

Smalltalk

Smalltalk es un lenguaje orientado a objetos puro: todas las entidades manipuladas son objetos.

Ej. la expresión

$$2 + 4$$

es el resultado de enviar el mensaje '+' con argumento 4 al objeto 2.

Smalltalk Squeak es un entorno de desarrollo de aplicaciones orientadas a objetos.

Incluye:

- Un lenguaje: Smalltalk.
- Un modelo de objetos, que define cómo se comportan los objetos.
- Un entorno de programación, que permite añadir, modificar y borrar clases.
- Una jerarquía de clases, que pueden ser reutilizadas directamente y de las que se puede consultar su código.
- Un entorno de ejecución interpretado.

Objetos

Todo el procesamiento en Smalltalk se lleva a cabo mediante el envío de mensajes a objetos. Los objetos son instancias de una clase concreta.

'esto es una cadena'	una cadena de caracteres
1234	un número entero
1.234e-5	un número real
\$a	un carácter
#(1 2 3)	un array de tres elementos
#('array' 'de' 4 'elementos')	un array de cuatro elementos
#(1 ('dos' 3) 'cuatro')	un array de tres elementos
true	el valor lógico verdadero
false	el valor lógico falso
[i := i + 1]	un bloque de código

Clases

Una clase define el comportamiento de sus instancias especificando su estado (variables de instancia) y su comportamiento (métodos para responder a los mensajes que se le envían).

Patrón para crear una nueva clase:

```
NombrePadre subclass: #NombreClase  
    instanceVariableNames: "  
    classVariableNames: "  
    poolDictionaries: "  
    category: "
```

Todo objeto es instancia de una clase, y todo objeto conoce la instancia a la que pertenece.

Ejemplos:

```
1234 class
```

```
 #(Francesca Jackie Marisa Bree) class
```

```
'Rakesh Vijay Charles Daniel Tyler' class
```

