



Ruby

Serialización
Unit Tests

Files




Ruby posee un conjunto de métodos para operaciones de entrada salida (I/O), implementadas en el Kernel de Ruby.

Todos los métodos son provistos por la clase IO, como *read*, *write*, *gets*, *puts*, *readline*, *getc* y *printf*

- *puts*: muestra en STDOUT el valor del parámetro y agrega un `\n`
- *gets*: lee de STDIN un string, incluyendo el `\n`
- *putc*: envía a STDOUT un caracter. Si recibe un string sólo muestra el primer caracter
- *print*: similar a *puts* pero no agrega un `\n` al final

Estos mismos métodos se pueden usar con una instancia de File

Files



```
putc 64      # => @
putc ?.      # => .
putc ??      # => ?
putc '\n'    # => \
putc 0xa     # => imprime un enter
putc "\n"    # => idem
```

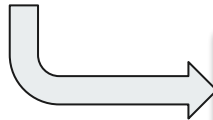


```
$stdout.puts 'es lo mismo que puts'
```

Files

Además de operar con entrada y salida estándar se pueden manipular archivos físicos
Para crear o abrir archivos se puede usar `File.new` o `File.open`. Este último puede ser asociado a un bloque

```
aFile = File.new('filename', 'mode')  
# ... process the file  
aFile.close
```



r, r+, w, w+, a, a+

```
File.open('filename', 'mode') do |aFile|  
  # ... process the file  
end
```

Files



El método `sysread` lee un `String` desde un archivo

El método `syswrite` escribe en un archivo abierto para escritura

El método `each_byte` debe estar asociado a un bloque, y lee un archivo byte a byte

El método de clase `read` retorna un `String` con el contenido completo del archivo

```
f = File.new('testfile', 'a+')
f.syswrite("Una primer fila de prueba\n")

f.sysread(5)  => in `sysread': end of file reached
(EOFError)
```

Files

Ejemplos

```
f = File.new('testfile', 'r')    # o r+

f.sysread(5)                      #=> "Una p"
f.sysread(5)                      #=> "rimer"
f.sysread(25)                     #=> " fila de prueba\nUna prim"

f2 = File.new('testfile', 'r')
f.sysread(5)                      #=> "er fi"
f2.sysread(5)                     #=> "Una p"

buff = String.new
t = f2.sysread(3, buff)           #=> t y buff valen "rim"
```

Files

```
f = File.new('example.txt', 'w')
f.puts("Argentina")
f.puts("Uruguay")
f.close
```

```
p File.size?( 'example.txt' )    # => 18 o 20
```

```
File.new('example.txt', 'r').each_byte {
  |ch| puts ch; puts ?-}
```

A-r-g-e-n-t-i-n-a-
-U-r-u-g-u-a-y-
-

```
v = IO.readlines('example.txt')
# v = ["Argentina\n", "Uruguay\n"]
```

```
IO.foreach('example.txt') { |line| puts line }
```

Argentina
Uruguay

Files



```
# Manipulación de archivos
File.rename('old_name', 'new_name')
File.delete('file_name')
file = File.new('file_name', 'w')
file.chmod(0755)
File.open('file.rb') if File::exists?( 'file.rb' )
File::directory?( '/usr/local/bin' )
File.file?( 'file_name' )

f = File.new('example.txt', 'r')
f.sysseek("Argentina\n".size + 1, IO::SEEK_SET)
p f.sysread(15)           # => "Uruguay\n"
```


Serialización de objetos

YAML: *YAML Ain't Markup Language* (en un principio *Yet Another Markup Language*)

```
require 'yaml'

date1 = Date.new(2018, 5, 23)

serialized_object = YAML.dump(date1)
p serialized_object    # => "--- 2018-05-23\n...\n"

date2 = YAML.load(serialized_object)

p date2                # => #<Date: 2018-05-23 ((2458262j,0s,0n),+0s,2299161j)>
p date2.class          # => Date

puts date2.eql?(date1)  # => true
puts date2 == date1     # => true
```

Cargar datos con YAML sólo si la fuente es confiable

Serialización de objetos

```
class Container
  def initialize(obj1, obj2)
    @obj1 = obj1
    @obj2 = obj2
  end
end

var1 = Container.new(Date.new(2000, 1, 1), [1, 2, 3, 4])
serialized_object = YAML.dump(var1)

puts serialized_object.inspect
# "--- !ruby/object:Container\nobj1: 2000-01-01\nobj2:\n- 1\n- 2\n- 3\n- 4\n"
```

