

Laboratorio de Computación III

Clase XVI

Andrés Fortier

Hoy

- Repaso de bloques.
- Un poco de tests.
- Un par de mensajes nuevos de colecciones.

Repaso - Bloques

- Contienen una secuencia de expresiones que se pueden ejecutar.
- Se escriben enmarcando dichas expresiones entre corchetes.
- Ejemplos:
 - `[1+2].`
 - `[]`. *Bloque sin expresiones.*
 - `[| sum |
sum:= 1 + 5.
sum ** 2
].`

Repaso - Bloques

- Los bloques pueden tomar parámetros.
- Ejemplos:
 - $[:x \mid x + 1]$.
 - $[:x :y :z \mid (x + y) * z]$.

Repaso - Bloques

- Para evaluar un bloque se utiliza el mensaje `#value` o alguna de sus variantes.
- Ejemplos:
 - `[1+5] value. “println => 6”`
 - `[:x | x + 1] value: 7. “println => 8”`

Repaso - Bloques

- Los bloques retornan el resultado de la última expresión evaluada.
- Ejemplos:
 - `[]` value. *“println => nil”*
 - `[4]` value. *“println => 4”*
 - `[:x |
 x + 1.
 3 + 3.
]` value: 99. *“println => 6”*

Repaso - Bloques

- Los bloques son objetos y pueden evaluarse muchas veces.
- Ejemplos:
 - | bloque |
bloque := [1+1].
bloque value. “*println* => 2”
bloque value. “*println* => 2”
 - | bloque |
bloque := [:x | x+1].
bloque value: 3. “*println* => 4”
bloque value: 18. “*println* => 19”

Repaso - Bloques en acción

CuentaBancaria>>extraer: unMonto

```
(self puedeExtraer: unMonto)
  ifTrue:[self realizarExtracción: unMonto].
```

True>>ifTrue: aBlock

^aBlock value.

- Supongamos que se puede hacer la extracción.

```
(self puedeExtraer: unMonto)
  ifTrue:[self realizarExtracción: unMonto].
```

~>

```
true
  ifTrue:[self realizarExtracción: unMonto].
```

~>

```
[self realizarExtracción: unMonto] value.
```

~>

```
self realizarExtracción: unMonto.
```


Repaso - Iteradores

```
SequenceableCollection>>do: aBlock
```

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

```
#(1 3 5) do: [:numero | Transcript show: numero]
```

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

~>

```
1 to: 3 do:  
  [:index | aBlock value: (self at: index)]
```

~>

```
aBlock value: (self at: 1).  
aBlock value: (self at: 2).  
aBlock value: (self at: 3).
```

~>

```
aBlock value: 1.  
aBlock value: 3.  
aBlock value: 5.
```

Repaso - Iteradores

```
SequenceableCollection>>do: aBlock
```

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

```
#(1 3 5) do: [:numero | Transcript show: numero]
```

```
aBlock value: 1.  
aBlock value: 3.  
aBlock value: 5.
```

~>

```
[:numero | Transcript show: numero] value: 1.  
[:numero | Transcript show: numero] value: 3.  
[:numero | Transcript show: numero] value: 5.
```

~>

```
Transcript show: 1.  
Transcript show: 3.  
Transcript show: 5.
```

Tests de unidad

- Conforme tengan que modelar problemas de mayor envergadura los tests serán mas complejos.
- En muchos casos van a ver que el código necesario para el setup se repite.
- Y sabemos que no hay que repetir código.

Tests de unidad

```
EstacionDeServicioTest>>testCantidadDeSurtidores
```

```
| estacionSinSurtidores estacionConSurtidores |
```

```
estacionSinSurtidores:= EstacionDeServicio new.  
self assert: estacionSinSurtidores cantidadDeSurtidores = 0.
```

```
estacionConSurtidores:= EstacionDeServicio new.  
estacionConSurtidores  
    agregarSurtidor: (Surtidor conCapacidadMaxima: 1000);  
    agregarSurtidor: (Surtidor conCapacidadMaxima: 3000).  
self assert: estacionConSurtidores cantidadDeSurtidores = 2.
```

```
...
```

