CHAPTER 6

Some collection katas with words

This chapter proposes some little challenges around words and sentences as a way to explore Pharo collections.

6.1 Isogram

An isogram is a word or phrase without a repeating letter. The following words are examples of isograms in english and french:

- egoism, sea, lumberjacks, background, hacking, pathfinder, pharo
- antipode, altruisme, absolument, bigornaux

Isograms are interesting words also because they are often the basis of simple cifers. Isograms of length 10 are commonly used to encode numbers. This way salespeople of products can get access to the original cost of the product and control their sale.

Using the *pathfinder* cipher we can decide that *p* represents the number 1, *a* represents the number 2 and so on. The price tag for an item selling for 1100 Euros may also bear the cryptic letters *frr* written on the back or bottom of the tag. A salesman familiar with the pathfinder cipher will know that the original cost of the item is 500 Euros and he can control his sale.

Since we will essentially manipulate strings, let us start with some basic knowledge on strings.

6.2 About strings

A string in Pharo is in fact an array of characters. We can access string elements using the message `at: anIndex`. Since all collections in Pharo have their first elements at index 1, the message `at: 1` returns the first element of a string.

```
'coucou' at: 1
>>> $c
```

```
'coucou'at: 3
>>> $u
```

As with any collection, we can apply iterators such `select:`, `do:`, or `collect:`. Here we select all the characters that are after, hence bigger, than character `$m`.

```
'coucou' select: [ :aChar | aChar > $m ]
>>>'ouou'
```

We can also apply all kinds of operations on the collection. Here we reverse it.

```
'coucou' reverse
>>>  'uocuoc'
```

We can also find the index of a string inside another one using the message `findString: aString startingAt: anIndex`.

```
'coucou' findString: 'ou' startingAt: 1
>>> 2
```

```
'coucou' findString: 'ou' startingAt: 3
>>> 5
```

We simply present some of the possible messages that can be sent to a string. We select some that you can use in the following or in the next chapter. Now let us solve our problem to identify if a string is an isogram.

6.3 A solution using sets

We can do a simple (and not really efficient) implementation using sets. Sets are collections that only contain one occurence of each element added to them. Adding twice the same element only adds one.

Note that sets in Pharo can contain any objects, even sets themselves. This is not the case in all languages. In Pharo, there are no restriction about set elements.

You can convert a string into a set of characters sending the string the message `asSet`.

```
'coucou' asSet
>>> a Set($u $c $o)
```

Now this is your turn: Imagine how using a set of characters, you can determine if a string is a isogram.

Hints

If the size of a set with the contents of a string and this string are the same, it means that the string does not contain any letter twice! Bingo we can simply identify an isogram.

To get the size of a collection use the message `size`

```
'coucou' size
>>> 6
```

Now we convert `'coucou'` into a set using the message `asSet`.

```
'coucou' asSet size
>>> 3
```

Note that the message `asSet` is equivalent to the following script:

```
| s1 |
s1 := Set new.
'coucou' do: [ :aChar | s1 add: aChar ].
s1
>>> a Set($u $c $o)
```

- Here we define a variable `s1`
- We iterate over all the characters of the string `'coucou'`, and we add each character one by one to the set `s1`.
- We return the set.
- The set contains only three elements `$c`, `$o`, `$u` as expected.

Checking expression

So now we can get to the expression that verifies that `'pharo'` is an isogram.

```
| s |
s := 'pharo'.
s size = s asSet size
>>> true
```

And that `'phaoro'` is not!

```
| s |
s := 'phaoro'.
s size = s asSet size
```

```
>>> false
```

Adding a method to the class String

Now we can define a new method to the class `String`. Since you may propose multiple implementations, we postfix the message with the implementation strategy we use. Here we define `isIsogramSet`

```
String >> isIsogramSet
  "Returns true if the receiver is an isogram, i.e., a word using no
    repetitive character."
  "'pharo' isIsogramSet
  >>> true"
  "'phaoro' isIsogramSet
  >>> false"

  ... Your solution here ...
```

And we test that our method is correct.

```
'pharo' isIsogramSet
>>> true
```

```
'phaoro' isIsogramSet
>>> false
```

Wait! We do not want to have to check manually all the time!

