An Overview of Essential Collections

Damien Cassou, Stéphane Ducasse and Luc Fabresse

W3S07

http://www.pharo.org
What You Will Learn

- Some basic collections
- Essential API to program collections
- Difference between literal and dynamic arrays
Pharo has a rich hierarchy of collection

Common API: size, do:, select:, includes:, collect:...

First element is at index 1

Can contain any object
Most Common Collections

- OrderedCollection (dynamically growing)
- Array (fixed size, direct access)
- Set (no duplicates)
- Dictionary (key-based, aka. maps)
Essential Collection In a Nutshell
Common API Overview

Common messages work on all collections

1. **creation**: with: anElt, with:with:, withAll: aCollection
2. **accessing**: size, at: anIndex, at: anIndex put: anElt
3. **testing**: isEmpty, includes: anElt, contains: aBlock,
4. **adding**: add: anElement, addAll: aCollection
5. **removing**: remove: anElt, remove: anElt ifAbsent: aBlock, removeAll: aCollection
6. **enumerating**: do: aBlock, collect: aBlock, select: aBlock, reject: aBlock, detect: aBlock, ...
7. **converting**: asBag, asSet, asOrderedCollection, asSortedCollection, asArray
Variable Size Object Creation

- Message `new` instantiates one object
- Message `new: size` creates an object specifying its size

```plaintext
Array new: 4
> #(nil nil nil nil)

Array new: 2
> #(nil nil)

(OrderedCollection new: 1000)
```
With Specific Elements

OrderedCollection withAll: #(7 7 3 13)
> an OrderedCollection(7 7 3 13)

Set withAll: #(7 7 3 13)
> a Set( 7 3 13)

Remember: no duplicate in Sets
Creation with Default Value

OrderedCollection new: 5 withAll: 'a'
> an OrderedCollection('a' 'a' 'a' 'a' 'a')
#("Calvin" 'hates' "Suzie") at: 2
> 'hates'

#("Calvin" 'hates' "Suzie") asOrderedCollection at: 2
> 'hates'
Collections can be Heterogenous

Collections can contain any sort of objects

```
#('calvin' (1 2 3))
> #('calvin' #(1 2 3))
```

- An array composed of a string and an array

```
| s |
s := Set new.
s add: Set new;
  add: 1;
  add: 2.
s asArray
> an Array(1 2 a Set())
```

- A set containing an empty set and some numbers
Iteration

- Using message `do: aBlock`
- But many iterators (see Iterators Lecture)

```plaintext
#('Calvin' 'hates' 'Suzie')
do: [:each | Transcript show: each ; cr ]
```
Arrays

- Fixed size collection
- Direct access: at: and at:put:
- Has literal syntax: #( ... )
- Can also be created using new:

```plaintext
#('Calvin' 'hates' 'Suzie') size
> 3
```

is equivalent to

```plaintext
(Array new: 3)
  at: 1 put: 'Calvin';
  at: 2 put: 'hates';
  at: 3 put: 'Suzie') size
> 3
```
Accessing Elements

Getting the size of a collection

```plaintext
#('Calvin' 'hates' 'Suzie') size > 3
```

Accessing the 2nd element using `at: anIndex`

```plaintext
#('Calvin' 'hates' 'Suzie') at: 2
> 'hates'
```

Remember collection index starts at 1
Accessing Out of Bounds Elements

#('Calvin' 'hates' 'Suzie') at: 55
> SubscriptOutOfBounds Error
Modifying Elements

Use the message `at: anIndex put: anObject`
Modifying the second element of the receiver

```smalltalk
#('Calvin' 'hates' 'Suzie') at: 2 put: 'loves'
> #( 'Calvin' 'loves' 'Suzie')
```
Literal Arrays

Literal arrays contain objects that have a textual (literal) representation: numbers, strings, nil, symbols

```
#(45 'milou' 1300 true #tintin)
> #(45 'milou' 1300 true #tintin)
```

They are instances of the class `Array`

```
#(45 38 1300 8) class
> Array
```
Literal arrays are equivalent to a dynamic version

A literal array

```
#(45 38 'milou' 8)
> #(45 38 'milou' 8)
```

An array

```
Array with: 45 with: 38 with: 'milou' with: 8
> #(45 38 'milou' 8)
```
OrderedCollection

- Sequenceable
- Growing size
- add:, remove:

```plaintext
| ordCol |
ordCol := OrderedCollection new.
ordCol add: 'Reef'; add: 'Pharo'; addFirst: 'Pharo'.
ordCol > an OrderedCollection('Pharo' 'Reef' 'Pharo')
ordCol add: 'Seaside'.
ordCol > an OrderedCollection('Pharo' 'Reef' 'Pharo' 'Seaside')
```
Conversion

#('Pharo' 'Reef' 'Pharo' 'Pharo') asOrderedCollection
> an OrderedCollection('Pharo' 'Reef' 'Pharo' 'Pharo')
Set

- No duplicates
- Growing size
- add:, remove:
- Can contain any object including other Sets

```plaintext
#('Pharo' 'Reef' 'Pharo' 'Pharo') asSet
> a Set('Pharo' 'Reef')

Set with: (Set with: 1) with: (Set with: 2)
```
Conversion

Collections can be converted simply to other collections

- asOrderedCollection
- asSet
- asArray
Dictionary

- Key/values
- Growing size
- Accessing at:, at:ifAbsent:
- Changing/adding at:put:, at:ifAbsentPut:
- Iterating: do:, keysDo:, keysAndValuesDo:
Dictionary Creation

| days |
days := Dictionary new.
days
  at: #January put: 31;
  at: #February put: 28;
  at: #March put: 31.
Alternate Dictionary Creation

```
| days |
days := Dictionary new.
days
at: #January put: 31;
at: #February put: 28;
at: #March put: 31.
```

is equivalent to

```
{ #January -> 31.
 #February -> 28.
 #March -> 31} asDictionary
```
Pairs

(#January → 31) key
> #January

(#January → 31) value
> 31
Dictionary Access

<table>
<thead>
<tr>
<th>days</th>
</tr>
</thead>
</table>
days := Dictionary new.
days
  at: #January put: 31;
at: #February put: 28;
at: #March put: 31.

days at: #January
> 31

days at: #NoMonth
> KeyNotFound Error

days at: #NoMonth ifAbsent: [0]
> 0
Dictionary Iteration

days do: [:each | Transcript show: each; cr ]

prints

31
28
31

Why? Because

Dictionary >> do: aBlock

^self valuesDo: aBlock
Keys and Values Iteration

define days keysAndValuesDo:
    [:k :v | Transcript show: k asString, ' has ', v printString, ' days'; cr]

shows:

  January has 31 days
  February has 28 days
  March has 31 days
Summary

- Easy to use collections
- Common vocabulary
- Simple conversion between them
- Easy to discover!