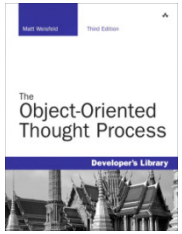
Introducción a Smalltalk

Alicia Díaz

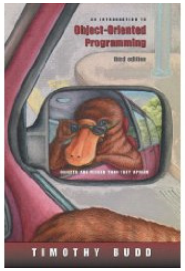
alicia.diaz@lifia.info.unlp.edu.ar

Bibliografía de POO

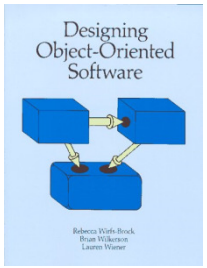
Sobre la teroría de la POO:



The Object-Oriented Thought Process, Matt Weisfeld, Third Edition, Pearson Education, Addison Wesley. ISBN-13: 978-0-672-33016-2



Introduction to Object-Oriented Programming, An (3rd Edition), Thimoty Budd,Addison Wesley; 3 edition (2001), ISBN-10: 0201760312

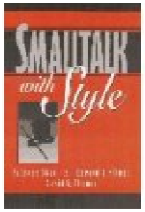


Designing Object-Oriented Software. Rebecca Wirfs-Brock, Brian Wilkerson (Contributor), Lauren Wiener . Prentice Hall PTR; (January 1991), ISBN 0136298257

Disponibles en la Biblioteca de la Facultad de Informática

Bibliografía de SmallTalk:

Pharo by Example
http://pharobyexample.org/
http://pharobyexample.org/es/PBE1-sp.pdf



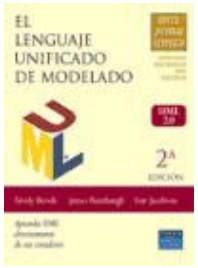
Smalltalk With Style, Suzanne Skublics, Edward J. Klimas, David A. Thomas, John Pugh (Foreword)
Pearson Education POD; 1 edition (May 21, 2002) ISBN: 0131655493

Smalltalk Best Practice Patterns , Kent Beck
Prentice Hall PTR; 1st edition (October 3, 1996) ISBN: 013476904X

Bajarlos de: http://www.iam.unibe.ch/~ducasse/FreeBooks.html

Bibliografía de UML:

UML, no es un lenguaje de programación ,es una notación…



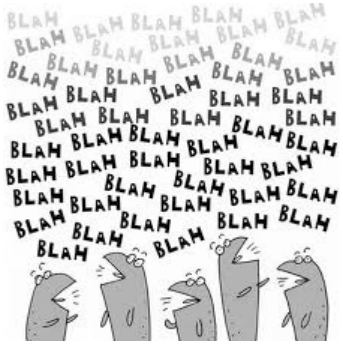
EL LENGUAJE UNIFICADO DE MODELADO. BOOCH GRADY, JACOBSON IVAR , RUMBAUGH JAMES, ADDISON-WESLEY IBEROA, Edición 2000, ISBN 8478290281

UML GOTA A GOTA. FOWLER MARTIN, SCOTT KENDALL, ADDISON-WESLEY IBEROA. Edición 1999 ISBN 9684443641

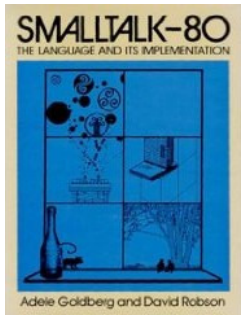
Disponibles en la Biblioteca de la Facultad de Informática

Algunas palabras sobre ST

- ¿Qué es Smalltalk?
  - Un lenguaje de programación OO puro
  - Una libreria de clases completa
  - Un ambiente de programación interactivo (Pharo)
    - El ambiente en sí mismo esta desarrollado en Smalltalk
    - Tiene su propio
      - Compilador
      - Debugger
      - Browser de la librería de clases
    - Es extensible



Algo de historia



Smalltalk-80 fue desarrollado en *Xerox Palo Alto Research Center* entre fines de los 70s y principio de los 80s (Alan Kay, Dan Ingalls, Adele Goldberg)

En 1997 se presenta Squeak como un dialecto de Smalltalk derivado directamente de Smalltalk-80 (Dan Ingalls, Alan Kay).
http://www.squeak.org/

En 2009 Pharo surge a partir de Squeak como una iniciativa open-source. Se centra en las técnicas de ingeniería de software y desarrollo modernos. (Pharo Board- Pharo Community)
http://pharo.org/

¿porqué SmallTalk?

