Chapter 1

Pharo In a Nutshell

Programming is an important skill to properly tell a computer what it has to do. This chapter gives an introduction of the Pharo programming language. Although we tried to make this chapter smooth and easy to read, having some basic programming knowledge is expected.

Pharo is an object-oriented programming language, class-based, and dynamically typed. The chapter therefore begins with a brief introduction of what programming with objects is all about. The focus will subsequently moves toward Pharo.

1.1 Going over an example, Step-by-step

Instead of giving a long rhetorical description about object-orientation, let us pick a simple example. The following code snippet opens a Roassal view with a label showing 'Hello World':

```
v := RTView new.
v add: (RTLabel elementOn: 'Hello World').
v
```

To execute the script given above, you need to type it in a playground, and press the green triangle (Figure 1.1). This script displays the message Hello World. The word RTView refers to a class. A class is an object factory and its name is easily recognizable because of its first letter, which is always a capital letter. A class is like a baking pan for cakes: creating an object is like backing a cake. All the cakes produced by a pan have the same physical aspects, but attributes, such as ingredients, may vary.

The first line creates an object view. An object is created using the message new. For example, the expression RTView new creates a view, String new

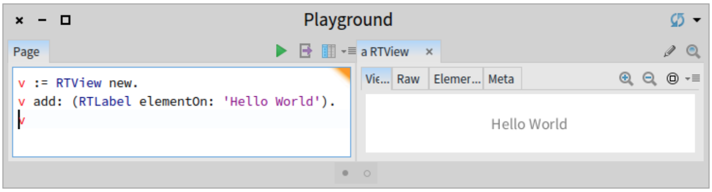

Figure 1.1: Hello World.

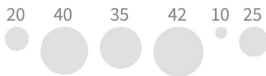

Figure 1.2: Circles are lined up.

creates an empty string character, Color new creates a black color. The view, produced by executing RTView new, is said to be the object (for instance) produced by RTView.

The expression object asString sends the message asString to an object, referenced by the variable object. In Pharo, a class is also an object, which means that objects are created by sending a message to a class. The message new is sent to the class RTView, which has the effect of creating an object. This object is assigned to the variable v using the operator := .

In the second line, the message elementOn: is sent to the class RTLabel. An argument is provided to that message, which is a string character 'Hello World' as argument. These Roassal instructions simply creates an element that has a shape label. That element is passed as argument to the message add:. The effect of add: is simply to add the element in the view. The view, referenced by v, understands the message add: because the class RTView defines a method add:.

Consider this script (Figure 1.2):

```
values := #(20 40 35 42 10 25).
v := RTView new.
elements := (RTEllipse new size: #yourself) elementsOn: values.
v addAll: elements.
RTHorizontalLineLayout on: elements.
elements @ RTLabeled.
v
```

This example renders 6 circles, each having a proper size. The expression #(20 40 35 42 10 25) defines an array, containing a few numbers. The expression RTEllipse new size: #yourself creates an object of the class RTEllipse by sending the message new. The message size: is sent to that ellipse object, with the symbol #yourself as argument. This message size: configures the size of the ellipses: The size of each circle is computed with the model object when creating the element. In particular, the message #yourself will be sent to each element of the array values. For example, the size of the circle representing the value 35 has a diameter of 35 pixels. Circles are then lined up using a dedicated layout, invoked by sending the message on: to it with the elements as argument. The expression elements @ RTLabeled labels each elements.

Most visualization engines and data analysis environments operate on the principle illustrated above: scripts are typed in a workspace or a webpage, and executed to produce a visualization. This approach to build a visualization or a small application is appealing since it is self-contained: all one needs to know is within the linear script and the expressed logic is made explicit. However, this way of developing software artifacts has serious limitations. Maintenance and evolution are seriously diminished. For example, a 200-line long script is painful to modify and confusing to look at. If not properly structured, adapting a complex visualization may have the fantastic ability to consume a ridiculously large amount of time. This is a situation well known to journalists, data scientists, and also software engineers. Fortunately, a couple of decades ago the Software Engineering research community produced a way of programing that is able to cope with the inherent complexity of software artifact development. Object-oriented programming is the most successful way to handle complex and large development.

