

# Programación Orientada a Objetos

Departamento de Computación, FCEyN, UBA

6 de junio de 2017

# Enfoque Conceptual/Aplicado

- ▶ Vamos a introducir los **conceptos fundamentales** del paradigma
  - ▶ Objetos y modelo de cómputo.
  - ▶ Clasificación y herencia
  - ▶ Super y static method dispatch
- ▶ Ilustraremos estos conceptos con el lenguaje **Smalltalk**

# Objetos - La visión mística (o metáfora)

- ▶ Todo programa es una simulación. Cada entidad del sistema que se está simulando se representa en el programa a través de una entidad u **objeto**.
  - ▶ Personificar los objetos físicos o conceptuales de un dominio del mundo real en objetos en el dominio del programa
  - ▶ los objetos en el programa tienen las características y capacidades del mundo real que nos interese modelar.
- ▶ Todas las componentes de un sistema son objetos

# Modelo de Computación - La visión mística (o metáfora)

- ▶ El modelo de computación consiste en el envío de mensajes: Un sistema está formado por *objetos* que comunican a través del **intercambio de mensajes**.
- ▶ **Mensajes** es una solicitud para que un objeto lleve a cabo una de sus operaciones
- ▶ El **receptor**, es decir, el objeto que recibe el mensaje, determina cómo llevar a cabo la operación.

## Ejemplo: "unCirculo radio"

- ▶ unCirculo es el objeto receptor
- ▶ radio es el mensaje que se le envía.

# Objetos

- ▶ El conjunto de mensajes a los que un objeto responde se denomina **interfaz** o **protocolo**
- ▶ La forma en que un objeto lleva a cabo una operación está descrita por un **método** (describe la implementación de las operaciones)
- ▶ La forma en que un objeto lleva a cabo una operación puede depender de un **estado** interno
  - ▶ El estado se representa a través de un conjunto de **colaboradores internos** (también llamados **atributos** o **variables de instancia**)

## unRectangulo

- ▶ interfaz: area
- ▶ atributos: alto y ancho
- ▶ método: area  
 $\text{^alto*ancho}$

# Objetos

La única manera de interactuar con un objeto es a través del envío de mensajes

- ▶ la implementación de un objeto no puede depender de los detalles de implementación de otros objetos
- ▶ principio básico heredado de los Tipos Abstractos de Datos, antecesores de los Objetos:

## Principio de ocultamiento de la información

El estado de un objeto es **privado** y solamente puede ser consultado o modificado por sus métodos

# Mensajes - Tipos y Sintaxis en Smalltalk

## Tipos

- ▶ Unarios:
  - ▶ anArray **size**
  - ▶ 1 **factorial**
- ▶ Binarios:
  - ▶ 1 + 2
  - ▶ December, 2014
- ▶ Keyword
  - ▶ anArray **at:** 1 **put:** 2
  - ▶ aList **add:** 2
  - ▶ Rectangulo deBase: 10 cm yAltura: 20 cm

## Prioridad

unarios > binarios > keywords (de izquierda a derecha)

anArray at: 2 \* 10 factorial \* 3 put: name size

== (anArray at: ((2 \* (10 factorial)) \*3) put: (name size))

# Detrás del modelo de cómputo: Method dispatch

- ▶ La interacción entre objetos se lleva a cabo a través de **envío de mensajes**
- ▶ Al recibir un mensaje se activa el método correspondiente
- ▶ Para poder procesar este mensaje es necesario hallar la **declaración del método** que se pretende ejecutar
- ▶ El proceso de establecer la asociación entre el mensaje y el método a ejecutar se llama **method dispatch**
- ▶ Si el method dispatch se hace en tiempo de
  - ▶ **compilación** (i.e. el método a ejecutar se puede determinar a partir del código fuente): se habla de **method dispatch estático**
  - ▶ **ejecución**: se habla de **method dispatch dinámico**



# Corrientes

¿Quién es responsable de conocer los métodos de los objetos?

