Práctica Nº 6 - 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.

Objetos - Mensajes

Ejercicio 1 ★

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

a) `10 numberOfDigitsInBase: 2.`

b) `10 factorial.`

c) `20 + 3 * 5.`

d) `20 + (3 * 5).`

e) `December first, 1985.`

f) `1 = 2 ifTrue: [ 'what!?' ].`

g) `1@1 insideTriangle: 0@0 with: 2@0 with: 0@2.`

h) `'Hello World' indexOf: $o startingAt: 6.`

i) `(OrderedCollection with: 1) add: 25; add: 35; yourself.`

j)
```
Object subclass: #SnakesAndLadders
instanceVariableNames: 'players squares turn die over'
classVariableNames: ''
poolDictionaries: ''
category: 'SnakesAndLadders'.
```

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 corraborar 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

b) Mensaje unario

c) Mensaje binario

d) Mensaje keyword

e) Colaborador

f) Variable local

g) Asignación

h) Símbolo

i) Carácter

j) Array

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 corraborarlo 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

¿Qué implementan los siguientes métodos? Dar una expresión válida que los utilice. Pensar un buen nombre para cada uno de ellos.

a)
```
String << magicMethod1
    |res|
    res := ''.
    self do: [:char | res := char asString, res].
    ^res.
```

b)
```
Integer << magicMethod2
    self <= 1 ifTrue: [ ^false ].
    2 to: (self-1) do: [ :each |  self \\ each = 0 ifTrue: [ ^false ] ].
    ^true
```

Ejercicio 8 ★

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

```
SomeClass << foo: x
  |aBlock 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 9 ★

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 "currificado".

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 10 ★

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` b_{i+1}`)`

Ejercicio 11 ★

I. Agregar a la clase `Collection` un método con la siguiente interfaz:

`detectMin: aBlock`

donde `aBlock` es un bloque con un parámetro de entrada cuya evaluación devuelve un número. El método debe evaluar el bloque en todos los elementos de la colección receptora, y devolver el mínimo de todos los valores obtenidos. Se asume que la colección receptora no está vacía.

Por ejemplo, la evaluación de la siguiente expresión:

`#('uno' 'dos' 'tres') detectMin: [:x | x size]` devuelve 3

II. La clase `Point` se utiliza para representar puntos en el plano. El punto (x, y) se representa en *Smalltalk* como `(x @ y)`. La clase cuenta, entre otros, con el método `#dist:` para obtener la distancia (`Float`) entre el objeto receptor y otro punto.

Agregar a la clase `Collection` el método `closestTo: aPoint`, que devuelva una colección de la misma clase que la receptora con los puntos que están a menor distancia del punto pasado como parámetro. Descartar todos los elementos que no sean puntos. Si la colección no tiene puntos, devolver una colección vacía.

Por ejemplo:

```
|c| c := OrderedCollection with: 2@2 with: 3@3 with: 2@1 with: 1@2.
c closerTo: 1@1.
```

Devuelve `OrderedCollection((2@1) (1@2))`

Method Dispatch - Self - Super

Ejercicio 12

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 instacia 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 13 ★

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 14

En una memorable publicidad, el fútbol se definía como "un grupo de hombres en paños menores corriendo detrás de una pelota". Siguiendo con el agudo análisis publicista del arte del balompié, se pueden distinguir tres tipos de jugadores: los que antes de correr protestan y corren solo para quitar la pelota (a los que llamaremos *defensores*), los que solo corren para patear al arco (a los que llamaremos *delanteros*) y la extraña especie de aquellos pocos privilegiados que pueden tomar el esférico con sus manos (*arqueros*).

class Jugador superclass Object	class Defensor superclass Jugador	class Delantero superclass Jugador	class Arquero superclass Jugador
#jugar self correrLaPelota #jugarCon: unJugador unJugador jugar. self jugar	#jugar self protestar. super jugar. self quitarPelota	#jugar super jugar. self patearAlArco	#jugar self atajar

Si asumimos que todos los métodos que no aparecen en la tabla están implementados en la misma clase donde son invocados y contamos con las siguientes instancias:

palermo := Delantero new

ibarra := Defensor new

garcía := Arquero new

Responder las siguientes preguntas respecto de la ejecución de las expresiones:

- garcía jugarCon: palermo.
- ibarra jugar.

a) ¿Cuántas veces se manda el mensaje #jugar? ¿A qué objetos?

b) ¿Cuántas veces se manda el mensaje #correrLaPelota? ¿A qué objetos? Pista: ¿qué herramienta puede usarse para comprobar qué mensajes se envían, y a quiénes?

Ejercicio 15

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: $2 \cdot \pi \cdot radio$, y el del cuadrado: $4 \cdot lado$. Consideramos que un círculo no tiene lados. Aproximar π por 3,14.

Ejercicio 16 ★

