Ruby Tutorial

...o como pasar un buen rato programando

Lección 1

- Introducción
- Instalación
- El Primer Programa
- Números en Ruby
- Strings y diversión
- Variables
- Alcance de las variables

Lección 2

- Introduciendo Datos
- Normas en los nombres
- Los métodos
- Los métodos: argumentos
- Rangos
- Arrays

Lección 3

- Bloques
- Más malabares con strings
- Expresiones Regulares
- Condicionales
- Bucles
- Números Aleatorios

Lección 4

- Clases y Objetos
- Accesores
- Ficheros: lectura/escritura
- Cargando librerías
- Herencia de clases
- Modificando clases
- Congelando objetos
- Serializando objetos

Lección 5

- Control de acceso
- Excepciones
- Módulos
- Constantes
- Hashes y Símbolos
- La clase Time

Lección 6

- self
- Duck Typing
- Azúcar Sintáctico
- Test de unidades

contacto

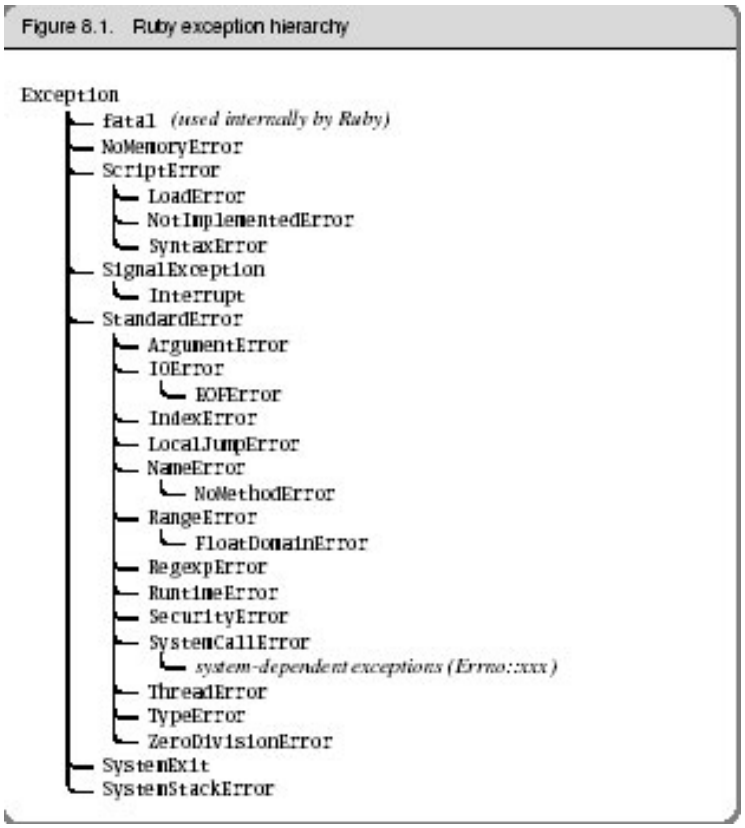
e-mail

Excepciones

Lanzando una excepción

Una **excepción** es un tipo especial de objeto de la clase **`Excepticon`**. Lanzar una excepción significa parar la ejecución de un programa para salir del mismo o para tratar con el problema que se ha producido. Para tratar el problema hace falta **`raise`**; de no ser así, el programa termina y avisa del error. Lo que hará raise (lanzar), será lanzar una "excepción" para manejar el error.

Ruby tiene una serie de clases, **`Exception`** y sus hijas, que ayudan a manejar los errores que pueden ocurrir. La siguiente figura enseña la jerarquía en Ruby:



Nota: la figura anterior es sacada del libro *Programming Ruby*.

```
def lanzar_excepcion
  puts 'Estoy antes del raise'
  raise 'Se ha producido un error' # lanza una excepción con el mensaje entre ''
  puts 'Estoy despues del raise'
end

lanzar_excepcion
```

El método raise procede del módulo Kernel. Por defecto, **`raise`** crea una excepción de la clase **`RuntimeError`**. Para lanzar una excepción de una clase específica, se puede poner el nombre de la clase como argumento de raise.

```
def inverse(x)
  raise ArgumentError, 'El argumento no es numerico' unless x.is_a? Numeric
  1.0 / x
end
puts inverse(2)
puts inverse('patata') # da un error que es manejado por raise
```

Hay que recordar que los métodos que actúan como preguntas, se les pone un **`?`** al final: **`is_a?`** pregunta al objeto cuál es su tipo. Y **`unless`** cuando se pone al final de una instrucción, significa que NO se ejecuta cuando la expresión a continuación es verdadera.

