Iniciación a Ruby

1. Notas generales:

- Lo primero a tener en cuenta en Ruby es que los programas **no se compilan**, Ruby es un lenguaje interpretado, se va ejecutando **línea a línea** esto provoca que:

  1. Como no se compila, no se saben los errores que puede tener el código hasta que se pasa por **todas las líneas**. Si hay líneas que rara vez se ejecutan, se podrían acumular errores inconscientemente.
  2. No es necesario poner punto y coma al final de cada sentencia, sino que hay que poner una sentencia por línea, las cuales se interpretan igual que en los demás lenguajes.
  3. No hay que escribir llaves al implementar un bloque (`class X, for, while, def...`) se indica su fin con la palabra reservada `end` que implica el fin de bloque.
  4. Para la gestión de proyectos se usa un **Rakefile**.
  5. Para separar una sentencia muy larga en varias líneas se utiliza \. Por ejemplo, la línea de código

  `Ruby string = "line #1line #2line #3"`

  podemos reescribirla como `Ruby string = "line #1"\ "line #2"\ "line #3"`

- Normalmente en Ruby se usa la notación **snake_case**, que consiste en separar las palabras con un guión bajo, sin embargo en **PDOO** se va a utilizar **camelCase** por "razones técnicas".
- Ruby es un lenguaje de **tipado dinámico**, es decir, no se indica explícitamente el tipo de las variables. Esto implica que una variable puede **variar su tipo** a lo largo del programa.
- Para comentar una línea se usa `#`, para comentar varias se utiliza `=begin` y `=end`, siempre en líneas separadas y sin tabular.
- Existe `return` pero habitualmente no se usa, cuando se llega al final de la función, se devuelve la última salida que se ha producido.
- La sintaxis de la declaración de funciones es la siguiente:

Declaración de métodos sin parámetros `Ruby def nombre_metodo # cosas # cosas lo_que_quiero_devolver end`

Declaración de métodos con parámetros `Ruby def nombre_metodo(mi_var, otra_var) # cosas # cosas lo_que_quiero_devolver end`

2. Clases:

- Debemos fijarnos que en Ruby (como en Java) todo son clases, vamos a tener que trabajar con la idea de que hagas lo que hagas vas a usar clases. Por ello hay que tener claro como son las clases en Ruby.
- Todas las clases deben tener un método `initialize` que sería lo que nosotros denominamos un **constructor** y es **ÚNICO**, luego veremos como crear varios constructores. Para evitar posibles errores en Ruby, en PDOO siempre vamos a darles un estado inicial a todas las variables en el `intialize`.
- Los datos miembro de una instancia (**atributos de instancia**) siempre son privados y los métodos públicos (por defecto), **cuidado** porque que algo sea privado en Ruby tiene mucha fuerza:

1. Si un método es privado entonces **No** podemos asignar lo que devuelva el método a nada, ni siquiera entre objetos de la misma clase, solo puede asignarse a algo si es el mismo objeto que invoca. Sin embargo **Si** se pueden llamar normalmente.
2. Con los atributos pasa igual, no podemos consultarlos y/o modificarlos de otros objetos, a no ser que tengamos un `attr_` especificado (ahora lo veremos).
3. Solo se pueden acceder a atributos con total libertad si es del propio objeto, y asignar si es tu propio método privado, es decir si al llamar a los métodos o acceder a las variables no es del tipo `var = obj.metodo_privado` o `obj.variable`, incluso `self.obj` daría fallo. Solo cuando podemos poner `var = metodo_privado`, `@variable` es cuando tenemos acceso ("no hay nada delante con un punto").
4. La implementación de los métodos set/get se puede hacer de la manera habitual, aunque hay una opción más rápida y es usando los `attr_`. Estos "accessors" son muy cómodos y rápidos pero tienen el inconveniente de que no se pueden hacer comprobaciones de ningún tipo en ellos. Se usan:
    - `attr_accesor :var` permite consultar/modificar el atributo var
    - `attr_reader :var` permite consultar el atributo var
    - `attr_writer :var` permite modificar el atributo var

- Cabe añadir que la manera de hacer referencia a los **atributos de una instancia de la clase** siempre se hace añadiendo una **arroba** antes, es decir: `@var`.
- Para **atributos estáticos o de clase**, es decir, atributos que son únicos de manera que todos los objetos de clase van a "compartir" ese atributo, si un objeto lo modifica entonces lo va a modificar para todos los demás objetos.

- Para **métodos estáticos**, es decir, métodos que pueden ser llamados sin tener que crear un objeto (como es el `new`, que llamamos para crear un objeto) solo tenemos que poner `self.` antes del nombre del método, es decir: "'Ruby class MiClase

  attr_accesor :var attr_accesor :var_estatica

  def initialize @var = "hola" @@var_estatica = 3 end

  def self.metodo_estatico puts "metodo_estatico" end end

MiClase.metodo_estatico # Imprime "metodo_estatico" miClase1 = MiClase.new # Crea un objeto de clase MiClase miClase2 = MiClase.new miClase1.var_estatica = 1 puts miClase2.var_estatica # Imprime "1" "'

- Como ya hemos visto cuando hacemos `new`, creamos un nuevo objeto, que lo que hace es llamar al método initialize donde tenemos que darle valor a los atributos.
- Para ayudar a la legibilidad del código, aconsejo declarar los `attr_` al principio para tener claro los atributos de la clase, aunque luego todos van a aparecer en el `initialize` puedes ver rápidamente los atributos de la clase.
- Dijimos antes que si solo podemos crear un único metodo `initialize` como podemos tener varios constructores. La idea es crear varios métodos estáticos `self.` que dentro llamen al `new.`, de esta manera tendremos el initialize que le paseremos todas los parámetros, y otros metodos (constructores) definiendo ya las variables que no pasamos. Además si queremos evitar que se use el constructor general `initialize` entonces tendremos que ponerle `private` para versiones de Ruby en 2.0, para anteriores pondremos `private_class_method :new`.

Ej. 1 "'Ruby class MiClase

```
attr_reader :var1
attr_reader :var2
attr_reader :var3
attr_reader :var4

private def initialize(var1, var2, var3, var4)
  @var1 = var1
  @var2 = var2
  @var3 = var3
  @var4 = var4
end

def self.constructor1(var1, var2)
  new(var1, var2, 3, 0)
end
```

```
  def self.constructor2(var3, var4)
    new([], 2, var3, var4)
  end
end
```

Ej. 2

```Ruby class MiClase
  attr_reader :var1
  attr_reader :var2
  attr_reader :var3
  attr_reader :var4

  private_class_method :new

  def initialize(var1, var2, var3, var4)
    @var1 = var1
    @var2 = var2
    @var3 = var3
    @var4 = var4
  end

  def self.constructor1(var1, var2)
    new(var1, var2, 3, 0)
  end

  def self.constructor2(var3, var4)
    new([], 2, var3, var4)
  end
end
```

3. Control de flujo

1. If