1.2 Pillars of Object-oriented programming

Object-oriented programming simplifies the programming activity. Handling objects, instead of functions or code snippets, uses a metaphor that is familiar to us, humans: an object may react upon some actions, have a behavior on its own, and may hide details about how it is physically built.

Let us bring a bit of theory in all this. There are five essential ingredients to an object-oriented system:

- *Encapsulation*: in our daily life, we are used to handling information that is not publicly accessible: social security numbers, body weight, just to name a few. Encapsulation in object-oriented systems is about letting objects have private information. Private information may reflect detail that is not directly necessary to a service consumer. In case private information has to be publicly exposed, *asking a question* is a polite and cordial way to access it. In object-oriented programming, *asking a question* or *giving an order* is called *sending a message*. Encapsulation in object-oriented programming enables abstractions and information hiding.

- *Composition*: a complex problem may be solved by cutting it down into smaller problems, hopefully easier to solve.
- *Distribution of responsibility*: in our daily life, each of us has duties and responsibilities. Having a clear separation of concerns is key to having a good object-oriented design. This is what makes systems easy to understand and maintain. For example, instead of asking someone's weight in order to select what may be eaten, it is better for everybody to let that person make a responsible choice. This example is not far stretched: many difficulties in software maintenance are directly rooted from improperly assigned responsibilities in software.
- *Message sending*: electronic mails are the base of our daily communication: a person, called the *sender*, sends an email to another person, called the *receiver*. In object-oriented programming, objects communicate in a similar fashion: computation is carried out by sending messages between objects. An object sends messages to other objects. After sending a message, a reply is returned. In object-orientation, sending a message is often perceived as a way to delegate responsibility.
- *Inheritance*: general concepts have to be specialized to address particular requirements. Inheritance allows one to define conceptual hierarchy, reuse code, and support polymorphism. Inheritance may say that an ellipse and a square are two graphical shapes.

These five pillars are not particularly tied to a programming language. So, in theory, it is perfectly doable to have an object-oriented design in a procedural language such as C. However, having a programming language that enforces these principles greatly alleviates the task of the programmer.

There are numerous object-oriented languages around and Pharo is one of them. Pharo differs from other languages by offering an homogeneous way of expressing computation: everything is an object, therefore computation happens by sending messages. When objects are taken seriously, there is no need for primitive types (*e.g.*, `int`, `float`), language operators, and even external compilable files! Considering only message sending significantly reduces the amount of technological details associated with the program execution that most mainstream programming languages unnecessarily expose.

1.3 Message sending

Sending a message is the elementary unit of computation. Understanding how to send a message is key to feel comfortable in Pharo. Consider the expression:

```
'the quick brown fox jumped over the lazy dog' replaceAllRegex: 'fox' with: 'cat'
```

This expression sends to the string object 'the quick brown fox jumped over the lazy dog' a message having the name #replaceAllRegex:with: and two arguments, 'fox' and 'cat', themselves two string objects. The result of sending this message is 'the quick brown cat jumped over the lazy dog', another string.

In Pharo, a character string (often simply called a string) is a collection of characters written between two accents (*e.g.,* 'fox'). A string is a plain object, which means one can send messages to it. A message is composed of two essential ingredients: a name and a collection of arguments. It may be that the set of arguments is empty. For example, the expression 'fox' asUppercase, which evaluates to 'FOX', sends the message #asUppercase to the string 'fox'. No arguments are involved here.

Message sending is at the heart of the Pharo language, and is therefore well expressed within its syntax. There are three kinds of message sending:

- *Unary message*: a unary message is a message that does not take any argument. The expression 'fox' asUppercase sends a unary message to the string 'fox'.

- *Binary message*: a binary message has exactly one argument and its name is not made of alphanumerical characters. Instead one or two characters are common for binary messages, such as +, /, -, <, >>. The expression 2 + 3 sends to the object 2 a binary message named + with the argument 3. You may notice that this expression has therefore a different semantic than 3 + 2, although the result is obviously the same. Note that the expression 3 + 2 * 2 returns 10, and not 7 as one may expect. If you wish to enforce mathematical priorities in arithmetic operations, use parenthesis, as in 3 + (2 * 2).

