

## Práctica N° 9 - Programación Orientada a Objetos

Para resolver esta práctica, recomendamos usar el entorno *Pharo*, que puede bajarse del sitio web indicado en la sección *Enlaces* de la página de la materia.

Los ejercicios marcados con el símbolo ★ constituyen un subconjunto mínimo de ejercitación. Sin embargo, aconsejamos fuertemente hacer todos los ejercicios.

### Objetos - Mensajes

#### Ejercicio 1 ★

En las siguientes expresiones, identificar mensajes. Indicar el objeto receptor y los colaboradores en cada caso.

- |                                |  |
|--------------------------------|--|
| a) 10 numberOfDigitsInBase: 2. | g) 1@1 insideTriangle: 0@0 with: 2@0 with: 0@2.  |
| b) 10 factorial.               | h) 'Hello World' indexOf: \$o startingAt: 6.   |
| c) 20 + 3 * 5.                 | i) (OrderedCollection with: 1) add: 25; add: 35; yourself.   |
| d) 20 + (3 * 5).               | j) Object subclass: #SnakesAndLadders<br>instanceVariableNames: 'players squares turn die over'<br>classVariableNames: ''<br>poolDictionaries: ''<br>category: 'SnakesAndLadders'. |
| e) December first, 1985.       |  |
| f) 1 = 2 ifTrue: [ 'what!?' ]. |  |

#### Ejercicio 2

Para cada una de las expresiones del punto anterior, indicar cuál es el resultado de su evaluación. Para este punto se recomienda utilizar el Workspace de *Pharo* para corroborar las respuestas.

#### Ejercicio 3

Dar ejemplos de expresiones válidas en el lenguaje *Smalltalk* que contengan los siguientes conceptos entre sus sub-expresiones. En cada caso indicar por qué se adapta a la categoría y describir que devuelve su evaluación.

- |                    |                   |             |
|--------------------|-------------------|-------------|
| a) Objeto          | e) Colaborador    | i) Carácter |
| b) Mensaje unario  | f) Variable local | j) Array    |
| c) Mensaje binario | g) Asignación     |             |
| d) Mensaje keyword | h) Símbolo        |             |

### Bloques - Métodos - Colecciones

#### Ejercicio 4 ★

Para cada una de las siguientes expresiones, indicar qué valor devuelve o explicar por qué se produce un error al ejecutarlas. Para este ejercicio recomendamos pensar qué resultado debería obtenerse y luego corroborarlo en el Workspace de *Pharo*.

- a) [:x | x + 1] value: 2
- b) [|x| x := 10. x + 12] value
- c) [:x :y | |z| z := x + y] value: 1 value: 2
- d) [:x :y | x + 1] value: 1
- e) [:x | [:y | x + 1]] value: 2
- f) [|:x | x + 1] value
- g) [:x :y :z | x + y + z] valueWithArguments: #(1 2 3)
- h) [|z| z := 10. [:x | x + z]] value value: 10

¿Cuál es la diferencia entre [|x y z| x + 1] y [:x :y :z| x + 1]?

## Ejercicio 5

Dada la siguiente implementación:

```
Integer << factorialsList
| list |
list := OrderedCollection with: 1.
2 to: self do: [ :aNumber | list add: (list last) * aNumber ].
^list.
```

Donde `UnaClase << unMetodo` indica que se estará definiendo el método `#unMetodo` en la clase `UnaClase`.

¿Cuál es el resultado de evaluar las siguientes expresiones? ¿Quién es el receptor del mensaje `#factorialsList` en cada caso?

- a) `factorialsList: 10.`
- b) `Integer factorialsList: 10.`
- c) `3 factorialsList.`
- d) `5 factorialsList at: 4.`
- e) `5 factorialsList at: 6.`

## Ejercicio 6 ★

Mostrar un ejemplo por cada uno de los siguientes mensajes que pueden enviarse a las colecciones en el lenguaje Smalltalk. Indicar a qué evalúan dichos ejemplos.

- a) `#collect:`
- b) `#select:`
- c) `#inject: into:`
- d) `#reduce: (o #fold:)`
- e) `#reduceRight:`
- f) `#do:`

## Ejercicio 7 ★

Suponiendo que tenemos un objeto `obj` que tiene el siguiente método definido en su clase

```
SomeClass << foo: x
| aBlock y z |
z := 10.
aBlock := [x > 5 ifTrue: [z := z + x. ^0] ifFalse: [z := z - x. 5]].
y := aBlock value.
y := y + z.
^y.
```

¿Cuál es el resultado de evaluar las siguientes expresiones?

- a) `obj foo: 4.`
- b) `Message selector: #foo: argument: 5.`
- c) `obj foo: 10.` (Ayuda: el resultado no es 20).

## Ejercicio 8 ★

Implementar métodos para los siguientes mensajes:

- a) `#curry`, cuyo objeto receptor es un bloque de dos parámetros, y su resultado es un bloque similar al original pero “curricado”.

Por ejemplo, la siguiente ejecución evalúa a 12.

```
| curried new |
curried := [ :x :res | x + res ] curry.
new := curried value: 10.
new value: 2.
```

- b) `#flip`, que al enviarse a un bloque de dos parámetros, devuelve un bloque similar al original, pero con los parámetros en el orden inverso.

- c) `#timesRepeat:`, cuyo objeto receptor es un número natural y recibe como colaborador un bloque, el cual se evaluará tantas veces como el número lo indique.

