

PARADIGMAS DE PROGRAMACIÓN

Smalltalk/V - Smalltalk Express
2008

Paradigmas de Programación

SMALLTALK

1. **SMALLTALK** es un lenguaje orientado a objetos puro, pues todas las entidades que maneja son objetos. El lenguaje se basa en conceptos tales como objetos y mensajes.
2. **SMALLTALK** es descendiente del lenguaje SIMULA y tiene sus orígenes en el Centro de Estudios de Palo Alto de Xerox, en los comienzos de 1970. Su desarrollo se basa en gran parte en las ideas de Alan Kay. Las tres versiones principales del lenguaje son **SMALLTALK-72**, **SMALLTALK-76** y **SMALLTALK-80**.
3. **SMALLTALK** es mucho más que un lenguaje de programación, es un ambiente completo de desarrollo de programas. Éste integra de una manera consistente características tales como un editor, un compilador, un debugger, utilitarios de impresión, un sistema de ventanas y un manejador de código fuente.
4. **SMALLTALK** elimina la frontera entre aplicación y sistema operativo, modelando todos los elementos como objetos.

Paradigmas de Programación

La programación en **SMALLTALK** requiere de al menos los siguientes conocimientos:

1. los conceptos fundamentales del lenguaje: manejo de clases y objetos, mensajes, clases y herencia.
2. la sintaxis y la semántica del lenguaje.
3. cómo interactuar con el ambiente de programación de **SMALLTALK** para construir nuevas aplicaciones **SMALLTALK**.
4. las clases fundamentales del sistema, tales como numéricas, colecciones, gráficas y las clases de interfase del usuario.

Diseñar nuevas aplicaciones **SMALLTALK**, requiere de conocimientos sobre las clases existentes en el sistema **SMALLTALK**.

La programación en **SMALLTALK** se denomina

Programación por extensión

Las nuevas aplicaciones son construidas por extensión de las bibliotecas de clases de **SMALLTALK**.

Paradigmas de Programación

SMALLTALK: CONCEPTOS BÁSICOS

Los conceptos básicos son:

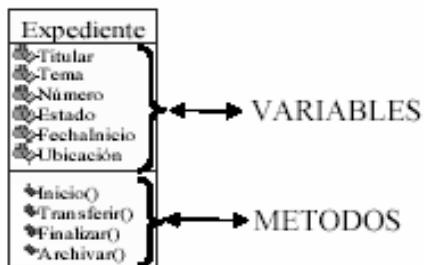
- Clase
- Instancia
- Mensaje
- Herencia

La programación en SMALLTALK consiste en:

- Crear clases.
- Crear instancias.
- Especificar la secuencia de mensajes entre objetos.

Paradigmas de Programación

Una **clase** contiene toda la información necesaria para **crear nuevos objetos**
y
permite agrupar bajo un mismo nombre
las **variables** y los **métodos** que manipulan esas variables.



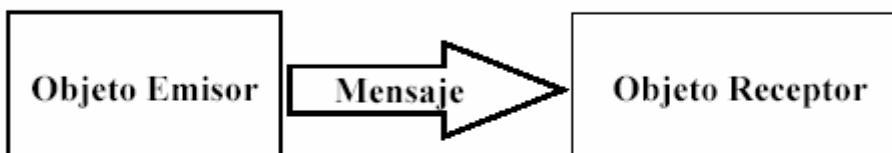
A partir de una clase se pueden crear tantos elementos como se deseen. A estos elementos creados se los denomina
Instancias u Objetos de la clase

Por lo tanto

dos instancias diferentes de una misma clase
comparten los mismos métodos y la misma lista de variables con valores diferentes.

MENSAJE

Es una petición a un **objeto** para que brinde algún **servicio** que el objeto puede realizar.



El texto del envío de un mensaje está compuesto de:

- el nombre del objeto destinatario denominado **RECEPTOR DEL MENSAJE**.
- un **SELECTOR**, que es el nombre del **método** (punto de entrada en el objeto receptor).
- y eventualmente, **PARÁMETROS** del método que se quiere activar, denominados **ARGUMENTOS DEL MENSAJE**.

HERENCIA

- Permite crear nuevas clases partiendo de otras previamente definidas con características semejantes a la que se quiere crear.
- El mecanismo de herencia nos permite definir las propiedades y comportamientos particulares de una nueva clase de objetos y **heredar** las propiedades y comportamientos comunes ya existentes.

SINTAXIS SMALLTALK

La sintaxis de SMALLTALK es:

objeto mensaje

- 3 factorial
significa que el mensaje factorial es enviado al objeto 3 (instancia de la clase Integer) y por lo tanto se ejecutará el método de nombre factorial que se encuentra en la clase Integer o en sus superclases.
- 'HOLA' size
- #(4 8 6 3) at: 2
- \$A asciiValue

Paradigmas de Programación

EXPRESIÓN

Smalltalk es un lenguaje basado en expresiones. Una expresión es una secuencia de caracteres que puede ser evaluada.