> **Important** When you verify two times the same things, better write a test! Remember you write a test once and execute it million times!

6.4 Defining a test

To define tests we could have extended the `StringTest` class, but we prefer to define a little class for our own experiment. This way we will create also a package and move the methods we define as class extension to the that package.

> **Important** To define a method as a class extension of package `Foo`, just name the protocol of the method `*Foo`.

We define the class `GramCheckerTest` as follow. It inherits from `TestCase` and belong to the package `LoopStarGram`.

```
TestCase subclass: #GramCheckerTest
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'LoopStarGram'
```

Now we are ready to implement our first automated test for this chapter.

Test methods are special.

- A test method should start with `'test'`.
- A test method is executed automatically when we press the icons displaying the method.
- A test method can contain expressions such as `self assert: aTrueExpression` or `self deny: aFalseExpression`.

Here

- Our method is named `testIsogramSetImplementation`.
- We check (`assert:`) that 'pharo' is an isogram i.e., that `'pharo' isIsogramSet` returns `true`.
- We check (`deny:`) that 'phaoro' is *not* an isogram i.e., that `'pharo' isIsogramSet` returns `false`.

```
GramCheckerTest >> testIsogramSetImplementation

  self assert: 'pharo' isIsogramSet.
  self deny: 'phaoro' isIsogramSet.
```

Important When you write a test, make sure that you test different situations or results. Why? Because imagine that your methods always return true, you would never be sure that not all the string are isograms. So always check for positive and negative results.

Messages `assert:` and `deny:` are equivalent as follows: assert (something) is equals to deny(something not) and assert (something not) is equivalent to deny (something). Hence the following expressions are strictly equivalent.

```
  self assert: 'phaoro' isIsogramSet not.
  self deny: 'phaoro' isIsogramSet.
```

Testing several strings

Now we do not want to write a test per string. We want to test multiple strings at the same time. For that we will define a method in the test class that returns a collection of strings. Here we create a methods returning an array of isograms.

```
GramCheckerTest >> isograms
  ^ #('pharo' 'pathfinder' 'xavier' 'francois' 'lumberjacks'
    'altruisme' 'antipode')
```

Then we define a new test method `testAllIsogramSet` that simply iterates over the string array and for each verifies using `assert:` that the element is indeed an isogram.

In Pharo, there are multiple ways to express loops on collection, the easiest is to send the message `do:` to the collection. The `do:` message executes the block to each element of the collection one by one.

> **Important** The `do:` message executes its argument taking each elements of the receiver collection one by one. Note the way we express it, we ask the collection to iterate on itself. Note also that we do not have to worry about the size of the collection and the index of an element as this is often the case in other languages.

```
GramCheckerTest >> testAllIsogramSet

  self isograms do: [ :word |
    self assert: word isIsogramSet ]
```

Since we said that we should also tests negative let us to the same for non isograms. We create another method that returns non isogram strings and we enhance our testing method.

```
GramCheckerTest >> notIsograms
  ^ #('phaoro' 'stephane' 'idiot' 'freedom')
```

And we make our test using both.

```
GramCheckerTest >> testAllIsogramSetImplementation

  self isograms do: [ :word |
    self assert: word isIsogramSet ].
  self notIsograms do: [ :word |
    self deny: word isIsogramSet ]
```

6.5 Some fun: Obtaining french isograms

Now we would like to find some isograms in french. http://www.pallier.org/ressources/dicofr/liste.de.mots.francais.frgut.txt contains 330 000 french words one by line. So we would like to get all the lines.