## 2 Alternativas conocidas:

- ▶ Clasificación
- ▶ Prototipado

# Clasificación

## Clases

- ▶ Modelan **conceptos abstractos** del dominio del problema a resolver
- ▶ Se utilizan principios de diseño para decidir cuando crearlas
- ▶ Definen el comportamiento y la forma de un conjunto de objetos (**sus instancias**)
- ▶ **Todo** objeto es instancia de alguna clase

# Ejemplo

## clase Point

Variables de instancia 'xCoord' e 'yCoord'

Métodos

x

~xCoord

y

~yCoord

dist: aPoint

"Answer the distance between aPoint and the receiver."

| dx dy |

dx := aPoint x - xCoord.

dy := aPoint y - yCoord.

~ (dx \* dx + (dy \* dy)) sqrt

# Componentes de una clase

## Componentes de una clase

- ▶ un nombre
- ▶ definición de variables de instancia
- ▶ métodos de instancia
- ▶ por cada método se especifica
  - ▶ su nombre
  - ▶ parámetros formales
  - ▶ cuerpo

## Además en Smalltalk

- ▶ Las clases son objetos
- ▶ definición de variables de clase
- ▶ métodos de clase

# Ejemplo

## clase INode

### Métodos de clase

```
l: leftchild r:rightchild  
    "Creates an interior node"  
    ...
```

Vars. de instancia 'left right'

### Métodos de instancia

```
sum  
    ^ left sum + right sum
```

## clase Leaf

### Métodos de clase

```
new: anInteger  
    "Creates a leaf"  
    ...
```

Vars. de instancia 'value'

### Métodos de instancia

```
sum  
    ^value
```

## Ejemplos

- 1) Leaf new: 5
- 2) (INode l: (Leaf new: 3) r: (Leaf new: 4)) sum

# Self

Pseudo variable que, durante la evaluación de un método, referencia al receptor del mensaje que activó dicha evaluación.

- ▶ no puede ser modificada por medio de una asignación.
- ▶ se liga automáticamente al receptor cuando comienza la evaluación del método.

## clase INode

### Métodos de clase

```
l: leftchild r:rightchild
  "Creates an interior node"
  ...
```

Var. de instancia 'left right'

### Métodos de instancia

```
l
  ^left
r
  ^right
sum
  ^ (self l) sum + (self r) sum
```

# Uso común de self

## clase Point

### Métodos de clase

```
x: xInteger y: yInteger
    "Creates an instance of Point"
    ^self new setX: xInteger setY: yInteger
```

### Variables de instancia 'x' e 'y'

### Métodos de instancia

```
x
    ^x
```

```
y
    ^y
```

```
setX: xValue setY: yValue
    x := xValue.
    y := yValue.
```

```
dotProduct: aPoint
    ^ (self x * aPoint x) + (self y * aPoint y)
```

## ¿Quién es self?

```
(Point x: 10 y: 20) dotProduct: (Point x: 1 y: 1)
```

# Jerarquía de clases

- ▶ Es común que nuevas clases aparezcan como resultado de la extensión de otras existentes incluyendo
  - ▶ adición o cambio del comportamiento de uno o varios métodos
  - ▶ adición de nuevas variables de instancia o clase
- ▶ Una clase puede **heredar de** o **extender** una clase pre-existente (la **superclase**)
- ▶ La transitividad de la herencia da origen a las nociones de **ancestros** y **descendientes**



# Ejemplo

```
Object subclass: #Point
instanceVariableNames: 'x y'
```

## Métodos de clase

```
x: p1 y: p2
    ^self new setX: p1 setY: p2
```

## Métodos de instancia

```
x
    ^x

y
    ^y
```

```
setX: xValue setY:yValue
    x := xValue.
    y := yValue.
```

## USO

```
ColorPoint x: 10 y: 20 color: red.
```

```
Point subclass: #ColorPoint
instanceVariableNames: 'color'
```

## Métodos de clase

```
x: p1 y: p2 color: aColor
    |instance|
    instance := self x: p1 y: p2.
    instance color: aColor.
    ^instance
```

