

Práctica 9: Excepciones

Ejercicio 1: Explique claramente a qué se denomina excepción.

Condición inesperada o inusual, que ocurre durante la ejecución del programa y no puede ser manejada en el contexto local. Denota un comportamiento anómalo e indeseable que supuestamente ocurre raramente, pero es necesario controlarlo. Pero, suelen ocurrir con frecuencia.

Ejercicio 2: ¿Qué debería proveer un lenguaje para el manejo de las excepciones? ¿Todos los lenguajes lo proveen?

Para que un lenguaje trate excepciones debe proveer mínimamente:

- Un modo de definir las
- Una forma de reconocerlas
- Una forma de lanzarlas y capturarlas
- Una forma de manejarlas especificando el código y respuestas
- Un criterio de continuación

No todos los lenguajes de programación proporcionan un mecanismo completo para el manejo de excepciones. Algunos lenguajes más antiguos o de bajo nivel pueden tener un enfoque más limitado o no proporcionar un manejo de excepciones integrado. Sin embargo, la mayoría de los lenguajes modernos y populares, como Java, C++, Python y C#, sí brindan soporte completo para el manejo de excepciones. Estos lenguajes tienen palabras clave y sintaxis específicas para lanzar, capturar y manejar excepciones de manera efectiva.

Ejercicio 3: ¿Qué ocurre cuando un lenguaje no provee manejo de excepciones? ¿Se podría simular? Explique cómo lo haría.

Cuando un lenguaje de programación no proporciona un mecanismo integrado para el manejo de excepciones, puede ser más difícil manejar situaciones excepcionales o errores de manera adecuada. Sin un sistema de manejo de excepciones, los errores pueden propagarse y afectar el flujo normal del programa o, en el peor de los casos, provocar fallas inesperadas.

Si un lenguaje no ofrece soporte nativo para el manejo de excepciones, aún es posible simular un mecanismo de manejo de excepciones utilizando técnicas alternativas. Una forma común de hacerlo es mediante el uso de estructuras de control condicionales (por ejemplo, if-else) y la utilización de valores de retorno especiales para indicar errores.

Se podría simular de la siguiente manera:

1. Identificar situaciones excepcionales
2. Devolver valores de error (ya que no hay excepciones)
3. Validar valores de retorno (después de llamados a funciones o métodos)
4. Propagación manual de errores
5. Usar bloques try-catch simulados con if-else.

Ejercicio 4: Cuando se termina de manejar la excepción, la acción que se toma luego es importante. Indique

01. ¿Qué modelos diferentes existen en este aspecto?

Reasunción: se refiere a la posibilidad de retomar la ejecución normal del programa después de manejar una excepción. El controlador de excepciones realiza las acciones necesarias para manejar la excepción (medidas correctivas) y luego el programa continúa su ejecución a partir del punto donde se produjo la excepción. •

Terminación: el controlador de excepciones realiza las acciones necesarias para manejar la excepción, pero no se retorna al punto donde se produjo la excepción (invocador), continúa su ejecución a partir de la finalización del manejador.

02. Dé ejemplos de lenguajes que utilizan cada uno de los modelos presentados anteriormente. Por cada uno responda respecto de la forma en que trabaja las excepciones.

- a. ¿Cómo se define?
- b. ¿Cómo se lanza?
- c. ¿Cómo se maneja?
- d. ¿Cuál es su criterio de continuación?

Reasunción

- PL/1:

a. Las excepciones en PL/I se denominan CONDITIONS y se definen mediante la sentencia ON CONDITION. Pueden ser predefinidas (integradas, por ejemplo, ZERODIVIDE) o definidas por el usuario. La sintaxis es:

ON CONDITION(Nombre-excepción) Manejador

Los manejadores se vinculan dinámicamente a las excepciones, y el manejador definido más recientemente tiene prioridad (manejo basado en una pila de manejadores). El alcance de un manejador termina cuando finaliza la ejecución de la unidad donde fue declarado. Excepciones con el mismo nombre se enmascaran entre sí.

b. Las excepciones se lanzan explícitamente con la palabra clave SIGNAL seguida del nombre de la condición:

SIGNAL CONDITION(Nombre-excepción)

c. Los manejadores se declaran con la sentencia ON CONDITION, que especifica el código a ejecutar cuando ocurre la excepción. El manejador puede ser una sola instrucción o un bloque (entre begin y end). Por ejemplo:

```
ON CONDITION(PEPE) begin
  /* Código del manejador */
end;
```

d. Reasunción: Después de que el manejador procesa la excepción, la ejecución se reanuda en la instrucción inmediatamente siguiente a donde se lanzó la excepción. Esto permite que el programa continúe su flujo normal tras manejar la excepción.