Hay cuatro tipos de expresiones:

1. Literales
2. Nombres de Variables
3. Expresiones de mensajes
4. Expresiones de bloque

Paradigmas de Programación

VARIABLES Y ASIGNACIÓN

TODA VARIABLE EN SMALLTALK ES UN OBJETO PUNTERO QUE PERMITE REFERENCIAR OTRO OBJETO

Los nombres de variables en SMALLTALK son identificadores que consisten en una secuencia de letras y dígitos que **comienza con una letra**.

EXPRESIÓN DE ASIGNACIÓN

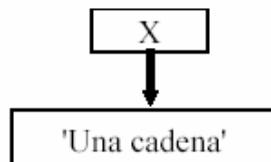
variable := expresión (SMALLTALK V)

Paradigmas de Programación

VARIABLES Y ASIGNACIÓN

X := 'Una cadena'

la variable X apunta al objeto 'Una cadena'



Una variable puede contener diferentes punteros a objetos a lo largo de la ejecución

X := 'Una cadena'.
X := 23 factorial.
X := #(1 2 3).

Paradigmas de Programación

TIPOS DE VARIABLES

Hay dos tipos de variables:

- **variables privadas**
- **variables compartidas**

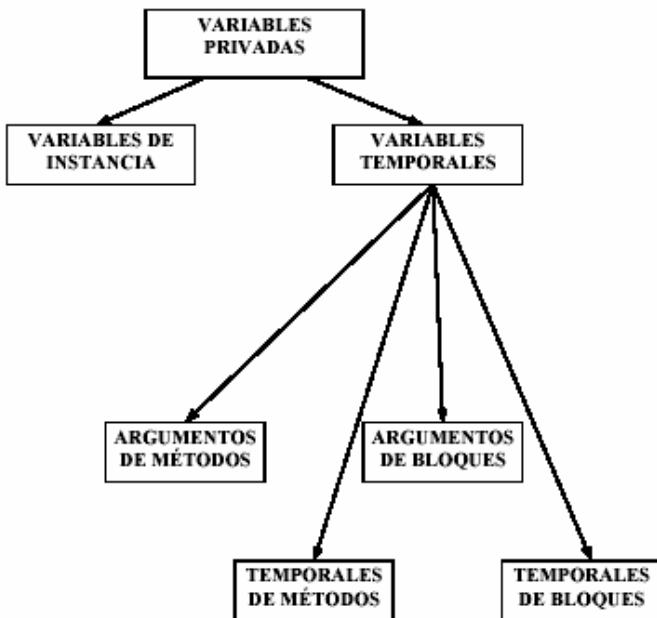
Las **VARIABLES PRIVADAS** sólo son accesibles por **un objeto**.

Las **VARIABLES COMPARTIDAS** pueden ser accedidas por **más de un objeto**.

Los nombres de las **variables privadas** deben comenzar con **una letra minúscula** y los nombres de las **variables compartidas** deben comenzar **con letra mayúscula**.

Paradigmas de Programación

CLASIFICACIÓN DE VARIABLES PRIVADAS



Paradigmas de Programación

VARIABLES DE INSTANCIA

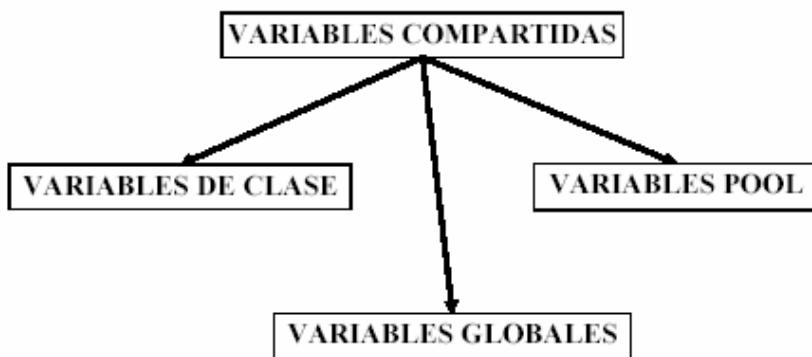
- Aunque todas las instancias de una clase tienen el mismo conjunto de variables de instancia, **sus valores son únicos a cada una de las mismas.**
- Sólo pueden ser accedidas en métodos de instancia disponibles en la clase.

Las variables de instancia existen durante todo el tiempo de vida de un objeto y representan el estado del objeto.