- *Keyword message*: a keyword message is a message that is neither unary nor binary. A keyword message accepts one or more object arguments. Consider the example 'the quick brown fox jumped over the lazy dog' includesSubstring: 'fox'. This expression evaluates to true. The name of the keyword message is #includesSubstring: and the argument is 'fox'. Each argument is preceded by a keyword. For example, the message replaceAllRegex: 'fox' with: 'cat' contains two keywords and therefore two arguments. Arguments are inserted within the message name.

Sending a message triggers a mechanism that searches for a method to execute. This mechanism, often called "method lookup", begins from the class of the object up and goes to the superclass if not found.

1.4 Object creation

An object is a bundle of data to which messages can be sent to. An object is created most of the time by sending the new message to a class. This is revealing the true nature of classes, being an object factory. A class may produce as many different objects as new is sent to it. Objects produced from a unique class are different but understand the same set of messages and have the same variables. Differences between two or more objects issued from the same class are the the values given to these variables. For example, consider the following expression:

```
Object new == Object new
```

This expression sends three messages, twice the message new and once the message ==, used to compare object identities. The expression evaluates to false, since the two objects are different, *i.e.,* located at different physical memory location.

The expression Point new creates a point by sending the message new to the class Point. There are several ways to create a point:

- Point new creates a point *(0, 0)*. All classes in Pharo understand the message new. Except when explicitly prohibited, an object is created by sending new to the class.

- Point x: 5 y: 10 creates a point *(5, 10)*. This expression sends the message x:y: to the class Point, with 5 and 10 as arguments. The class Point defines the class method x:y:. The difference between new and x:y: is that the latter allows one to create and initialize a point with a given value for x and y.

- 2 @ 3 sends to the object 2 the message named @ with the argument 3. The effect is the same than Point x: 2 y: 3, which is to create the point *(2, 3)*.

Each class has its way to create objects. For example, a point is not created the same way as is a color. Creating an object is also commonly mentioned as "instantiating a class" and an object is often referenced as "instance".

A class is an object factory and an object is necessarily created from a class. An object associates values to the attributes defined by the class of the object. As discussed above, objects interact by sending messages. An object is able to understand messages corresponding to methods defined in its class, and methods defined in the chain of superclasses.

1.5 Class creation

A class is a factory of objects, often regarded as an abstraction of objects. You need to create classes as soon as you wish to bundle logic and data together (*i.e.*, "doing hands on work").

A class belongs to a package. You may want to create a dedicated package to contain the classes you will define. A package is created by right-clicking on the package list in a system browser. We will define a class Tweet:

1. Open a system browser from the World menu
2. Right click on the top left list panel, and define a package called TweetsAnalysis.
3. Create the class Tweet. Classes are created by filling the following template in a code browser:

```
Object subclass: #NameOfSubclass
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'Announcements-Core'
```

The text NameOfSubclass has to be replaced by the name of the class you wish to create. After the keyword instanceVariableNames: you need to provide the instance variables, and after classVariableNames: the class variables. Right click on the code and select the menu **accept** to effectively create the class. You should have

```
Object subclass: #Tweet
   instanceVariableNames: 'content sender date'
   classVariableNames: ''
   package: 'TweetsAnalysis'
```

You should obtain something similar to Figure 1.3. We have defined the class Tweet, contained in the package TweetsAnalysis. The class contains three instance variables, content, sender, and date. No methods have been defined so far. Note that in Pharo, an instance variable name begins with a minuscule letter.

1.6 Method creation

A method is an executable piece of code. A method is composed of instruction statements typically aiming to carry out a computation. We will define

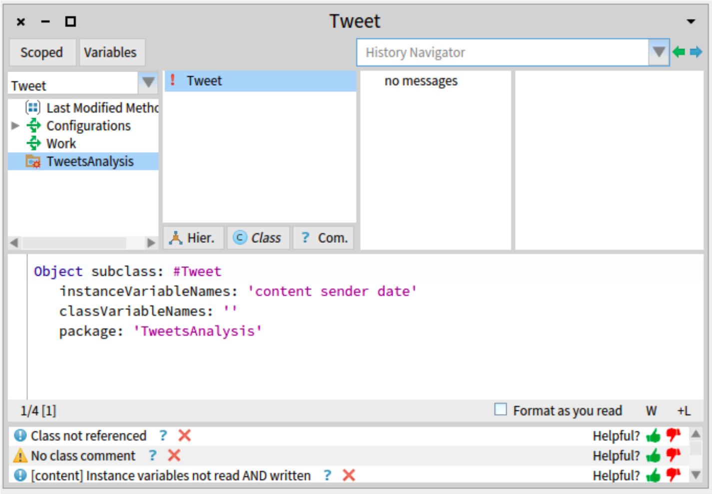

Figure 1.3: The class Tweet.

a small mathematical example to illustrate the creation of a method. We will therefore leave out our Twitter example for a short while.