```
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)
...	...	...

Metaprogramación

Ejercicio 17 ★

I. Dar una expresión que permita determinar la cantidad de subclases de la clase Object.

II. Dar una expresión que permita obtener la cantidad de clases en *Smalltalk* (utilizar la clase Smalltalk)

III. Dar una expresión que agregue a todas las instancias de `Array` el método `'palindrome'`, que devuelve verdadero si el `Array` es capicúa.

Ejercicio 18 ★

Implementar la clase `PluggableProxy`, de tal manera que cada vez que reciba un mensaje para un objeto determinado evalúe el bloque asociado a dicho mensaje y luego redirija el mensaje al objeto elegido como destino. Por ejemplo:

```
| proxyOfOne |
proxyOfOne := PluggableProxy of: 1 whenReceving: #printString doBefore:
                                                    [Transcript show: 'About to print 1'; cr].
```

Si se evalúa:

- `proxyOfOne printString` ⇝ Debe poner en el trancript: "`About to print 1`", y en la siguiente línea: "`1`".
- `proxyOfOne + 2` ⇝ Debe devolver 3.

Recordar:

- Cuando se envía a un objeto un mensaje que éste no sabe responder, el objeto recibe en lugar del mensaje original el mensaje `doesNotUnderstand: unMensaje`, donde `unMensaje` representa al mensaje original.
- El mensaje `#selector` responde el selector correspondiente a un mensaje.
- En Smalltalk existe la clase `MessageSend`, que representa el envío de un mensaje. Posee un mensaje de clase `#receiver:selector:` y un mensaje de instancia `#value`.

Ejercicio 19 ★

En este ejercicio incorporaremos al lenguaje Smalltalk la capacidad de crear clases unitarias (en inglés *singleton*). Así como cada clase puede crear subclases de sí misma al recibir el mensaje `subclass: nombreSubclase`, ahora también podrá crear clases con una única instancia.

Implementar el método de instancia `singletonSubclass: nombreSubclase` para la clase `Class`. El comportamiento esperado es el siguiente: cuando una clase reciba el mensaje `singletonSubclass:` con un símbolo, debe crear una subclase de sí misma cuyo nombre sea el símbolo pasado como parámetro. Además, la nueva clase deberá redefinir el método `new`, de manera que si ya existe una instancia no cree una nueva, y en cambio devuelva la instancia ya existente.

Está permitido incluir en la clase que se crea variables de instancia y/o de clase según sea necesario. Al escribir el código, poner especial atención en el objeto receptor de cada mensaje (la clase original, la nueva subclase, alguna metaclase, etc.)

Algunos mensajes útiles:

- `Class>>subclass: instanceVariableNames: classVariableNames: category:`
  crea una subclase de la clase receptora y la devuelve. El nombre de la subclase debe ser un símbolo, y los otros argumentos son strings. Los nombres de las variables se separan con espacios. Para este ejercicio usar la categoría 'Singleton', sin poolDictionaries (usar ''), y con las variables que sean necesarias.
- `Behavior>>compile:` recibe el código completo de un método como string (con un espacio entre el nombre y la implementación) y agrega el método al objeto receptor (que es una clase para métodos de instancia, o una metaclase para métodos de clase).