Terminación

- ADA:

a. Las excepciones se declaran en la sección de declaración de variables de una unidad de programa (por ejemplo, subprograma, bloque, paquete) con la sintaxis:

MiExcepcion: exception;

Tienen el mismo alcance que las variables en esa unidad. Ada también proporciona excepciones predefinidas como Constraint_Error, Program_Error, Storage_Error, Tasking_Error y Name_Error.

b. Las excepciones se lanzan explícitamente con la palabra clave raise, tanto para excepciones definidas por el usuario como predefinidas:

raise MiExcepcion;

Las excepciones también pueden ser lanzadas implícitamente por el entorno de ejecución (por ejemplo, Constraint_Error por división por cero).

c. Los manejadores se definen en una sección exception al final de un bloque, subprograma o paquete, usando la palabra clave when:

begin

-- Código

exception

when MiExcepcion => -- Manejador para MiExcepcion

-- Código del manejador

when Constraint_Error => -- Manejador para Constraint_Error

-- Código del manejador

when others => -- Captura cualquier excepción no especificada

-- Código del manejador

end;

La cláusula when others captura cualquier excepción no manejada, pero debe ser el último manejador para evitar efectos secundarios. Si no se encuentra un manejador en la unidad actual, la excepción se propaga dinámicamente a la unidad que la llamó.

d. Terminación: Después de manejar una excepción, el bloque o unidad actual termina, y el control pasa a la instrucción siguiente al manejador. Si no se encuentra un manejador, la excepción se propaga por la pila de llamadas, pudiendo terminar el programa si no se maneja. El programa puede continuar normalmente, volver a un estado anterior, propagar la excepción más arriba o terminar, dependiendo de las acciones del manejador y el diseño del lenguaje.

- CLU:

a. Las excepciones se definen en el encabezado del procedimiento usando la palabra clave signals, especificando qué excepciones puede lanzar el procedimiento:

Procedure UNO() signals error1;

Las excepciones son explícitamente declaradas por el usuario, y el documento no menciona excepciones predefinidas.

b. Las excepciones se lanzan explícitamente con la palabra clave signal seguida del nombre de la excepción:

signal error1;

c. Los manejadores se definen usando una cláusula exception when dentro del procedimiento o programa principal:

exception when error1 -> y := y + 7; x := x + 2; resignal; end;

Los manejadores pueden realizar acciones correctivas y opcionalmente relanzar la excepción con resignal. Pueden existir múltiples manejadores en diferentes niveles (por ejemplo, dentro de procedimientos o en el programa principal), y la excepción se propaga si no se maneja completamente.

d. Terminación: Después de manejar una excepción, el bloque actual termina, y el control no regresa al punto donde se lanzó la excepción. En su lugar, la ejecución continúa después del manejador, o la excepción

se propaga más arriba (por ejemplo, con `resignal`) a un manejador de nivel superior, pudiendo terminar el programa si no se maneja.

- C++:

a. Las excepciones en C++ son objetos, típicamente instancias de una clase derivada de `std::exception` o clases personalizadas definidas por el programador. Ejemplo:

```
class MiExcepcion : public std::exception {  
public:  
    const char* what() const noexcept override { return "MiExcepcion"; }  
};
```

b. Las excepciones se lanzan con la palabra clave `throw` seguida de un objeto:

```
throw MiExcepcion();
```

c. Las excepciones se manejan usando bloques `try` y `catch`:

```
try {  
    // Código que podría lanzar una excepción  
} catch (MiExcepcion& e) {  
    // Manejar MiExcepcion  
} catch (std::exception& e) {  
    // Manejar otras excepciones estándar  
} catch (... ) {  
    // Capturar cualquier otra excepción  
}
```

Los manejadores se añaden al bloque donde se lanza la excepción, y el bloque `catch` especifica el tipo de excepción a manejar.

d. Terminación: Después de capturar y manejar una excepción, el bloque actual termina, y la ejecución continúa después del bloque `catch`. Si no se encuentra un manejador, la excepción se propaga por la pila de llamadas, pudiendo causar la terminación del programa con una llamada a `std::terminate()` si no se maneja.

