CHAPTER 1

CHAPTER 2

Some Collection Katas with Words

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

2.1 Isogram

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

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

Isograms are fun 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.

2.2 About strings

A strings in Pharo is in fact an array of characters.

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

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

As with any collection, we can apply iterators such `select:`, `do:`, or `collect:`.

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

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

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

We can also find the index of a string inside another one.

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

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

2.3 A Set-Based solution

We can do a simple implementation using Sets. Sets are collections that only contains one element. Adding twice the same element only adds one. Note that Set in Pharo can contain any objects even Sets themselves. 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)
```

Imagine how using a set of characters, you can determine if a string is a isogram.

Hints

If the size of set and the 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 we will propose multiple implementations we postfix the message with the implementation strategy we use. Here we define `isIsogramSetImplementation`

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

  ^ self size = self asSet size
```

And we test that our method is correct.

```
'pharo' isIsogramSetImplementation
>>> true
```

```
'phaoro' isIsogramSetImplementation
>>> true
```

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

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

2.4 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.

> **Note** To define a method as 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' isIsogramSetImplementation` returns `true`.
- We check (`deny:`) that 'phaoro' is *not* an isoGram i.e., that `'pharo' isIsogramSetImplementation` returns `false`.

```
GramCheckerTest >> testIsogramSetImplementation

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

> **Note** 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:

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

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.

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

```
GramCheckerTest >> testAllIsogramSetImplementation

    self isograms do: [ :word |
```

```
      self assert: word isIsogramSetImplementation ]
```

Since we said that we should also tests negative let us to the same for non isograms.

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

And we make our test using both.

```
GramCheckerTest >> testAllIsogramSetImplementation

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

Some Fun

Now we would like to find some isograms in french. So we would like to get all the lines of http://www.pallier.org/ressources/dicofr/liste.de.mots.francais.frgut.txt

This page returns text that is latin1 (iso-8859-1) encoded, but describes it as 'text/plain' without further qualification. Zn then assumes the encoding is utf8 (the most reasonable default today). 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 Zn

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

The above will give you an array of 336531 words (it is a bit slow because it is lot of data).

```
self select: #isIsogramFastest

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

2.5 A dictionary-based solution

To do blbl

```
String >> isIsogramDictionaryImplementation
  "'ALTRUISME' isIsogramDictionaryImplementation"
  | letters |
  letters := Dictionary new.
  self do: [ :aChar |
      letters at: aChar
        ifPresent: [^ false]
        ifAbsent: [ letters at: aChar put: 1].
      ].
  ^ true
```

2.6 A Bag-based solution

To do blbl

```
String >> isIsogramUsingBagImplementation
  "'ALTRUISME' isIsogramUsingBagImplementation"

  | bag |
  bag := Bag new.
  self do: [ :each |
      bag add: each.
      ].
  ^ bag sortedCounts first key = 1
```

2.7 A fast solution

To do blbl

```
String >> isIsogramFatestImplementation
  "'ALTRUISME' isIsogramFatestImplementation"
  1 to: self size -1 do: [ :ix |
  (self
    findString: (self at: ix) asString
    startingAt: ix +1
    caseSensitive: false) ~~ 0 ifTrue: [ ^ false ]
  ].
  ^ true
```

2.8 Comparing solutions

Pharo proposes tools to measure execution speed (for example the message `timeToRun` and `millisecondsToRun:` shown below). When an operation is too fast you cannot measure well (Check the chapter on profiling of Deep Into Pharo), so you should execute multiple times the same expressions.

Here measuring an expression alone does not really help as you can see.

```
['PHRAO' isIsogramSetImplementation ] timeToRun.
>>> 0
Time millisecondsToRun: ['PHRAO' isIsogramSetImplementation ]
>>> 0
```

Pharo libraries also offers the bench method that gives the number of execution possible per second.

[[[['ALTRUISME' isIsogramSetImplementation] bench '334,371 per second'

['ALTRUISME' isIsogramFastestImplementation] bench '546,823 per second'

['ALTRUISME' isIsogramUsingBagImplementation] bench '142,432 per second'

['ALTRUISME' isIsogramDictionaryImplementation] bench '147,276 per second']]]

So now we will look at other kind of words or sentences.

2.9 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 testfirst

```
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
```

We can

```
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"
  | alphabet |
  alphabet := 'abcdefghijklmnopqrstuvwxyz'.
  alphabet do: [ :aChar |
    (self includes: aChar)
```

```
      ifFalse: [ ^ false ]
    ].
  ^ true
```

2.10 Handling alphabet

```
GramCheckerTest >> testIsPangramIN

  self assert: ('The quick brown fox jumps over the lazy dog'
    isPangramIn: 'abcdefghijklmnopqrstuvwxyz').
  self assert: ('ma papa mama' isPangramIn: 'apm').
```

```
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"

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

```
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"

  ^ self isPangramIn: 'abcdefghijklmnopqrstuvwxyz'.
```

Execute all the tests to verify that we did change anything.

If we keep to use french words that do not need accents, we can verifie 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
```

2.11 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.
```

Propose a definition for the method `detectFirstMissingLetterFor:`

```
String >> detectFirstMissingLetterFor: alphabet
  ...
```

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 and empty string when there is missing letter.

Here is a possible implementation. Verify that the test is passing.

```
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 ]
    ].
  ^ ''
```

About the return values of detectFirstMissingLetterFor:

Returning objects that are not polymorphic such as a single character or a string is really bad design. Why? Because the user of the method will not have the

```
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 ]
    ].
  ^ ''
```

> **Note** 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 return always 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 character since the method is returning the first 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).
```

Implement

```
String >> detectAllMissingLettersFor: alphabet

  ...
```

```
String >> detectAllMissingLettersFor: alphabet

  | res |
  res := Set new.
  alphabet do: [ :aChar |
    (self includes: aChar)
      ifFalse: [ res add: aChar ]
    ].
  ^ res
```

We create a Set instead of an Array because Arrays in Pharo have a fixed size that should be known at creation time. We could have created an array of size 26. In addition we do not care about the order of the missing letters. A Set is a collection whose size can change, and in which we can add the element one by one, therefore it fits our requirements.

Now our test does not work because it is verifying 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 convert the receiver into a Set as shown in the second check.