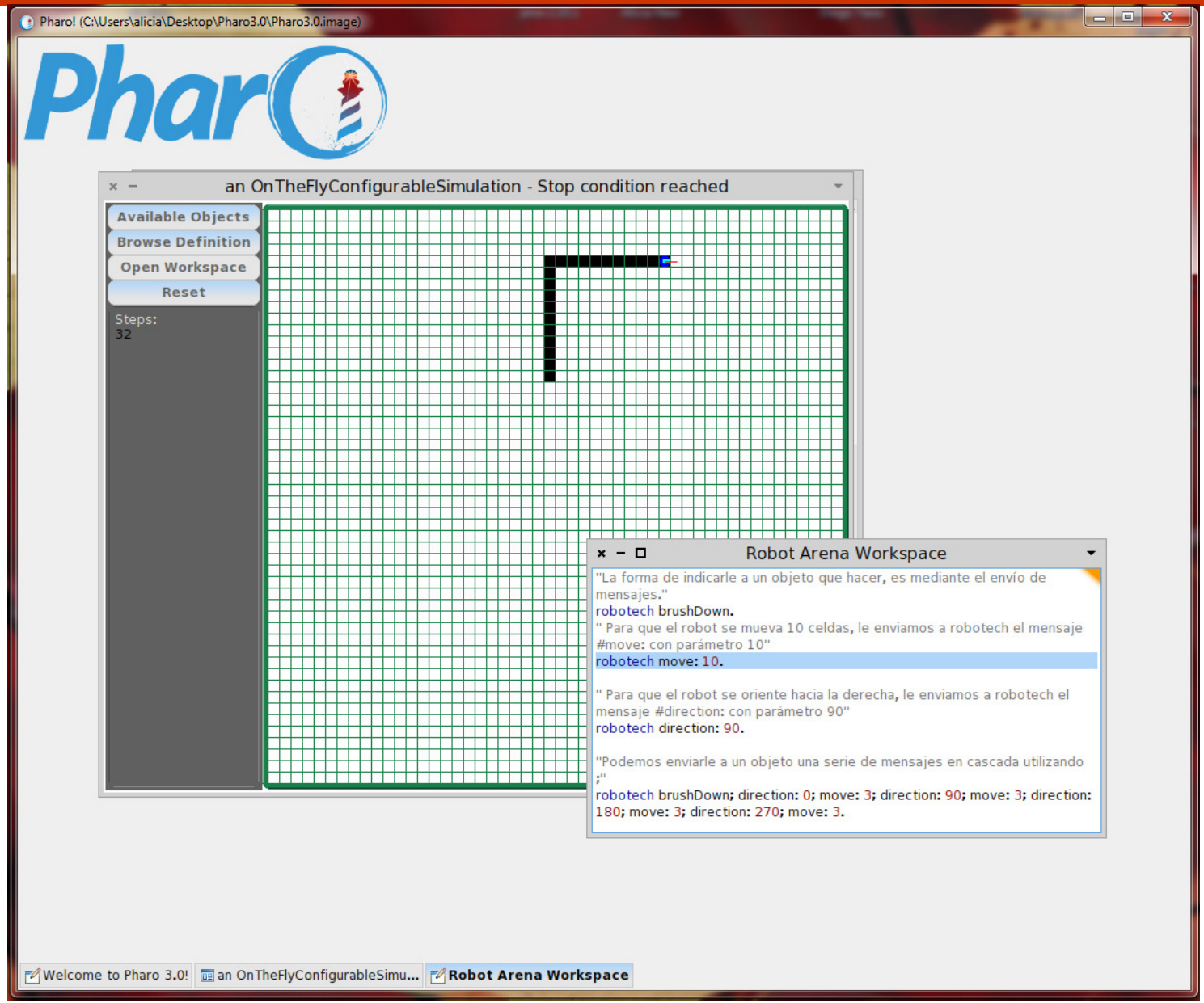
- Hay muchas razones que iremos aprendiendo durante el curso

- el lenguaje es muy simples y los conceptos subyacentes son simples y uniformes en todo el lenguaje
- Es uno de los pocos leguajes OO puros
- No hay separación entre el lenguaje y el ambiente de programación.
    - El ambiente en sí es un universo viviente de objetos
- El código escrito por programadores expertos es muy compacto y legible
- El código fuente es parte del ambiente, está disponible para su estudio, extensión y modificación
- El ambiente de desarrollo es muy potente
- El lenguaje y el ambiente es reflexivo: es posible cambiar los programas en run-time
- La compilación es incremental, favoreciendo el desarrollo y testing
- Es posible interrumpir la ejecución e inspeccionar el estado del programa e incluso modificar código y objetos
- Es fuertemente tipado, un objeto no puede responder a un mensaje que no se le programó
- Es dinamicamente tipado: cuando un programa tiene que evaluar su definición, este lo resuleve en run-time y no en la compilación
- es portable, permite que aplicaciones bien implementadas corran en más de 10 plataformas distintas sin necesidad de hacer cambios
- Es muy maduro, se está desarrollando desde los 70s.