- Java:

a. Las excepciones son objetos, instancias de la clase `Throwable` o sus subclases (`Error` y `Exception`). Los programadores pueden definir excepciones personalizadas extendiendo `Exception`:

```
class MiExcepcion extends Exception {  
    public MiExcepcion(String message) { super(message); }  
}
```

b. Las excepciones se lanzan con la palabra clave `throw`, y los métodos deben declarar excepciones comprobadas con `throws` en su firma:

```
public void metodo() throws MiExcepcion {  
  
    throw new MiExcepcion("Ocurrió un error");  
  
}
```

c. Las excepciones se manejan usando bloques `try`, `catch` y `finally`:

```
try {
    // Código que podría lanzar una excepción
} catch (ArithmeticException e) {
    // Manejar ArithmeticException
} catch (Exception e) {
    // Manejar otras excepciones
} finally {
    // Código que siempre se ejecuta
}
```

El bloque finally asegura la ejecución del código independientemente de si ocurre una excepción. Posee propagación dinámica (stack tree): una vez lanzadas, en caso de no ser tratadas por la unidad que las generó, se propagan dinámicamente. Las excepciones pueden propagarse a bloques try externos si no se manejan.

d. Terminación: Después de manejar una excepción, el bloque try actual termina, y la ejecución continúa después del bloque catch o finally. Si no se maneja, la excepción se propaga por la pila de llamadas, pudiendo terminar el programa si no se encuentra un manejador.

- Python:

a. Las excepciones son objetos, típicamente instancias de clases integradas como ValueError, TypeError o clases personalizadas derivadas de Exception:

```
class MiExcepcion(Exception):
```

```
    pass
```

b. Las excepciones se lanzan con la palabra clave raise:

```
raise MiExcepcion("Ocurrió un error")
```

Si sólo aparece la palabra clave raise, se levanta anónimamente, se vuelve a lanzar la última excepción que estaba activa en el entorno actual. Si no hay ninguna, se levanta RuntimeError.

c. Las excepciones se manejan usando bloques try y except:

```
try:
```

```
    # Código que podría lanzar una excepción
    x = int(input("Ingrese un número: "))
```

```
except ValueError:
```

```
    print("Entrada inválida")
```

```
except (TypeError, MiExcepcion):
```

```
    print("Múltiples excepciones manejadas")
```

```
except:
```

```
    print("Captura cualquier otra excepción")
```

```
else:
```

```
    print("No ocurrió ninguna excepción")
```

```
finally:
```

```
print("Siempre se ejecuta")
```

El bloque else se ejecuta si no ocurre ninguna excepción, y finally se ejecuta siempre. Un except sin nombre de excepción debe ser el último y actúa como comodín.

d. Terminación: Después de manejar una excepción, el bloque try termina, y la ejecución continúa después del bloque except, else o finally. Si no se encuentra un manejador, Python primero busca estáticamente (en bloques try anidados) y en caso de no encontrarlo, lo propaga dinámicamente (por la pila de llamadas). Una excepción no manejada termina el programa con un mensaje de error estándar.

- PHP:

a. Las excepciones son objetos, instancias de la clase Exception o sus subclases:

```
class MiExcepcion extends Exception {}
```

b. Las excepciones se lanzan con la palabra clave throw:

```
throw new MiExcepcion("Ocurrió un error");
```

c. Las excepciones se manejan usando bloques try, catch y finally:

```
try {
    // Código que podría lanzar una excepción
    echo inverse(0);
} catch (Exception $e) {
    echo "Excepción capturada: ", $e->getMessage(), "\n";
} finally {
    echo "Bloque finally\n";
}
```

Cada bloque try debe tener al menos un bloque catch. Las excepciones pueden relanzarse dentro de un bloque catch, y finally asegura la ejecución del código independientemente de las excepciones. Las excepciones no capturadas generan un error fatal a menos que se haya definido un manejador global con set_exception_handler().

d. Terminación: Después de manejar una excepción, el bloque try termina, y la ejecución continúa después del bloque catch o finally. Si no se captura, la excepción provoca un error fatal, deteniendo el programa a menos que exista un manejador global definido.