Jerarquía de clases de Smalltalk

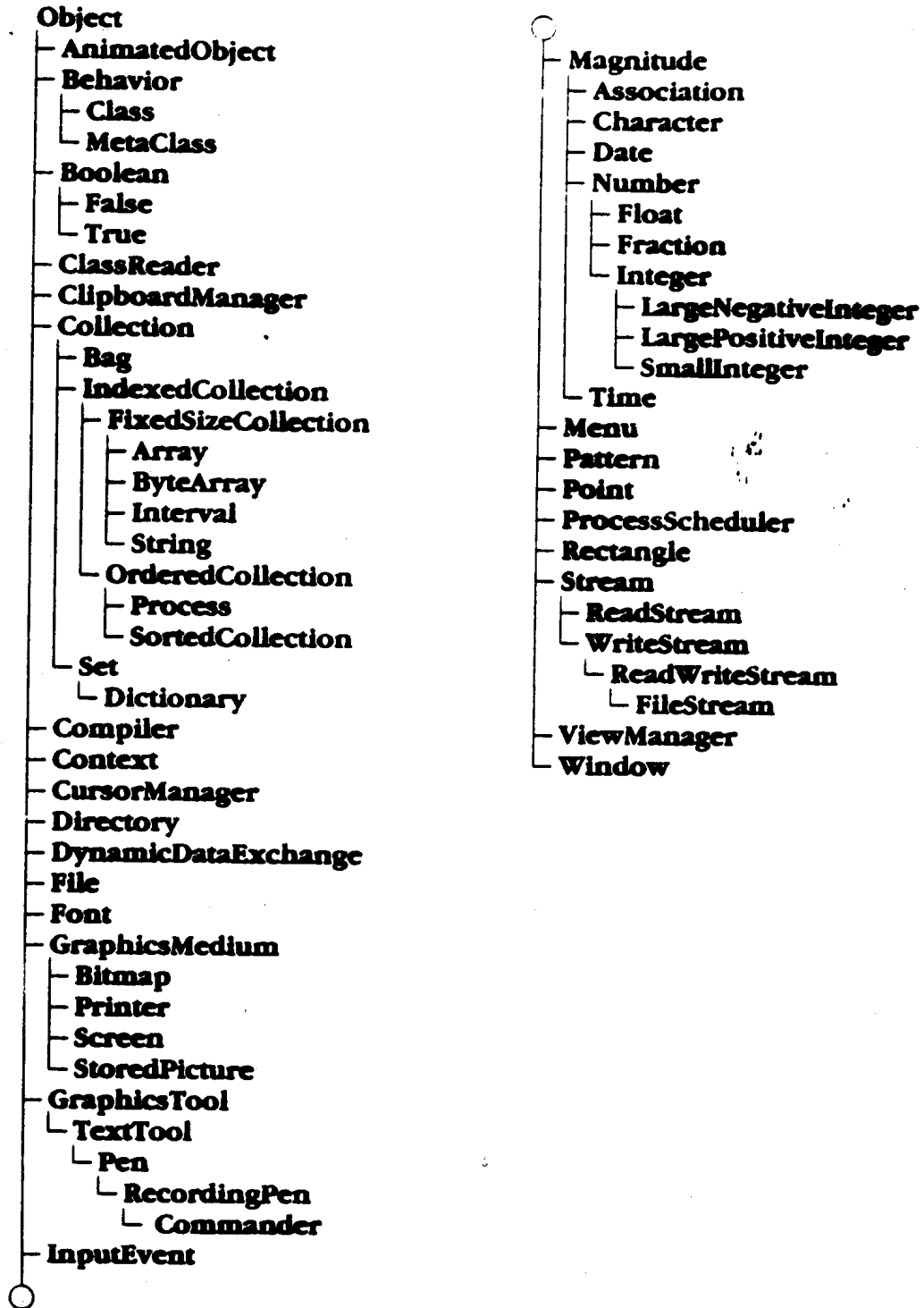


Figure 1.4
Partial Smalltalk/V
Class Hierarchy

Navegador de la jerarquía de clases

System Browser			
Kernel-Objects	Float	-- all --	*
Kernel-Classes	Fraction	arithmetic	+
Kernel-Methods	Integer	comparing	-
Kernel-Processes	LargeNegativeInteger	truncation and round o	/
Kernel-Magnitudes	LargePositiveInteger	converting	negated
Kernel-Numbers	Number	printing	reciprocal
Kernel-ST80 Remnants	Random	private	-----
Collections-Abstract		mathematical functions	
Collections-Unordered	instance	?	class
Number subclass: #Fraction instanceVariableNames: 'numerator denominator ' classVariableNames: '' poolDictionaries: '' category: 'Kernel-Numbers'			

Mensajes

Todo el procesamiento en Smalltalk se lleva a cabo mediante el envío de mensajes a objetos. El envío de mensajes es equivalente a la invocación de funciones en los lenguajes procedimentales.

Un mensaje se compone de tres partes:

- un objeto receptor,
- un selector de mensaje, y
- cero o más argumentos

Ejemplos: 'este es el momento' size

(1 3 5 7) at: 2

20 factorial

Un mensaje siempre devuelve un único objeto como resultado.

Los mensajes a los que un objeto puede responder se definen en el protocolo de su clase.

La forma en que el objeto responde a un mensaje se define en el cuerpo de los métodos.

Los métodos indican la implementación de los mensajes y representan el comportamiento de la clase.

Tipos de mensajes

Mensajes unarios.

No tienen argumentos.

Ejemplos: 5 factorial
 \$A asciiValue

Mensajes con argumentos.

Mensajes con uno o más argumentos.

Ejemplos: ‘ejemplo de cadena’ at: 6
 ‘ejemplo de cadena’ at: 1 put: \$E
 # (9 8 7 6 5) at: 3

Mensajes binarios.

Mensajes con un argumento, y uno o varios caracteres especiales (ni dígitos ni letras) como selectores.