VARIABLES TEMPORALES

- Son las variables que están definidas en los **métodos, bloques o programas** Smalltalk. Se utilizan en una actividad dada y después se destruyen.
- Representan un estado transitorio de un objeto, existen mientras dura la activación del **método, bloque o programa.**

VARIABLES COMPARTIDAS



VARIABLES DE CLASE

- Son variables compartidas por todas las instancias de una clase y sus subclases.
- Tienen el mismo valor para todas las instancias.
- Sólo pueden ser accedidas por los métodos de clase y los métodos de instancia de la clase y sus subclases.
- Se emplean para compartir un valor en el contexto de una clase

Paradigmas de Programación

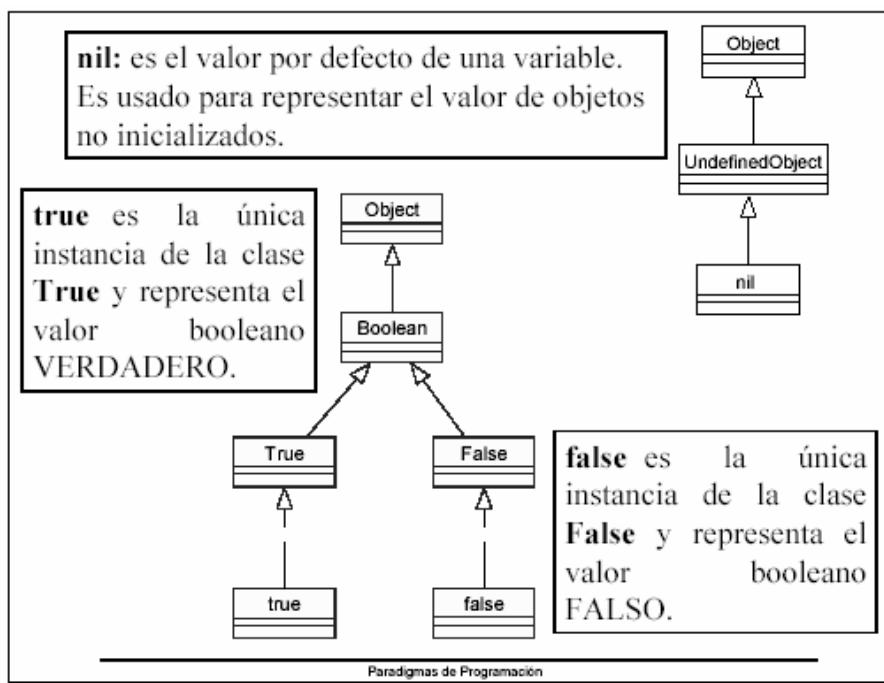
PSEUDO VARIABLES

Existen cinco pseudovariables:

nil, true, false, self y super.

- el valor de una **pseudovariable** no puede cambiarse con una expresión de asignación.
- **nil, true y false** son instancias de clases.
- **self y super** son variables que refieren a distintos objetos dependiendo del contexto donde se usen.
- las cinco son palabras reservadas y son globales

Paradigmas de Programación



self: es usada en métodos; su valor es siempre el **objeto** que recibe el mensaje que causa que el método que contiene **self** sea ejecutado.

super

super es usada en el cuerpo de métodos.

SmallTalk utiliza otra estrategia de búsqueda para ubicar el método con que **super** va a responder al mensaje.

La búsqueda comienza en la superclase inmediata (superior) de la clase que contiene el método en el cual **super aparece.**

Los mensajes a **super** son utilizados cuando se quiere utilizar un método de una superclase que es redefinido en una subclase.

EXPRESIONES DE MENSAJE

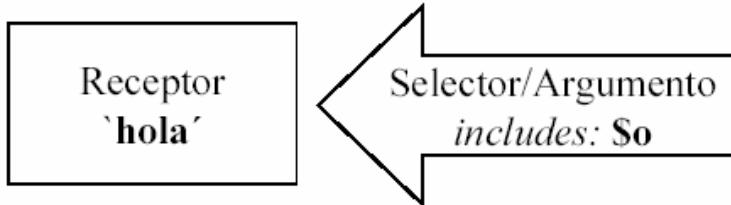
Las expresiones de mensajes en SMALLTALK describen que objeto es el receptor del mensaje, el nombre del mensaje y los argumentos

16.79 rounded.

‘hola que tal’ size.

numerador + (unaFracción verNumerador).

‘hola’ includes: \$o.



Paradigmas de Programación

TIPOS DE MENSAJE

SMALLTALK soporta tres tipos primitivos de mensajes:

- **UNARIOS** (unary)
- **BINARIOS** (binary)
- **de PALABRA CLAVE** (keyword)

Paradigmas de Programación

MENSAJES UNARIOS

Los mensajes **unarios no tienen argumentos**, su sintaxis comprende solamente un receptor y un selector.

objeto	selector	
7.5 isInteger → false	7.5	<i>isInteger</i>
\$A asLowerCase → \$a	\$A	<i>asLowerCase</i>
\$u isVowel → true	\$u	<i>isVowel</i>
3 factorial → 6	3	<i>factorial</i>

MENSAJES BINARIOS

Además del **receptor** y **selector**, la sintaxis de los mensajes binarios tiene **un único argumento**. Los selectores para los mensajes binarios son caracteres especiales simples o dobles.

