Programación Orientada a Objetos

Introducción a POO
Ruby

Objetivos de la materia

- Distinguir la ventaja de la utilización de la programación orientada a objetos en el diseño e implementación de programas frente a otros paradigmas.
- Diseñar e implementar jerarquías de objetos que sean una abstracción del mundo real y permitan resolver problemas.

NO son objetivos de la materia

- Implementación de estructuras de datos (lo verán EDA)
- Algoritmos complejos (también en EDA)
- Patrones de diseño (ya lo verán)
- Programación concurrente (*Threads*)

Cursada

- Asistencia no obligatoria
- Aprobar la cursada
  - Dos parciales
  - Autoevaluaciones
  - TPE Ruby
- Final
  - TPE en grupo (Java)

La ausencia a clases no justifica el desconocimiento de todo lo ocurrido en las mismas

La Cátedra asume que los alumnos poseen los conocimientos de las materias correlativas

ADT: Abstract Data Type

- Hiding
  - Alta cohesión
  - Bajo acoplamiento
- Packaging (encapsulamiento)
- Contrato

ADT: Ejemplos

- List
- Queue
- Set
- BST
- etc

OOP

- Sub-paradigma del Paradigma Imperativo
- Un programa es un conjunto de objetos que interactúan a través de mensajes
- Los objetos son *instancias* de una clase
- Una clase encapsula propiedades y métodos
- Un objeto tiene un estado (los valores de sus atributos)
- El estado de un objeto se modifica a través de los métodos definidos en la clase

OOP

- TAD
  - Ocultamiento de la información
  - Encapsulamiento
+ Herencia
+ Polimorfismo

Conceptos: Clase

Clase: Modelo o molde a partir del cual se pueden crear objetos. Define las características y comportamiento de los objetos de su tipo.

```ruby
class Date
 @year
 @month
 @day

 def initialize(day, month, year)
   @day = day
   @year = year
   @month = month
 end

end
```

Herencia

- Definir una nueva clase en base a una clase existente
- La subclase contiene (hereda) los atributos y métodos de la clase “padre”
- Algunos lenguajes permiten la herencia múltiple

Polimorfismo

- “Que posee varias formas diferentes”
- De sobrecarga: distintas clases con métodos con el mismo nombre
- Paramétrico: dentro de una misma clase métodos con el mismo nombre pero distintos parámetros
- De redefinición: una clase hija sobreescribe un método de la clase padre

Variable de instancia

Variable que se relaciona con una única instancia de una clase.

```
class Date
 @year
 @month
 @day

 def initialize(day, month, year)
   @day = day
   @year = year
   @month = month
 end

end
```

Variable de clase

También llamadas estáticas. Es una variable propia de la clase. Todos los objetos de esa clase comparten su valor

```
class Date
 @year
 @month
 @day
 @@pattern

 def initialize(day, month, year)
   @day = day
   @year = year
   @month = month
 end

end
```

Método y mensaje

- Un mensaje es enviado por un objeto para invocar comportamiento
- Un método se implementa en una clase , y determina cómo debe actuar el objeto cuando recibe un mensaje.
- Los objetos interactúan enviándose mensajes unos a otros. Tras la recepción de un mensaje el objeto actuará. La acción puede ser el envío de otros mensajes, el cambio de su estado, o la ejecución de cualquier otra tarea que se requiera que haga el objeto.

Método de clase

Puede ser invocado sin que exista una instancia.

Necesarios para crear una nueva instancia.

```
class Date
 @year
 @month
 @day
 @@pattern

 def initialize(day, month, year)
   @day = day
   @year = year
   @month = month
 end
```

```
  def Date.class_method1
    ...
  end

  def self.class_method2
    ...
  end
end

d = Date.new(15, 12, 2017)
Date.class method1
Date.class_method2
```

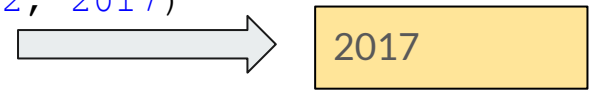
Acceso a propiedades

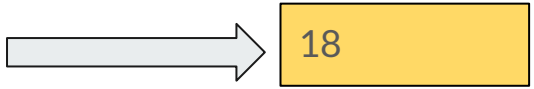
```
class Date
 @year
 @month
 @day
 @@pattern

 ...
end

d = Date.new(15, 12, 2017)

puts d.year
```

undefined method `year'