Ejemplos: 3 + 4 suma
 5 * 7 multiplicación
 5 // 2 división entera
 2 / 6 división real

Smalltalk evalúa las expresiones aritméticas de izquierda a derecha estrictamente, sin precedencias entre operadores.

Ejemplo: 3 + 4 * 2 devuelve 14, no 11.

Es posible usar paréntesis para controlar el orden de precedencia.

Ejemplo: 3 + (4 * 2) devuelve 11.

Creación de objetos

Ya hemos creado algunos objetos.

Ejemplos:

```
'bigger', ' string'
```

```
 #(1 2 3)
```

Otra forma de crear objetos es enviando mensajes (constructores) a las clases.

Ejemplos:

```
Array new: 10
```

```
Array new
```

```
Date today
```

```
Time now
```

Referencias a objetos (variables)

En Smalltalk todas las variables son referencias (punteros) a objetos, no estando asociadas a ningún tipo o clase en particular.

Existen varias categorías de variables, dependiendo de su ámbito.

Variables globales

Son variables persistentes, cuyo valor se guarda de una ejecución para otra del entorno.

Ejemplos: Display
 Transcript
 Disk

Las variables globales comienzan con una letra mayúscula y se declaran de forma implícita.

Ejemplo: Sammy := 'Sammy Jones'

Variables de clase

Variables de clase son variables globales cuyo ámbito es una clase, por lo que son accesibles por todas las instancias de una clase.

```
Object subclass: #Factura
  instanceVariableNames: "
  classVariableNames: 'contador'
  poolDictionaries: "
```

Se usan para compartir información dentro de una clase.

Las variables de clase empiezan con una letra mayúscula.

Ejemplo: Número de instancias de una clase.

Variables de instancia

Las variables de instancia de un objeto son las definidas en su clase. El número de variables de instancia es el mismo para todas las instancias de una misma clase.

```
Object subclass: #Fraction
```

```
instanceVariableNames: 'numerator denominator'
```

```
classVariableNames: "
```

```
poolDictionaries: "
```

Variables temporales

Se denominan variables temporales porque Smalltalk las descarta cuando dejan de usarse.

Se declaran colocándolas entre barras verticales en la primera línea de una serie de expresiones. Sus nombres han de empezar por minúscula.

Ejemplo

```
| temp index factorials |
```

```
factorials := #( 3 4 5 6 ).
```

```
index := 1.
```

```
factorials size timesRepeat: [
```

```
    temp := factorials at: index.
```

```
    factorials at: index put: temp factorial.
```

```
    index := index + 1].
```

```
^factorials
```

Métodos

Los métodos son el código en Smalltalk, indican los algoritmos que determinan el comportamiento de un objeto.

Ejemplo: $(1/7)$ numerator

El método numerator está definido en la clase Fraction.

```
numerator
  ^numerator
```

Ejemplo: $(2/3) * (5/7)$

El método * está definido en la clase Fraction.

```
* aNumber
  ^(numerator * aNumber numerator)
  / (denominator * aNumber denominator)
```

self

self es una variable especial que representa el objeto que recibe el mensaje. Podemos hacer referencia a self en el cuerpo de un método

Ejemplo:

fraction

"Answer the receiver minus its integral part."

^self - self truncated

Si ahora evaluamos

(22/7) fraction devuelve 1/7

(2/3) fraction devuelve 2/3

super es una variable similar a self, que también hace referencia al objeto receptor y que está relacionada con la herencia.

Expresiones de asignación

Asignan objetos (punteros a objetos) a variables.

Ejemplos:

asignación de objetos

factorials := # (3 4 5 6).

index := 1.

asignación de resultados de mensajes (objetos)

temp := factorials at: index.

index := index + 1.

Expresiones de retorno (return)

El carácter ^ indica que lo que le sigue es el valor a ser devuelto como resultado de la serie de expresiones.