Los selectores de carácter simple incluyen operadores de comparación y aritméticos tales como:

+ - * / < > =

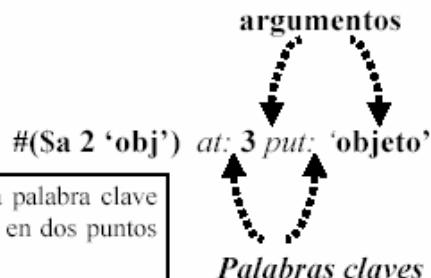
Los selectores de carácter doble incluyen operadores tales como:

~= (not =) <= // (división entera)

mensaje binario	objeto	selector	argumento
1 + 6.2e2 → 621.0	1	+	6.2e2
‘abc’ ~= ‘de’ → true	‘abc’	~=	‘de’
5 // 2 → 2	5	//	2
\$a < \$b → true	\$a	<	\$b

MENSAJES DE PALABRA CLAVE

Estos mensajes contienen una o más palabras claves, donde cada palabra clave tiene un argumento simple asociado con ella.



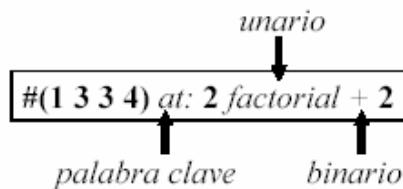
El selector en un mensaje de más de una palabra clave se forma concatenando todas las palabras claves.

Selector → at:put:

Paradigmas de Programación

En **SMALLTALK** la relación de **precedencia en la evaluación de expresiones** es la siguiente:

1. expresiones entre paréntesis.
2. expresiones unarias (evaluadas de izquierda a derecha).
3. expresiones binarias (evaluadas de izquierda a derecha).
4. expresiones de palabra clave.
5. expresiones de asignación.



Paradigmas de Programación

MÉTODOS

- Los métodos de una clase son los procedimientos que se activan ante la recepción de un mensaje.
- La forma en que un objeto responde a un mensaje está descripta en un Método.
- Cada clase contiene la lista de métodos que le permiten a sí misma y a sus instancias responder a los mensajes que le son enviados.

CLASIFICACIÓN de MÉTODOS:

Los **mensajes** pueden ser enviados a las **clases** o las **instancias**. Los **métodos** que se activan ante dichos **mensajes** también deberán estar asociados a las **clases** o a las **instancias**:

- **Métodos de Clase:** sólo empleado por clases
- **Métodos de Instancia:** sólo empleado por instancias

Paradigmas de Programación

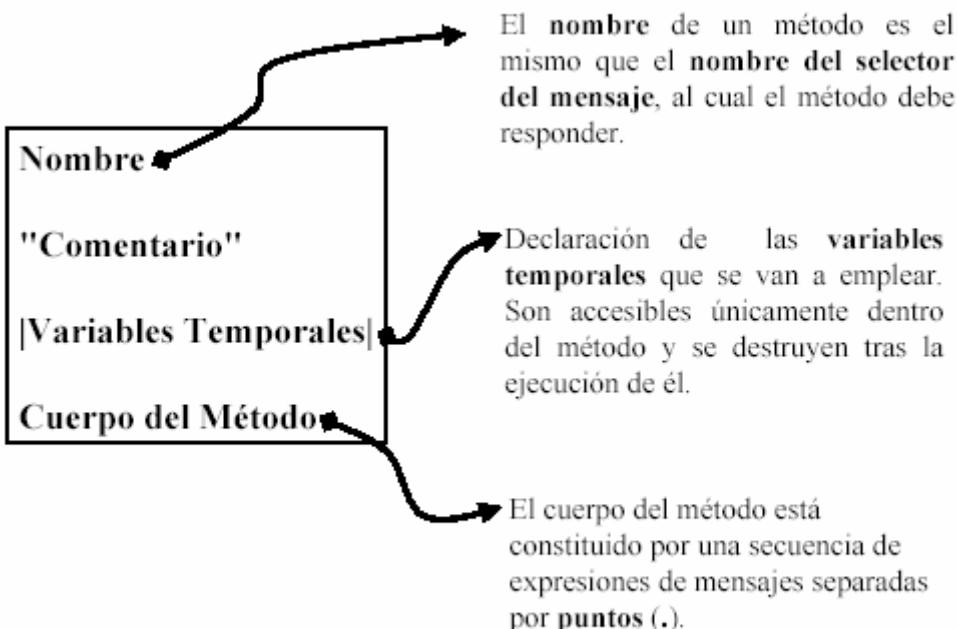
ACTIVACIÓN DE METODOS

Cuando una instancia recibe un mensaje, el selector del mensaje podrá o no corresponder con un método local a la clase de la instancia:

- **Si el selector corresponde** a un método local de la clase del objeto, entonces hay activación de ese método.
- **Si el selector no corresponde** a un método local a la clase del objeto, entonces se busca el método en la superclase de la clase del objeto y así recursivamente hasta encontrar la primera clase del árbol de herencia que posea un método idéntico al selector del mensaje.

