Biblioteca de Clases Smalltalk

2do Cuatrimestre 2025

Biblioteca de clases de Smalltalk

Funcionalidades básicas disponibles:

- Creación de instancias, comportamiento común (clase **Object**)
- Testeo y control condicional simple y compuesto, operaciones lógicas (clase **Boolean** y subclases)
- Operaciones y comparaciones numéricas (clase **Number** y subclases)
- Iteraciones (clases **Number**, **BlockClosure**, **Collection**...)
- Arreglos, listas, conjuntos... (**Collection** y subclases)
- Inspección / Depuración

2

Clase Object

- **Object** es la clase que se encuentra en la raíz de la jerarquía de herencia de **SMALLTALK**.
- **Object** provee el protocolo común para:
  - **creación de instancias**
  - **comparación entre objetos**
  - **copia de objetos**
  - **acceso al objeto clase**
  - **errores/depuración**
  - **inspección**
  - ...

3

Métodos de clase de Object

- **Para crear un nuevo objeto es necesario enviar un mensaje a la clase correspondiente.**

new

Método Unario. Retorna una nueva instancia del receptor.

Persona *new.*
OrderedCollection *new.*

new: tamaño

Método de palabra clave. Retorna una nueva instancia del receptor con un número de variables de instancia indexadas dado por tamaño (clase Array).

Array *new:* **5.**

4

Inicialización de variables de instancia

- El valor por defecto de toda variable de instancia es **nil**.
- En muchos casos este valor no es adecuado para la nueva instancia → ***new*** debe ser redefinido **para realizar la inicialización**.

5

Inicialización de variables de instancia

Redefinición del método de clase ***new*** en una subclase de **Object**:

```
new
  "Crea, inicializa y retorna una instancia del receptor"
  | instancia |
  instancia := super new.
  instancia inicializar.
  ^ instancia
```

```
new
  "Crea, inicializa y retorna una instancia del receptor"
  ^ super new inicializar; yourself
```

6

Métodos de instancia de Object

= unObjeto	Comparación de igualdad.
== unObjeto	Comparación de identidad.
~= unObjeto	No iguales.
~~ unObjeto	No idénticos.
yourself	Devuelve el receptor.
isArray isBoolean isCharacter isInteger isNil notNil	Devuelve true si el receptor es del tipo indicado.

7

Métodos de instancia de Object

asString	Devuelve la representación del receptor como String.
copy shallowCopy	Copia superficial del receptor.
deepCopy	Copia profunda.
class	Devuelve la clase del receptor.
isKindOf: *unaClase*	Devuelve true si *unaClase* es la clase o es ancestro de la clase del receptor.
inspect	Abre una ventana de *Inspección* del objeto receptor.
error: *unString*	Produce un mensaje de error.

8

Estructuras de Control

En **Smalltalk** todo es un **objeto**. Por lo tanto, las **estructuras de control** se implementan mediante el **envío de mensajes a objetos**.

- **Selección o alternativa** (*If Then*, *If Then Else*) se implementa por medio del envío de mensajes a los objetos:
  - **true**
  - **false**
- **Iteración o repetición** (*For*, *While*, *etc*) se implementan por medio del envío de mensajes a objetos de las clases:
  - **Number**
  - **BlockClosure**
  - **Collection**

9

Selección

- Existen dos objetos de tipo **Boolean** en el lenguaje y algunos mensajes enviados a estos objetos permiten la implementación de la **SELECCIÓN** en ***Smalltalk***.
- Los mensajes que se envían a los objetos **true** y **false** llevan como parámetro **un bloque sin argumentos**. La evaluación de dicho bloque depende del mensaje y del objeto al que se lo envía.
- Estos mensajes provistos por ***Smalltalk*** permiten el testeo y control condicional:
  - **simple**
  - **compuesto**

10

Testeo y control condicional simple

Esta estructura de control se implementa mediante el mensaje de palabra clave

ifTrue:

- Sintaxis:

  <objetoBooleano> ***ifTrue:*** *<bloque>*

- Semántica:
  - Si el receptor del mensaje es el objeto **true**, luego se evalúa el bloque pasado como argumento, y retorna el objeto retornado por la evaluación del bloque.
  - Si el receptor del mensaje es el objeto **false**, no se evalúa el bloque y retorna **nil**.

