Iterators

Stéphane Ducasse

http://stephane.ducasse.free.fr/
stephane.ducasse@inria.fr

Objectives

- Understand iterators
- Offer an overview

Pharo code is Compact!

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

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

- Yes in Java 8.0 it will be finally simpler.
- But it is like that in Pharo since day one!
- Iterators are deep into the core of the language.

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) collect: [ :each | each abs ]
> #(2 3 4 35 4)
```

- `collect:` sends the message `abs` (absolute) to each element of the receiver
- and returns the resulting collection.
- What we see is that 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!

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

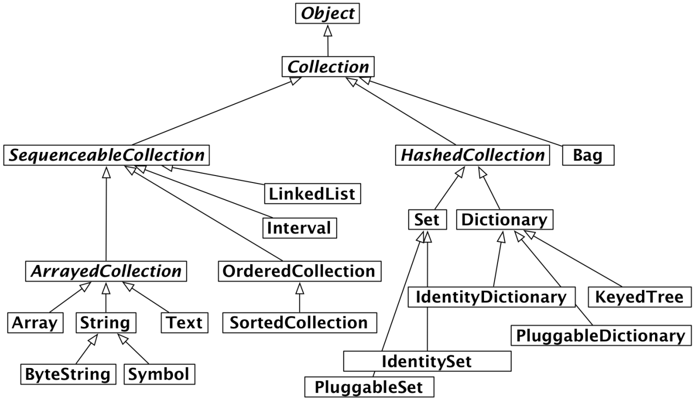
- We can also do it that way!

```
| 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
```

- Here we copied the definition of `collect:`, to show how we could expressed the same behavior but this is error prone, verbose and tedious.

Part of the collection hierarchy

- Iterators work polymorphically on the entire collection hierarchy.



Part of the Collection hierarchy.

Think objects!

- With iterators we ask the collection to iterate on itself.
- As a client we do not have to know the internal details of the collection.
- Each collection can implement differently the iterator.

Basic Iterators Overview

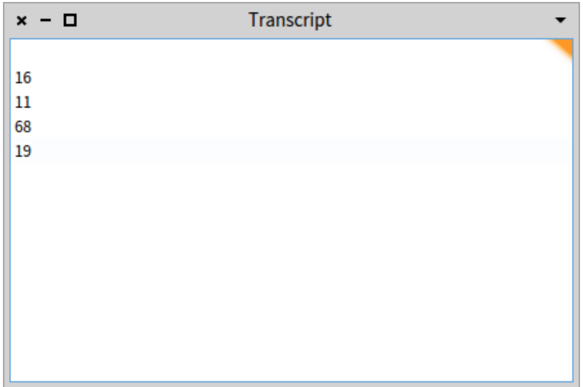
- `do:` (iterate)
- `collect:` (iterate and collect results)
- `select:` (select matching elements)
- `reject:` (reject matching elements)
- `detect:` (get first element matching)
- `detect:ifNone:` (get first element matching or a default value)
- `includes:` (test inclusion)
- and a lot more...

do: an action on each element

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

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

- Here we print each elements and insert a carriage return.



```
16
11
68
19
```

select: elements matching a criteria

- Select some elements

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

select: elements matching a criteria

- Select some elements

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

```
> #(11 19)
```

With unary messages, no block needed

When a block expects a single argument, 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: elements matching a criteria

- Filter some elements

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

reject: elements matching a criteria

- Filter some elements

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

```
> #(16 68)
```

detect: the first elements that...

- First element that matches

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

detect: the first elements that...

- First element that matches

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

```
> 11
```

detect:ifNone:

- First element that matches else return a value

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

detect:ifNone:

- First element that matches else return a value

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

```
> 0
```

Some other 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:`

Exercises

Propose some expressions to illustrate ther uses of

- `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:`

Exciting ones

- How to produce?

```
#('a' 'b' 'c') message
> 'a, b, c'

#('a') message
> 'a'

#() message
> ''
```

]]]

Exciting ones

- How to produce?

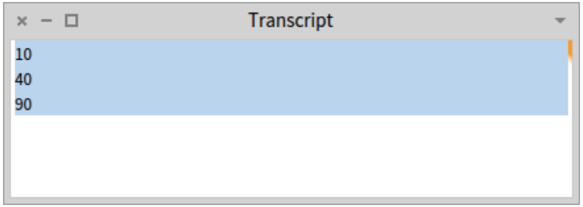
```
#('a' 'b' 'c') message
> 'a, b, c'

#('a') message
> 'a'

#() message
> ''
```

- Use `doSeparatedBy:`

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

Iterating two structures

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



```
10
40
90
```

- with:do: requires two structures of the same length.

Grouping elements

- `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:`

```
#( #(1 2) #(3) #(4) #(5 #(6 7 3))) collect: [ :each | each ]

> #(#(1 2) #(3) #(4) #(5 #(6 7 3)))
```

```
#( #(1 2) #(3) #(4) #(5 #(6 7 3))) flatCollect: [ :each | each ]

> #(1 2 3 4 5 #(6 7 3))
```

Opening the box

- You can learn and discover the system.
- You can define your own.
- How `do:` is implemented?

```
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