Paradigmas de Programación

SINTAXIS DE UN MÉTODO



Paradigmas de Programación

EVALUACIÓN DE MÉTODOS Y EXPRESIONES DE RETORNO

Cuando un método finaliza su ejecución **siempre** retorna al emisor del mensaje **un objeto**.

El **objeto** returnedo por un método es:

1. **el objeto receptor del mensaje**

o

2. el resultado de una **expresión de retorno** (expresión de mensaje precedida por ^).



Paradigmas de Programación

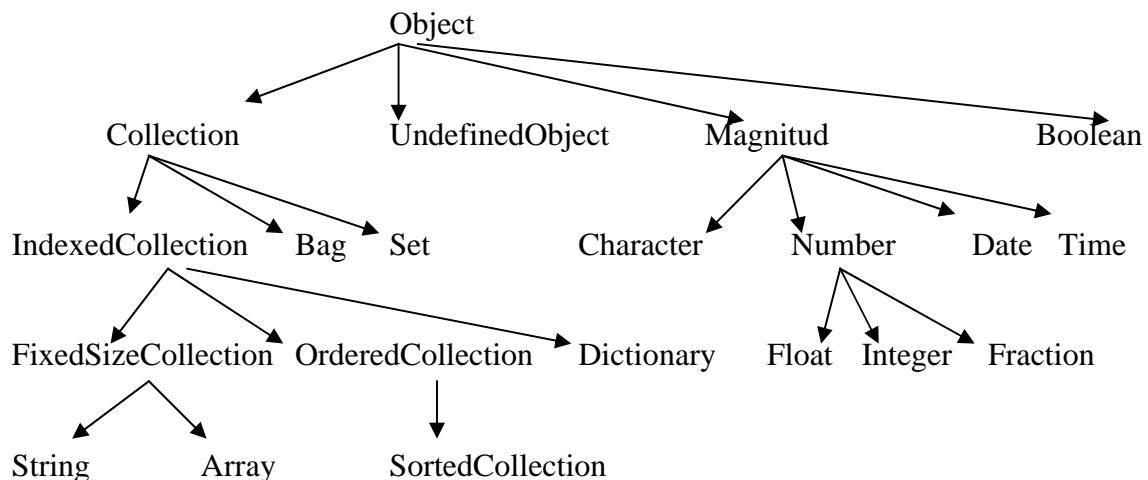
BIBLIOTECA DE CLASES DE SMALLTALK

Funcionalidades básicas:

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

Paradigmas de Programación

Jerarquía de clases (subconjunto)



NÚMEROS

Descripción:

Los números son objetos que representan valores numéricos y responden a mensajes que calculan resultados matemáticos.

Representación:

Secuencia de dígitos precedidos o no de un signo '-' y/o con un punto decimal.

Ejemplos:

27.5 -35.7 -128 25

Notación científica: 25.53e2 -8.126e-3.

Paradigmas de Programación

CARACTERES

Descripción:

Los caracteres son objetos que representan los símbolos que forman un alfabeto.

Representación:

Expresión precedida por el signo \$ seguida por cualquier carácter.

Ejemplos:

\$a \$A \$3 \$+ \$\$

Paradigmas de Programación

La variable de instancia de los objetos de esta clase es asciiInteger que equivale a la representación entera del carácter correspondiente.

Los caracteres son tomados como objetos inmutables o constantes, que no pueden variar.

Los objetos de esta clase soportan operaciones de comparación además de :

<code>unObjeto isUpperCase</code>	(devuelve V o F si está en mayúscula o no)
<code>unObjeto isLowerCase</code>	(devuelve V o F si está en minúscula o no)
<code>unObjeto asUpperCase</code>	(lo pasa a mayúscula)
<code>unObjeto asLowerCase</code>	(lo pasa a minúscula)
<code>unObjeto isAlphabetic</code>	(devuelve V si es letra del alfabeto, F en caso contrario)
<code>unObjeto asciiValue</code>	(devuelve el ascii correspondiente)
<code>unObjeto isVowel</code>	(devuelve V o F si es o no una vocal minúscula o mayúscula)
<code>unObjeto isDigit</code>	(devuelve V o F si es o no \$0 o \$1 o ... \$9)
<code>unObjeto isAlphanumeric</code>	(devuelve V o F si contiene o no letras y numeros)
<code>unObjeto isLetter</code>	(devuelve V o F si es \$a, \$b,..., \$A, \$B,... o no lo es)
<code>unObjeto isSeparator</code>	(devuelve V o F si es o no: un espacio, tabulador, CR, salto)

SECUENCIA DE CARACTERES

Descripción:

Son objetos que representan una cadena de caracteres. Responden a mensajes para acceder a caracteres individuales, sustituir secuencias, compararlas con otras secuencias y concatenarlas.