Método de instancia

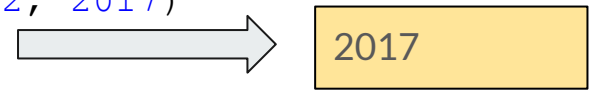
```ruby
class Date
 @year
 @month
 @day
 ...
 def year
  @year
 end
 def set_year(new_year)
  @year = new_year % 100
 end

end
d = Date.new(15, 12, 2017)
puts d.year

d.set year(2018)
puts d.year
```

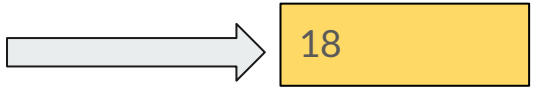




Método de instancia

```
class Date
 @year
 @month
 @day
 ...
 def year
  @year
 end
 def set_year(new_year)
  @year = new_year % 100
 end

end
d = Date.new(15, 12, 2017)
puts d.year

d.set_year 2018
puts d.year
```

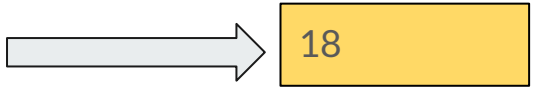




Método de instancia

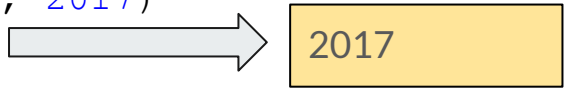
```ruby
class Date
 @year
 @month
 @day
 ...
 def year
  @year
 end
 def year=(new_year)
  @year = new_year % 100
 end

end
d = Date.new(15, 12, 2017)
puts d.year

d.year = 2018
puts d.year
```

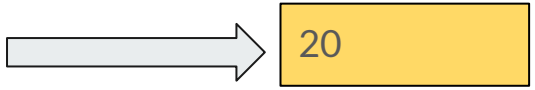
"Virtual attribute"



Métodos: valores por defecto

```ruby
class Date
 @year
 @month
 @day
 ...
 def year
  @year
 end
 def set year(new year = 2020)
  @year = new_year % 100
 end

end
d = Date.new(15, 12, 2017)
puts d.year

d.set year
puts d.year
```





Métodos: validación de parámetros

```ruby
class Date
 @year
 @month
 @day
 ...

 def month=(new_month)
  raise 'invalid month' if not (1..12).include? new_month
  @month = new_month
 end

 def year=(new_year)
  raise 'invalid year' if new_year == nil
  @year = new_year % 100
 end

end
```

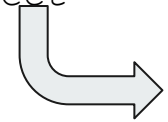
Métodos: validación de parámetros

```ruby
class Date
 @year
 @month
 @day
 ...

 def month=(new_month)
  raise 'invalid month' unless (1..12).include? new_month
  @month = new_month
 end

 def year=(new_year)
  raise 'invalid year' if new_year.nil?
  @year = new_year % 100
 end

end
```

Método to_s vs inspect

```ruby
class Date
 @year
 @month
 @day
 @@pattern
 ...

 def to_s
     "#{@day}/#{@month}/#{@year}"
 end
end

d = Date.new(1, 10, 2017)

puts d

puts d.inspect
```

1/10/2017



#<Date:0x0000000002b9d7f8 @day=1, @year=2017, @month=5>

Herencia

Dentro de un método, *super* envía un mensaje al mismo método de la clase padre del objeto

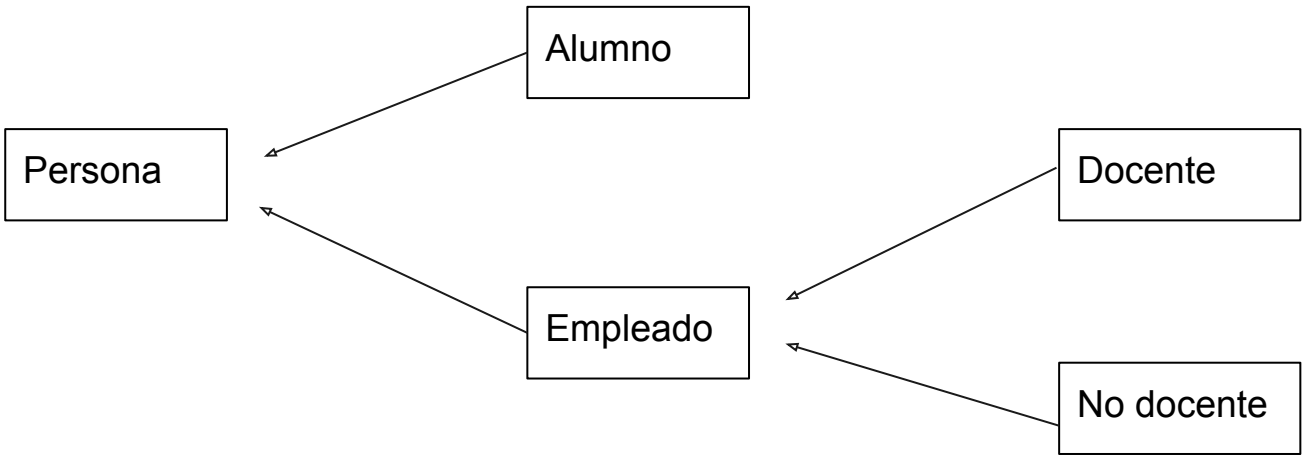
```
class DateTime < Date
  @hour=0
  @minute=0
  @second=0

  def initialize(day, month, year, hour=nil, minute=nil, second=nil)
   super(day, month, year)
   @hour, @minute, @second = hour, minute, second
  end

  def to_s
    super + "%s" % ( " #{@hour}:#{@minute}:#{@second}" unless
@hour.nil?)
  end
end
```

