MANUAL DE RUBY

(PARTE IV)

Luis José Sánchez González

1. MÉTODOS

Los métodos constituyen las acciones que podemos realizar o aplicar sobre los objetos. A un objeto concreto se le aplica un método y el objeto hace algo o da una respuesta.
Esto debe ocurrir sin que tengamos necesariamente algún tipo de conocimiento o nos importe cómo realiza el objeto, interiormente, el trabajo. De hecho, en muchos casos utilizaremos métodos que no hemos creado nosotros y sólo nos importará qué hace el método pero no cómo lo hace.

En ruby, se llama a un método con la notación punto (como en C++ o Java). El objeto con el que nos comunicamos se nombra a la izquierda del punto.

Ejemplo:

```
"hola mundo".length
```

Aquí estamos aplicando el método **length** a un objeto que es la cadena **"hola mundo"**. Lo único que nos interesa es que ese método nos devuelve el tamaño de la cadena, que podremos visualizar o guardarlo en una variable, etc. Pero no nos preocupamos cómo **length** realiza esos cálculos.

Otros objetos pueden hacer una interpretación diferente de **length**.

Ejemplo:

```
frase = "hola mundo"
puts frase.length
dias_de_la_semana = ["lunes", "martes", "miercoles", "jueves", "viernes",
"sabado", "domingo"]
puts dias_de_la_semana.length
```

La decisión sobre cómo responder a un método se hace al vuelo, durante la ejecución del programa, y la acción a tomar puede cambiar dependiendo del objeto.

En este ejemplo vemos como aplicamos **length** primero sobre una cadena y luego sobre un array. Lógicamente, aunque en este caso los métodos se llamen igual, se habrán definido de forma diferente, es decir, habrá un método **length** definido en la clase **String** y otro **length** definido en la clase **Array**.

Este comportamiento libera al programador de la carga de memorizar una gran cantidad de nombres de funciones, ya que una cantidad relativamente pequeña de nombre de métodos, que corresponden a conceptos que sabemos cómo expresar en lenguaje natural, se pueden aplicar a diferentes tipos de datos siendo el resultado el que se espera.

Cuando un objeto recibe un mensaje que no conoce, es decir, cuando aplicamos un método que no está definido para la clase a la que pertenece el objeto en cuestión, salta un error:

Ejemplo:

```
a=5
a.length
```

Nos daría el mensaje:

```
NoMethodError: undefined method `length' for 5:Fixnum
```

Podemos averiguar todos los métodos a los que responde un objeto:

```
puts "Métodos a los que puede responder una variable de tipo cadena (incluyendo los métodos de las superclases)"

puts String.instance_methods

puts "Métodos a los que puede responder una variable de tipo cadena (sólo los que son específicos de cadena)"

puts String.instance_methods(false)
```

Si se pasan argumentos a un método, éstos van normalmente entre paréntesis.

```
objeto.metodo(arg1, arg2)
```

pero se pueden omitir, si su ausencia no provoca ambigüedad

```
objeto.metodo arg1, arg2
```

Ejemplo:

```
frase = "Hola, ¿qué tal?, ¿cómo te va?"
mi_array = frase.split ','
puts mi_array
```

En este caso

```
mi_array = frase.split ','
```

sería equivalente a

```
mi_array = frase.split(',')
```

2. CLASES

El mundo real está lleno de objetos que podemos clasificar. En terminología de POO, una categoría de objetos, como por ejemplo perro , se denomina**clase** y cualquier objeto determinado que pertenece a una clase se conoce como **instancia** de esa clase.

Generalmente, en ruby y en cualquier otro lenguaje orientado a objetos, se definen primero las características de una clase y luego se crean las instancias.

Ejemplo:

```
class Perro
        def ladra
                puts "guau guau"
        end
end

milu = Perro.new
milu.ladra
```

En ruby, la definición de una clase es la región de código que se encuentra entre las palabras reservadas **class** y **end**.

Dentro de este área, **def** inicia la definición de un método, que como se ha dicho anteriormente, corresponde con algún comportamiento específico de los objetos de esa clase.

El método **new** de cualquier clase, crea una nueva instancia. Dado que **milu** es un **Perro**, según la definición de la clase, sólo hay una cosa que puede hacer, ladrar.

Seguiremos la norma de definir las clases con la primera letra en mayúscula. Si el nombre de la clase está compuesto por más de una palabra, utilizaremos tantas mayúsculas como palabras. Por ejemplo, podrían ser nombres de clases **InstitutoPublico** o **VehiculoMotorizado**.