03. ¿Cuál de esos modelos es más inseguro y por qué?

El modelo de reasunción (utilizado por PL/I) es más inseguro que el modelo de terminación porque permite que la ejecución continúe en el punto donde ocurrió la excepción, lo que puede llevar a estados inconsistentes o errores poco notables si el manejador no corrige completamente el problema. El modelo de terminación, utilizado por Ada, CLU, C++, Java, Python y PHP, es más seguro porque fuerza la terminación del bloque actual, evitando la ejecución en un estado potencialmente corrupto y fomentando un manejo explícito de excepciones a través de la propagación.

Ejercicio 5: La propagación de los errores, cuando no se encuentra ningún manejador asociado, no se implementa igual en todos los lenguajes. Realice la comparación entre el modelo de Java, Python y PL/1, respecto a este tema. Defina la forma en que se implementa en un lenguaje conocido por Ud.

Lenguaje	Tipo de Propagación	Comportamiento si no se maneja	Características Específicas
Java	Dinámica (pila de llamadas)	Terminación con traza de pila	Excepciones comprobadas (<code>throws</code>), propagación hasta <code>main</code> , mensaje de error estándar.
Python	Híbrida (estática en bloques <code>try</code> anidados, luego dinámica)	Terminación con traza de pila	Búsqueda estática y dinámica, sin declaración de excepciones, mensaje de error estándar.
PL/I	Limitada (reasunción)	Comportamiento indefinido o terminación (depende del entorno)	Enfocado en manejo local, manejadores dinámicos, comportamiento depende del compilador.
JavaScript	Dinámica (pila de llamadas)	Error en consola (navegadores) o terminación (Node.js)	Propagación simple, sin excepciones comprobadas, comportamiento varía según el entorno.

Ejercicio 6: Sea el siguiente programa escrito en Pascal

```

...
Procedure Manejador;
Begin ... end;
Procedure P(X:Proc);
begin
  ....
  if Error then X;
  ....
end;
Procedure A;
begin
  ....
  P(Manejador);
  ....
end;
....

```

¿Qué modelo de manejo de excepciones está simulando? ¿Qué necesitaría el programa para que encuadre con los lenguajes que no utilizan este modelo? Justifique la respuesta.

El código simula un modelo de reasunción. En este modelo, cuando se detecta un "error" (simulado por la condición `if Error`), se invoca un manejador (`X`), y después de que el manejador se ejecuta, el control regresa al punto donde ocurrió el error para continuar la ejecución normal.

Los lenguajes que no utilizan el modelo de reasunción (como Java, Python, Ada, C++, etc.) emplean el modelo de terminación. Para que el programa en Pascal encuadre con estos lenguajes, se necesitarían los siguientes cambios:

Interrupción del Flujo y Propagación:

- En lugar de continuar la ejecución tras invocar el manejador, el programa debería interrumpir el flujo actual de P y propagar el error a un nivel superior (por ejemplo, a A o al programa principal). Esto requeriría un mecanismo para salir del procedimiento P y transferir el control al llamador.
- Solución: Usar una estructura de control como exit o un mecanismo de salto (por ejemplo, una variable de estado o una excepción simulada) para indicar que el error debe ser manejado fuera de P.

Definición de un Bloque de Manejo Externo:

- En lenguajes de terminación, el manejo de excepciones ocurre en bloques separados (por ejemplo, try-catch en Java). El programa debería reestructurarse para que Manejador se ejecute como parte de un bloque de manejo en A o en un nivel superior, en lugar de ser llamado directamente dentro de P.
- Solución: Modificar A para incluir un bloque que simule un try-catch, donde P se ejecute dentro de ese bloque y Manejador se invoque solo si se propaga el error.

Simulación de Excepciones:

- Pascal no tiene excepciones nativas, por lo que el "error" se simula con la condición if Error. Para emular el modelo de terminación, se podría usar una variable global o un parámetro de retorno para indicar que ocurrió un error, y el llamador (A) decidiría cómo manejarlo sin reanudar P.
- Solución: Introducir un parámetro de salida en P (por ejemplo, P(var ErrorOccurred: Boolean; X: Proc)) que indique si ocurrió un error, y en A verificar ese valor para decidir si ejecutar Manejador.