Por ejemplo, luego de la siguiente ejecución, `count` vale 20 y `copy` 18.

```
| count copy |  
count := 0.  
10 timesRepeat: [copy := count. count := count + 2].
```

### Ejercicio 9 ★

Agregar a la clase `BlockClosure` el método de clase `generarBloqueInfinito` que devuelve un bloque `b1` tal que:

```
b1 value devuelve un arreglo de 2 elementos #(1 b2)  
b2 value devuelve un arreglo de 2 elementos #(2 b3)  
:  
bi value devuelve un arreglo de 2 elementos #(i bi+1)
```

## Method Dispatch - Self - Super

### Ejercicio 10

Indique en cada caso si la frase es cierta o falsa en *Smalltalk*. Si es falsa, ¿cómo podría corregirse?

- I. Todo objeto es instancia de alguna clase y a su vez, estas son objetos.
- II. Cuando un mensaje es enviado a un objeto, el método asociado en la clase del receptor es ejecutado.
- III. Al mandar un mensaje a una clase, por ejemplo `Object new`, se busca en esa clase el método correspondiente. A este método lo clasificamos como método de instancia.
- IV. Una *Variable de instancia* es una variable compartida por todas las instancias vivas de una clase, en caso de ser modificada por alguna de ellas, la variable cambia.
- V. Las *Variables de clase* son accesibles por el objeto clase, pero al mismo tiempo también son accesibles y compartidas por todas las instancias de la clase; es decir, si una instancia modifica el valor de dicha variable, dicho cambio afecta a todas las instancias.
- VI. Al ver el código de un método, podemos determinar a qué objeto representará la pseudo-variable *self*.
- VII. Al ver el código de un método, podemos determinar a qué objeto representará la pseudo-variable *super*.
- VIII. Un *Método de clase* puede acceder a las variables de clase pero no a las de instancia, y por otro lado, siempre devuelven un objeto instancia de la clase receptora.
- IX. Los métodos y variables de clase son los métodos y variables de instancia del objeto clase.

### Ejercicio 11 ★

Suponiendo que `anObject` es una instancia de la clase `OneClass` que tiene definido el método de instancia `aMessage`. Al ejecutar la siguiente expresión: `anObject aMessage`

- I. ¿A qué objeto queda ligada (hace referencia) la pseudo-variable *self* en el contexto de ejecución del método que es invocado?
- II. ¿A qué objeto queda ligada la pseudo-variable *super* en el contexto de ejecución del método que es invocado?
- III. ¿Es cierto que `super == self`? ¿es cierto en cualquier contexto de ejecución?

## Ejercicio 12

Se cuenta con la clase **Figura**, que tiene los siguientes métodos:

```
perimetro
  ^((self lados) sumarTodos).
lados
  ^self subclassResponsibility.
```

donde **sumarTodos** es un método de la clase **Collection**, que suma todos los elementos de la colección receptora. El método **lados** debe devolver un **Bag** (subclase de **Collection**) con las longitudes de los lados de la figura.

**Figura** tiene dos subclases: **Cuadrado** y **Círculo**. **Cuadrado** tiene una variable de instancia **lado**, que representa la longitud del lado del cuadrado modelado; **Círculo** tiene una variable de instancia **radio**, que representa el radio del círculo modelado.

Se pide que las clases **Cuadrado** y **Círculo** tengan definidos su método **perímetro**. Implementar los métodos que sean necesarios para ello, respetando el modelo (incompleto) recién presentado.

Observaciones: el perímetro de un círculo se obtiene calculando:  $2 \cdot \pi \cdot \text{radio}$ , y el del cuadrado:  $4 \cdot \text{lado}$ . Consideramos que un círculo no tiene lados. Aproximar  $\pi$  por 3,14.

## Ejercicio 13 ★

```
Object subclass: Counter [
  |count| "Instance variable."
  class << new [
    ^super new initialize: 0.
  ]

  initialize: aValue [
    count := aValue.
    ^self.
  ]

  next [
    self initialize: count+1.
    ^count.
  ]

  nextIf: condition [
    ^condition ifTrue: [self next]
      ifFalse: [count]
  ]
]
```

```
Counter subclass: FlexibleCounter [
  |block| "Instance variable"
  class << new: aBlock [
    ^super new useBlock: aBlock.
  ]

  useBlock: aBlock [
    block := aBlock.
    ^self.
  ]

  next [
    self initialize: (block value: count).
    ^count.
  ]
]
```

En la siguiente expresión:

```
aCounter := FlexibleCounter new: [:v | v+2 ]. aCounter nextIf: true.
```

Se desea saber qué mensajes se envían a qué objetos (dentro del contexto de la clase) y cuál es el resultado de dicha evaluación. Recordar que `:=` y `^` no son mensajes.

Recomendación, utilizar una tabla parecida a la siguiente:

Objeto	Mensaje	Resultado
FlexibleCounter	new:	un contador flexible (unCF de ahora en adelante)
...	...	...

### Ejercicio 14

Considerar las siguientes definiciones:

```
Object subclass: A [  
  a: x b: y  
  ^ x a: (y c) b: self.  
  
  c  
  ^ 2.  
]
```

```
A subclass: B [  
  a: x b: y  
  ^ y c + x value.  
  
  c  
  ^ 1.  
]
```

```
B subclass: C [  
  a: x b: y  
  ^ x.  
  
  c  
  ^ [self a: super c b: self].  
]
```

Hacer una tabla donde se indique, en orden, cada mensaje se envía, qué objeto lo recibe, con qué colaboradores, en qué clase está el método respectivo, y cuál es el resultado final de cada colaboración tras ejecutar el siguiente código:

```
(A new) a: (B new) b: (C new)
```