Sintaxis Básica de Smalltalk

Objetos, Mensajes, Variables

El objeto "robotech" y sus mensajes

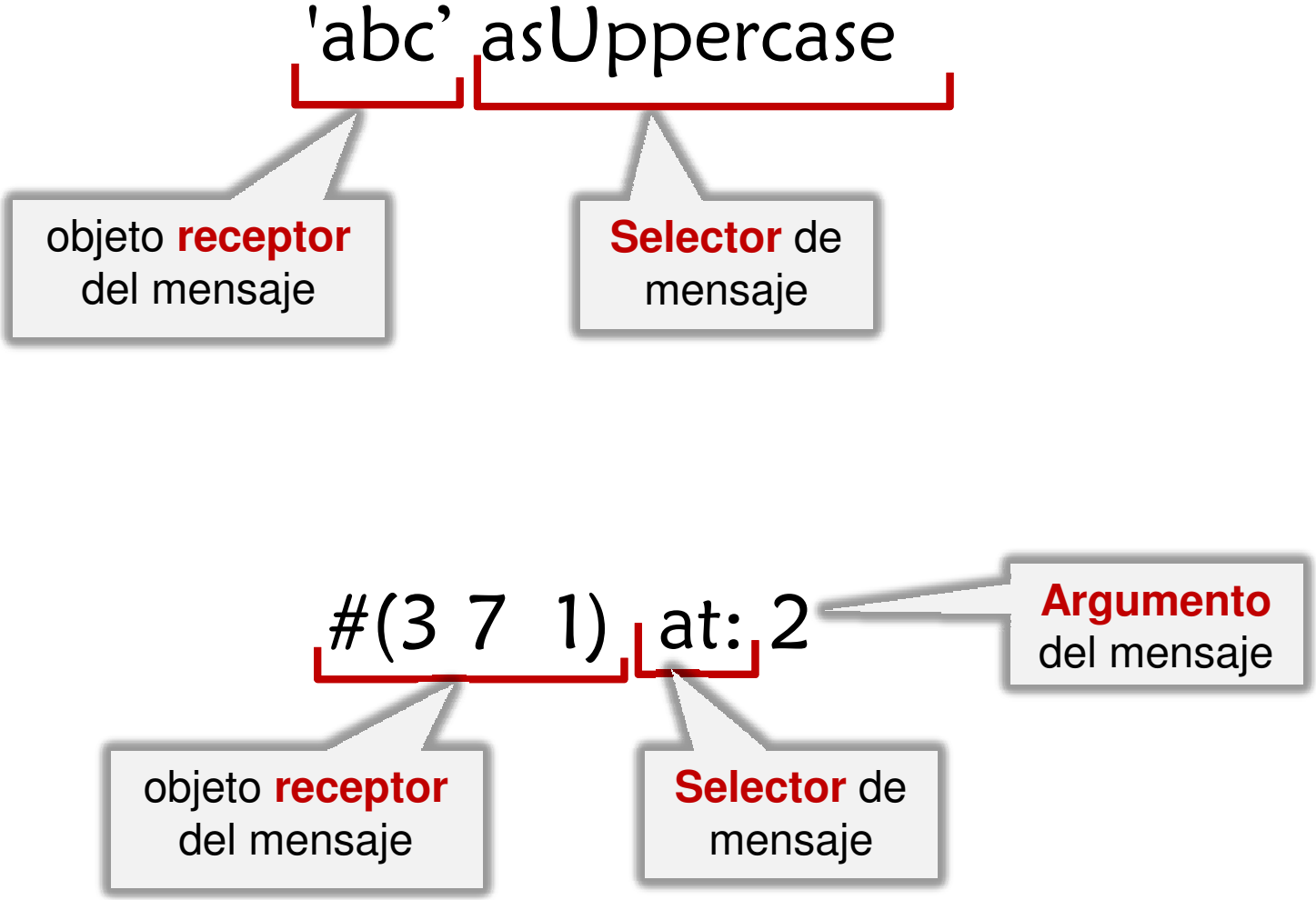


Otros objetos

- Literales
  - numbers: 3 -5 0.56 1.3e5 16rA
  - characters: $A $1 $$
  - strings: 'hello' 'A' 'haven''t'
  - symbols: #Fred #dog
  - arrays: #(3 7 1)
    #(3 $A 'hello' #Fred #(4 'world'))
  - punto 3@5
  - bloques: lo veremos más tarde