Ejercicio 7:

<pre> Program Principal; var x:int; b1,b2:boolean; Procedure P (b1:boolean); var x:int; Procedure Manejador1 begin x:=x + 1; end; begin x:=1; if b1=true then Manejador1; x:=x+4; end; Procedure Manejador2; begin x:=x * 100; end; </pre>	<pre> Begin x:=4; b2:=true; b1:=false; if b1=false then Manejador2; P(b); write (x); End. </pre>
---	--

a) Implemente este ejercicio en PL/1 utilizando manejo de excepciones

...

Prog Principal


```

DCL x DEC FIXED (3,2) INIT (0);
// como se instancian los boolean en pl1?

PROC P
    DCL x DEC FIXED (3,2) INIT (0);
    ON CONDITION Manejador1 begin
        x = x + 1;
    end;
    x = 1;
    if(b1 = true) then SIGNAL CONDITION Manejador1
    x = x + 4
    ON CONDITION Manejador2 begin
        x = x * 100;
    end;
end;

Begin
    x = 4;
    b2 = true;
    b1 = false;
    // el manejador 2 deberia estar acá porque sino se rompe todo
    ON CONDITION Manejador2 begin
        if(b1 = false) then SIGNAL CONDITION Manejador2;
        P;
        PUT SKIP LIST(x);
    end;
    ...

```

b) ¿Podría implementarlo en JAVA utilizando manejo de excepciones? En caso afirmativo, realízelo.

Teóricamente no, porque esto es con el modelo de reasunción, pero se puede simular algo parecido:

```
public class Principal {
    static int x;
    static boolean b1, b2;

    public static void main(String[] args) {
        x = 4;
        b2 = true;
        b1 = false;

        try {
            if (!b1) {
                manejador2();
            }
            P(b1);
        } catch (Exception e) {
            System.out.println("Excepción atrapada: " + e.getMessage());
        }

        System.out.println(x);
    }

    public static void P(boolean b1_param) throws Exception {
        int x_local = 1;

        if (b1_param) {
            x_local = manejador1(x_local);
        }

        x_local = x_local + 4;
        // x_local no afecta al x global, pero lo hace en Pascal. Para que sea igual:
        x = x_local;
    }

    public static int manejador1(int x) {
        return x + 1;
    }

    public static void manejador2() {
        x = x * 100;
    }
}
```

Ejercicio 8: Sean los siguientes, procedimientos de un programa escrito en JAVA:

```

public static void main (String[] argos){
    Double array_doubles[] = new double[500];
    for (int i=0; i<500; i++){
        array_doubles[i]=7*i;
    }
    for (int i=0 ; i<600 ; i=i+25){
        try{
            system.out.println("El elemento en "+ i + " es " + acceso_por_indice (array_doubles,i));
        }
        catch(ArrayIndexOutOfBoundsException e){
            system.out.println(e.toString());
        }
        catch(Exception a){
            system.out.println(a.toString());
        }
        finally{
            system.out.println("sentencia finally");
        }
    }
}

Public static double acceso_por_indice (double [] v, int indice) throws Exception; ArrayIndexOutOfBoundsException{
    if ((indice>=0) && (indice<v.length)){
        Return v[indice];
    }
    else{
        if (indice<0){
            // caso excepcional
            Throw new ArrayIndexOutOfBoundsException(" el índice" + indice + " es un número negativo");
        }
        else{
            // caso excepcional
            Throw new Exception(" el indice" + indice + " no es una posición válida");
        }
    }
}

```

a) Analizar el ejemplo y decir qué manejadores ejecuta y en qué valores quedan las variables. JUSTIFIQUE LA RESPUESTA.

“i” toma los siguientes valores: Iteraciones de 0, 25, 50, 75, ..., hasta 575. El arreglo tiene un tamaño de 500 elementos (posición 0 hasta 499). Para $i \geq 500$, el índice está fuera de rango y el método `acceso_por_indice` lanza una `Exception` genérica, no una `ArrayIndexOutOfBoundsException`.

Para $i = 0$ hasta $i = 475 \rightarrow$ no hay excepción, se accede correctamente y se imprime el valor.

Se imprime:

- El elemento en 0 es 0.0 (bloque try) y sentencia finally (bloque finally).
- ...
- El elemento en 475 es 3325.0 (bloque try) y sentencia finally (bloque finally)

Para $i = 500, 525, 550, 575 \rightarrow$ índice fuera de rango, se lanza `Exception` genérica desde `acceso_por_indice`.