Manejando una excepción

Para tratar una excepción, se pone el método que puede causar el error dentro de un bloque **`begin…end`**. Dentro de este bloque, se pueden poner varios `rescue` para cada tipo de error que pueda surgir:

```
def raise_and_rescue
  begin
    puts 'Estoy antes del raise'
    raise 'Un error ha ocurrido' # simulamos un error
    puts 'Estoy después del raise'
  rescue
    puts 'Estoy rescatado del error.'
  end
  puts 'Estoy después del bloque'
end

raise_and_rescue
```

La salida es:

```
Estoy antes del raise
Estoy rescatado del error.
```

```
Estoy después del bloque
```

Observar que el código interrumpido por la excepción, nunca se ejecuta. Una vez que la excepción es manejada (por el rescue), la ejecución continúa inmediatamente después del bloque `begin` fuente del error.

Al escribir `rescue` sin parámetros, el parámetro `StandardError` se toma por defecto. En cada rescue se pueden poner varias excepciones a tratar. En el caso de poner múltiples rescues:

```
begin
  #
rescue UnTipoDeExcepcion
  #
rescue OtroTipoDeExcepcion
  #
else
  # Otras excepciones
end
```

Ruby compara la excepción que produce el error, con cada rescue hasta que sea del mismo tipo; o sea una superclase de la excepción. Si la excepción no concuerda con ningún rescue, usar **`else`** se encarga de manejarla.

Para saber acerca del tipo de excepción, hay que mapear el objeto `Exception` a una variable usando `rescue`:

```
begin
  raise 'Test de excepcion'
rescue Exception => e
  puts e.message            # Test de excepción
  puts e.backtrace.inspect   # ["nombre de fichero:linea de la excepción"]
end
```

Si además de manejar la excepción, se necesita que se ejecute un código, se usará la instrucción **`ensure`**: lo que haya en ese bloque, siempre se ejecutará cuando el bloque `begin`…`end` termine.

Excepciones más comunes

He aquí algunas excepciones más comunes, con la causa que las origina y un ejemplo:

- **`RuntimeError`** - la excepción que se lanza por defecto. Ejemplo:

```
raise
```

- **`NoMethodError`** - el objeto no puede manejar el mensaje/método que se le envía. Ej:

```
string = 'patata'
string.multiplicarse
```

- **`NameError`** - el intérprete encuentra un identificador que no puede resolver ni como método, ni como variable. Ej:

```
a = variable_sin_definir
```

- **`IOError`** - lectura de un *stream* cerrado, escribir a un sistema de sólo lectura y operaciones parecidas. Ej:

```
STDIN.puts("No escribas a STDIN!")
```

- **`Errno::`***`error`* - errores relaccionado con el fichero IO. Ej:

```
File.open(-12)
```

- **`TypeError`** - un método recibe un argumento que no puede manejar. Ej:

```
a = 3 + "no puedo sumar un string a un número!"
```

- **`ArgumentError`** - causado por un número incorrecto de argumentos. Ej:

```
def m(x)
end
m(1,2,3,4,5)
```

Ejemplo

```
begin
  # Abre el fichero y lo lee
  File.open('origen.txt', 'r') do |f1|
    while line = f1.gets
      puts line
    end
  end

  # Crea un nuevo fichero y escribe en él
  File.open('destino.txt', 'w') do |f2|
    f2.puts "Creado por Satish"
  end
rescue Exception => msg
  # dispone el mensaje de error
  puts msg
end
```

page_revision: 3, last_edited: 26 Jul 2009, 18:17 GMT-05 (491 days ago)

Other interesting sites

H A R I T' S

Have fun!!

Mikity

Iatkos

Geo Spatial Data Wiki