Ejemplo: ^ factorials

Expresiones compuestas

Anidamiento de mensajes

Siempre que aparezca un objeto en una expresión, podemos usar en su lugar un mensaje que devuelva un objeto del mismo tipo.

Ejemplos: 'hola' size + 4

'ahora' size + # (1 2 3 4) size

(1 12 24 36) includes: 4 factorial

4 factorial between: 3 + 4 and: 'hola' size * 7

Orden de evaluación:

- mensajes unarios,
- mensajes binarios
- mensajes con argumentos.

Es posible modificar el orden de evaluación usando paréntesis.

Ejemplos: 'hola' at: (# (5 3 1) at: 2)

'hola' at: # (5 3 1) at: 2

error: el objeto 'hola' no entiende el mensaje at:at:.

Serie de expresiones

El punto es el separador de expresiones.

Ejemplo:

```
Turtle black.  
Turtle home.  
Turtle go: 100.  
Turtle turn: 120.  
Turtle go: 100.  
Turtle turn: 120.  
Turtle go: 100.  
Turtle turn: 120
```

Los objetos tienen un estado.

Los mensajes cambian el estado de un objeto.

Mensajes en cascada

Un mensaje en cascada es una forma de escribir una serie de mensajes que son enviados al mismo receptor.

Ejemplo:

```
Turtle  black;  
        home;  
        go: 100;  
        turn: 120;  
        go: 100;  
        turn: 120;  
        go: 100;  
        turn: 120
```

Comparación de objetos

La comparación de objetos se realiza enviando mensajes.

Las comparaciones normales (<, <=, =, >=, > y se implementan como mensajes

Ejemplos: 3 < 4
 #(1 2 3 4) = #(1 2 3 4)

Todos los objetos entienden el mensaje de igualdad (=). Muchos objetos definen mensajes de ordenación (<=).

Ejemplo: 'hello' <= 'goodbye'

Existen mensajes que permiten comprobar el estado de un objeto.

Ejemplos: \$a isUpperCase
 ('hello' at: 1) isVowel
 7 odd

Expresiones Booleanas

Los mensajes and: y or: son recibidos por los objetos true o false, y como argumento tienen un bloque cuya última expresión también devuelve true o false.

Ejemplos: (c < \$0 or: [c > \$9])
 (c > \$0 and: [c <= \$9])
 (c isDigit or: [c >= \$A and: [c <= \$F]])

Expresiones condicionales

Utilizamos bloques y mensajes para ejecuciones condicionales.

Ejemplo:

```
| max a b |  
a := 5 squared.  
b := 4 factorial.  
a < b  
    ifTrue: [max := b]  
    ifFalse: [max := a].  
^max
```

Ejemplo:

```
3 < 4  
    ifTrue: ['the true block']  
    ifFalse: ['the false block']
```

Recursión

Ejemplo:

factorial

“Answer the factorial of the receiver”

self > 1

ifTrue: [^(self - 1) factorial * self]

self < 0

ifTrue: [^self error: ‘negative factorial’].

^1

fibonacci

“Answer the nth fibonacci number,
where n is the receiver”

^self < 3

ifTrue: [1]

ifFalse: [

(self - 1) fibonacci + (self - 2) fibonacci]

Iteradores

Bucles simples: timesRepeat:

Ejemplo:

```
Window turtleWindow: 'Turtle Graphics'.
```

```
Turtle
```

```
    black;
```

```
    home.
```

```
3 timesRepeat: [
```

```
    Turtle
```

```
        go: 100;
```

```
        turn: 120]
```

Ejemplo:

```
"compute the value of the first integer in a string"
```

```
| string index answer c |
```

```
string := '1234 is the number'.
```

```
answer := 0.
```

```
index := 1.
```

```
string size timesRepeat: [
```

```
    c := string at: index.
```

```
    (c < $0 or: [ c > $9 ] )
```

```
        ifTrue: [^answer].
```

```
    answer := answer * 10 + c asciiValue - $0 asciiValue.
```

```
    index := index + 1].
```

```
^answer
```