Se imprime:

- `java.lang.Exception`: el índice 500 no es una posición válida (bloque catch) y sentencia finally (bloque finally)
- `java.lang.Exception`: el índice 525 no es una posición válida (bloque catch) y sentencia finally (bloque finally)

- `java.lang.Exception`: El índice 550 no es una posición válida (bloque `catch`) y sentencia `finally` (bloque `finally`)
- `java.lang.Exception`: El índice 575 no es una posición válida (bloque `catch`) y sentencia `finally` (bloque `finally`)

b) La excepción ¿se propaga o se maneja en el mismo método? ¿Qué instrucción se agrega para poder propagarla y que lleve información?

En este diseño la excepción se propaga. “acceso_por_indice” no la maneja, sino que la lanza (`throw`) y el método `main` la captura en el bloque `try-catch`.

```
throw new Exception("Mensaje con información");
```

Esto crea una excepción con un mensaje, lo cual permite que quien la reciba tenga contexto del error.

c) ¿Como modificaría el método “acceso_por_indice” para que maneje él mismo la excepción?

```
public class Main {
    public static void main(String[] args) {
        double array_doubles[] = new double[500];

        // Llenar el array con múltiplos de 7
        for (int i = 0; i < 500; i++) {
            array_doubles[i] = 7.0 * i;
        }

        for (int i = 0; i < 600; i += 25) {
            System.out.println("El elemento en " + i + " es " + acceso_por_indice(array_doubles, i));
            System.out.println("sentencia finally");
        }
    }

    public static double acceso_por_indice(double[] v, int indice) {
        try {
            if (indice >= 0 && indice < v.length) {
                return v[indice];
            } else if (indice < 0) {
                throw new ArrayIndexOutOfBoundsException("El índice " + indice + " es un número negativo");
            } else {
                throw new Exception("El índice " + indice + " no es una posición válida");
            }
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Excepción atrapada en acceso_por_indice: " + e.getMessage());
            return -1;
        } catch (Exception e) {
            System.out.println("Excepción general atrapada en acceso_por_indice: " + e.getMessage());
            return -1;
        }
    }
}
```

Ejercicio 9: Indique diferencias y similitudes entre Python y Java con respecto al manejo de excepciones.

Similitudes:

- Estructura `try-catch`: Ambos lenguajes utilizan bloques `try` para envolver el código que puede generar excepciones y bloques (`catch` en Java, `except` en Python) para capturar y manejar las excepciones.

- Jerarquía de excepciones: En ambos lenguajes, las excepciones se organizan en una jerarquía de clases, con una clase base (Throwable en Java, Exception en Python) de la cual derivan otras excepciones más específicas.
- Manejo de múltiples excepciones: Tanto Python como Java permiten capturar múltiples tipos de excepciones en un solo bloque o en bloques separados.
- Bloque finally: Ambos lenguajes ofrecen un bloque finally que se ejecuta siempre, independientemente de si se lanza una excepción o no, útil para liberar recursos.
- Propagación de excepciones: Si no se captura una excepción, se propaga hacia arriba en la pila de llamadas en ambos lenguajes.

Diferencias:

- Sintaxis:
 - En Java, se usa try, catch, y finally. Ejemplo:

```
java

try {
    // Código
} catch (Exception e) {
    // Manejo
} finally {
    // Limpieza
}
```

- En Python, se usa try, except, else, y finally. Ejemplo:

```
python

try:
    # Código
except Exception as e:
    # Manejo
else:
    # Código si no hay excepción
finally:
    # Limpieza
```

- Excepciones verificadas vs no verificadas:
 - Java distingue entre excepciones verificadas (checked, deben declararse con throws o manejarse) y no verificadas (unchecked, derivadas de RuntimeException). Ejemplo: IOException es verificada.
 - Python no tiene excepciones verificadas; todas son no verificadas, y no es necesario declararlas ni manejarlas explícitamente.
- Declaración de excepciones:
 - En Java, las excepciones verificadas deben declararse en la firma del método con la palabra clave throws. Ejemplo: public void metodo() throws IOException.
 - En Python, no se requiere declarar excepciones en la firma de una función.
- Bloque else:
 - Python tiene un bloque else que se ejecuta si no ocurre ninguna excepción en el bloque try, lo cual no existe en Java.