11

Ejemplos de *ifTrue*:

5 > 3 ***ifTrue:*** ['mayor el primero'] → 'mayor el primero'

3 > 5 ***ifTrue:*** ['mayor el primero'] → **nil**

A := 'cama'. B := 'casa'.

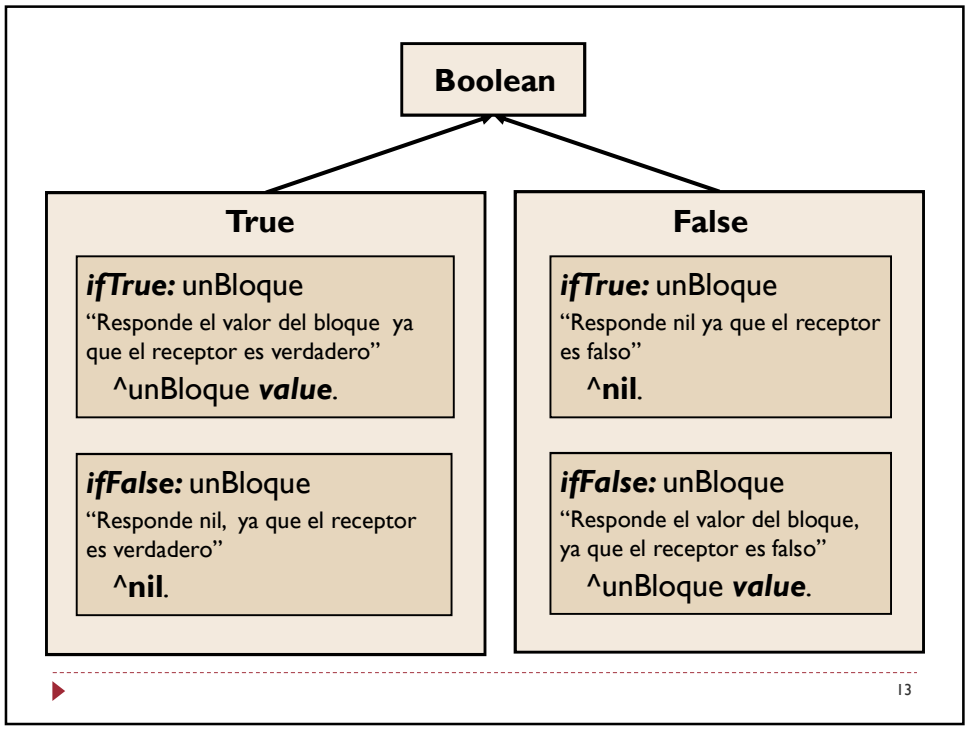
A > B ***ifTrue:*** [A ***asSymbol***] → **nil**

Mensaje *ifFalse*:

<objetoBooleano> ***ifFalse:*** *<bloque>*

Comportamiento inverso al de ***ifTrue:***

3 > 5 ***ifFalse:*** ['mayor el segundo'] → 'mayor el segundo'



Testeo y control condicional compuesto

Corresponde a la estructura de control *"If Then Else"* y se implementa mediante el mensaje ***ifTrue:ifFalse:***

<objetoBooleano> ***ifTrue:*** *<bloque1>* ***ifFalse:*** *<bloque2>*

3 > 5 ***ifTrue:*** ['mayor el primero']
ifFlase: ['mayor el segundo'] → 'mayor el segundo'

True

ifTrue: bloque1 ***ifFalse:*** bloque2
"Responde el valor del bloque verdadero"
^bloque1 ***value***.

False

ifTrue: bloque1 ***ifFalse:*** bloque2
"Responde el valor del bloque falso"
^bloque2 ***value***.