We will use `ZnClient`, the HTTPClient that comes with Pharo. This page returns text that is latin1 (iso-8859-1) encoded, but describes it as 'text/plain' without further qualification. `ZnClient` then assumes the encoding is utf8 (the most reasonable default today). So we have to tell the client which encoding to use using mime types. Mime-types can specify the encoding as follows: `'text/plain;charset=utf8'` or `'text/plain;charset=latin1'`.

Here is how to override the default in `ZnClient`. Since this is a lot of data, execute the expression using the **Inspect It** menu or shocut to get an inspector instead of a simple **DoIt**.

```
| lines |
lines := (ZnDefaultCharacterEncoder
  value: ZnCharacterEncoder latin1
  during: [
    ZnClient new
      get:
    'http://www.pallier.org/ressources/dicofr/liste.de.mots.francais.frgut.txt'
    ]) lines.
```

The expression above will give you an array of 336531 words (it is a bit slow depending on your internet connection because it is lot of data).

Once you get the inspector opened, you can start to play with the data. Make sure that you select `self` and in the text pane you can execute the following expressions:

The first one will select all the words that are isogram and you will see them in the second list that will appear on the right.

```
self select: #isIsogramSet
```

Now you can select again all the isogram longer or equal to 10.

```
self select: [:each | each size >= 10 ]
```

We have other ways to implement isograms and we will discuss such implementations in the next chapter. Now we will play with pangrams.

6.6 Pangrams

The definition of a pangram is the following: *A Pangram or holoalphabetic sentence for a given alphabet is a sentence using every letter of the alphabet at least once.*

Here are examples of english pangrams:

- the five boxing wizards jump quickly
- the quick brown fox jumps over the lazy dog

Let us write a test first. Yes we want to make sure that we will be able to control if our code is correct and we do not want to repeat typing the test.

```
GramCheckerTest >> testIsEnglishPangram

  self assert: 'The quick brown fox jumps over the lazy dog'
    isEnglishPangram.
  self deny: 'The quick brown fox jumps over the  dog'
    isEnglishPangram
```

Imagine a solution

Imagine that we have a collection or string representing the alphabet. A solution is to check that the potential pangram string contains each of the characters of the alphabet, as soon as we see that one character is missing we stop and know that the sentence is not a pangram.

```
'The quick brown fox jumps over the lazy dog' isEnglishPangram
>>> true
'The quick brown fox jumps over the dog' isEnglishPangram
>>> false
```

A first version

Here is a first version. We define a variable `isPangram` that will represent the information we know about the receiver. We set it to true to start. Then we iterate over an alphabet character by character and as soon as the character is not included in the receiver we set the variable to false. At the end we return the variable `isPangram`.

```
String >> isEnglishPangram
  "Returns true is the receiver is a pangram i.e., that it uses all
    the characters of a given alphabet."

  | isPangram |
  isPangram := true.
  'abcdefghijklmnopqrstuvwxyz' do: [ :aChar |
    (self includes: aChar)
      ifFalse: [ isPangram := false ]
    ].
  ^ isPangram
```

This first implementation has a problem. Can you see which one? If the sentence does not contain $a, we will know it immediately still we will look for all the other letters. So this is clearly inefficient.

A better version

Instead for testing all characters, even if we know one is missing, what we should do is to stop looking as soon as we identify that there is one missing character and return the result.

The following definition is doing this and it deserves a word of explanation.

The expression `^ something` returns the value `something` to the caller method. The program execution exits at that point: it does not execute the rest of method. The program execution returns to the method caller. Usually we use `^ something` as last statement of a method when they need to return a special value. Now `^ anExpression` can occur anywhere and in particu-

lar inside a loop. In such a case the loop is stopped, the method execution is stopped and the value is returned.

```
String >> isEnglishPangram
  "Returns true is the receiver is a pangram i.e., that it uses all
    the characters of a given alphabet."

  'abcdefghijklmnopqrstuvwxyz' do: [ :aChar |
    (self includes: aChar)
      ifFalse: [ ^ false ]
    ].
  ^ true
```

Note that we do not need the variable `isPangram` anymore. We return true as last expression because we assume that if the execution arrives to the this point, it means that all the characters of the alphabet are in the receiver, else the execution would have been stopped and false would have been returned.

