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 los 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

require "net/http"
response_body = nil

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

```ruby
  res = Net::HTTP.get_response(uri)
  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?

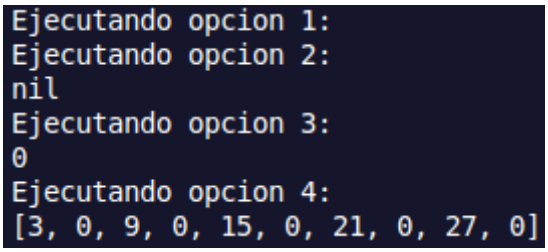
   Permite reintentar el bloque begin después de un rescue (vuelve a ejecutar desde el inicio del begin). Útil ante fallos transitorios (E/S, red, locks). Para evitar loops infinitos es recomendable limitar intentos con un contador, cambiar condiciones antes de reintentar (sleep/backoff, refrescar token, 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:

   - Ámbito:
     - 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?



- 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?

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

   Es un paquete de código reutilizable que contiene una biblioteca, herramienta o funcionalidad que puede ser instalada y utilizada en distintos proyectos. Permite distribuir y compartir código Ruby de manera sencilla a través del ecosistema RubyGems, que es el sistema de gestión de paquetes oficial de Ruby.
   Una gema suele tener la siguiente estructura básica:

```
mi_gema/
├── lib/
│   ├── mi_gema.rb           # archivo principal (require base)
│   └── mi_gema/
│       └── version.rb       # define la versión de la gema
├── mi_gema.gemspec          # metadatos de la gema
├── README.md                # documentación
├── Gemfile (opcional)
├── LICENSE.txt
└── Rakefile (opcional)
```

   mi_gema.gemspec es el archivo más importante ya que define el nombre, versión, autor, resumen y archivos incluidos en la gema.

```ruby
Gem::Specification.new do |spec|
  spec.name          = "mi_gema"
  spec.version       = "0.1.0"
  spec.authors       = ["Agustina Sol Rojas"]
  spec.summary       = "Una gema de ejemplo"
  spec.description   = "Provee funcionalidades para demostrar cómo se estructura una gema Ruby."
  spec.files         = Dir["lib/**/*.rb"]
  spec.require_paths = ["lib"]
```

```
end
```

10. ¿Cuáles son las principales diferencias entre el comando gem y Bundler? ¿Hacen lo mismo?

    gem gestiona gemas individuales, mientras que Bundler gestiona colecciones de gemas y sus versiones dentro de un proyecto.
    gem es el comando del gestor de paquetes RubyGems, que viene instalado con Ruby. Este permite instalar, listar, actualizar y eliminar gemas global o localmente.
    Bundler es una herramienta que se apoya en RubyGems para gestionar las dependencias de un proyecto Ruby completo. Garantiza que todas las gemas de un proyecto (y sus versiones exactas) sean las correctas y compatibles entre sí.

11. ¿Dónde almacenan las gemas que se instalan con el comando gem? ¿Y aquellas instaladas con el comando bundle?

    Las gemas instaladas con gem se instalan globalmente (por defecto) en el directorio de gemas de RubyGems. Se puede ver la ubicación exacta con gem environment. Tambien se puede usar el comando gem which nombre_gema que muestra la ruta exacta del archivo principal de una gema.
    Cuando se instalan gemas mediante Bundler depende de cómo esté configurado el entorno. Si no se configuro nada especial, Bundler usa el mismo directorio global de RubyGems, es decir, las gemas van al mismo lugar que las de gem install. Si se ejecuta bundle install --path vendor/bundle entonces Bundler instala todas las gemas dentro del proyecto, en la carpeta vendor/bundle/. Se pueden ver dónde está instalada una gema específica con bundle show nombre_gema

12. ¿Para qué sirve el comando gem server? ¿Qué información podés obtener al usarlo?

    Permite levantar un servidor web local con información sobre todas las gemas que se tienen instaladas en el sistema. Por defecto, inicia un servidor en el puerto 8808 al cual se puede acceder a través de http://localhost:8808. gem server --port

4000 permite especificar otro puerto. Al usarlo se puede obtener lista de gemas, versiones, documentación RDoc/ri y dependencias

13. 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?

    Semantic Versioning es un estándar para asignar números de versión a proyectos de software de forma predecible y significativa. El objetivo es que el número de versión comunique claramente el tipo de cambios realizados en el código, de modo que otros desarrolladores sepan si una actualización puede romper compatibilidad o no. La especificación oficial es: https://semver.org/.
    Una versión SemVer tiene este formato MAJOR.MINOR.PATCH donde MAJOR se incrementa cuando hay cambios incompatibles con versiones anteriores, MINOR cuando se agregan nuevas funcionalidades compatibles, PATCH cuando se hacen correcciones de errores o pequeños ajustes sin agregar ni romper nada.

14. 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

15. Utilizando el proyecto creado en el punto anterior como referencia, contestá las siguientes preguntas:
    1. ¿Qué finalidad tiene el archivo Gemfile?

       El Gemfile indica qué gemas y versiones necesita un proyecto Ruby. Sirve para que todos los desarrolladores trabajen con las mismas dependencias.

    2. ¿Para qué sirve la directiva source del Gemfile? ¿Cuántas veces puede estar en un mismo archivo? Muchas veces si se quiere, identifica donde buscar las gemas.

       Define de dónde (qué repositorio) se descargarán las gemas. Normalmente es “source "https://rubygems.org" ”. Se pueden tener varias directivas source, aunque lo habitual es una sola. Bundler buscará gemas en ese(los) origen(es).

    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?

       Cuando no se especifica una versión Bundler instala la última versión disponible al momento del bundle install. Si mañana se publica una nueva versión, no se actualizará automáticamente en tu proyecto, porque el archivo Gemfile.lock fija la versión instalada, para actualizar se debe ejecutar bundle update colorputs. Si se quiere permitir actualizaciones solo dentro de la misma versión mayor, se puede limitar haciendo gem "colorputs", "~> 7.2"

    4. ¿Qué ocurrió la primera vez que ejecutaste prueba.rb? ¿Por qué?

       Falló con un error porque la gema aún no estaba instalada en el sistema ni en el entorno Bundler.

5. ¿Qué cambió al ejecutar bundle install?

   Se descargó e instaló la gema colorputs y se creó el archivo Gemfile.lock para registrar la versión exacta instalada.

6. ¿Qué diferencia hay entre bundle install y bundle update?

   bundle install instala las gemas en base a lo que figura en Gemfile.lock (sin cambiar versiones) mientras que bundle update actualiza las gemas a sus versiones más recientes (dentro de las restricciones del Gemfile) y reemplaza el Gemfile.lock.

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?

   Con ruby prueba_dos.rb falla, porque Bundler no está. Con bundle exec ruby prueba_dos.rb funciona, ya que Bundler ejecuta el script dentro del contexto de las gemas del Gemfile.
   Agregaría un require explícito al inicio del archivo:

```ruby
require 'bundler/setup'
Bundler.require

puts "Chau!", :red
```