Clase práctica: Smalltalk

(Parte 1)

Paradigmas de Lenguajes de Programación

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

24 de octubre de 2017

Clase de hoy

1. Repaso: Objetos / Mensajes / Métodos / Method Dispatch
2. Ejercicios Integer - Seguimiento
3. Bloques
4. Ejercicios Bloques
5. Métodos de clase
6. Ejercicio de parcial

Repaso del paradigma que utilizaremos hoy

¿Qué es un programa?

- Software como modelo computable de la realidad
- Cómputo basado en objetos enviándose **mensajes**
- Evitar soluciones procedurales **delegando responsabilidades** en objetos

Repaso del paradigma que utilizaremos hoy

¿Qué es un programa?

- Software como modelo computable de la realidad
- Cómputo basado en objetos enviándose **mensajes**
- Evitar soluciones procedurales **delegando responsabilidades** en objetos

Los **mensajes** que responden los objetos dan su comportamiento. El comportamiento denota **QUÉ** es un objeto.

Repaso del paradigma que utilizaremos hoy

¿Qué es un programa?

- Software como modelo computable de la realidad
- Cómputo basado en objetos enviándose **mensajes**
- Evitar soluciones procedurales **delegando responsabilidades** en objetos

Los **mensajes** que responden los objetos dan su comportamiento. El comportamiento denota **QUÉ** es un objeto.

Ejemplos

- Unarios: `2 pesos`
- Binarios: `December first, 1985`[a]
- Keyword: `'hola mundo' indexOf: $o startingAt: 6`

[a] notar que first es unario, ¿dónde está el binario?.

Objetos distintos para el Qué y el Cómo

¿Qué saben hacer los objetos?

Enviar mensajes y colaborando entre sí. A esta secuencia de Colaboraciones lo llamamos **método**, y definen el **CÓMO**.

Objetos distintos para el Qué y el Cómo

¿Qué saben hacer los objetos?

Enviar mensajes y colaborando entre sí. A esta secuencia de Colaboraciones lo llamamos **método**, y definen el **CÓMO**.

Observaciones

- Los métodos son objetos
- La ejecución de un método también es un objeto
- Ambos se pueden inspeccionar dentro del entorno de programación

Objetos distintos para el Qué y el Cómo

¿Qué saben hacer los objetos?

Enviar mensajes y colaborando entre sí. A esta secuencia de Colaboraciones lo llamamos **método**, y definen el **CÓMO**.

Observaciones

- Los métodos son objetos
- La ejecución de un método también es un objeto
- Ambos se pueden inspeccionar dentro del entorno de programación

Comunicación entre objetos en Smalltalk

- Dirigida
- Sincrónica
- Siempre hay respuesta
- El receptor no conoce al emisor (siempre responde igual sin importar el emisor)

Ejercicios Integer - Seguimiento

Implementar

- Implementar el método `mcm: aNumber` en donde corresponda para poder calcular el mcd entre dos números.
  Recordar que el mcm se calcula cómo $mcm(a, b) = \frac{a.b}{gcd(a,b)}$.
  Asuma que cuenta con el mensaje *gcd* implementado.

Seguimiento

- Realizar un seguimiento de la expresión `6 mcm: 10` completando la siguiente tabla:

Objeto Receptor	Selector del mensaje	Objeto resultado
6	mcm:	??
??	??	??
??	??	??

Métodos de clase

¿Qué ocurre cuando mandamos un mensaje a una clase

- Lo mismo que siempre!
- Las clases son objetos
- Como todo objeto, tienen sus colaboradores internos y su clase.

Cómo funciona el new

Dada la siguiente implementación:

```
Person class >> newWithName: nombre
  instancia := (self new).
  instancia firstName: nombre.
  nInstancias := nInstancias + 1.
  ^ instancia.
```

¿Qué ocurre si ejecutamos la siguiente colaboración?

```
Person newWithName: 'roberto'
```

Closures

Permiten representar un conjunto de colaboraciones, en definitiva, código (al cual no me importa ponerle un nombre).

Sintaxis

```
[ :x :y |
  | v |
  v := x.
  v * y
]
```

Closures

Permiten representar un conjunto de colaboraciones, en definitiva, código (al cual no me importa ponerle un nombre).

Sintaxis

```
[ :x :y |
  | v |
  v := x.
  v * y
]
```

¿Bloque, Lambda o Closure?

- Bloque: término genérico, designa una *porción de código*
- Expresión lambda: proveniente del mundo funcional (Lisp)
- Closure: bloque con **binding lexicográfico** (también un objeto, ¡obviamente!)

Pero señor McClure... ¿Qué es el binding lexicográfico?

Closures: Seguimiento

Los closures se ligan al contexto de ejecución donde son creadas.
Tanto las variables como el return.