Important When you define a method returning a boolean value, always think that you should at least return a true and a false value. This sounds like a stupid advice but developing such basic reflex is important.

Important The execution of any expression preceded by a ^ (a caret) will cause the method to exit at that point, returning the value of that expression. A method that terminates without explicitly returning some expression will implicitly return `self`.

6.7 Handling alphabet

A pangram is only valid within a given alphabet. The web site http://clagnut.com/blog/2380/ describes pangrams in many different languages. Now we could follow one gag in Gaston Lagaffe with the 'Il y a des poux. Parmi les poux, il y a des poux papas et des poux pas papas. Parmi les poux papas, il y a des poux papas papas et des poux papas non papas....' and all their descendance. 'les poux papas et les poux pas papas' is not a pangram in french but a pangram in the alphabet 'apouxetl'.

We would like to be able to specify the alphabet to be used to verify. Yes we define a new test.

```
GramCheckerTest >> testIsPangramIn

  self assert: ('The quick brown fox jumps over the lazy dog'
    isPangramIn: 'abcdefghijklmnopqrstuvwxyz').
  self assert: ('les poux papas et les poux pas papas' isPangramIn:
    'apouxetl').
```

You can do it really simply:

```
String >> isPangramIn: alphabet
  "Returns true is the receiver is a pangram i.e., that it uses all
    the characters of a given alphabet."
  "'The quick brown fox jumps over the lazy dog' isPangramIn:
    'abcdefghijklmnopqrstuvwxyz'
  >>> true"
  "'tata' isPangramIn: 'at'
  >>> true"

  ... Your solution ...
```

```
String >> isEnglishPangram
  "Returns true is the receiver is a pangram i.e., that it uses all
    the characters of a given alphabet."
  "'The quick brown fox jumps over the lazy dog' isEnglishPangram
  >>> true"
  "'The quick brown fox jumps over the dog' isEnglishPangram
  >>> false"

  ... Your solution ...
```

Execute all the tests to verify that we did not change anything.

If we keep to use french words that do not need accents, we can verify that some french sentences are also pangrams.

```
'portez ce vieux whisky au juge blond qui fume' isEnglishPangram
>>> true

'portons dix bons whiskys à l''avocat goujat qui fume au zoo.'
    isEnglishPangram
>>> true
```

6.8 Identifying missing letters

Building a pangram can be difficult and the question is how we can identify missing letters. Let us define some methods to help us. But first let us write a test.

We will start to write a test for the method `detectFirstMissingLetterFor:`. As you see we just remove one unique letter from our previous pangram and we are set.

```
GramCheckerTest >> testDetectFirstMissingLetter

  self assert: ('the quick brown fox jumps over the lzy dog'
    detectFirstMissingLetterFor: 'abcdefghijklmnopqrstuvwxyz')
    equals: $a.
  self assert: ('the uick brown fox jumps over the lazy dog'
    detectFirstMissingLetterFor: 'abcdefghijklmnopqrstuvwxyz')
```

```
    equals: $q.
```

Your work: Propose a definition for the method `detectFirstMissingLetterFor:`.

```
String >> detectFirstMissingLetterFor: alphabet
  "Return the first missing letter to make a pangram of the receiver
    in the context of a given alphabet.
  Return '' otherwise"

  ... Your solution ...
```

In fact this method is close to the method `isPangramIn: alphabet`. It should iterate over the alphabet and check that the char is included in the string. When this is not the case, it should return the character and we can return an empty string when there is no missing letter.

About the return values of detectFirstMissingLetterFor:

Returning objects that are not polymorphic such as a single character or a string (which is not a character but a sequence of characters) is really bad design. Why? Because the user of the method will be forced to check if the result is a single character or a collection of characters.

> **Important** Avoid as much as possible to return objects that are not polymorphic. Return a collection and an empty collection. Not a collection and nil. Write methods returning the same kind of objects, this way their clients will be able to treat them without asking if they are different. This practice reinforces the **Tell do not ask principle**.