Mensajes que retornan **true** o **false**

Los siguientes mensajes binarios, entre otros, retornan los objetos **true** o **false**:

Mensaje	Significado
==	equivalentes (idénticos)
~~	no equivalentes
=	iguales
~=	no iguales
<, <=	menor, menor o igual
>, >=	mayor, mayor o igual

Este comportamiento los hace aptos para expresar las **condiciones booleanas** que a su vez reciben los mensajes que implementan la **selección o alternativa**.

15

Mensajes que representan Operaciones lógicas

Para poder expresar **condiciones lógicas compuestas**, están definidos los métodos:

- **not** (unario)
- **&** (representa el operador **AND** y es binario)
- **|** (representa el operador **OR** y es binario)

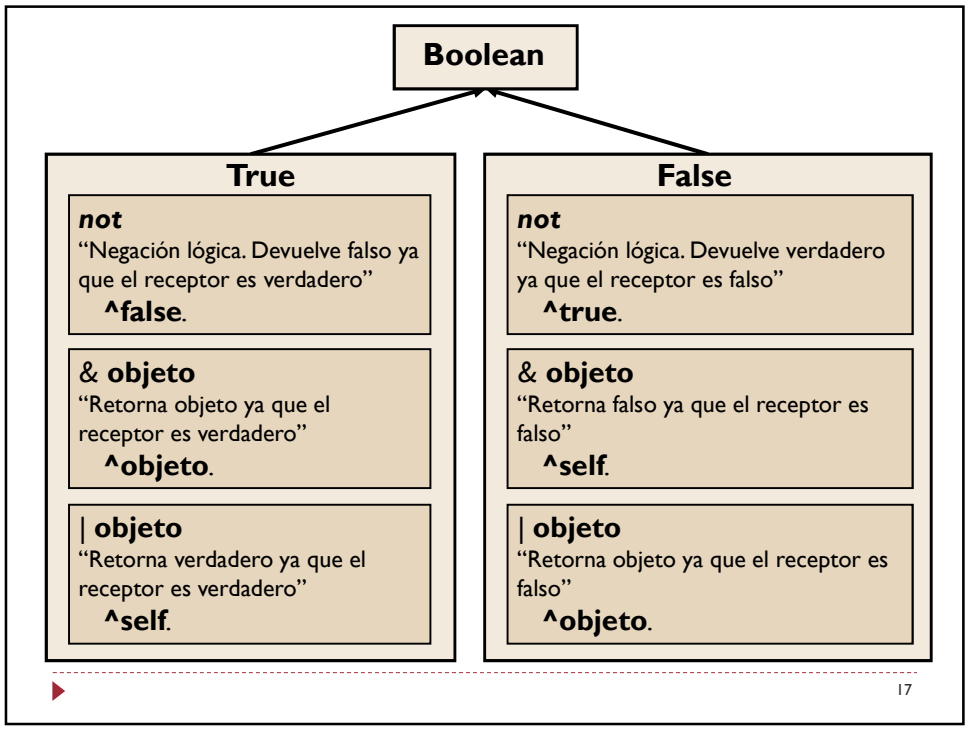
en las clases **True** y **False**.

(5 > 3) ***not***. → **false**

(3 > 5) ***not***. → **true**

```
Numero := 5.
Numero > 0 & (Numero < 50).
```
→ **true**

16



Iteración

La iteración en ***Smalltalk*** es soportada mediante el envío de mensajes a objetos de las clases **Number**, **BlockClosure** y **Collection**.

Los ciclos que se estudiarán son:

- Un equivalente al ciclo de repeticiones un determinado número de veces:

  to:do: **to:by:do:** **timesRepeat:**

- Un equivalente al ciclo *"while"*:

  whileTrue: **whileFalse:**

- **Recorrer los elementos de una colección (otra forma de iteración):**

  do: **detect:** **collect:** **reject:** **select:**

18

Ciclo de repeticiones un determinado número de veces — Ciclo *to:do:*

Este tipo de *iteración* se implementa como método de instancia de la clase **Number**.

- Sintaxis:

  valorInicial **to:** *valorFinal* **do:** **[** **:***parámetro* **|** <*cuerpo*> **]**

- Semántica:
  - El método ***to:do:*** evalúa una vez el bloque argumento por cada valor en el intervalo **[valorInicial , valorFinal]**, partiendo de **valorInicial** y siguiendo con los sucesivos valores obtenidos al incrementar en 1 el valor anterior.
  - El argumento del bloque toma sucesivamente los valores del intervalo en cada evaluación del bloque.