Representación:

Secuencia de caracteres encerrados entre apóstrofes
'secuencia'

Ejemplos:

'Hola' 'secuencia de caracteres' 'Region 001'

Se pueden concatenar cadenas separándolas con coma:

'Esto es', ' una tira', ' de caracteres'

Produce como resultado: **'Esto es una tira de caracteres'**.

Paradigmas de Programación

Un objeto de la clase String es una cadena de caracteres encerrados entre comillas simples. Es un objeto indexado y cada componente es un carácter:

Ejemplo: 'hola' es igual a \$h, \$o, \$l, \$a ya que la coma concatena.

Un objeto de esta clase además de las operaciones de comparación, entiende los siguientes mensajes, entre otros:

<code>unObjeto isUpperCase</code>	
<code>unObjeto isLowerCase</code>	
<code>unObjeto asLowerCase</code>	
<code>unObjeto asUpperCase</code>	
<code>unObjeto size</code>	(devuelve la cantidad de caracteres corrientes)
<code>unObjeto, unObjeto</code>	(concatena)

unObjeto <i>a</i> : unaPosic	(me devuelve el carácter de la posición unaPosic)
unObjeto <i>at: unaP put: unCar</i>	(accede al elemento de la posición unaP y lo modifica con unCar)
unObjeto <i>copyFrom: aPos1 to : aPos2</i>	(retorna la subcadena comprendida entre los índices indicados)
unObjeto <i>asInteger</i>	(si el contenido de unObjeto es numérico hace la conversión)
unObjeto <i>asFloat</i>	(ideo si es real)
unObjeto <i>asDate</i>	(lo convierte a fecha si el contenido aparece como mes, día y año en ese orden y separados por blancos)

Mensajes que entienden todos los objetos

unObjeto <i>new</i>	(crea una instancia vacía)
unObjeto <i>class</i>	(me devuelve la clase del objeto receptor)
unaClase <i>superClass</i>	(me devuelve la superclase de una clase)
unObjeto <i>isKindOf: aClass</i>	(me devuelve V o F si unObjeto pertenece a la clase aClass o no)
unObjeto <i>isNil</i>	(devuelve true o false si es nil o no)
unObjeto <i>notNil</i>	
unObjeto <i>yourself</i>	(devuelve el objeto receptor del mensaje)
unObjeto <i>inspect</i>	(abre una ventana especial que me permite inspeccionar al objeto receptor para ver su estado y/o modificarlo)

Resumen de métodos más usados.

Literales	
"Esto es un comentario"	comentario.
'Hola, esto es una cadena. ¿No es cierto?'	constante de cadena (consta de caracteres).
\$a, \$x, \$;	constantes de carácter.
#(1 2 3 4)	array literal (lista constante).
Valores Booleanos	
true, false	constantes.
unObjeto = unObjeto	igual.
unObjeto ~= unObjeto	no igual.
unObjeto == unObjeto	idéntico.
unObjeto ~~ unObjeto	no idéntico.
unObjeto > unObjeto	mayor.
unObjeto >= unObjeto	mayor o igual.
unObjeto < unObjeto	menor.
unObjeto <= unObjeto	menor o igual.
unBooleano unBooleano	o.
unBooleano & unBooleano	y.
unBooleano or: [Expresión Booleana]	cortocircuito o ^
unBooleano and: [Expresión Booleana]	cortocircuito y ^
unBooleano not	negación.
unObjeto isNil	devuelve true si el receptor es el objeto nil, si no false..
unObjeto notNil	devuelve true si el receptor no es el objeto nil, si no false.
^ Puede que no se evalúe la segunda expresión.	

Números

<code>+,-,*,/</code>	<i>operaciones típicas.</i>
<code>unNumero // unNumero</code>	<i>divide y trunca hacia -∞</i>
<code>unNumero \ unNumero</code>	<i>resto tras truncar por //</i>
<code>unNumero negated</code>	<i>invierte el signo del numero.</i>
<code>unNumero abs</code>	<i>valor absoluto.</i>
<code>unNumero sqrt</code>	<i>raíz cuadrada.</i>
<code>unNumero raisedTo: unNumero</code>	<i>eleva el primer número a la potencia indicada por el segundo.</i>
<code>unNumero squared</code>	<i>eleva un número al cuadrado</i>

Conversiones

<code>unNumero o unaCadena asInteger</code>	<i>convierte a objeto entero.</i>
<code>unNumero o unaCadena asFloat</code>	<i>convierte a objeto punto flotante.</i>
<code>unEntero asCharacter</code>	<i>convierte a objeto carácter.</i>
<code>unCarácter asciiValue</code>	<i>inverso de asCharacter.</i>
<code>unObjeto printString</code>	<i>devuelve una cadena que representa al receptor.</i>