Ejemplo:

```
"Draw a polygon flower"  
| sides length |  
Window turtleWindow: 'Turtle Graphics'.  
sides := 5.  
length := 240 // sides.  
Turtle  
    black;  
    home;  
    north.  
sides timesRepeat: [  
    Turtle go: length.  
    sides - 1 timesRepeat: [  
    Turtle  
        turn: 360 // sides;  
        go: length] ]
```

Ejemplo

```
"Draw a polygon flower"  
| sides length |  
sides := 5.  
length := 240 // sides.  
Window turtleWindow: 'Turtle Graphics'.  
Turtle  
    black; up;  
    home; turn: 90;  
    down; north.  
sides timesRepeat: [  
    Turtle  
        up;  
        go: length // 2;  
        down;  
        go: length.  
sides - 1 timesRepeat: [  
    Turtle  
        turn: 360 // sides;  
        go: length ] ]
```

Ejemplo:

```
| string index c |  
string := 'Now is the time'.  
index := 1.  
string size timesRepeat: [  
    c := string at: index.  
    string  
        at: index  
        put:  
            (c isVowel  
                ifTrue: [ c asUpperCase ]  
                ifFalse: [ c asLowerCase ] ).  
    index := index + 1 ].  
^string
```

Bucles condicionales: whileTrue:, whileFalse:

Ejemplo (whileFalse):

```
"copy a disk file"  
| input output |  
input := File pathName: 'tutorial\chapter.2'.  
output := File pathName: 'tutorial\copychap.2'.  
[input atEnd]  
    whileFalse: [output nextPut: input next].  
input close.  
output close
```

Ejemplo (WhileTrue):

```
"draw several polygons"  
| sides |  
Window turtleWindow: 'Turtle Graphics'.  
sides := 3.  
[sides <= 6]  
    whileTrue: [  
        sides timesRepeat: [  
            Turtle  
                go: 60;  
                turn: 360 // sides].  
        sides := sides + 1]
```

Bucles con índice: to:do:, to:by:do:

Ejemplo (to:do:):

```
"draw several polygons"
Window turtleWindow: 'Turtle Graphics'.
3 to: 6 do: [ :sides |
    sides timesRepeat: [
        Turtle
            go: 60;
            turn: 360 // sides ] ]
```

Ejemplo (to:by:do:):

```
"compute the sum of 1/2, 5/8, 3/4, 7/8, 1"
| sum |
sum := 0.
1/2 to: 1 by: 1/8 do: [ :i |
    sum := sum + i ].
^sum
```

Otros iteradores: do:

Ejemplo (do: sobre strings):

```
"count vowels in a string"  
| vowels |  
vowels := 0.  
'Now is the time' do: [ :char |  
    char isVowel  
        ifTrue: [ vowels := vowels + 1 ] ].  
^vowels
```

Ejemplo (do: sobre arrays):

```
"draw several polygons"  
Window turtleWindow: 'Turtle Graphics'.  
#( 3 4 12 24 ) do: [ :sides |  
    sides timesRepeat: [  
        Turtle  
            go: 20;  
            turn: 360 // sides ] ]
```

Ejemplo (do: sobre ficheros):

```
"Strip all line feed characters (ascii 10)
from a disk file. Answer the number of characters
stripped."
| input output stripped |
stripped := 0.
input := File pathName: 'tutorial\striplf.pc'.
output := File pathName: 'tutorial\striplf.mac'.
input do: [ :char |
    char = 10 asCharacter
        ifTrue: [ stripped := stripped + 1 ]
        ifFalse: [ output nextPut: char ] ].
input close.
output close.
^stripped
```

Otros iteradores: *select:*, *reject:*. *collect:*

Ejemplo (*select:*):

"count the vowels in a string"

('Now is the time' *select:* [:c | c isVowel])

size

El mensaje *select:* itera sobre el receptor y devuelve todos los elementos para los que el bloque se evalúa a true.