Expresión Smalltalk

- Sintaxis básica



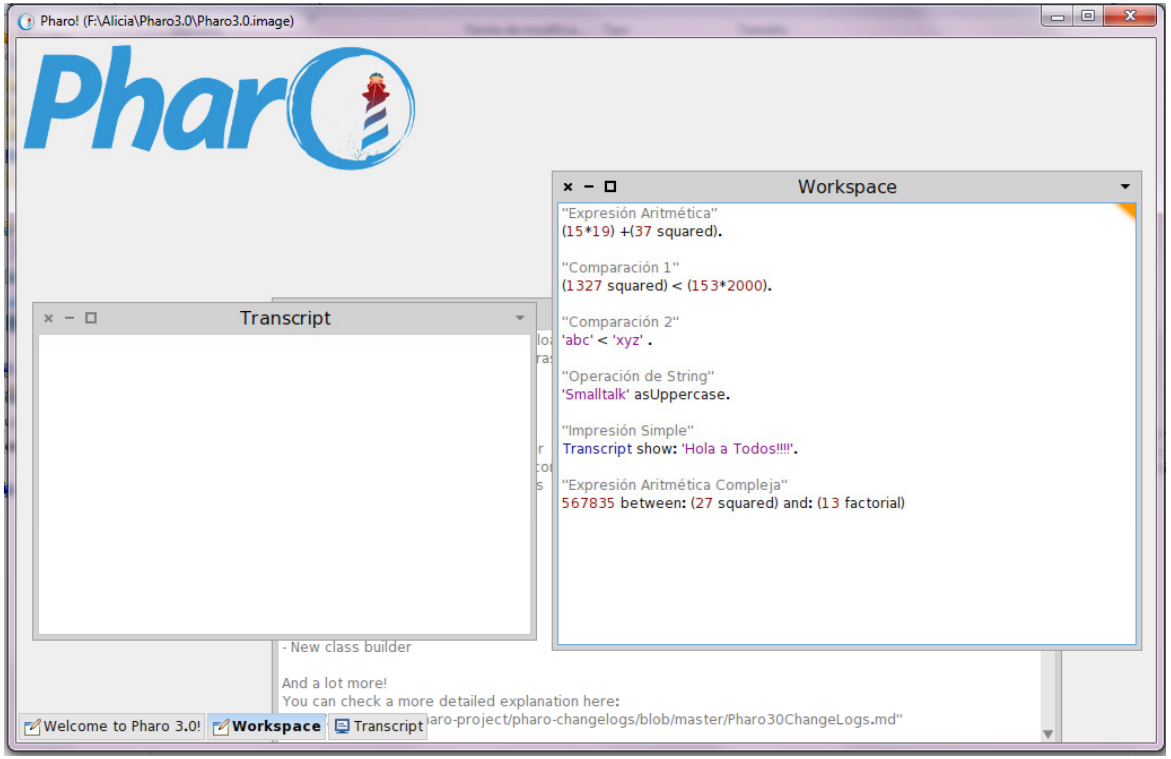
Otros ejemplos de expresiones Smalltalk

"Expresión Aritmética"
(15*19) +(37 **squared**)

"Comparación 1"
(1327 **squared**) < (153*2000)

"Comparación 2"
'abc' < 'xyz'

"Operación de String"
'Smalltalk' **asUppercase**



"Impresión Simple"
Transcript **show:** 'Hola a Todos!!!!'

"Expresión Aritmética Compleja"
567835 **between:** (27 **squared**) **and:** (13 **factorial**)

ST tiene 3 tipos de mensajes

- Unarios, Binarios y de Palabra Clave

- Difieren en:
  - La estructura del nombre del mensaje (su selector), y
  - La cantidad de argumentos que el mensaje espera

Mensajes unarios

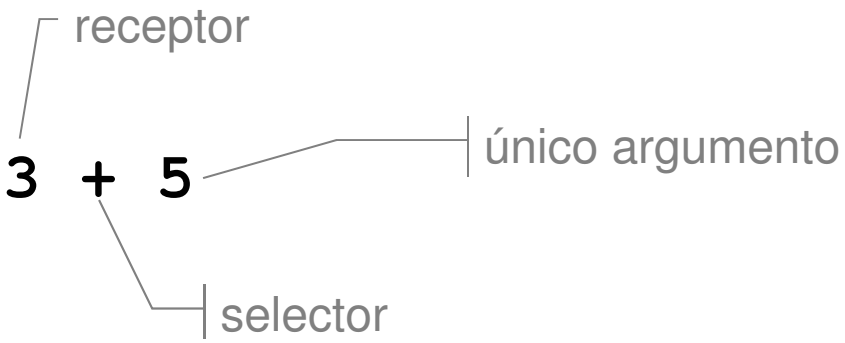
- No tienen argumentos


```
'ABC' asLowercase  →  'abc'
```

```
#( 3 4 5 6) size  →  4
```

Mensajes Binarios

- Usan uno o dos caracteres especiales como selector (+, *, @, >=, =, “,” , ...) y tienen un solo argumento



8

‘hi’ , ‘All’