Relación entre clases e instancias

- La herencia crea una relación “es-una” entre clases
- La **composición** crea una relación de dependencia “tiene-un” entre instancias
  - La clase *persona* tiene cero o más *tarjetas de crédito*
  - La clase *remito* contiene una lista de *items*
- La **asociación** son dos objetos trabajando juntos.
  - Un alumno tiene asociado un plan, un conjunto de materias aprobadas y un conjunto de cursos en los que está inscripto
  - Un curso tiene una lista de alumnos activos

¿Herencia o Composición?

En SGA se debe manejar información de personal docente, personal no docente y alumnos.
Posible solución: modelarlo usando herencia



Clase abstracta (Ruby)

```ruby
class Transport
   @capacity = 0
   @model = 0 # year of construction
   def capacity
     @capacity
   end
   def initialize
     raise 'this method should be overriden'
   end
end


class Ship < Transport

 def initialize
   #
 end

end
```

```ruby
u = Ship.new      OK


t = Transport.new ERROR
```

Clase abstracta (Java)

```java
public abstract class Transport {
    private int capacity = 0;
    private Integer model = null;

    ...
}
```

Herencia y mensajes

```ruby
class GF
 def initialize
   puts 'In GF class'
 end
 def gfmethod
   puts 'GF method call'
 end
 def smethod
   puts 'smethod in GF'
 end

end

# class F sub-class of GF
class F < GF
 def initialize
   puts 'In F class'
 end
end
```

```ruby
# class S sub-class of F
class S < F
 def initialize
   super
   puts 'In S class'
 end
 def smethod
   super
   puts 'smethod in S'
 end
end

son = S.new
```

```
In F class
In S class
```

Herencia y mensajes

```ruby
class GF
 def initialize
   puts 'In GF class'
 end
 def gfmethod
   puts 'GF method call'
 end
 def smethod
   puts 'smethod in GF'
 end

end

# class F sub-class of GF
class F < GF
  # def initialize
  #   puts 'In F class'
  # end
end
```

```ruby
# class S sub-class of F
class S < F
 def initialize
   super
   puts 'In S class'
 end
 def smethod
   super
   puts 'smethod in S'
 end
end

son = S.new
```

```
In GF class
In S class
```

Herencia y mensajes

```ruby
class GF
 def initialize
   puts 'In GF class'
 end
 def gfmethod
   puts 'GF method call'
 end
 def smethod
   puts 'smethod in GF'
 end

end

# class F sub-class of GF
class F < GF
 def initialize
   puts 'In F class'
 end
end
```

```ruby
# class S sub-class of F
class S < F
 def initialize
   # super
   # puts 'In S class'
 end
 def smethod
   super
   puts 'smethod in S'
 end
end

son = S.new
son.gfmethod
son.smethod
```

```
GF method call
smethod in GF
smethod in S
```

Herencia y mensajes

```ruby
class GF
 def initialize
   puts 'In GF class'
 end
 def gfmethod
   puts 'GF method call'
 end
 def smethod
   puts 'smethod in GF'
 end

end

# class F sub-class of GF
class F < GF
 def initialize
   puts 'In F class'
 end
end
```

```ruby
# class S sub-class of F
class S < F

 def smethod
   super
   puts 'smethod in S'
 end
end

son = S.new
son.gfmethod
son.smethod
```

```
In F class
GF method call
smethod in GF
smethod in S
```

UML (Unified Modeling Language)

Es un lenguaje visual de modelado de software creado en 1997 para unificar los diferentes estándares existentes hasta el momento.