ^ CHBlock new
    code: aRBBlockNode;
    yourself
]
```

Closures are executed when they receive the message `value` or one of its variants such as value `value:`, `value:value:`... On the reception of such messages, their bodies should be executed.

We follow the design of Pharo and we add a new primitive responsible for the block body execution. We define the method `value` on the class `CHBlock` as a primitive number 201.

```
CHBlock >> value [
  <primitive: 201>
  "If the fallback code executes it means that block evaluation
    failed.
  Return nil for now in such case."
  ^ nil
]
```

We now need to implement the new primitive in the evaluator. A first version of it is to just visit the body of the block's code. Remember that primitives are executed in their own frame already, so the block's body will share the frame created for the primitive method.

```
CHInterpreter >> initializePrimitiveTable [
  ...
  primitives at: 201 put: #primitiveBlockValue.
  ...
]

CHInterpreter >> primitiveBlockValue [
  ^ self visitNode: self receiver code body
]
```

So far we implemented only a simple version of closures. We will extend it in the following sections.

10.3 Closure temporaries

Our simplified closure implementation does not yet have support for closure temporaries. Indeed, a closure such as the following will fail with an interpreter failure because `temp` is not defined in the frame.

```
  [ | temp | temp ] value
```

To solve this we need to declare all block temporaries when activating the block, as we did previously for methods. As a first attempt to make our test green, let's declare block temporaries once the block is activated:

```
CHInterpreter >> primitiveBlockValue [
  "Initialize all temporaries to nil"
  aSequenceNode temporaryNames do: [ :e | self tempAt: e put: nil ].
  ^ self visitNode: self receiver code body
]
```

We are now able to execute the following expression

```
[ | a b |
  a := 1.
  b := 2.
  a + b ] value
```

10.4 Removing Logic Repetition

The handling of temporaries in `primitiveBlockValue` is very similary to a sequence of messages we wrote when activating a normal method in method `executeMethod:withReceiver:andArguments:` below:

```
CHInterpreter >> executeMethod: anAST withReceiver: anObject
    andArguments: aCollection [
  | result |
  self pushNewFrame.
  self tempAt: #self put: anObject.
  anAST arguments with: aCollection do: [ :arg :value | self tempAt:
    arg name put: value ].
  anAST temporaryNames do: [ :tempName | self tempAt: tempName name
    put: nil ].
  result := self visitNode: anAST body.
  self popFrame.
  ^ result
]
```

We solve this repetition by moving temporary initialization to the `visitSequenceNode:` method, since both method nodes and block nodes have sequence nodes inside them.

```
CHInterpreter >> visitSequenceNode: aSequenceNode [
  "Initialize all temporaries to nil"

  aSequenceNode temporaryNames do: [ :e | self tempAt: e put: nil ].

  "Visit all but the last statement without caring about the result"
  aSequenceNode statements allButLast
    do: [ :each | self visitNode: each ].
  "Return the result of visiting the last statement"
  ^ self visitNode: aSequenceNode statements last
]
```

```
CHInterpreter >> primitiveBlockValue [
  ^ self visitNode: self receiver code body
]
```

```
CHInterpreter >> executeMethod: anAST withReceiver: anObject
    andArguments: aCollection [
  | result |
  self pushNewFrame.
  self tempAt: #self put: anObject.
  anAST arguments with: aCollection do: [ :arg :value | self tempAt:
    arg name put: value ].
  result := self visitNode: anAST body.
  self popFrame.
  ^ result
]
```

The resulting code is nicer and simpler. This is a clear identication that the refactoring was a good move.

10.5 Capturing the Defining Context

As we stated before, a closure is not just a function, it is a function that captures the context (set of variables that can be accessible) of its definition. Block closures capture their *defining* context or enclosing context, i.e., the context in which they are created. Blocks are able to read and write their own temporary variables, but also all the variables accessible to its enclosing context such as a temporary variable accessible during the block definition.

In this section we evolve our closure execution infrastructure to support closure temporaries and to provide access to the enclosing environment.

The defining execution context gives the closure access to that context's receiver, arguments and temporaries. Moreover, it is a fairly common mistake to think that the captured context is the caller context, and not the defining context. This is the case, in the example above, where the context where the block closure is defined is both the defined and the caller. However, as soon as we work on more complex scenarios, where blocks are sent as arguments of methods, or stored in temporary variables, this does not hold anymore.

A first scenario to check that our block properly captures the defining context is to evaluate `self` inside a block. In our current design, the receiver specified in the block's frame is the block itself. Indeed, the expression `[ ... ] value` is a message send where the block is the message receiver and `value` is the message. However, the `self` variable should be bound to the instance of `CHInterpretable`.

```
CHInterpretable >> readSelfInBlock [
  ^ [ self ] value
]

CHInterpreterTest >> testReadSelfInBlock [
  self assert: (self executeSelector: #readSelfInBlock) equals:
    receiver
]
```

To make this test pass, we need to implement two different things in the evaluator.

- First we need to capture the defining context at block *definition* time in `visitBlockN-ode:`.
- Second we need to use *that* captured context to resolve variables.

Capturing the defining context is as simple as storing the current `topFrame` at the moment of the method creation.

We extend `CHBlock` with a `definingContext` instance variable and corresponding accessors (omitted here after).