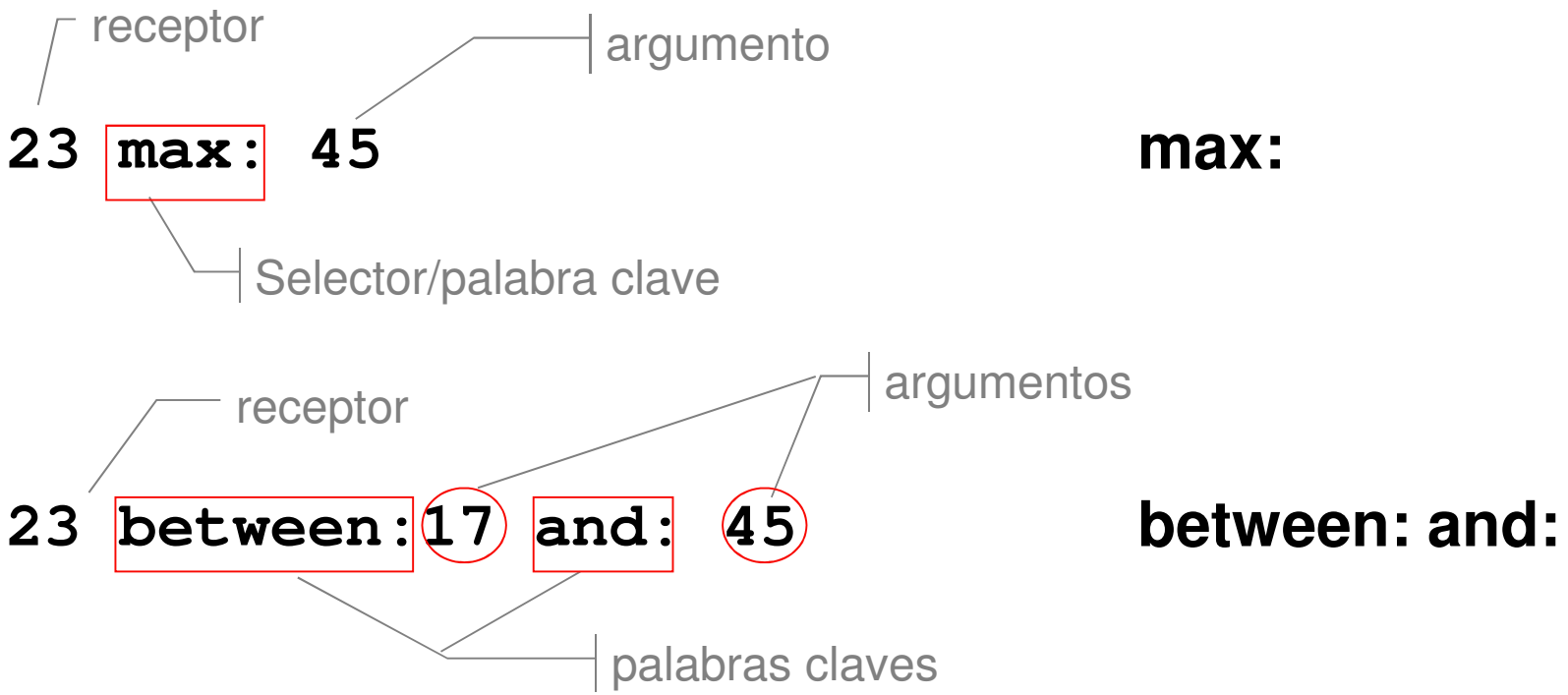
‘hiAll'

23 <= 25

true

Mensajes de Palabra Clave

- Tienen uno o más argumentos
- Cada argumento es precedido por una palabra clave
- El selector tiene tantas palabras clave como argumentos
- Cada palabra es seguida por ' : '



Mensajes de Palabra Clave

- Otro ejemplo:

```
#('blue' 'green' 'yellow') at: 2
```

```
'green'
```

```
#('blue' 'yellow' 'green') at: 3 put: 'red'
```

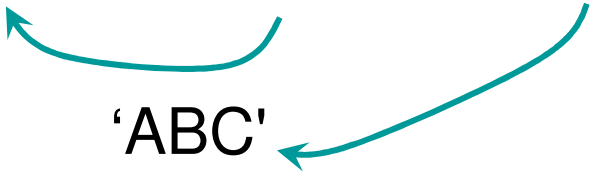
```
#('blue' 'yellow' 'red')
```

Cadenas Mensajes

- Cada mensaje devuelve un objeto

…. entonces podemos escribir cadenas de mensajes

```
'abc' asUppercase reverse
```



'ABC'

'CBA'

No siempre es tan simple…..

```
1 + 3 factorial
```

Veamos que devuelve…

```
#(1 3 5) at:2 + 1
```

y acá que pasará? ..veamos…

Objetos y Mensajes

- Reglas de precedencia
  - los mensajes se ejecutan de izquierda a derecha
  - primero, las expresiones parentizadas
  - luego, los mensajes unarios,
  - luego, los binario
  - y por último los de palabra clave

- **Única regla a tener en cuenta para poder leer y escribir programas Smalltalk**

```
5 factorial between: 3 squared and: 3 * 5 + 9
```

```
(5 factorial) between: (3 squared) and: (3 * 5 + 9)
```

Los paréntesis permiten alterar la precedencia: `(1+3) factorial`

Sentencias

- Cada expresión ST representa una sentencia
- Una secuencia de expresiones representa un fragmento de código ST y sus componentes son sentencias
- Las sentencia se separan con ‘ . ’

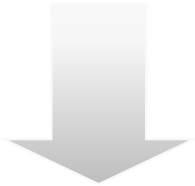
```
3 factorial.
2 squared

Transcript clear.
Transcript show: 'Hello!'
```

Shortcut o mensajes en cascadas

- Todos los mensajes van al mismo receptor y se separan con ' ; '

```
Transcript clear.
Transcript show: 'Hello!'.
Transcript cr.
Transcript show: 'How are you?'
```



```
Transcript clear;
        show: 'Hello!';
        cr;
        show: 'How are you?'
```

Objetos y Clases

Objetos y Clases

- Todo objeto es una instancia de una clase

Una clase permite que todos los objetos del mismo tipo compartan la misma definición

Por ejemplo, **3/5** y **4/7** son instancias de la clase **Fraction**

La clase define las propiedades y funcionalidades (comportamiento) de sus instancias

Estado interno: numerator, denominator
Comportamiento: numerator , denominator, negative, isZero

Veamos la clase Fraction y Point ...

Métodos

Métodos

- ¿Qué es un método?
  - Es la contraparte funcional del mensaje.
  - Expresa la forma de llevar a cabo la semántica propia de un mensaje particular (el *cómo*).
- son identificados por su clase y selector de mensaje
- El código de un método puede realizar básicamente 3 cosas:
  - Modificar el estado interno del objeto.
  - Colaborar con otros objetos (enviándoles mensajes).
  - Retornar y terminar.