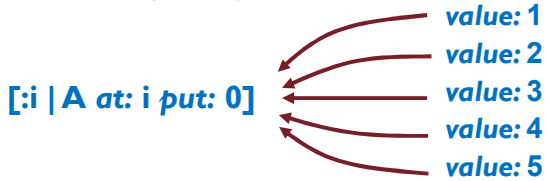
19

Ciclo de repeticiones un determinado número de veces — Ciclo *to:do:*

- Ejemplo:

```
A := Array new: 5.
1 to: 5 do: [:i | A at: i put: 0].
A.                                  → #(0 0 0 0 0)
```

En este ejemplo, la implementación del método ***to:do:*** en la clase **Number** se encarga de que los siguientes mensajes sean enviados al bloque argumento:



20

Ciclo de repeticiones un determinado número de veces — Ciclo *to:by:do:*

Sintaxis:

valorInicial **to:** *valorFinal* **by:** *incremento*
do: **[** :*parámetro* | <*cuerpo*> **]**

- El método ***to:by:do:*** es una variación del anterior que especifica la cantidad (***incremento***) en la cual el argumento del bloque debe incrementarse en cada evaluación.
- Para **incrementos positivos**, la evaluación repetitiva termina cuando el índice del ciclo **es mayor que el *valorFinal***. Para **incrementos negativos**, el ciclo finaliza cuando el índice del ciclo **es menor que *valorFinal***.

```
A := Array new: 5.
1 to: 3 by: 2 do: [:i | A at: i put: 0].
A.                                  → #(0 nil 0 nil nil)
```

21

Posibles implementaciones de los ciclos

Number

```
to: valorFinal do: bloque
  self <= valorFinal
      ifTrue: [ bloque value: self.
                  ^self + 1 to: valorFinal do: bloque]
      ifFalse: [^nil].
```

```
to: valorFinal by: inc do: bloque
        | dif |
  inc < 0 ifTrue: [ dif := self - valorFinal ]
          ifFalse: [ dif := valorFinal  - self ].
  dif < 0 ifTrue: [^nil ]
          ifFalse: [ bloque value: self.
                  ^self + inc to: valorFinal by: inc do: bloque].
```

22

Ciclo de repeticiones un determinado número de veces — CICLO *timesRepeat:*

Otro ciclo iterativo simple, con un número específico de repeticiones, se puede llevar a cabo usando el mensaje ***timesRepeat:*** definido en la clase **Integer**.

Este mensaje lleva como parámetro un bloque sin argumentos que se evaluará tantas veces como lo indique el entero receptor del mismo.

Sintaxis:
entero ***timesRepeat:*** *bloque*

```
| s i |
s := 'hola' copy.   i := 0.
s size timesRepeat: [i := i + 1. s at: i put: (s at: i) asUppercase].
 ^s
```

→ **'HOLA'**

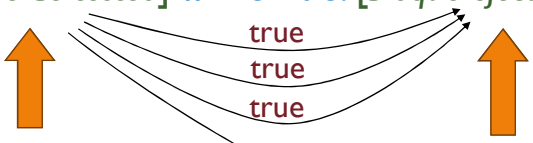
23

Ciclo While

Este tipo de *iteración* se implementa como método de la clase **BlockClosure.**

Existen dos mensajes a objetos de la clase **BlockClosure** que proveen un equivalente funcional de los ciclos *"while"* de otros lenguajes. Estos mensajes son ***whileTrue:*** y ***whileFalse:***

[bloque de testeo] ***whileTrue:*** *[bloque ejecutable]*



- ✓ bloque sin argumento.
- ✓ se evalúa en cada iteración.
- ✓ su evaluación retorna **true** o **false**

- ✓ bloque sin argumento.
- ✓ se evalúa en cada iteración.