Files: JSON

JSON (JavaScript Object Notation) es un formato de texto para el intercambio de datos

Se basa en dos estructuras

- Una colección de pares nombre/valor (tabla de hash)
- Una lista ordenada de valores

Para serializar clases propias se puede extender el método `to_json`

```
require 'json'
require 'time'

json_string = JSON.generate Time.new
puts json_string # => # "2018-03-08 10:06:26 -0300"

time = JSON.parse(json_string)
p time # "2018-03-08 10:06:26 -0300"
p time.class # => String

t2 = Time.parse(time)

p t2.inspect # "2018-03-08 10:06:26 -0300"
p t2.class # => Time
```

JSON no brinda soporte para Time o clases arbitrarias.
Object#to_json es lo mismo que #to_s.to_json

Files: JSON

Para que JSON soporte una clase se deben extender los métodos `to_json` y `json_create`

```
range = (1..10)
range_json = JSON.generate range
p range_json      # => "\"1..10\""
r = JSON.parse(range_json)
p r.inspect       # => "\"1..10\""
```

```
class Range
  def to_json(*a)
    {
      'json_class' => self.class.name,    # = 'Range'
      'data'       => [ first, last, exclude_end? ]
    }.to_json(*a)
  end

  def self.json_create(o)
    new(*o['data'])
  end
end
```

Files: JSON

Ahora se puede serializar y deserializar una instancia de Range con JSON

El método `JSON#parse` además del string a “parsear” recibe un hash de opciones

```
range_json = JSON.generate range
p range_json      # => "{\"json_class\":\"Range\",\"data\":[1,10,false]}"
r = JSON.parse(range_json)
p r.inspect       # => "{\"json_class\"=>\"Range\", \"data\"=>[1, 10, false]}"
p r.class         # => Hash
```

```
range_json = JSON.generate range
p range_json      # => "{\"json_class\":\"Range\",\"data\":[1,10,false]}"
r = JSON.parse(range_json, :create_additions => true)
p r.inspect       # => "1..10"
p r.class         # => Range
```

Files: JSON

```
class Container
  def initialize(obj1, obj2)
    @obj1 = obj1
    @obj2 = obj2
  end

  def to_s
    "Contains:\n    #{@obj1.class} : #{@obj1} , #{@obj2.class} : #{@obj2} \n"
  end

  def to_json(*a)
    {
      "json_class" => self.class.name,
      "data"       => {"obj1" => @obj1, "obj2" => @obj2 }
    }.to_json(*a)
  end

  def self.json_create(o)
    new(o["data"]["obj1"], o["data"]["obj2"])
  end
end
```

Files: JSON

```
a = Container.new(Container.new('hello', 5), Container.new(1.5, 'Bye'))
```

```
json_string = a.to_json
```

```
puts json_string
```



```
{"json_class":"Container","data":{"obj1":{"json_class":"Container","data":{"obj1":"hello","obj2":5}}, "obj2":{"json_class":"Container","data":{"obj1":1.5,"obj2":"Bye"}}}}
```

```
puts JSON.parse(json_string, :create_additions => true)
```



```
Contains:  
  Container: Contains:  
    String: hello world , Integer: 5  
  , Container: Contains:  
    Float: 1.5 , String: Bye
```

Files: JSON

```
file = File.new('containers.json', 'w')

a = Container.new(Container.new('hello world', 5), Container.new(1.5, 'Bye'))
json_string = a.to_json

file.puts(json_string)
file.puts((Container.new('Key', (1..20))).to_json)
file.close

v = []
IO.foreach( 'containers.json' ) { |line|
  v << JSON.parse(line, :create_additions => true)
}
```


Unit Testing



Unit testing permite encontrar errores en forma temprana en el proceso de programación

Para que un *test unit* tenga calidad suficiente debe ser

- Automatizable
- Completo
- Reutilizable
- Independientes
- Profesionales: las pruebas deben tener la misma consideración que el código

Como cada unidad posee su “testeo” automatizado

- No es necesario ejecutar el programa completo y “jugar” con él
- No se crea un error al solucionar otro
- Permite “integración continua”
- Permite *Test Driven Development*: desarrollo guiado por pruebas