```
Object subclass: #CHBlock
  instanceVariableNames: 'code definingContext'
  classVariableNames: ''
  package: 'Champollion-Core'
```

Since a block is created when the block node is visited we extend the previous block creation to store the current context. Note that this is this context that will be let block access to the temporaries and arguments is use at the moment the block is created.

```
CHInterpreter >> visitBlockNode: aRBBlockNode [
  ^ CHBlock new
    code: aRBBlockNode;
    definingContext: self topFrame;
    yourself
]
```

10.6 Accessing Captured Receiver

Resolving the block variables is a trickier case, as it can be resolved in many different ways. For now we choose to set the correct values and override the incorrect ones in the current frame upon block activation. SD: unclear to me.

This solution will work as far as this primitive does not fail. We leave for the reader to think what happens in such a case.

The first variable we want to provide access to from a block is `self` which is the original receiver of the method at the time the block was created. The following method is worth explaining

- First we grab the block itself. It is simple since the method `primitiveBlockValue` is executed during the evaluation of the message `value` sent to a block. Therefore `self receiver` returns the block currently executed.
- Second remmeber that self in a block refers to the receiver of the method at the time the block was created. So we need to set as receiver the receiver that we found in the context of the block creation. This is what `theBlock definingContext receiver` is returning.
- Finally we are evaluating the block body.

```
CHInterpreter >> primitiveBlockValue [
  | theBlock |
  theBlock := self receiver.
  self receiver: theBlock definingContext receiver.
  ^ self visitNode: theBlock code body
]
```

```
CHInterpreter >> receiver: aValue [
  ^ self tempAt: #self put: aValue
]
```

Note that in the `primitiveBlockValue` we use the frame of message `value` execution. The evaluation of the block body uses this frame. When the evaluation is done such frame is simply pop as any other method evaluations (See `executeMethod:withReceiver:andArguments:`), therefore there is no worries to be made when we changed the value of receiver.`receiver` is not a state of the interpreter but refer to the current frame.

Now that we can correctly resolve the receiver, instance variable reads and writes should work properly too. We leave as an exercise for the reader to verify their correctness.

10.7 Looking up Temporaries in Lexical Contexts

A problem we have not solved yet involves the reads and writes of temporary variables that are not part of the current frame. This is the case when a block tries to access a temporary of a parent lexical scope, such as another surrounding scope, or the home method. The method `increaseEnclosingTemporary` is an example of such a situation: the block `[ temp := temp + 1 ]` will access during its execution the temporary variable that was defined outside of the block. Note that the execution of the block could happen in another method and still be block should be able to access the temporary variable `temp`.

Our next scenario checks that blocks can correctly read and write temporaries of their enclosing contexts. In our test, the enclosing environment creates a temporary. The block reads that value and increases it by one. When the block executes and returns, the value of its temporary should have been updated from `0` to `1`.

```
CHInterpretable >> increaseEnclosingTemporary [
  | temp |
  temp := 0.
  [ temp := temp + 1 ] value.
  ^ temp
]

CHInterpreterTest >> testIncreaseEnclosingTemporary [
  self assert: (self executeSelector: #increaseEnclosingTemporary)
    equals: 1
]
```

should add some diagrams here This scenario is resolved by implementing a temporary variable lookup in the block's *defining* context. Of course, a block could be defined inside another's block context, so our lookup needs to be lookup through the complete context chain. The lookup should stop when the current lookup context does not have a defining context i.e., it is a method and not a block.

To simplify temporary variable lookup we define first a helper method `lookupFrameDefiningTemporary:` that returns the frame in which a temporary is defined. This method returns a frame. It has to walk from a frame to its defining frame up to a method. However, so far the only object in our design knowing the defining frame is the block (via its instance variable `definingContext`), and we do not have any way to access a block from its frame.

One possibility is to store a block reference in its frame when it is activated, and then go from a frame to its block to its defining frame and continue the lookup. Another possibility, which we will implement, is to directly store the defining context in the frame when the block is activated.

```
CHInterpreter >> primitiveBlockValue [
  | theBlock |
  theBlock := self receiver.
  self receiver: (theBlock definingContext at: #self).
  self tempAt: #__definingContext put: theBlock definingContext.
  ^ self visitNode: theBlock code body
]
```

```
CHInterpreter >> lookupFrameDefiningTemporary: aName [
  | currentLookupFrame |
  currentLookupFrame := self topFrame.
  [ currentLookupFrame includesKey: aName ]
    whileFalse: [ currentLookupFrame := currentLookupFrame at:
```

```
    #__definingContext ].
  ^ currentLookupFrame
]
```

should add some diagrams here Now we need to redefine temporary reads and writes. Temporary reads need to lookup the frame where the variable is defined and read the value from it. This is what what the method `visitTemporaryNode:` does.