24

BlockClosure

```
whileTrue: bloque
self value
        ifTrue: [ bloque value.
                  self whileTrue: bloque. ].
^nil
```

```
| vocales string indice |
vocales := 0.
indice := 1.
string := 'String con varias vocales'.
[ indice <= string size ]
    whileTrue: [ (string at: indice) isVowel
                    ifTrue: [vocales := vocales + 1].
                 indice := indice +1 ].
^vocales  → 8
```

25

Clase **Collection**

Collection es una **clase abstracta** que define el **protocolo común** de las clases diseñadas para almacenar **colecciones de objetos** (conjuntos, arreglos, listas, etc.)

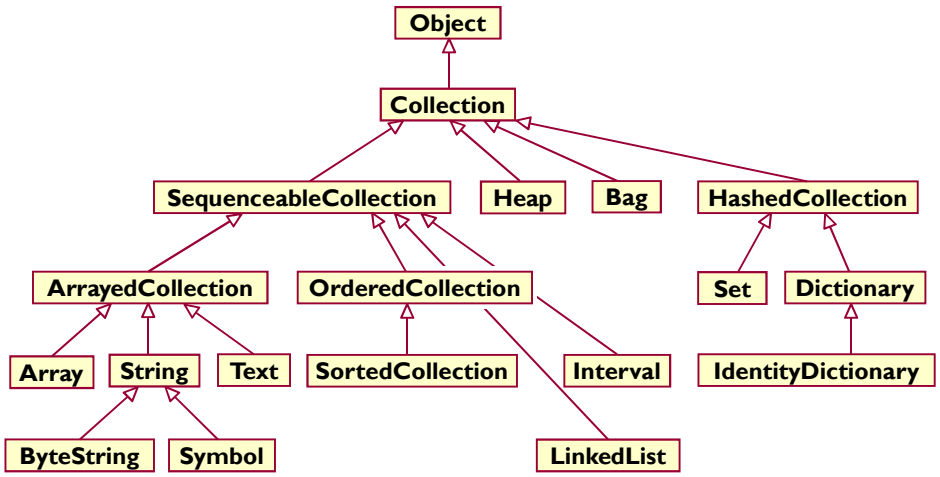
Las subclases de **Collection** proveen las distintas estructuras de datos básicas requeridas para almacenar los grupos de datos.

Collection provee el protocolo para:

- acceder a un elemento
- almacenar un elemento
- acceder en un orden particular a todos los elementos
- ejecutar un bloque para cada elemento
- etc.

26

Clase **Collection** (Pharo 11)