Ejemplo (*reject:*):

"answer all digits whose factorial is

less than the digit raised to the 4th power"

#(1 2 3 4 5 6 7 8 9) *reject:* [:i |

i factorial >= (i * i * i * i)]

Ejemplo (*collect:*):

"square each element in the array"

#(1 13 7 10) *collect:* [:i | i * i]

Implementación de los iteradores

BlockContext

whileTrue: aBlock

" Evaluate the argument, aBlock, as long as the value of the receiver is true."

```
self value ifTrue: [ aBlock value.
```

```
^self whileTrue: aBlock ]
```

Number

timesRepeat: aBlock

"Evaluate the argument, aBlock, the number of times represented by the receiver."

```
| count |
```

```
count := 1.
```

```
[count <= self]
```

```
whileTrue:
```

```
    [aBlock value.
```

```
    count _ count + 1]
```

to: stop do: aBlock

" Evaluate aBlock for each element of the interval (self to: stop by: 1)."

```
| nextValue |
```

```
nextValue := self.
```

```
[nextValue <= stop]
```

```
whileTrue:
```

```
    [aBlock value: nextValue.
```

```
    nextValue _ nextValue + 1]
```

Collection

do: aBlock

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

select: aBlock

```
| aStream |  
aStream := WriteStream on: (self species new: self size).  
1 to: self size do:  
    [:index |  
        (aBlock value: (self at: index))  
        ifTrue: [aStream nextPut: (self at: index)]]].  
^ aStream contents
```

reject: aBlock

"Evaluate aBlock with each of the receiver's elements as the argument.
Collect into a new collection like the receiver only those elements for
which aBlock evaluates to false. Answer the new collection."

```
^self select: [:element | (aBlock value: element) == false]
```

collect: aBlock

```
| result |  
result := self species new: self size.  
1 to: self size do:  
    [:index | result at: index put: (aBlock value: (self at: index))].  
^ result
```

Herencia

Una clase tiene una superclase 'inmediata' y posiblemente una o más subclases, con la clase **Object** en la cima de la jerarquía.

Las clases situadas más arriba en la jerarquía representan características más generales, mientras que clases más abajo en la jerarquía representan características más específicas.

Un objeto hereda todas las variables de instancia definidas en sus superclases más las contenidas en su propia clase.

Los métodos también se heredan. Cuando se envía un mensaje a un objeto, Smalltalk busca el método correspondiente en la clase del objeto. Si lo encuentra, lo ejecuta. En caso contrario, repite el procedimiento en la superclase del objeto. Este proceso continua hasta llegar a la clase Object. Si no se encuentra el método se produce un error.

Índice

Objetos.....	2
Clases.....	3
Jerarquía de clases de Smalltalk	4
Navegador de la jerarquía de clases.....	5
Mensajes.....	6
Tipos de mensajes.....	7
Mensajes unarios.....	7
Mensajes con argumentos.....	7
Mensajes binarios.....	7
Creación de objetos	8
Referencias a objetos (variables).....	9
Variables globales.....	9
Variables de clase.....	9
Variables de instancia	10
Variables temporales.....	10
Métodos	11
self	12
Expresiones de asignación.....	13
Expresiones de retorno (return).....	13
Expresiones compuestas	14
Anidamiento de mensajes.....	14

Serie de expresiones	15
Mensajes en cascada	15
Comparación de objetos.....	16
Expresiones Booleanas.....	16
Expresiones condicionales.....	17
Recursión.....	18
Iteradores	19
Bucles simples: <code>timesRepeat:</code>	19
Bucles condicionales: <code>whileTrue:</code> , <code>whileFalse:</code>	23
Bucles con índice: <code>to:do:</code> , <code>to:by:do:</code>	24
Otros iteradores: <code>do:</code>	25
Otros iteradores: <code>select:</code> , <code>reject:</code> , <code>collect:</code>	27
Implementación de los iteradores	28
Herencia	30
Índice.....	31