Anatomía de un método

- Tiene:
  - La firma: selector del mensaje con un nombre de variable por cada palabra clave en el selector
  - Un comentario
  - Una declaración de variables temporales (entre |…|)
  - Una secuencia de sentencias
  - Si el método devuelve un objeto, se usa el ^ como carácter de retorno seguido del objeto.

Anatomía de un método

- El método **#nearesPointOnLineFrom: to:** de la clase **Point**.

```
Point>>nearestPointOnLineFrom: point1 to: point2
        "Answer the closest point to the receiver on the line determined by (point1, point2)."
        "43@55 nearestPointOnLineFrom: 10@10 to: 100@200"

        | dX dY newX newY dX2 dY2 |

        dX := point1 x - point2 x.
        dY := point1 y - point2 y.
        dX = 0   ifTrue: [dY = 0  ifTrue: [^point1]
                                  ifFalse: [^point1 x @ y]]
                 ifFalse: [dY = 0 ifTrue: [^x @ point1 y]].
        dX2 := dX * dX.
        dY2 := dY * dY.
        newX := x * dX2 + (point2 x *dY2) + (y - point2 y * dX * dY) / (dX2 + dY2).
        newY := x - newX * dX / dY + y.
        ^newX @ newY
```

Secuencia de Sentencias

2 tipos de Métodos en ST

- Métodos de instancias
  - Son los que se pueden enviar a las instancias de una clase
    - `'abc' asUppercase`
    - `3/5 numerator`
    - `miCuentaBancaria saldo`
- Métodos de clase
  - Son los que se pueden enviar a las clases
    - `CuentaBancaria new`
    - `Fraction numerator:3 denominator:5`
    - `Date today`

Variables

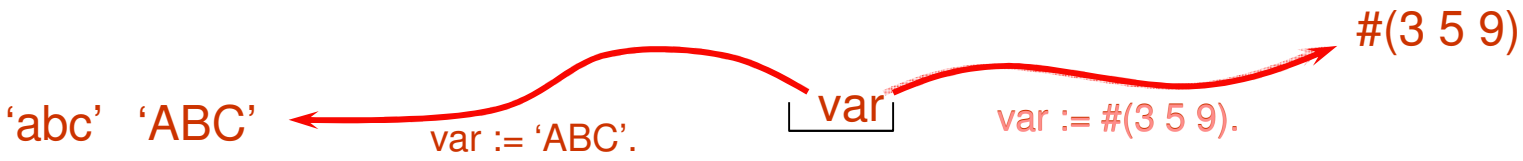
Variables

- Una variable se liga a un objeto a través del operador de asignacion :=

```
var := #(3 5 9).
var size
```

- no necesitan ser tipadas porque son punteros a objetos
- Las variables no son objetos
- Pueden ser ligadas a distintos objetos

```
var := 'ABC'.
var asLowercase
```



Variables

- El nombre de una variable es una secuencia de caracteres y dígitos comenzando con una letra:
  - price
  - taxRate07
  - empleadosContratados

- Se acostumbra usar la notación *Camel* para construir el nombre de la variable
  - consiste en escribir los identificadores con la primera letra de cada palabra en mayúsculas y el resto en minúscula:
    - endOfFile.

Categoría de variables

- Variables temporales
- Variables de instancia
- Pseudo-variables: self, super, nil, true, false
- Variables de clases o compartidas

... veamos cada categoría

Variables Temporales

- Sirven para guarda el resultado de una ejecución que será utilizado más tarde
- deben ser declaradas antes que aparezca la sentencia que las usa
  - |variableName| o |variableName1 variableName2 variableName3|

- Se usan en el workspace o como variables auxiliares en un método
- Su alcance esta limitado al método o fragmento de código que las contenga
- Comienzan con minúscula

  - En el workspace

```
| today myBirthday daysToMyBirthday |

today:= Date today.
myBirthday :=  Date year: 2014 month: 12 day:13.
daysToMyBirthday := myBirthday - today.
^ daysToMyBirthday asDays
```

Veamos como funcionan …

Variables Temporales

- En un método

```
Point>>nearestPointOnLineFrom: point1 to: point2
        "Answer the closest point to the receiver on the line determined by (point1, point2)."
        "43@55 nearestPointOnLineFrom: 10@10 to: 100@200"

        | dX dY newX newY dX2 dY2 |

        dX := point1 x - point2 x.
        dY := point1 y - point2 y.
        dX = 0   ifTrue: [dY = 0  ifTrue: [^point1]
                                  ifFalse: [^point1 x @ y]]
                 ifFalse: [dY = 0 ifTrue: [^x @ point1 y]].
        dX2 := dX * dX.
        dY2 := dY * dY.
        newX := x * dX2 + (point2 x *dY2) + (y - point2 y * dX * dY) / (dX2 + dY2).
        newY := x - newX * dX / dY + y.
        ^newX @ newY
```

Variables de instancias

- Sirven para describir las propiedades o estado interno de un objeto
- Son declaradas cuando se define una clase y las poseen cada objeto de esa clase
- Son creadas cuando se crea una instancia de una clase
- Sus valores pueden ser modificados durante la vida útil del objeto
  - Por manipulación directa *en un método de la clase a la que pertenece el objeto que las posee*
  - a través de métodos setters
- Su alcance esta limitado a los *método de instancia* de los objetos que las poseen.