## Métodos de instancia

```
color: aColor
    color := aColor

color
    ^color
```

# Herencia

- ▶ Hay dos tipos de herencia
  - ▶ **Simple**: una clase tiene una única clase padre (salvo la clase raíz object)
  - ▶ **Múltiple**: una clase puede tener más de una clase padre
- ▶ La gran mayoría de los lenguajes OO utilizan **herencia simple**
- ▶ La herencia múltiple complica el proceso de method dispatch

# Inconveniente con herencia múltiple

- ▶ Supongamos que
  - ▶ clases  $A$  y  $B$  son incomparables y  $C$  es subclase de  $A$  y  $B$
  - ▶  $A$  y  $B$  definen (o heredan) dos métodos diferentes para  $m$
  - ▶ se envía el mensaje  $m$  a una instancia  $C$
- ▶ ¿Qué método debe ejecutarse?
- ▶ Dos soluciones posibles:
  - ▶ Establecer un **orden de búsqueda** sobre las superclases de una clase
  - ▶ Si se heredan dos métodos diferentes para el mismo mensaje debe ser **redefinidos** en la clase nueva

# Method Dispatch Estático

- ▶ Method dispatch dinámico es uno de los pilares de la POO (junto con la noción de clase y de herencia)
- ▶ Por cuestiones de eficiencia (o diseño, como el caso de C++) muchos lenguajes también cuentan con method dispatch estático
- ▶ Sin embargo, hay algunas situaciones donde method dispatch estático es **requerido**, más allá de cuestiones de eficiencia
- ▶ Un ejemplo es el **super**

# Method Dispatch Estático

Supongamos que queremos extender la clase point del siguiente modo:

```
Object subclass: #Point
```

```
Métodos de instancia
```

```
setX: xValue setY: yValue
```

```
    x:= xValue.
```

```
    y:= yValue.
```

```
Point subclass: #ColorPoint
```

```
Métodos de instancia
```

```
setX: xValue setY: yValue setColor: aColor
```

```
    x:= xValue.
```

```
    y:= yValue.
```

```
    color:= aColor.
```

¡setX: setY: setColor: duplica código **innecesariamente**!

# Method Dispatch Estático

- ▶ Esto es un ejemplo claro de mala práctica de programación en presencia de herencia
- ▶ Deberíamos recurrir al código ya existente del método `initialize` de `point` para que se encargue de la inicialización de `x` e `y`

```
Object subclass: #Point
```

```
Métodos de instancia
```

```
setX: xValue setY: yValue
```

```
    x:= xValue.
```

```
    y:= yValue.
```

```
Point subclass: #ColorPoint
```

```
Métodos de instancia
```

```
setX: xValue setY:yValue setColor: aColor
```

```
    self setX: xValue setY: yValue.
```

```
    color:= aColor.
```

# Method Dispatch Estático

- ¿La siguiente variante funciona?

```
Object subclass: #Point
```

```
Métodos de instancia
```

```
setX: xValue setY: yValue
```

```
    x:= xValue.
```

```
    y:= yValue.
```

```
Point subclass: #BluePoint
```

```
Métodos de instancia
```

```
setX: xValue setY: yValue
```

```
    self setX: xValue setY: yValue.
```

```
    color:= 'azul'.
```

# Method Dispatch Estático

## Super

- ▶ Pseudovariable que **referencia al objeto que recibe el mensaje**
- ▶ **Cambia** el proceso de activación al momento del envío de un mensaje.
- ▶ Una expresión de la forma "super msg" que aparece en el cuerpo de un método m provoca que el **method lookup** se haga desde el padre de la **clase anfitriona** de m