Entrada y Salida

<code>Prompter prompt: unTexto caption: unTítulo</code>	<i>abre una ventana con unTítulo, mostrando unTexto, a la espera del ingreso de una cadena que devuelve.</i>
<code>Prompter on: unValor prompt: unaCadena caption: unaCadena</code>	<i>pide un dato para ser ingresado.</i>
<code>Transcript nextPutAll: unaCadena</code>	<i>añade una cadena a la ventana Transcript.</i>
<code>Transcript nextPut: unCarácter</code>	<i>añade un carácter a la ventana Transcript.</i>
<code>Transcript cr</code>	<i>añade un retorno de carro.</i>
<code>Transcript space</code>	<i>añade un espacio en blanco.</i>
<code>Transcript tab</code>	<i>añade un tabulador.</i>
<code>Transcript comma</code>	<i>añade una coma.</i>
<code>unObjeto printOn: Transcript</code>	<i>imprime una representación textual del receptor en Transcript.</i>

Asignación, Retorno y Estructuras de Control

<code>variable := unObjeto ^unObjeto</code>	<i>una asignación. una sentencia de devolución.</i>
<code>unBooleano ifTrue: [Sentencias]</code>	
<code>unBooleano ifFalse: [Sentencias]</code>	
<code>unBooleano ifTrue: [Sentencias] ifFalse: [Sentencias]</code>	
<code>[Expresión Booleana] whileTrue: [Sentencias]</code>	
<code>[Expresión Booleana] whileFalse: [Sentencias]</code>	
<code>unEntero timesRepeat: [Sentencias]</code>	
<code>unNúmero to: unNúmero by: unNúmero do: [:índice Sentencias]</code>	
<code>unNúmero to: unNúmero do: [:índice Sentencias]</code>	

Colecciones

UnaClaseColeccion new: tamaño	genera una colección de tamaño fijo.
UnaClaseColeccion new	genera una colección de tamaño dinámico.
SortedCollection new: [:unElem :otroElem Comparación entre ambos]	genera una colección ordenada según el criterio de comparación indicado en el bloque.
unaColeccion size	devuelve la cantidad de elementos de la colección.
unaColeccion isEmpty	devuelve true si la colección no tiene elementos, si no false
unaColeccion notEmpty	devuelve true si la colección tiene elementos, si no false
unaColeccion copyFrom: unEntero to: unEntero	genera una colección con los elementos entre las 2 posiciones.
unaColeccion at: clave	devuelve el elemento en la clave.
unaColeccion at: clave put: unObjeto	sustituye el elemento de la clave por el indicado.
unaColeccion add: unObjeto	añade un elemento al final (crece la colección).
unaColeccion addAll: unaColeccion	añade todos los elementos al final (crece la colección).
unaColeccion addFirst: unObjeto	añade un elemento al principio (crece la colección).
unaColeccion addLast: unObjeto	añade un elemento al final (crece la colección).
unaColeccion remove: unObjeto	recupera y elimina el elemento (reduce la colección).
unaColeccion removeFirst	recupera y elimina el primer elemento (reduce la colección).
unaColeccion removeLast	recupera y elimina el último elemento (reduce la colección).
unaColeccion removeAll	recupera y elimina todos los elementos (reduce la colección).
unaColeccion first	devuelve el primer elemento de la colección.
unaColeccion last	devuelve el último elemento de la colección.
unaColeccion asSortedCollection: [:unElem :otroElem Comparación entre ambos]	devuelve una colección ordenada según el criterio de comparación indicado en el bloque.
unaColeccion asSet	devuelve la colección como Set.
unaColeccion asOrderedCollection	devuelve la colección como OrderedCollection.
unaColeccion asBag	devuelve la colección como Bag.
unaColeccion asArray	devuelve la colección como Array.
unaColeccion select: [:unElem Expresión booleana]	devuelve una colección con los elementos para los que la expresión booleana devuelve true.
unaColeccion reject: [:unElem Expresión booleana]	devuelve una colección con los elementos para los que la expresión booleana devuelve false.
unaColeccion detect: [:unElem Expresión Booleana]	devuelve el primer elemento para el que la expresión booleana devuelve true. En caso de que ninguno cumpla evalúa el segundo bloque.
ifNone: [Sentencias]	
unaColeccion collect: [:unElem Expresión]	devuelve una colección con los objetos resultantes de la evaluación de la expresión para cada elemento.
unaColeccion do: [:unElem Expresión]	Evaluá la expresión con cada uno de los elementos de la colección.
unaColeccion includes: unObjeto	Devuelve true o false si el objeto se encuentra o no en la colección.
unaColeccion occurrencesOf: unObjeto	Devuelve la cantidad de ocurrencias del objeto en la colección.
unaColeccion inject: unValorInicial into: [:elValorIntermedio :índice expresiónQueActualiza]	Devuelve el último valor calculado (especial para ciclos de acumulación)

Nota: las listas y las colecciones ordenadas tienen el rango entero comprendido entre 1 y el tamaño de la colección.

Operaciones de recorrido de colecciones en general.

* *do* : [:miembro| unBloque] donde miembro es el parámetro de bloque

Este mensaje liga automáticamente cada elemento de la colección a la variable local miembro y ejecuta el bloque unBloque sobre miembro.