The Fibonnacci sequence is a well known sequence of numbers obtained with the formula F(n) = F(n-1) + F(n-2). Terminal cases are given with F(0) = 0 and F(1) = 1.

We will implement the Fibonacci formula as a method defined on the class Integer. This class describes all the integer numbers in Pharo. First, let us open a system browser on this class. Spotter is a tool for searching in Pharo (Figure 1.4). We will therefore search for the Integer class and opens a system browser on it.

Enter Integer in Spotter and select the corresponding class by pressing the Enter key or clicking on it using the mouse. Select the arithmetic protocol (third list panel) and enter the following code in the lower text pane:

```
fibonacci
   self <= 1 ifTrue: [ ^ self ].
   ^ (self – 1) fibonacci + (self – 2) fibonacci
```

After having entered the code, right click on it and select **Accept**. Accepting a method compiles it and makes it executable.

Open a playground, type and execute 10 fibonacci. You will see 55, its result (Figure 1.5).

The self word refers to a pseudo-variable that designates the object having received the message. When executing the expression 10 fibonacci, self

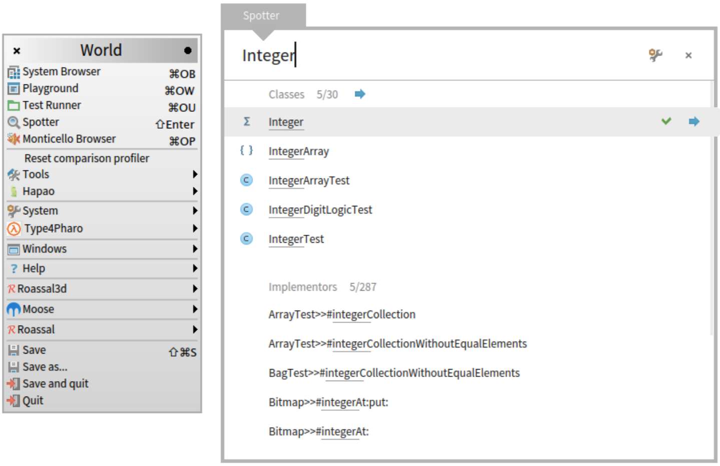

Figure 1.4: The World menu and Spotter.

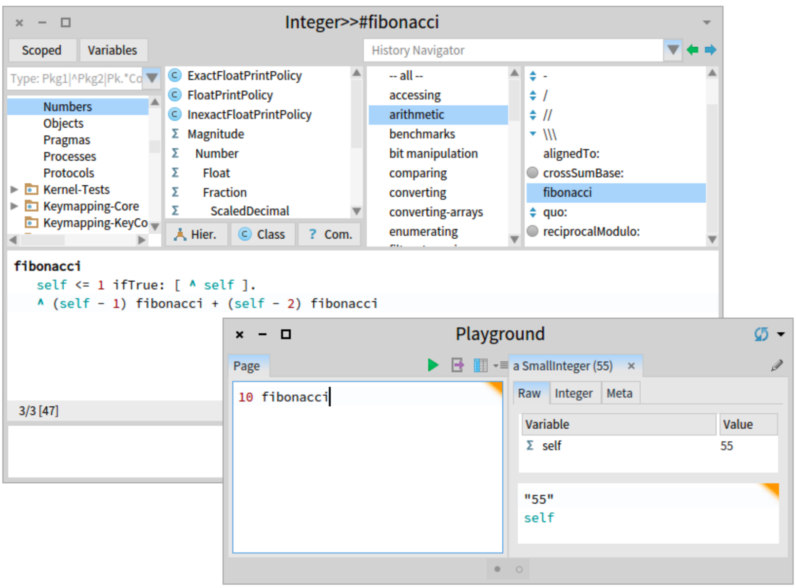

Figure 1.5: Fibonacci of 10.

refers to the object 10. The expression self <= 1 is true if self is either 1 or smaller. If this is the case, then we exit the method with ifTrue: [↑ self]. The caret character (↑) is a return statement: it exits the method and returns a value. If self is greater or equals 2, then the result is the sum of (self - 1) fibonacci and (self - 2) fibonacci.