Como sabemos, en ruby no es necesario definir las variables. Simplemente se crean en el momento en que le damos el primer valor. De la misma forma, no es necesario a priori definir los atributos de una clase. Cuando tengamos que utilizar estos atributos, nos referiremos a ellos con un símbolo '@' delante. Este tipo especial de variables se llamará **variable de instancia** y estará disponible para todos los métodos de la clase.

Existe un método especial, de nombre **initialize**, al que se llama en el momento de crear una instancia de una clase. Servirá normalmente para inicializar las variables de instancia.

Veamos un ejemplo del uso de las **variables de instancia** y de **initialize**:

```
class Coche
	def initialize
		@num_plazas = 4
		@plazas_libres = 4
	end

	def monta(personas)
		if personas > @plazas_libres
			puts "No hay sitio, hay #{@plazas_libres} plazas libres."
		  else
			@plazas_libres = @plazas_libres - personas
		end
	end

	def apea(personas)
		if personas > @num_plazas - @plazas_libres
			puts "No se puede, en el coche solo hay #{@num_plazas -
@plazas_libres} personas."
		  else
			@plazas_libres = @plazas_libres + personas
		end
	end

	def asientos_libres
		return @plazas_libres
	end

end

kitt = Coche.new
puts kitt.asientos_libres
kitt.monta(3)
puts kitt.asientos_libres
kitt.monta(7)
kitt.apea(2)
puts kitt.asientos_libres
```

El método **initialize** también puede llevar parámetros. Nos podría interesar establecer el número de plazas de que dispone el coche en el momento de crearlo. Se puede indicar un valor por defecto.

El nuevo método podría quedar así:

```
class Coche
	def initialize(plazas)
		@num_plazas = plazas
		@plazas_libres = plazas
	end

	...
```

Indicando un valor por defecto quedaría así:

```
class Coche
        def initialize(plazas = 5)
                @num_plazas = plazas
                @plazas_libres = plazas
        end

        ...
```

3. HERENCIA

La clasificación de los objetos en nuestra vida diaria es evidentemente jerárquica. Sabemos que todos los gatos son mamíferos y que todos los mamíferos son animales. Las clases inferiores heredan características de las clases superiores a las que pertenecen. Si todos los animales respiran, entonces los gatos respiran.

Ejemplo:

```
class Animal
        def respira
                puts "inhalar y exhalar"
        end
end

class Gato<Animal
        def maulla
                puts "miau"
        end
end

garfield = Gato.new
garfield.respira
garfield.maulla
```

Aunque no se dice cómo respira un gato, todo gato heredará ese comportamiento de la clase **Animal** dado que se ha definido **Gato** como una subclase de **Animal**. En terminología POO, la clase inferior es una subclase de la clase superior que es una superclase. Por lo tanto, desde el punto de vista del programador, los gatos obtienen gratuitamente la capacidad de respirar; a continuación se añade el método **maulla**, así nuestro gato puede respirar y maullar.

Existen situaciones en las que ciertas propiedades de las superclases no deben heredarse por una determinada subclase. Aunque en general los pájaros vuelan, los pingüinos son una subclase de los pájaros que no vuelan.

Ejemplo:

```
class Pajaro
      def aseo
              puts "me estoy limpiando las plumas."
      end

      def vuela
              puts "estoy volando."
      end
end

class Pinguino<Pajaro
      def vuela
              puts "Lo siento, yo sólo nado."

     end
end
```

```
tux = Pinguino.new
tux.aseo
tux.vuela
```

En vez de definir exhaustivamente todas las características de cada nueva clase, lo que se necesita es añadir o redefinir las diferencias entre cada subclase y superclase.

Ejercicios

1. Crea las clases **animal, mamifero, ave, gato, perro, canario, pinguino** y **lagarto**. Crea, al menos, tres métodos específicos de cada clase y redefine el/los método/s cuando sea necesario.
2. Crea la clase **fracción**, con los métodos **visualiza**, **invierte**, **multiplica por un número** y **divide entre un número**.
3. Realiza el programa **Expocoches Campanillas** que gestiona la venta de entradas de una exposición de vehículos (ver ejercicio 11 del documento introductorio a la POO).
4. En la película La Jungla de Cristal 2, el malo propone a John McLane y a su amigo un problema. Para desactivar una bomba tienen que colocar sobre una maleta una garrafa que contenga exactamente 4 litros de agua, pero sólo disponen de una garrafa de 5 litros y otra de 3 litros. ¿Qué hacemos para que en la garrafa de 5 litros queden exactamente 4 litros?

   Define la clase **garrafa** y los métodos **llena**, **vacía**, y **queda libre**. Utiliza un menú para ir realizando los pasos necesarios para resolver el problema.