Iterators

Damien Cassou, Stéphane Ducasse and Luc Fabresse
What You Will Learn

- Understand the power of iterators
- Offer an overview of iterators
Pharo code is Compact!

```
ArrayList<String> strings = new ArrayList<String>();
for(Person person: persons)
    strings.add(person.name());
```

is expressed as

```
strings := persons collect: [ :person | person name ]
```

- Yes in Java 8.0 it is finally simpler

```
strings = persons.stream().map(person -> person.getName())
```

- But it is like that in Pharo since day one!
- Iterators are deep into the core of the language and libraries
A First Iterator - collect:

collect: applies the block to each element and returns a collection (of the same kind than the receiver) with the results

```
\#(2 \ -3 \ 4 \ -35 \ 4) \text{ collect: } [ : \text{each} | \text{each abs} ]
> \#(2 \ 3 \ 4 \ 35 \ 4)
```

- collect: evaluates the block for each element (using value:)
- In the block, each element is sent abs (absolute)
- collect: returns a new collection (of the same kind of the receiver) with all results
- [Think object] We ask the collection to do something for us
Another collect: Example

We want to know if each elements is odd or even

```
#(16 11 68 19) collect: [ :i | i odd ]

> #(false true false true)
```
Choose your camp!

We can also do it that way! (We copied the definition of `collect`):

```
| result |
aCol := #(16 11 68 19).
result := aCol species new: aCol size.
1 to: aCollection size do:
    [ :each | result at: each put: (aCol at: each) odd ].
^ result
```
Part of the Collection Hierarchy

Iterators work polymorphically on the entire collection hierarchy. Below a part of the Collection hierarchy.
Think objects!

- With iterators we **tell** the collection to **iterate on itself**
- As a client we do not have to know the internal logic of the collection
- Each collection can implement differently the iterator
Basic Iterators Overview

- do: (iterate)
- collect: (iterate and collect results)
- select: (select matching elements)
- reject: (reject matching elements)
- detect: (get first element matching)
- detect:ifNone: (get first element matching or a default value)
- includes: (test inclusion)
- and a lot more...
do: an Action on Each Clement

- Iterates on each elements
- Applies the block on each elements

```small
#(16 11 68 19) do: [:each | Transcript show: each ; cr ]
```

Here we print each element and insert a carriage return
select: Elements Matching a Criteria

To select some elements, use select:

```
#(16 11 68 19) select: [:i | i odd ]
> #(11 19)
```
With Unary Messages, No Block Needed

When a block is just one message, we can pass an unary message selector

```
#(16 11 68 19) select: [ :i | i odd ]
```

is equivalent to

```
#(16 11 68 19) select: #odd
```
reject: Some Elements Matching a Criteria

To filter some elements, use reject:

```javascript
#(16 11 68 19) reject: [:i | i odd ]
> #(16 68)
```
To find the first element that matches, use `detect`:

```
#(16 11 68 19) detect: [ :i | i odd ]
> 11
```
detect:ifNone:

To find the first element that matches else return a value, use `detect:ifNone:`

```
#(16 12 68 20) detect: [ :i | i odd ] ifNone: [ 0 ] > 0
```
Some Powerful Iterators

- **anySatisfy**: (tests if one object is satisfying the criteria)
- **allSatisfy**: (tests if all objects are satisfying the criteria)
- **reverseDo**: (do an action on the collection starting from the end)
- **doWithIndex**: (do an action with the element and its index)
- **pairsDo**: (evaluate aBlock with my elements taken two at a time.)
- **permutationsDo**: ...
Iterating Two Structures

To iterate with:do:

```
#(1 2 3) with: #(10 20 30)
do: [ :x :y | Transcript show: (y * x) ; cr ]
```

with:do: requires two structures of the same length
Use do:separatedBy:

```ruby
String streamContents: [:s |
    #('a' 'b' 'c')
    do: [:each | s << each ]
    separatedBy: [ s << ',', ']
]
> 'a, b, c'
```
Grouping Elements

To group elements according to a grouping function:
groupedBy:

```
#(1 2 3 4 5 6 7) groupedBy: #even
> a PluggableDictionary(false → #(1 3 5 7) true → #(2 4 6) )
```
Flattening Results

How to remove one level of nesting in a collection? Use `flatCollect`:

```ruby
#( #(1 2) #(3) #(4) #(5 6)) collect: [:each | each ]
> #(1 2 3 4 5 6)
```

```ruby
#( #(1 2) #(3) #(4) #(5 6)) flatCollect: [:each | each ]
> #(1 2 3 4 5 6)
```
Opening The Box

- You can learn and discover the system
- You can define your own iterator
- For example how `do:` is implemented?

```plaintext
SequenceableCollection >> do: aBlock
"Evaluate aBlock with each of the receiver's elements as the argument."

1 to: self size do: [:i | aBlock value: (self at: i)]
```
Analysis

- Iterators are really powerful because they support polymorphic code
- All the collections support them
- New ones are defined
- Missing controlled navigation as in the Iterator design pattern
Summary

- Iterators are your best friends
- Simple and powerful
- Enforce encapsulation of collections and containers
A course by

and

in collaboration with

Except where otherwise noted, this work is licensed under CC BY-NC-ND 3.0 France
https://creativecommons.org/licenses/by-nc-nd/3.0/fr/