Tests de unidad

```
EstacionDeServicioTest>>testLitrosACargar
```

```
| estacionSinSurtidores estacionConSurtidores |
```

```
estacionSinSurtidores:= EstacionDeServicio new.  
self assert: estacionSinSurtidores litrosACargar = 0.
```

```
estacionConSurtidores:= EstacionDeServicio new.  
estacionConSurtidores  
    agregarSurtidor: (Surtidor conCapacidadMaxima: 1000);  
    agregarSurtidor: (Surtidor conCapacidadMaxima: 3000).  
self assert: estacionConSurtidores litrosACargar = 4000.
```

```
...
```

Tests de unidad

```
EstacionDeServicioTest>>testLitrosACargar
```

```
| estacionSinSurtidores estacionConSurtidores |
```

```
estacionSinSurtidores:= EstacionDeServicio new.  
self assert: estacionSinSurtidores litrosACargar = 0.
```

```
estacionConSurtidores:= EstacionDeServicio new.  
estacionConSurtidores  
    agregarSurtidor: (Surtidor conCapacidadMaxima: 1000);  
    agregarSurtidor: (Surtidor conCapacidadMaxima: 3000).  
self assert: estacionConSurtidores litrosACargar = 4000.
```

```
...
```


Tests de unidad

- Definimos estacionSinSurtidores y estacionConSurtidores como variables de instancia del test.
- Y luego definimos el mensaje #setUp

```
EstacionDeServicioTest>>setUp
```

```
    estacionSinSurtidores:= EstacionDeServicio new.
```

```
    estacionConSurtidores:= EstacionDeServicio new.
```

```
    estacionConSurtidores
```

```
        agregarSurtidor: (Surtidor conCapacidadMaxima: 1000);
```

```
        agregarSurtidor: (Surtidor conCapacidadMaxima: 3000).
```

Tests de unidad

```
EstacionDeServicioTest>>testLitrosACargar
```

```
self assert: estacionSinSurtidores litrosACargar = 0.
```

```
self assert: estacionConSurtidores litrosACargar = 4000.
```

```
...
```

Tests de unidad

- El framework de test nos asegura que el mensaje `#setUp` se envía antes de ejecutar cada test.
- A favor: factorizar código.
- En contra: si no usan buenos nombres de v.i. los tests se vuelven menos legibles y hay que ir al `setUp` para entender que objetos hay en esas v.i.

Colecciones - #inject:into:

- Se utiliza para convertir los elementos de una colección en un único objeto final.
- También conocido como *fold*, se origina en la programación funcional.
- Recordemos el caso del mensaje #sum
 - #(1 2 3 4) sum
- Puede escribirse como:
 - #(1 2 3 4) inject: 0 into: [:subtotal :valor | subtotal + valor]

Colecciones - #inject:into:

- Algunos ejemplos:

```
| numeros |
```

```
numeros := #(1 2 3 4).
```

“Obtener el máximo de una colección de números”

```
numeros
```

```
  inject: numeros first
```

```
  into: [:maximo :numero | maximo max: numero].
```

```
EstacionDeServicio >> litrosACargar
```

“Sumar los litros a cargar de los surtidores”

```
^surtidores
```

```
  inject: 0
```

```
  into: [:litros :surtidor | litros + surtidor litrosACargar].
```


Colecciones - #inject:into:

“Convertir un array de strings en un string”

```
#('la' 'mesa' 'está' 'servida')  
  inject: "  
  into: [:resultado :string | resultado , ' ' , string]
```

- ¿Alguien ve un problema con esto?

```
#('la' 'mesa' 'está' 'servida')  
  inject: "  
  into: [:resultado :string | resultado , ' ' , string]
```

→ '  la mesa está servida'

Colecciones - #reduce:

```
#('la' 'mesa' 'está' 'servida')  
  reduce: [:resultado :string | resultado , ' ' , string]
```

→ 'la mesa está servida'

```
 #(1 2 3 4) reduce: [:max :number | number max: max]
```

→ 4

```
| fechas |  
fechas := OrderedCollection new.  
fechas add: (Fecha dia: 2 mes: 4 año: 2015).  
fechas add: (Fecha dia: 3 mes: 8 año: 2015).  
fechas add: (Fecha dia: 1 mes: 1 año: 2016).  
fechas reduce: [:max :fecha | fecha max: max].
```