Variables de instancias

- Son declaradas cuando se define una clase y las poseen cada objeto de esa clase

```
Object subclass: #Point
        instanceVariableNames: 'x y'
        classVariableNames: ''
        category: 'Kernel-BasicObjects`
```

```
Number subclass: #Fraction
        instanceVariableNames: 'numerator denominator'
        classVariableNames: ''
        category: 'Kernel-Numbers`
```

```
OTFRobot subclass: #WalkingBrushRobot
        instanceVariableNames: 'battery state'
        classVariableNames: ''
        category: 'BotArena-Robots'
```

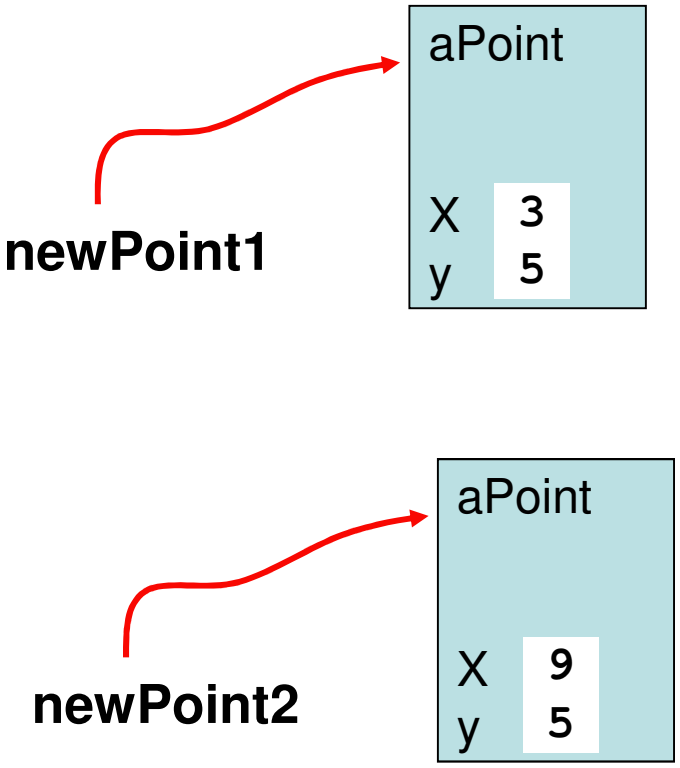
Variables de instancias

- Son creadas cuando se crea una instancia de una clase

```
| newPoint1 newPoint2|
newPoint1 := Point new.
newPoint2 := Point new.

newPoint1 x: 3.
newPoint1 y: 5.

newPoint2 x: 9.
newPoint2 y: 5.
```



Variables de instancias

- Sus valores pueden ser modificados durante la vida útil del objeto

  - Por manipulación directa *en un método de la clase a la que pertenece el objeto que las posee*

```
Point>>x: xInteger
        "Set the x coordinate."
        x := xInteger
```

Variable de instancia definida en Point

  - a través de métodos setters

```
| newPoint|
        newPoint := Point new.
        newPoint x: 3.
        newPoint y: 5.
```

Setters de Point

Pseudo Variables

- Variables que son asignadas a algún objeto por el compilador y no pueden ser modificadas a través del :=
- Su binding existe solamente durante la ejecución del método, sin embargo el objeto puede continuar existiendo
- El alcance de la variable es el método que las contiene
- Hay 3 tipos:
  - Argumento de los mensajes

```
CuentaBancaria>> extraer: unMonto
```