27

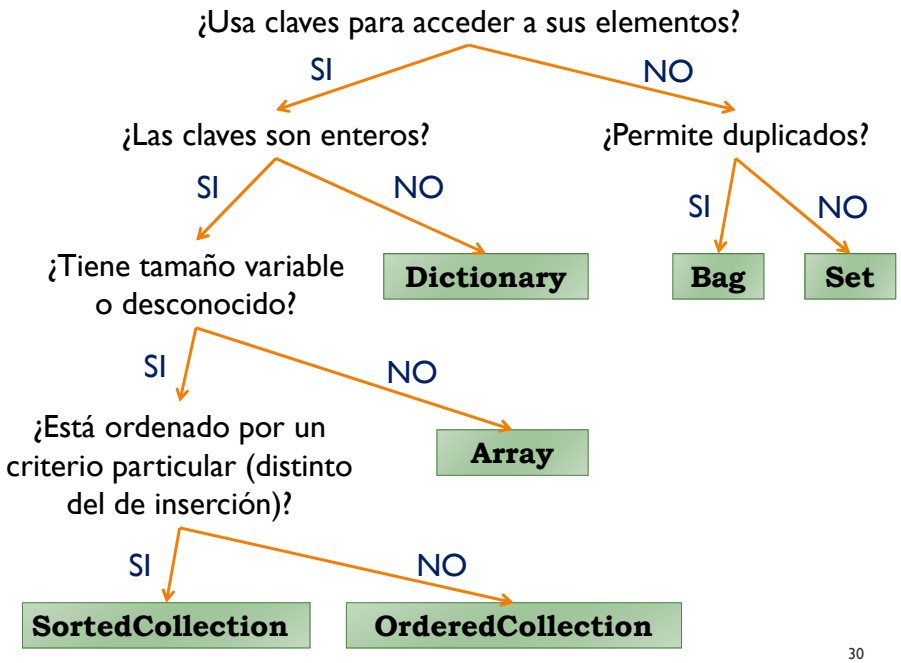
Tipos de Colecciones

SequenceableCollection	Colecciones cuyos elementos siguen una secuencia y son referenciados por números enteros.
ArrayedCollection	Colecciones de tamaño fijo
Array	Elementos pueden ser de cualquier tipo
String	Elementos = caracteres
IntegerArray	Elementos = enteros
Interval	Progresión aritmética finita
LinkedList	Lista enlazada con encadenamiento dinámico de elementos
OrderedCollection	Lista ordenada de tamaño variable, con orden determinado por la inserción de los elementos
SortedCollection	Lista ordenada automáticamente por un criterio especificado por un bloque

Tipos de Colecciones

Collection	
Bag	Bolsa de objetos, no hay ordenamiento, permite duplicados
HashedCollection	Clase abstracta para colecciones sin duplicados, cuyos objetos tienen el mismo código *hash* cuando son iguales
Set	Conjunto de objetos, no hay ordenamiento, no permite duplicados
Dictionary	Conjunto de asociaciones *clave-valor*, se almacena un único *valor* para cada *clave*, comparación con igualdad (=)
IdentityDictionary	Ídem anterior pero comparación entre *claves* con identidad (==)

29



30

Métodos de Instancia de **Collection**

Adición de Elementos

***add:* anObject**

Retorna **anObject**. Adiciona **anObject** a la colección receptora.

***addAll:* aCollection**

Retorna **aCollection**. Adiciona **aCollection** a la colección receptora.

31

Métodos de Instancia de **Collection**

Conversión

asArray

Retorna un **Array** conteniendo todos los elementos del receptor.

asBag
asOrderedCollection
asSet
asSortedCollection

***asSortedCollection:* aBlock**

Retorna un objeto de la clase **SortedCollection** conteniendo los objetos del receptor, ordenados de acuerdo con **aBlock**.

32

Métodos de Instancia de **Collection**

Iterativos

La clase **Collection** provee una variedad de esquemas de iteración. Todos los mensajes tienen como argumento un bloque con un parámetro que es evaluado una vez por cada elemento en el receptor.

objetoCollection ***selector:*** [*:parametro | <cuerpo bloque>*]

- ***do:*** aBlock
- ***collect:*** aBlock
- ***select:*** aBlock
- ***reject:*** aBlock
- ***detect:*** aBlock
- ***detect:*** aBlock ***ifNone:*** exceptionBlock

33

Iteraciones con colecciones

do: Evalúa el bloque argumento tantas veces como elementos aparezcan en la colección, asignando al parámetro del bloque cada elemento de la colección.

collect: Para cada elemento del receptor se evalúa el bloque con ese elemento como parámetro, y retorna una nueva colección que contiene los resultados de esas evaluaciones.

select: Crea y retorna una nueva colección con los elementos del receptor que verifican la expresión booleana final del bloque.

34

Iteraciones con colecciones

reject: Crea y retorna una nueva colección con los elementos del receptor que no verifican la expresión booleana final del bloque.

#(-1 2 -3 4 5) ***reject:*** [:i | i > 0] → #(-1 -3)

detect: Retorna el primer elemento de la colección que verifica la expresión booleana final del bloque.

#(-1 2 -3 4 5) ***detect:*** [:i | i > 0] → 2

35

Métodos de Instancia de **Collection**

Eliminación de componentes
(métodos que deben ser redefinidos por las subclases)

remove: **anObject**

Elimina **anObject** del receptor. Si no lo encuentra produce un error. Retorna **anObject**.

remove: **anObject** ***ifAbsent:*** **aBlock**

Ídem anterior, pero evalúa **aBlock** si no lo encuentra.

removeAll: **aCollection**

Elimina todos los elementos contenidos en **aCollection** del receptor. Retorna **aCollection**.