Another common pseudo-variable is super. The two pseudo-variables self and super reference the same object, the object that has received a message. The unique difference between self and super is characterized when when one sends a message to it, in particular:

- sending a message to self triggers the method lookup from the class of the object,
- sending a message to super triggers the method lookup from the superclass of the class that contains the call on super.

Coming back to our Tweet example. Define the following six methods on the class Tweet:

```
date
   ^ date

date: aDate
   date := aDate

content
   ^ content

content: aContent
   content := aContent

sender
   ^ sender

sender: aSender
   sender := aSender
```

These methods will enable one to set the content of a tweet and query about it.

Click on the **Class** button in the system browser. Clicking on it switches the system browser to the **class side**: methods defined on that side are class methods of the class Tweet. Define the method (Figure 1.6):

```
createFromURL: urlAsString
   "Method to be defined on the CLASS side"
   | content lines sender date |
   content := (ZnEasy get: urlAsString) contents readStream.
```

```
    lines := content contents lines collect: [ :l |
        | firstCommaIndex secondCommaIndex |
        firstCommaIndex := l indexOf: $,.
        secondCommaIndex := l indexOf: $, startingAt: (firstCommaIndex + 1).
        sender := l copyFrom: 1 to: (firstCommaIndex − 1).
        date := l copyFrom: (firstCommaIndex + 1) to: (secondCommaIndex − 1).
        content := l copyFrom: (secondCommaIndex + 1).
        { sender . date . content }
    ].
^ lines collect: [ :line |
        Tweet new
            sender: line first;
            date: line second;
            content: line third
    ]
```

The method createFromURL: fetches a CSV file we have prepared for that example. The file contains 1000 random tweets. It does a simple parsing of the content by identifying the comma.

Next, you can define the method:

```
createFromExample
    "Method to be defined on the CLASS side"
    ^ self createFromURL: 'http://bit.ly/exampleTweetCSV'
```

The url http://bit.ly/exampleCSV is an example we have prepared to illustrate our purpose. Open it in a web browser to see what it looks like. At that stage, evaluating the expression Tweet createFromExample returns a list of 1000 tweet objects, each tweet describing an entry of the online CSV file.

We will define two new methods on the class Tweet. Switch to the instance side (*i.e.,* unselect the **Class** button in the system browser), and define the following two instance methods:

```
words
    ^ self content substrings

isSimilarTo: aTweet
        ^ (self words intersection: aTweet words) size >= 6
```

The method words simply returns all the words defining the content of a tweet. It uses substrings which returns a list of words from a string. For example, the expression 'fox and dog' substrings return #('fox' 'and' 'dog'). The method isSimilarTo: takes as argument another tweet and returns true or false whether the tweet argument is similar to the tweet that receives the message isSimilarTo:. The notion of similarity we use here is: two tweets are similar if they have at least 6 words in common.

