Práctica 3: Excepciones, gemas y Bundler

Taller de Tecnologías de Producción de Software - Ruby

Cursada 2023

En esta tercera práctica del taller trataremos en mayor profundidad las excepciones como herramienta para el control del flujo de un programa en Ruby, e incorporaremos las librerías reutilizables que el lenguaje provee como elementos de primer nivel del mismo.

Excepciones

1. Investigá la jerarquía de clases que presenta Ruby para las excepciones. ¿Para qué se utilizan las siguientes clases?

   - `ArgumentError`
   - `IOError`
   - `NameError`
   - `NotImplementedError`
   - `RuntimeError`
   - `StandardError`
   - `StopIteration`
   - `SystemExit`
   - `SystemStackError`
   - `TypeError`
   - `ZeroDivisionError`

2. ¿Cuál es la diferencia entre `raise` y **`throw`**? ¿Para qué usarías una u otra opción?
3. ¿Para qué sirven `begin .. rescue .. else` y `ensure`? Pensá al menos 2 casos concretos en que usarías estas sentencias en un script Ruby.
4. ¿Para qué sirve `retry`? ¿Cómo podés evitar caer en un loop infinito al usarla?
5. ¿Para qué sirve `redo`? ¿Qué diferencias principales tiene con `retry`?
6. Analizá y probá los siguientes métodos, que presentan una lógica similar, pero ubican el manejo de excepciones en distintas partes del código. ¿Qué resultado se obtiene en cada caso? ¿Por qué?

```ruby
def opcion_1
  a = [1, nil, 3, nil, 5, nil, 7, nil, 9, nil]
  b = 3
  c = a.map do |x|
    x * b
  end
  puts c.inspect
rescue
  0
end

def opcion_2
  c = begin
        a = [1, nil, 3, nil, 5, nil, 7, nil, 9, nil]
        b = 3
        a.map do |x|
          x * b
        end
      rescue
        0
      end
  puts c.inspect
end

def opcion_3
  a = [1, nil, 3, nil, 5, nil, 7, nil, 9, nil]
  b = 3
  c = a.map { |x| x * b } rescue 0
  puts c.inspect
end

def opcion_4
  a = [1, nil, 3, nil, 5, nil, 7, nil, 9, nil]
  b = 3
  c = a.map { |x| x * b rescue 0 }
  puts c.inspect
end
```

7. Suponé que tenés el siguiente script y se te pide que lo hagas *resiliente* (tolerante a fallos), intentando siempre que se pueda recuperar la situación y volver a intentar la operación que falló. Realizá las modificaciones que consideres necesarias para lograr que este script sea más robusto.

```ruby
# Este script lee una secuencia de no menos de 15 números desde
    teclado y luego imprime el resultado de la división
# de cada número por su entero inmediato anterior.

# Como primer paso se pide al usuario que indique la cantidad de nú
    meros que ingresará.
cantidad = 0
while cantidad < 15
  puts 'Cuál es la cantidad de números que ingresará? Debe ser al
      menos 15'
  cantidad = Integer(gets)
end

# Luego se almacenan los números
numeros = 1.upto(cantidad).map do
  puts 'Ingrese un número'
  numero = Integer(gets)
end

# Y finalmente se imprime cada número dividido por su número entero
    inmediato anterior
resultado = numeros.map { |x| x / (x - 1) }
puts 'El resultado es: %s' % resultado.join(', ')
```

8. Partiendo del script que modificaste en el inciso anterior, implementá una nueva clase de excepción que se utilice para indicar que la entrada del usuario no es un valor numérico entero válido. ¿De qué clase de la jerarquía de `Exception` heredaría?

9. Dado el siguiente código:

```ruby
def fun3
  puts "Entrando a fun3"
  raise RuntimeError, "Excepción intencional"
  puts "Terminando fun3"
rescue NoMethodError => e
  puts "Tratando excepción por falta de método"
rescue RuntimeError => e
  puts "Tratando excepción provocada en tiempo de ejecución"
rescue
  puts "Tratando una excepción cualquiera"
ensure
  puts "Ejecutando ensure de fun3"
end

def fun2(x)
  puts "Entrando a fun2"
  fun3
  a = 5 / x
  puts "Terminando fun2"
end

def fun1(x)
  puts "Entrando a fun1"
  fun2 x
rescue
  puts "Manejador de excepciones de fun1"
  raise
ensure
  puts "Ejecutando ensure de fun1"
end

begin
  x = 0
  begin
    fun1 x
  rescue Exception => e
    puts "Manejador de excepciones de Main"
    if x == 0
      puts "Corrección de x"
      x = 1
      retry
    end
  end
  puts "Salida"
end
```

1. Seguí el flujo de ejecución registrando la traza de impresiones que deja el programa, analizando por qué partes del código va pasando y justificando esos pasos.
2. ¿Qué pasaría si se permuta, dentro de `fun3`, el manejador de excepciones para `RuntimeError` y el manejador de excepciones genérico (el que tiene el `rescue`

vacío)?

3. ¿El uso de `retry` afectaría el funcionamiento del programa si se mueve la línea `x = 0` dentro del segundo `begin` (inmediatamente antes de llamar a `fun1` con `x`)?

10. Implementá un método que reciba dos parámetros: un `String` que representa la ruta a un archivo y un bloque. El método deberá abrir el archivo indicado, leer su contenido y procesar cada linea del mismo con el bloque recibido, imprimiendo a la salida estándar el resultado de cada invocación al bloque.

    Considerá qué posibles excepciones pueden ocurrir, intentá manejarlas, y en caso que la ejecución del bloque falle para alguna de las líneas, en el lugar que iría la salida de la ejecución fallida deberá imprimirse `Error encontrado: mensaje de error (ClaseDelError)`. Finalmente, el método deberá retornar la cantidad de líneas procesadas correctamente.