  - **nil, true, false**

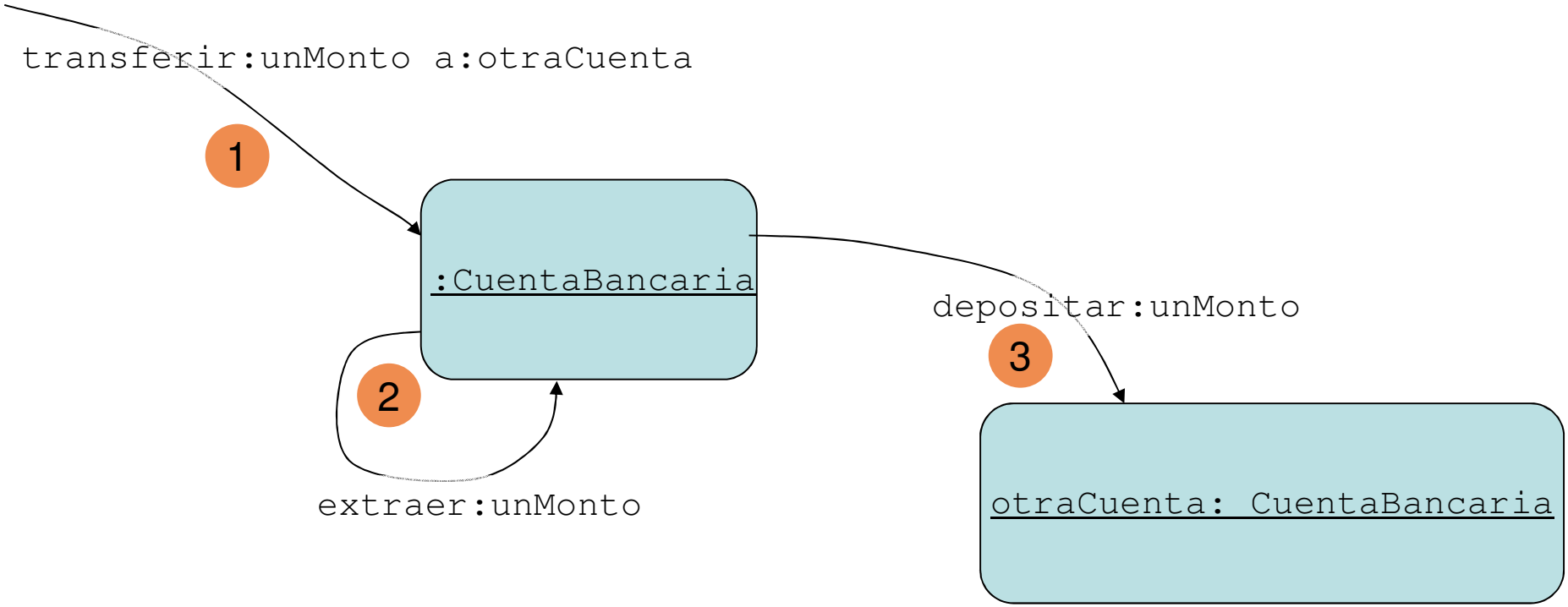
```
.....
^false
....
```

  - **self, super**

Pseudo Variables: **self**

- Una cuenta bancaria sabe “transferir un monto desde ella a otra cuenta bancaria”

CuentaBancaria>>transferir:unMonto a:otraCuenta



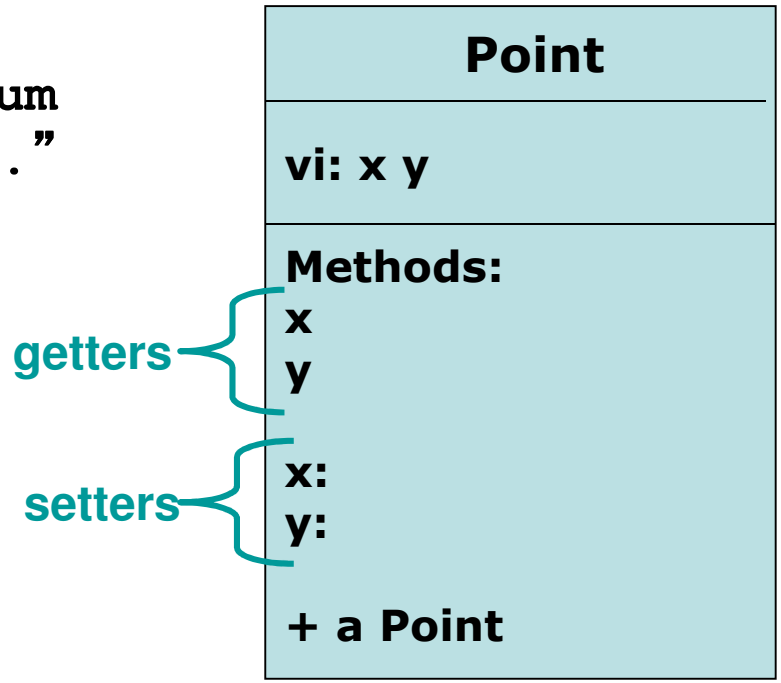

```
CuentaBancaria>>transferir:unMonto a:otraCuenta
                    self extaer:unMonto.
                    otraCuenta depositar:unMonto.
```

Otro ejemplo

- Método + de la clase Point
  - 3@5 + (4@9)

```
+ aPoint
"Answer a new point that is the sum
 of the receiver and the argument."

|newPoint|
newPoint := Point new.
newPoint x: self x + aPoint x.
newPoint y: self y + aPoint y.
^ newPoint
```



condicionales e iteradores

- Algunos Condicionales

```
aBoolean ifTrue:[ sentencias… ]

    a>b iftrue:[^"a es mayor que b"]


aBoolean   ifTrue:[ sentenciasTrue… ]
           ifFalse:[ sentenciasFalse… ]

    a>b    ifTrue:[^"a es mayor que b"]
           ifFalse:[^"b es mayor o igual que a"]
```

- Otros

```
ifFalse: ; ifFalse: ifTrue:
```

condicionales e iteradores

- Algunos Iteradores

- **timesRepeat:**

```
anInteger timesRepeat:[ sentencias… ]
        |sum|
        sum:=0.
        4 timesRepeat:[sum:= sum+10]
```

- **to:do:**

```
anInteger to:otherInteger do:[:index| sentencias(index)]
        |sum|
        sum:=0.
        1 to:10 do:[:i|sum:= sum+i]
```

- **to:do:by:**

¿En qué clase están definidos estos mensajes? …