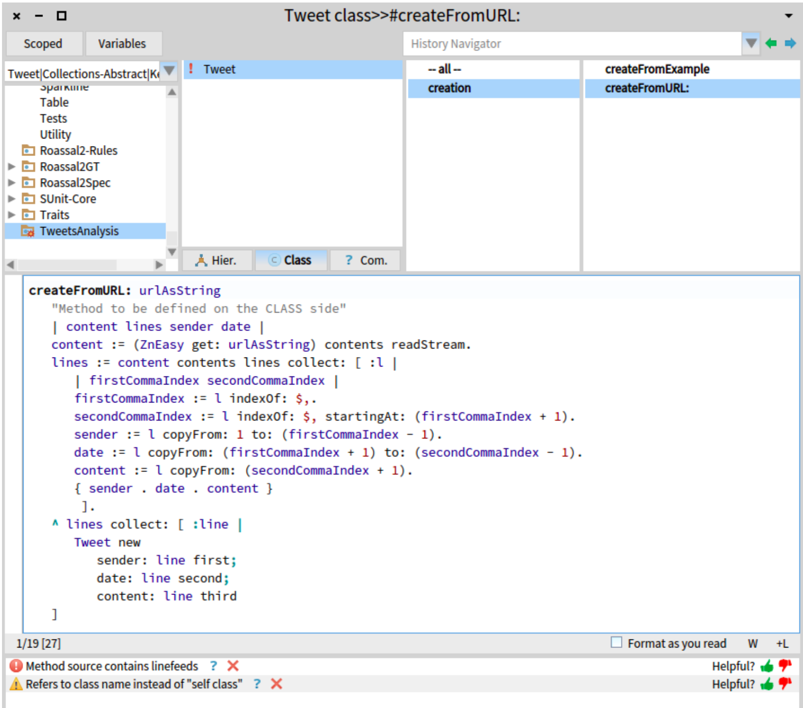

Figure 1.6: The method createFromURL:.

So, we have some objects and a way to establish a relation between them. This is more than enough to start to visualize them. Open a playground and type (Figure 1.7):

```
tweets := Tweet createFromExample.
positive := tweets
    select: [ :t | #('great' 'cool' 'super' 'fantastic' 'good' 'yes' 'okay' 'ok') includesAny:
    t words  ].

negative := tweets
    select: [ :t | #('bad' 'worse' 'down' 'no') includesAny: t words  ].

b := RTMondrian new.
b shape circle
    if: [ :t | positive includes: t ] color: Color blue;
    if: [ :t | negative includes: t ] color: Color red.
b interaction popupText: #content.
b nodes: positive, negative.
b edges connectToAll: [ :tweet |
        tweets select: [ :t | t isSimilarTo: tweet ] ].
```

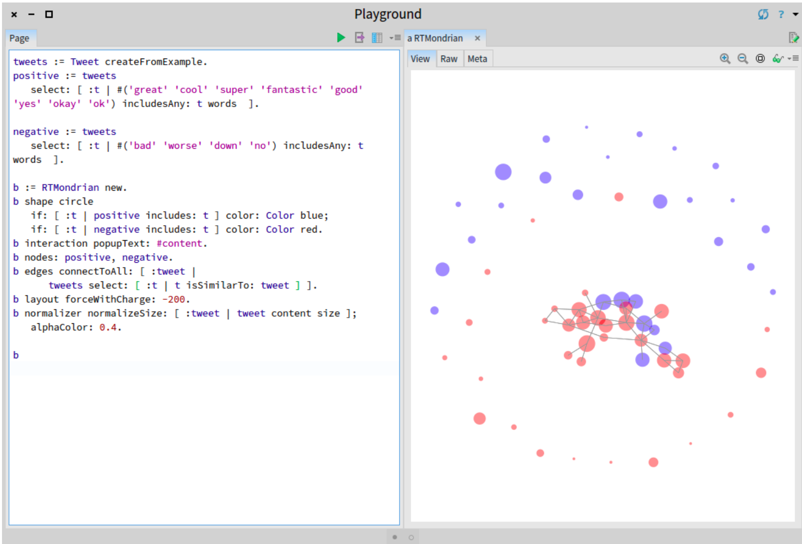

Figure 1.7: Positive and Negative tweets.

```
b layout forceWithCharge: −200.
b normalizer normalizeSize: [ :tweet | tweet content size ];
   alphaColor: 0.4.

b
```

The code given above does a very primitive classification of feeling. Tweets with a positive feeling are blue, while the negative ones are red. Among the 1000 tweets, only 60 have a feeling.

We see that only a few of the tweets have actually common words and most of them are negative.

1.7 Block closure

A block closure (also simply called "block") is a piece of code associated to an environment. A block is manipulable, as any object is (*i.e.*, a block may therefore be provided as message argument and be assigned to a variable). The expression [:value | value + 5] is a block closure that takes one parameter and return the sum between that argument and 5. This block may be evaluated with an argument using the message value:. Consider the following code snippet:\

```
b := [ :value | value + 5 ].
b value: 10. "Return 15"
b value: −5. "Return 0"
```

Recall the definition of the fibonacci method, defined on the class Integer:

```
fibonacci
   self <= 1 ifTrue: [ ^ self ].
   ^ (self − 1) fibonacci + (self − 2) fibonacci
```

The message ifTrue: takes a block [↑ self] as argument. In case that self <= 1 evaluates to true, the block is evaluated and triggers an early exit of the method. The expression ↑ self exits the method. The block uses the pseudo-variable self. A block may access variables defined in the outer lexical scope. A block may use temporary variables, instance variables, and argument variables.