    Por ejemplo, suponiendo un archivo `/tmp/datos.txt` que tiene el siguiente contenido:

```
2
ruby
5
7
perro
13

vaca
```

    Y suponiendo la siguiente ejecución del método:

```ruby
procesar_archivo('/tmp/datos.txt') { |linea| Integer(linea) }
```

    Su salida esperada sería:

```
2
Error encontrado: invalid value for Integer(): "ruby" (ArgumentError)
5
7
Error encontrado: invalid value for Integer(): "perro" (ArgumentError)
13
Error encontrado: invalid value for Integer(): "" (ArgumentError)
Error encontrado: invalid value for Integer(): "vaca" (ArgumentError)
```

    Y su valor de retorno esperado sería 4.

Librerías reutilizables en Ruby (Gemas) y Bundler

11. ¿Qué es una *gema*? ¿Para qué sirve? ¿Qué estructura general suele tener?

12. ¿Cuáles son las principales diferencias entre el comando `gem` y Bundler? ¿Hacen lo mismo?
13. ¿Dónde almacenan las gemas que se instalan con el comando `gem`? ¿Y aquellas instaladas con el comando `bundle`?

    > Tip: `gem which` y `bundle show`.

14. ¿Para qué sirve el comando `gem server`? ¿Qué información podés obtener al usarlo?
15. Investigá un poco sobre *Semantic Versioning* (o *SemVer*). ¿Qué finalidad tiene? ¿Cómo se compone una versión? ¿Ante qué situaciones debería cambiarse cada una de sus partes?
16. Creá un proyecto para probar el uso de Bundler:
    1. Inicializá un proyecto nuevo en un directorio vacío con el comando `bundle init`.
    2. Modificá el archivo `Gemfile` que generaste con el comando anterior y agregá ahí la gema `colorputs`.
    3. Creá el archivo `prueba.rb` y agregale el siguiente contenido:

       ```
       require 'colorputs'

       puts "Hola!", :rainbow_bl
       ```

    4. Ejecutá el archivo anterior de las siguientes maneras:
       - `ruby prueba.rb`
       - `bundle exec ruby prueba.rb`
    5. Ahora utilizá el comando `bundle install` para instalar las dependencias del proyecto.
    6. Volvé a ejecutar el archivo de las dos maneras enunciadas en el paso **4**.
    7. Creá un nuevo archivo `prueba_dos.rb` con el siguiente contenido:

       ```
       Bundler.require

       puts "Chau!", :red
       ```

    8. Ahora ejecutá este nuevo archivo:
       - `ruby prueba_dos.rb`
       - `bundle exec ruby prueba_dos.rb`
17. Utilizando el proyecto creado en el punto anterior como referencia, contestá las siguientes preguntas:
    1. ¿Qué finalidad tiene el archivo `Gemfile`?

2. ¿Para qué sirve la directiva `source` del `Gemfile`? ¿Cuántas veces puede estar en un mismo archivo?
3. Acorde a cómo agregaste la gema `colorputs`, ¿qué versión se instaló de la misma? Si mañana se publicara la versión `7.3.2`, ¿esta se instalaría en tu proyecto? ¿Por qué? ¿Cómo podrías limitar esto y hacer que sólo se instalen *releases* de la gema en las que no cambie la *versión mayor* de la misma con respecto a la que tenés instalada ahora?
4. ¿Qué ocurrió la primera vez que ejecutaste `prueba.rb`? ¿Por qué?
5. ¿Qué cambió al ejecutar `bundle install`?
6. ¿Qué diferencia hay entre `bundle install` y `bundle update`?
7. ¿Qué ocurrió al ejecutar `prueba_dos.rb` de las distintas formas enunciadas? ¿Por qué? ¿Cómo modificarías el archivo `prueba_dos.rb` para que funcione correctamente sin importar de cuál de las dos maneras indicadas es ejecutado?

18. Desarrollá una gema que empaquete toda la funcionalidad implementada en el módulo `Countable` que desarrollaste en la práctica anterior. Llamala con algún nombre distintivo y único. Por ejemplo, podrías llamarla `ruby_2023`.

    La forma de usarla sería algo similar a esto:

```ruby
require 'ruby_2023'

class MiClase
  include Ruby2023::Countable

  def hola
    puts "Hola"
  end

  def chau
    puts "Chau"
  end

  count_invocations_of :hola, :chau
end
```

> Tip: investigá el uso del comando `bundle gem`.

19. Si te animás, versioná tu gema en un repositorio Git público y publicala en RubyGems, el hosting público de gemas gratuito por excelencia de Ruby. Luego de hacerlo, agregá la gema como una dependencia en el `Gemfile` del proyecto de prueba de Bundler que hiciste en esta misma práctica, y utilizá tu gema para contar las invocaciones del método que quieras.

    1. ¿Qué pasos tuviste que realizar para publicar la gema?
    2. ¿Cómo podés cambiar la versión de tu gema y publicarla?

3. ¿Cómo incluiste tu gema en el proyecto de prueba?

> ¡No dejes de enviar un mensaje al foro compartiendo el link a la gema y tu experiencia!

Referencias

A la hora de aprender un nuevo lenguaje, una herramienta o un *framework*, es fundamental que te familiarices con sus APIs. Ya sea para conocer clases base del lenguaje o parte de la herramienta que estés comenzado a utilizar, las APIs que te provea serán la forma de sacarle provecho.

Por eso, te dejamos en esta sección algunos links para que puedas consultar la documentación de las herramientas que tratamos en esta práctica:

- RubyGems - https://rubygems.org
  - Guías
- Bundler - https://bundler.io
  - Motivación y breve ejemplo
  - Gemfile