Práctica 3

1. Investigá la jerarquía de clases que presenta Ruby para las excepciones. ¿Para qué se utilizan las siguientes clases?
   - ArgumentError: Se genera cuando los argumentos son incorrectos y no existe una clase de excepción más específica.
   - IOError: Se genera cuando falla una operación de E/S.
   - NameError: Se genera cuando un nombre dado es inválido o no está definido.
   - NotImplementedError: Se genera cuando una función no está implementada en la plataforma actual. Por ejemplo, métodos que dependen de fsync o fork pueden lanzar esta excepción si el sistema operativo o el runtime de Ruby no los soporta.
   - RuntimeError: Excepción por defecto que se lanza cuando se realiza una operación inválida. Es la que raise usa por defecto si no se especifica otra.
   - StandardError: Base común para la mayoría de errores “recuperables”; es lo que rescue captura sin clase. Los tipos de error más comunes son subclases de StandardError. Una cláusula rescue sin una clase Exception explícita capturará todos los StandardError (y solo esos).
   - StopIteration: Se genera para detener la iteración, en particular por Enumerator#next. Es capturada por Kernel#loop.
   - SystemExit: Se genera por exit para iniciar la terminación del script.
   - SystemStackError: Se genera en caso de desbordamiento de pila.
   - TypeError: Se genera al encontrar un objeto que no es del tipo esperado.
   - ZeroDivisionError: Se genera al intentar dividir un entero por 0.

2. ¿Cuál es la diferencia entre raise y throw? ¿Para qué usarías una u otra opción?

   Raise lanza una excepción que se maneja con rescue. Es para errores/condiciones anómalas. Se debe usar raise cuando algo salió mal o una precondición no se cumple y debes señalar un error que puede (o no) ser rescatado.
   Throw realiza una salida no local del flujo hasta el catch con la misma etiqueta. No es una excepción y no lo captura rescue (sí se ejecutan ensure al deshacer la pila). Se debe usar throw/catch para terminar antes un flujo normal y profundo (p. ej.,

salir de bucles/metodos anidados) cuando no es un error, sino un “no hace falta seguir”. Para un solo nivel, prefiere break/return.

3. ¿Para qué sirven begin .. rescue .. else y ensure? Pensá al menos 2 casos concretos en que usarías estas sentencias en un script Ruby
   - begin: Delimita el bloque “riesgoso”.
   - rescue: Captura y maneja excepciones.
   - else: Se ejecuta solo si NO hubo excepción.
   - ensure: Se ejecuta siempre (haya o no excepción)

```ruby
file = nil
begin
  file = File.open("config.yml", "r")
  data = file.read
  parsed = YAML.safe_load(data)  # puede fallar
rescue Errno::ENOENT
  puts "Archivo no encontrado, usando config por defecto"
  parsed = {}
rescue Psych::SyntaxError => e
  puts "YAML inválido: #{e.message}"
  parsed = {}
else
  puts "Configuración cargada correctamente"
ensure
  file&.close
end
```

```ruby
require "net/http"
response_body = nil

begin
  uri = URI("https://api.ejemplo.com/items")
  res = Net::HTTP.get_response(uri)
```

```ruby
    raise "HTTP #{res.code}" unless res.is_a?(Net::HTTPSuccess)
    response_body = JSON.parse(res.body)
  rescue JSON::ParserError
    puts "Respuesta no es JSON válido"
  rescue => e
    puts "Fallo en la solicitud: #{e.message}"
  else
    puts "Datos obtenidos: #{response_body.size} items"
  ensure
    puts "Fin de la operación (log, métricas, cleanup)"
  end
```

4. ¿Para qué sirve retry? ¿Cómo podés evitar caer en un loop infinito al usarla?
   Permite reintentar el bloque begin después de un rescue (vuelve a ejecutar desde el inicio del begin). Útil ante fallos transitorios (E/S, red, locks). Para evitar loops infinitos es recomendable limitar intentos con un contador, cambiar condiciones antes de reintentar (sleep/backoff, refrescar token, cambiar endpoint) o aplicar timeouts/circuit breaker

5. ¿Para qué sirve redo? ¿Qué diferencias principales tiene con retry?
   La sentencia Redo se usa para repetir la iteración actual del bucle. Redo siempre se usa dentro del bucle. Reinicia el bucle sin volver a evaluar la condición.
   Diferencias con retry:
   - Ambito:
     - redo: bucles/iteraciones; repite la iteración actual.
     - retry: en rescue; reinicia el bloque begin completo.
   - Flujo:
     - redo no evalúa de nuevo la condición del bucle ni avanza el iterador.
     - retry vuelve al comienzo del begin (reintenta toda la operación).

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?

```
Ejecutando opcion 1:
Ejecutando opcion 2:
nil
Ejecutando opcion 3:
0
Ejecutando opcion 4:
[3, 0, 9, 0, 15, 0, 21, 0, 27, 0]
```

- Opción 1:
    - El rescue está afuera del bloque, cubriendo todo el método. Cuando Ruby hace nil * 3, se lanza un error dentro de map que no se maneja ahí y sino que se sube hasta el nivel del método y es capturado por el rescue general. Por eso el método no imprime nada, ya que la excepción ocurre antes del puts. El bloque rescue devuelve 0
- Opción 2:
    - El rescue envuelve el bloque begin ... end. El error (nil * 3) ocurre dentro de ese bloque, y se captura ahí. Como el rescue no tiene valor de retorno explícito, devuelve nil haciendo que c = nil.
- Opción 3:
    - El rescue afecta toda la expresión a.map { |x| x * b }. Si cualquier excepción ocurre en todo el mapeo, se reemplaza todo el resultado por 0. La excepción sucede al intentar el primer nil * 3, entonces todo el map falla haciendo que c = 0.
- Opción 4:
    - Ahora el rescue está dentro del bloque del map. Eso significa que cada elemento individual maneja su propia excepción. Cuando x es un número se multiplica y cuando es nil se lanza error, que es capturado dentro del bloque y devuelve 0 solo para ese elemento.

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 quefalló. Realizá las modificaciones que consideres necesarias para lograr que este script sea más robusto.

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?