Ejercicio

¿Qué retorna cada envío de #value en el siguiente código si ejecutamos m2?

```
A >> m1
  | x y |
  y := 10.
  x := [y := y * 2].
  ^ x.
```

```
B >> m2
  | a aBlock anotherBlock |
  a := A new.
  aBlock := a m1.
  aBlock value.
  aBlock value.
  anotherBlock := a m1.
  anotherBlock value.
  aBlock value.
```

Closures: Ejercicios

Implementar los siguientes mensajes

- `#curry`
- `#timesRepeat:`

Ejemplos

```
|currificado nuevo|
currificado := [ :x :res | x + res ] curry.
nuevo := currificado value: 10.
```

nuevo value: 2 debe valer 12

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

count debe valer 20

copy debe valer 18

Ejercicios Integer - Seguimiento

Implementar

- Implementar el método `factorial` en donde corresponda para que los números sepan responder a este mensaje.

Seguimiento

Realizar un seguimiento completo de la expresión `factorial 2` completando la siguiente tabla:

Objeto Receptor	Selector del mensaje	Objeto resultado
2	factorial	??
??	??	??
??	??	??

Ejercicio de Parcial

Hacer el seguimiento que pide el ejercicio 16 de la guía (Counter / FlexibleCounter)

```
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.
```

Colecciones

Algunas conocidas

- Bag (Multiconjunto)
- Set (Conjunto)
- Array (Arreglo)
- OrderedCollection (Lista)
- SortedCollection (Lista ordenada)
- Dictionary (Hash)

El mensaje #with: with: ...

Forma de crear estas colecciones.

Ejemplo

```
Bag with: 1 with: 2 with: 4
```

Colecciones

Algunas conocidas

- Bag (Multiconjunto)
- Set (Conjunto)
- Array (Arreglo)
- OrderedCollection (Lista)
- SortedCollection (Lista ordenada)
- Dictionary (Hash)

El mensaje #with: with: ...

Forma de crear estas colecciones.

Ejemplo

```
Bag with: 1 with: 2 with: 4
#(1 2 4) = (Array with: 1 with: 2 with: 4)
```

Colecciones

Algunas conocidas

- Bag (Multiconjunto)
- Set (Conjunto)
- Array (Arreglo)
- OrderedCollection (Lista)
- SortedCollection (Lista ordenada)
- Dictionary (Hash)

El mensaje #with: with: ...

Forma de crear estas colecciones.

Ejemplo

```
Bag with: 1 with: 2 with: 4
#(1 2 4) = (Array with: 1 with: 2 with: 4)
Bag withAll: #(1 2 4)
```

Colecciones

Mensajes más comunes

- `add:` agrega un elemento.
- `at:` devuelve el elemento en una posición.
- `at:put:` agrega un elemento a una posición
- `includes:` responde si un elemento pertenece o no.
- `includesKey:` responde si una clave pertenece o no.

Colecciones

Mensajes más comunes

- `do:` evalua un bloque con cada elemento de la colección.
- `keysAndValuesDo:` evalua un bloque con cada par clave-valor.
- `keysDo:` evalua un bloque con cada clave.
- `select:` devuelve los elementos de una colección que cumplen un predicado (filter de funcional).
- `reject:` la negación del `select:`
- `collect:` devuelve una colección que es resultado de aplicarle un bloque a cada elemento de la colección original (map de funcional).
- `detect:` devuelve el primer elemento que cumple un predicado.
- `detect:ifNone:` como `detect:`, pero permite ejecutar un bloque si no se encuentra ningún elemento.
- `reduce:` toma un bloque de dos o más parámetros de entrada y hace fold de los elementos de izquierda a derecha (foldl de funcional).

Colecciones: Ejercicios

El mensaje #do:

La forma de iterar estas colecciones es propio de cada colección

#collect:

Implementemos el siguiente método en la clase Collection:

```
collect: aBlock
```

Al ejecutarse, retorna la colección resultante de aplicar ese bloque a cada elemento de la colección original.
Ejemplo: **res** debe contener 6, 7 y 9 luego de ejecutar lo siguiente

```
| s res|
s := Set with: 1 with: 2 with: 4.
res := s collect: [ :x | x + 5 ].
```

Colecciones: Ejercicios

El mensaje #do:

La forma de iterar estas colecciones es propio de cada colección

#collect:

Implementemos el siguiente método en la clase Collection:

```
collect: aBlock
```

Al ejecutarse, retorna la colección resultante de aplicar ese bloque a cada elemento de la colección original.
Ejemplo: **res** debe contener 6, 7 y 9 luego de ejecutar lo siguiente

```
| s res|
s := Set with: 1 with: 2 with: 4.
res := s collect: [ :x | x + 5 ].
```

- ¿Cómo decidimos qué clase de colección usar para el resultado? Array no responde a #add.
- ¿Cómo logró acceder desde el bloque al resultado parcial?

Colecciones: Ejercicios (2)

#minConsidering:

Agregar a la clase Collection un método con la siguiente interfaz:

```
minConsidering: aBlock
```

- 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.

Colecciones: Ejercicios (2)

#minConsidering:

Agregar a la clase Collection un método con la siguiente interfaz:

```
minConsidering: aBlock
```

- 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.

- ¿Cómo inicializar un primer valor?
- ¿Funciona para `Set`?