Ruby contiene en su librería estándar un “*framework*” para crear, organizar y correr pruebas llamado `Test::Unit`

Unit Testing: primer ejemplo

```
require 'test/unit'
```

```
class StringTest <  
  Test::Unit::TestCase
```

```
  def test_length  
    s = 'Hello, World!'  
    assert_equal(13, s.length)  
  end
```

```
end
```

```
require 'test/unit'
```

```
include Test::Unit  
class StringTest < TestCase
```

```
  def test_length  
    s = 'Hello, World!'  
    assert_equal(13, s.length)  
  end
```

```
end
```

1 tests, 1 assertions, 0 failures, 0 errors, 0 pendings, 0 omissions, 0 notifications

Test suite finished: 0.001393 seconds

Unit Testing

```
class NumberTest < TestCase
```

```
  def test_number
```

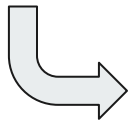
```
    assert_equal(25, 5 ** 2, 'El cuadrado de 5 es 25')
```

```
    assert_equal(25, 5 ** 5, '5 ** 5 es 25')
```

```
    assert_equal(5, 25.div(5), '25/5 = 5')
```

```
  end
```

```
end
```



No se ejecuta



Cada método test_
un único assert

5 ** 5 es 25.

<25> expected but was <3125>.

1 tests, 2 assertions, 1 failures, 0 errors, 0 pendings, 0 omissions, 0 notifications

Unit Testing negativo



```
# Verificamos que se lance la excepción  
def test_zero_division  
  assert_raise(ZeroDivisionError) do  
    n = 3 / 0  
  end  
end
```

```
def test_ok  
  assert nothing raised do  
    # codigo que no debería lanzar una excepción  
  end  
  
end
```

Unit Testing: Test suite




Si debemos testear muchos programas, podemos crear un test por cada uno. Para no tener que ejecutarlos uno x uno definimos un “Test Suite”

Si debemos testear muchos programas, podemos crear un test por cada uno. Para no tener que ejecutarlos uno x uno definimos un “Test Suite”

```
require 'test/unit'  
require 'test_library'  
require 'test_front'  
require 'test_anotherset'
```

Al incluir ‘test/unit’ Ruby se encarga de ejecutar los métodos test_ da cada una de las clases que extiendan TestCase, para cada uno de los módulos que comiencen con ‘test’. Incluso cada programa incluido puede ser un “Test suite”

Unit Testing: ejemplos



```
assert_empty("")           # -> pass
assert_empty([])          # -> pass
assert_empty({})          # -> pass
assert_empty(" ")         # -> fail
assert_empty([nil])       # -> fail
assert_compare(1, "<", 10)  # -> pass
assert_compare(1, ">=", 10) # -> fail
assert_boolean(true)      # -> pass
assert_boolean(nil)       # -> fail
assert_false(false)      # -> pass
assert_false(nil)         # -> fail
assert_include([1, 10], 1) # -> pass
assert_include(1..10, 5)   # -> pass
assert_include([1, 10], 5) # -> fail
assert_include(1..10, 20)  # -> fail
```

Unit Testing: ejemplos



```
assert_not_empty( " ")           # -> pass
assert_not_empty([ nil ])        # -> pass
assert_not_empty({ 1 => 2 })      # -> pass
assert_not_empty( "" )          # -> fail
assert_not_empty( [] )          # -> fail
assert_not_empty( {} )          # -> fail

assert_not_include([ 1, 10 ], 5)  # -> pass
assert_not_include( 1..10, 20 )  # -> pass
assert_not_include([ 1, 10 ], 1)  # -> fail
assert_not_include( 1..10, 5 )   # -> fail

assert_raise_message( "exception" ) { raise "exception" } # -> pass
assert_raise_message( /exc/i ) { raise "exception" }      # -> pass
assert_raise_message( "exception" ) { raise "EXCEPTION" } # -> fail
assert_raise_message( "exception" ) {}                    # -> fail
```

Etc



Ruby tiene mucho más para ofrecer que los temas vistos:

- Expresiones regulares
- Ruby Tk
- Acceso a bases de datos
- Multithreading
- Variables predefinidas
- Web services
- etc

Para más información ver <https://www.tutorialspoint.com/ruby/index.htm>