36

Métodos de Instancia de **Collection**

Consultas

***includes:* anObject**

Retorna true si el receptor contiene un elemento igual a **anObject**.

isCollection

Retorna **true** si el receptor es una instancia de **Collection** o una de sus subclases.

isEmpty

Retorna **true** si el receptor no contiene ningún elemento.

notEmpty

***occurrencesOf:* anObject**

Retorna el número de elementos del receptor que son iguales a **anObject**.

37

Clase **OrderedCollection**

Una **OrderedCollection** puede ser empleada como una lista, una pila o una cola.

A diferencia de las colecciones de tamaño fijo, ésta puede crecer incorporando nuevos elementos (arreglo dinámico).

Métodos de clase:

new

Crea una instancia sin elementos (vacía), reservando espacio interno para almacenar **10** elementos.

***new:* anInteger**

Crea una instancia sin elementos (vacía), reservando espacio interno para almacenar **anInteger** elementos.

38

Métodos de instancia de **OrderedCollection**

Adiciones

, aCollection

add: anObject

add: newObject ***after:*** oldObject

Retorna newObject.

add: anObject ***afterIndex:*** anInteger

Retorna anObject.

add: newObject ***before:*** oldObject

add: anObject ***beforeIndex:*** anInteger

addAllFirst: aCollection

addAllLast: aCollection

addFirst: anObject

addLast: anObject

39

Métodos de instancia de **OrderedCollection**

Ubicación y Cambio de elementos

after: anObject

after: anObject ***ifNone:*** aBlock

Retorna el elemento siguiente a anObject.

at: anInteger

at: anInteger ***put:*** anObject

Reemplaza el objeto ubicado en anInteger

before: anObject

before: anObject ***ifNone:*** aBlock

Retorna el elemento anterior a anObject.

40

Clase **SortedCollection**

A diferencia de **OrderedCollection**, las instancias de esta clase mantienen sus elementos ordenandos según un criterio que no coincide con el orden de inserción de elementos.

Métodos de clase:

new

Retorna una instancia de **SortedCollection** cuyos elementos están ordenados en forma ascendente

***sortBlock:* aBlock**

Retorna una instancia de **SortedCollection** cuyo ordenamiento estará dado por el bloque **aBlock**.

41

Métodos de instancia de **SortedCollection**

sortBlock

Retorna el bloque que determina el orden de la colección

***sortBlock:* aBlock**

Retorna el receptor. Asigna el bloque de ordenamiento del receptor como **aBlock** y reordena la colección.

***add:* anObject**
***addAll:* aCollection**

Agregan los elementos en forma ordenada a la colección.

~~***at:*** anInteger ***put:*** anObject~~
~~***add:*** newObject ***before:*** oldObject~~
~~***add:*** newObject ***after:*** oldObject~~
~~***addFirst:*** anObject~~
~~***addLast:*** anObject~~
~~***addAllFirst:*** aCollection~~
~~***addAllLast:*** aCollection~~

Reportan un error dado que el **sortBlock** define el orden de los elementos de la colección.

42

Clase **Dictionary**

- Una instancia de **Dictionary** es una colección de pares de objetos clave/valor (key/value).
- Las claves en una instancia de **Dictionary** deben ser únicas.
- Los elementos pueden ser agregados o extraídos como pares de objetos

  ***at:* key *put:* value**

  o como instancias de **Association**

  ***add:* anAssociation**
- Internamente una instancia de **Dictionary** almacena pares key/value como un conjunto de instancias de **Association**.

43

Clase **Dictionary** - Algunos métodos de instancia

keys

Retorna una instancia de **Array** que contiene todas las claves.

values

Retorna una instancia de **Array** que contiene todos los valores asociados a las claves del receptor.

at: aKey. ***at:*** aKey ***ifAbsent:*** aBlock ***keyAtValue:*** anObject ***keyAtValue:*** anObject ***ifAbsent:*** aBlock	Acceder a elementos.
removeKey: aKey. ***removeKey:*** aKey ***ifAbsent:*** aBlock	Eliminar asociaciones clave-valor

44