1.8 Control Structure

As illustrated in the Fibonacci example, a condition is expressed using the ifTrue:ifFalse: message. Obviously, it expects to have a boolean as receiver. This message takes two blocks as argument, the first one is evaluated in case the boolean receiver is true, or the second block is evaluated in case the receiver is false. Variant exists such as ifTrue: and ifFalse:. For example, true ifTrue: [5] evaluates to 5. The receiver can naturally be a combination of boolean expression such as (5 < 1) ifFalse: ['5 is not less than 1'].

1.9 Collection

A Collection is a very common data structure. As previously illustrated, the expression #(23 42 51) defines an array, instance of the class Array. This class, and its superclasses, defines a large number of methods. Two operations are very common in Pharo: transformation and filtering of collections.

A transformation is typically realized using collect:. For example, #(23 42 51) collect: [:v | v > 30] returns #(false true true). The initial array of numbers is transformed as an array of booleans.

Filtering is carried out using select:. For example, #(23 42 51) select: [:v | v > 30] returns #(42 51).

Both collect: and select: takes a block as argument. In case of select:, the block has to evaluate to a boolean.

Collections in Pharo are rooted into the Smalltalk programming language, and is often an inspiration for other programming languages. Pharo's collections are rich and expressive. We have just seen the example of Array. Another collection is OrderedCollection representing an expandable collections. Elements may be added and removed during the program execution. For example:

```
v := OrderedCollection new.
v add: 23.
v add: 42.
v add: 51.
elements := (RTBox new size: #yourself) elementsOn: v.
RTVerticalLineLayout on: elements.
RTView new
   addAll: elements;
   yourself
```

This small script shows three squares.

Another useful collection is Dictionary. A dictionary stores pairs of keys and values. For example, consider the following code snippet:

```
d := Dictionary new.
d at: #one put: 1.
d at: #two put: 2.
d at: #three put: 3.
```

The expression d at: #two returns the value 2.

1.10 Cascade

The last bit of syntax is yet to be described. A cascade allows one to send several messages to the same object receiver. For example, instead of writing:

```
v := OrderedCollection new.
v add: 23.
v add: 42.
v add: 51.
```

One could write:

```
v := OrderedCollection new.
v
   add: 23;
   add: 42;
   add: 51.
```

The cascade, noted ;, is a syntactic construction to make code more concise by avoiding text duplication. It is frequently used in this book.

1.11 A bit of Meta-programming

Pharo provides an expressive reflective API, which means one can programmatically get data about how Pharo code is structured and defined. Consider the following expression `RTShape methods size`. This expression returns the number of methods that the class `RTShape` defines. The message `methods` is sent to the class `RTShape`, which is also an object in Pharo. This message returns a collection of the methods defined on the class `RTShape`.

Many examples contained in Agile Visualization visualize software source code and therefore use the reflective API. Source code is convenient to illustrates visualization because it is already available (no need to rely on external data) and is complex enough to deserve to be visualized.

1.12 Summary and Further Reading

This chapter gave a brief introduction to object-oriented programming. From now on, you should be able to understand Pharo syntax. We recommend a number of books to further discover the World of Pharo:

- Pharo by Example: http://pharobyexample.org
- Deep Into Pharo: http://deepintopharo.com

Pharo is a beautiful, elegant, and simple language. Pharo has a small and concise syntax, which makes it each to learn. Its programming environment is also highly customizable.

Building a sophisticated visualization or any non-trivial software artifact often face complex development. Mastering object-orientation is not strictly necessary in order to use Roassal. However, having a good command of object-oriented programming will considerably alleviate development and maintenance effort.

Pharo offers a powerful *meta* architecture. Do you remember that an object is created by sending the message `new` to a class? In Pharo a class is also an object since we send `new` to it, as in the expression `Color new`. A class is therefore an object, itself an instance of another class, called a metaclass. And it does not stop here. A metaclass is also an object. Methods are also objects, a collection of bytecodes. Many parts of Pharo are truly beautiful, but going into more detail is out of the scope of this book.