## Código corregido

```
Object subclass: #Point
```

```
Métodos de instancia
```

```
setX: xValue setY: yValue
```

```
  x:= xValue.
```

```
  y:= yValue.
```

```
Point subclass: #BluePoint
```

```
Métodos de instancia
```

```
setX: xValue setY: yValue
```

```
  super setX: xValue setY: yValue.
```

```
  color:= 'azul'.
```



# Ejemplo - Super/Self

```
Object subclass: #C1
```

```
Métodos de instancia
```

```
m1
```

```
  ^self m2
```

```
m2
```

```
  ^13
```

```
C1 subclass: #C2
```

```
Métodos de instancia
```

```
m1
```

```
  ^22
```

```
m2
```

```
  ^23
```

```
m3
```

```
  ^super m1
```

```
C2 subclass: #C3
```

```
Métodos de instancia
```

```
m1
```

```
  ^32
```

```
m2
```

```
  ^33
```

Sigamos la ejecución de

(c2 new) m3. ----> Qué valor devuelve?

## Variante de super en algunos lenguajes

`super[A] n(...)`

- ▶ Similar super ya visto
- ▶ Salvo que la búsqueda comienza desde la clase  $A$
- ▶  $A$  debe ser una superclase de la clase anfitriona de  $m$ , el método que contiene la expresión super

# Lenguajes basados en Objetos

- ▶ Caracterizados por la ausencia de clases
- ▶ Constructores para la creación de objetos particulares

```
object cell is
  var contents := 0;
  method get() is return self.contents end;
  method set(n) is self.contentes:= n end;
end
```

- ▶ Procedimientos para generar objetos

```
procedure newCell(m) is
  object cell is
    var contents := 0;
    method get() is return self.contents end;
    method set(n) is self.contentes:= n end;
  end;
  return cell;
end;
```

# Prototipado

- ▶ Construye instancias concretas que se interpretan como representantes canónicos de instancias (llamados prototipos)
- ▶ Otras instancias se generan por clonación (copia shallow)

```
clonedCell := clone cell
```

- ▶ clone opera sobre instancias en lugar de sobre clases
- ▶ los clones se pueden cambiar

```
clonedCell.contents := 3
```

```
clonedCell.get := method () return self.contents + 1 end;
```

- ▶ Clonado y modificación de métodos es una forma limitada de herencia
- ▶ Hay otros mecanismos (embedding y delegación)

# Cálculo de Objetos (intuición) [Abadi&Cardelli,98]

## Syntaxis

$$a ::= x \mid [l_i : \varsigma(x_i)b_i^{i \in 1..n}] \mid a.l \mid a.l \leftarrow \sigma(x_i)b_i$$

## Semántica operacional

$$\frac{a \rightarrow v' \quad b_j[v'/x_j] \rightarrow v \quad j \in 1..n}{a.l_j \rightarrow v} [Sel] \quad \text{con } v' = [l_i : \varsigma(x_i)b_i^{i \in 1..n}]$$

$$\frac{a \rightarrow [l_i : \varsigma(x_i)b_i^{i \in 1..n}] \quad j \in 1..n}{a.l_j \leftarrow \varsigma(x)b \rightarrow [l_j : \varsigma(x)b, l_i : \varsigma(x_i)b_i^{i \in 1..n - \{j\}}]} [Upd]$$

## Ejemplo

```
zero = [  
    iszero =  $\varsigma(x)$ true,  
    pred =  $\varsigma(x)$ x,  
    succ =  $\varsigma(x)(x.iszero \leftarrow false).pred \leftarrow x]$ 
```

# Notas finales

- ▶ En este cálculo pueden codificarse las funciones (abstracciones lambda)
- ▶ Con las abstracciones lambda pueden escribirse métodos que reciben parámetros
- ▶ Se pueden representar clases (usando traits) y herencia.
- ▶ Existe la versión tipada