Ejemplos :

Suponiendo que direcciones es una colección con datos,

direcciones do : [: dato | dato printOn : Transcript].

Imprime cada elemento de la colección en la ventana del SystemTranscript

datos := #(1 2 3 4 5 6 7 8 9 10).

suma := 0.

suma los ele. de la colección

datos do : [: i | suma := suma + i].

^ suma

datos := #(1 2 3 4 5 6 7 8 9 10).

nueva := OrderedCollection new.

datos do : [: elem | nueva add : (elem * 2)].

^ nueva

devuelve una colec. nueva con el doble de cada elemento de datos

* *unVi to : unVf do* : [:i | unBloque].

Ejecuta unBloque tantas veces como sea necesario para que i vaya desde un valor inicial unVi hasta alcanzar un valor final unVf con paso 1 . (acá i funciona como índice o variable de control de alguna manera)

Ejemplos :

nueva := OrderedCollection new.

1 to : 5 do : [:j | nueva add : j factorial].

^ nueva

devuelve una nueva colección con el factorial de los primeros 5 números naturales

otra := OrderedCollection new.

2 to : 10 by : 2 do : [:k | otra add : k squared].

^ otra

devuelve una colección nueva con el cuadrado de los números pares entre 2 y 10

* *collect: aBlock* para cada elemento de la colección receptora evalúa aBlock usando dicho elemento como argumento y los devuelve en una colección de la misma clase que la colección receptora del mensaje.

datos := #(1 2 3 4 5 6 7 8 9 10).
 nueva := OrderedCollection new.
 nueva:= datos collect : [: elem | (elem * 2)].
 ^ nueva

devuelve una colección nueva con el doble de cada elemento de datos

* *select*: aBlock devuelve una nueva colección de la misma clase que la colección receptora del mensaje pero con los elementos del receptor que cumplen con aBlock

Ejemplo:
 datos := #(1 2 3 4 5 6 7 8 9 10).
 nueva := OrderedCollection new.
 nueva:= datos select: [: elem | (elem > 2)].
 ^ nueva

devuelve una colección nueva con los elementos de datos mayores a 2

* *reject*: aBlock devuelve una colección de la misma clase que la colección receptora del mensaje, con los elementos del receptor que no cumplen con aBlock

Ejemplo:
 datos := #(1 2 3 4 5 6 7 8 9 10).
 nueva := OrderedCollection new.
 nueva:=datos reject : [: elem | (elem = 2)].
 ^ nueva

devuelve una colección nueva con cada elemento de datos distinto de 2

* *detect*: aBlock devuelve el 1º elemento del receptor que cumple con aBlock, sino hay da error

* *detect*: aBlock *ifNone*: otroBlock idem anterior pero sino hay, ejecuta otroBlock

Ejemplos:
 datos := #(1 2 3 4 5 6 7 8 9 10).
 datos detect : [: elem | (elem < 2)].

devuelve el 1º elem de datos que sea menor que 2 si existe o
error en c.c

datos detect : [: elem | (elem < 2)] devuelve el 1º elem. De datos <2 o nil sino hay
ifNone:[^ nil].

* *sortBlock*: aBlock ordena una SortedCollection con el criterio dado por aBlock

Ejemplo:
 p:= SortedCollection new.
 suponer que acá se hace la carga de p con n°....
 p sortBlock: [a: b: | a <= b].

Ordena la colección ascendente

..... suponer ahora que p se carga con objetos de la clase Cliente
 p sortBlock: [x: y: | (x name size) < (y name size)] ordena alfabet. por apellido de los clientes

Operaciones con diccionarios

Dictionary <i>new</i>	crea una instancia vacía
unDic <i>at:</i> unaClave	devuelve el valor asociado a la clave
unDic <i>at: ifAbsent:[bloque]</i>	idem pero si la clave no existe no da error
unDic <i>at: unaClave put: unValor</i>	coloca clave y valor asociado
unDic <i>removeKey: unaClave</i>	elimina clave y valor, si no existe da error
unDic <i>removeKey: unaClave ifAbsent:[bloque]</i>	idem pero no da error si la clave no existe
unDic <i>do:[bloque]</i>	recorre los elementos del dicc y toma valores no claves
unDic <i>keysDo:[bloque]</i>	idem pero me permite tomar claves y valores asociados

Ejemplos:

unDic:=Dictionary new.

Otro:=Dictionary new.

unDic at: #manzana put:'roja'; at:#pera put:'verde'.

otro at: 'sauce' put:'arbol'; at:'pollo' put:'ave'.

unDic do:[:elem| elem printOn: Transcript]. Imprime los valores asociados a cada clave. La variable elem apunta a cada valor

otro keysDo:[:key| key printOn: Transcript.
 (otro at: key) printOn: Transcript].

Imprime las claves y sus valores asociados. La variable key apunta a cada clave.