We have two choices: either always return a collection as for that we convert the character into a string sending it the message `asString` as follow, or we can return a special character to represent that nothing happens for example Character space.

```
String >> detectFirstMissingLetterFor: alphabet
  "Return the first missing letter to make a pangram of the receiver
    in the context of a given alphabet.
  Return '' otherwise"

  alphabet do: [ :aChar |
    (self includes: aChar)
      ifFalse: [ ^ aChar asString ]
    ].
  ^ ''
```

Here we prefer to return a string since the method is returning the first character. In the following one we return a special character.

```
String >> detectFirstMissingLetterFor: alphabet
  "Return the first missing letter to make a pangram of the receiver
    in the context of a given alphabet.
  Return '' otherwise"

  alphabet do: [ :aChar |
    (self includes: aChar)
      ifFalse: [ ^ aChar ]
    ].
  ^ Character space
```

Now it is better to return all the missing letters.

Detecting all the missing letters

Let us write a test to cover this new behavior. We removed the character a and g from the pangram and we verify that the method returns an array with the corresponding missing letters.

```
GramCheckerTest >> testDetectAllMissingLetters

  self assert: ('the quick brown fox jumps over the lzy do'
    detectAllMissingLettersFor: 'abcdefghijklmnopqrstuvwxyz')
    equals: #($a $g).
  self assert: ('the uick brwn fx jumps ver the lazy dg'
    detectAllMissingLettersFor: 'abcdefghijklmnopqrstuvwxyz')
    equals: #($q $o).
```

Your work: Implement the method `detectAllMissingLettersFor:`.

```
String >> detectAllMissingLettersFor: alphabet

  ... Your solution ...
```

One of the problem that you can encounter is that the order of the missing letters can make the tests failed. You can create a Set instead of an Array.

Now our test does not work because it verifies that we get an array of characters while we get an ordered collection. So we change it to take into account the returned collection.

```
GramCheckerTest >> testDetectAllMissingLetters

  self assert: ('the quick brown fox jumps over the lzy do'
    detectAllMissingLettersFor: 'abcdefghijklmnopqrstuvwxyz')
    equals: (Set withAll: #($a $g)).
  self assert: ('the uick brwn fx jumps ver the lazy dg'
    detectAllMissingLettersFor: 'abcdefghijklmnopqrstuvwxyz')
    equals: #($q $o) asSet.
```

Instead of explicitely creating a Set, we could also use the message `asSet` that converts the receiver into a Set as shown in the second check.

6.9 Palindrome as exercise

We let as an exercise the identification if a string is a palindrom. A palindrome is a word or sentence that can be read in both way. 'KAYAK' is a palindrome.

```
GramCheckerTest >> testIsPalindrome

  self assert: 'ete' isPalindrome.
  self assert: 'kayak' isPalindrome.
  self deny: 'etat' isPalindrome.
```

Some possible implementations

Here is a list of possible implementation.

- You can iterate on strings and check that the first element and the last element are the same.
- You can also reverse the receiver (message `reverse`) and compare the character one by one. You can use the message `with:do:` which iterate on two collections.

```
'etat' reverse
>>> 'tate'
```

```
| res |
res := OrderedCollection new.
#(1 2 3) with: #(10 20 30) do: [ :f :s | res add: f * s ].
res
>>> an OrderedCollection(10 40 90)
```

You can also add the fact that space do not count.

```
  self assert: 'Elu par cette crapule' isPalindrome.
```

6.10 Conclusion

We got some fun around words and sentences. You should know more about strings and collection. In particular, in Pharo a collection can contain any objects. You also saw is that loops to not require to specify the first index and how to increment it. Of course we can do it in Pharo using the message `to:do:` and `to:by:do:`. But only when we need it. So play with some iterators such as `do:` and `select:`. The iterators are really powerful and this is really important to be fluent with them because they will make you save a lot of time.