```
CHInterpreter >> visitTemporaryNode: aTemporaryNode [
  | definingFrame |
  definingFrame := self lookupFrameDefiningTemporary: aTemporaryNode
    name.
  ^ definingFrame at: aTemporaryNode name
]
```

Temporary writes are similar to read. We need to lookup the frame where the variable is defined and write the value to it.

```
CHInterpreter >> visitAssignmentNode: aRBAssignmentNode [
  | rightSide |
  rightSide := self visitNode: aRBAssignmentNode value.
  aRBAssignmentNode variable variable isTempVariable
    ifTrue: [ | definingFrame |
      definingFrame := self
        lookupFrameDefiningTemporary: aRBAssignmentNode variable
    name.
      definingFrame at: aRBAssignmentNode variable name put:
    rightSide ]
    ifFalse: [ aRBAssignmentNode variable variable
          write: rightSide
          to: self receiver ].
  ^ rightSide
]
```

10.8 Block Non-Local Return

We have seen so far that blocks implicitly return the value of their last expression. For example the method `lastExpression` will return 43.

```
CHInterpretable >> lastExpression
  | tmp |
  tmp := 1.
  tmp := true ifTrue: [ tmp := 42. tmp := tmp + 1].
  ^ tmp
```

Now this is a complete different story when a block contains an explicit return statement. Return statements, instead, break the execution of the defining method, namely the home method, and return from it. For example, let's consider a method using `ifTrue:` to implement a guard which should stop the method execution if the guard fails:

```
CHInterpretable >> methodWithGuard
  true ifTrue: [ ^ nil ].
  ^ self doSomethingExpensive
```

put a figure here to show the stack, the blocks, their relationships. When executing this method, the message `doSomethingExpensive` will never be executed. The execution of the method `methodWithGuard` will be stopped by the return statement in the block `[^ nil]`.

More precisely, the block is not activated by `methodWithGuard`. `methodWithGuard` executes the message `ifTrue:` which in turn activates the `[^ nil]`. Still, this block knows the context of `methodWithGuard` as its defining context. When the block executes, the return statement should not return `nil` to the `ifTrue:` context: it should return *from* `methodWithGuard` with the `nil` value, as if it was the return value of the method. Because of this, we call such return inside blocks "non-local returns", because they return from a non-local context, its home context.

The block may have been passed around, when the block executes a return statement, it will return from the method that created the block. We say that the execution quits the home context of the block (the context of the method that defined it).

To implement non-local returns, we will first start by defining a new helper method: `homeFrameOf:` that returns the home frame of a frame. The home frame is the frame that has a defining context. Note that the home frame of a normal method frame is itself.

```
CHInterpreter >> homeFrame [
  | currentLookupFrame |
  currentLookupFrame := self topFrame.
  [ currentLookupFrame includesKey: #__definingContext ]
    whileTrue: [ currentLookupFrame := currentLookupFrame at:
    #__definingContext ].
  ^ currentLookupFrame
]
```

add a diagram A simple way to implement non-local returns in Pharo is by using exceptions: exceptions unwind automatically the call-stack, thus short-circuiting the execution of all methods automatically.

We define a new exception called `CHReturn`. It refers to the home frame and a value.

```
Error subclass: #CHReturn
  instanceVariableNames: 'value homeFrame'
  classVariableNames: ''
  package: 'Champollion-Core'
```

```
CHReturn >> homeFrame [
  ^ homeFrame
]
```

```
CHReturn >> homeFrame: aFrame [
  homeFrame := aFrame
]
```

```
CHReturn >> value [
  ^ value
]
```

```
CHReturn >> value: aValue [
  value := aValue
]
```

When we activate a method we then need to prepare ourselves to catch the exception indicating a return, and only manage it if the return is targetting the current method's context:

SD: we should explain more the `returnFrom homeFrame = thisFrame`

```
CHInterpreter >> execute: anAST withReceiver: anObject andArguments:
    aCollection [
  | result thisFrame |
  thisFrame := self pushNewFrame.

  self tempAt: #__method put: anAST.
  self tempAt: #self put: anObject.
  anAST arguments with: aCollection
    do: [ :arg :value | self tempAt: arg name put: value ].

  result := [ self visitNode: anAST ]
    on: CHReturn        "A return statement was executed"
    do: [ :return |
      return homeFrame = thisFrame
        ifTrue: [ return value ]
        ifFalse: [ return pass ] ].

  self popFrame.
  ^ result
]
```

When we visit a return we raise a return exception and we pass the context. SD: need more explanation.

```
CHInterpreter >> visitReturnNode: aReturnNode [
  CHReturn new
    value: (self visitNode: aReturnNode value);
    homeFrame: self homeFrame;
    signal
]
```

10.9 Conclusion

In this chapter we have extended our evaluator with block closures. Our block closure implementation required adding a kind of object to our runtime, `CHBlock`, to represent blocks containing some AST. Then we refined our evaluator to define a block evaluation primitive, and correctly set up the lexical context. Our lexical context implementation gives blocks access to the defining context's receiver and temporaries. We then shown a first implementation of non-local returns, using exceptions to unwind the stack.