- En Java, este comportamiento se logra colocando el código después del bloque catch, pero no es tan explícito.
- Tipado de excepciones:
 - En Java, los bloques catch requieren especificar el tipo exacto de excepción o una superclase. Ejemplo: `catch (FileNotFoundException e)`.
 - En Python, se puede omitir el tipo de excepción (`except:`), aunque no es una buena práctica, ya que captura todas las excepciones.
- Excepciones personalizadas:
 - En Java, se crean extendiendo `Exception` o `RuntimeException`. Ejemplo: `class MiExcepcion extends Exception {}`.
 - En Python, se crean extendiendo `Exception`. Ejemplo: `class MiExcepcion(Exception): pass`.
- Rendimiento y filosofía:
 - En Python, el manejo de excepciones es más flexible y se usa a menudo como parte del flujo de control.
 - En Java, el manejo de excepciones es más estricto debido a las excepciones verificadas.

Ejercicio 10: ¿Qué modelo de excepciones implementa Ruby? ¿Qué instrucciones específicas provee el lenguaje para manejo de excepciones y cómo se comportan cada una de ellas?

Ruby implementa un modelo de manejo de excepciones basado en objetos con criterio de continuación de tipo terminal, donde las excepciones son instancias de la clase `Exception` o sus subclases. Este modelo permite interrumpir el flujo normal de ejecución del programa cuando ocurre un error o una condición excepcional, transfiriendo el control a un bloque de código diseñado para manejar la situación.

En Ruby, cuando se lanza una excepción (con `raise`) y no se captura en un bloque `rescue`, la excepción se propaga hacia el método o bloque que llamó al código donde ocurrió el error. La búsqueda de un bloque `rescue` comienza en el bloque `begin` más cercano (o en el cuerpo implícito de un método) y continúa hacia arriba en la pila de llamadas hasta encontrar un `rescue` que coincida con el tipo de excepción o hasta llegar al nivel superior del programa.

Si no se encuentra un manejador, el programa termina con un mensaje de error que incluye el tipo de excepción, el mensaje y el stack trace. Ruby no distingue entre excepciones verificadas y no verificadas, por lo que todas las excepciones se propagan de la misma manera.

Instrucciones para manejo de excepciones en Ruby:

1. `begin`:

- Inicia un bloque donde se espera una excepción; el código se ejecuta hasta que ocurre un error, luego Ruby busca un `rescue` coincidente. Implícito en métodos si no se usa explícitamente.

```
begin
  result = 10 / 0
rescue ZeroDivisionError
  puts "No se puede dividir por cero"
end
```

2. `rescue`:

- Captura excepciones específicas o genéricas (StandardError por defecto), asignándolas opcionalmente a una variable. Compara la excepción con las clases especificadas y ejecuta el bloque correspondiente; si no hay coincidencia, la excepción se propaga.

```
begin
  raise ArgumentError, "Argumento inválido"
rescue ArgumentError => e
  puts "Error capturado: #{e.message}"
end
```

3. else:

- Ejecuta código si no ocurre ninguna excepción en el bloque begin.

```
begin
  puts "Ejecutando código sin errores"
rescue StandardError => e
  puts "Error: #{e.message}"
else
  puts "No hubo errores"
end
```

4. ensure:

- Ejecuta código siempre, con o sin excepción, ideal para liberar recursos. Se ejecuta incluso con return o errores no manejados.

```
begin
  file = File.open("archivo.txt")
rescue StandardError => e
  puts "Error: #{e.message}"
ensure
  file.close if file
  puts "Archivo cerrado"
end
```

5. raise:

- Lanza una excepción manualmente (RuntimeError por defecto o un tipo específico con mensaje). Puede relanzar la excepción actual si se usa sin argumentos.

```
def dividir(a, b)
  raise ArgumentError, "El divisor no puede ser cero" if b == 0
  a / b
end
begin
  dividir(10, 0)
rescue ArgumentError => e
  puts e.message
end
```

6. retry:

- Reinicia el bloque begin desde el principio tras manejar una excepción en rescue. Útil para reintentos, pero debe usarse con cuidado para evitar bucles infinitos.

```

intentos = 0
begin
  intentos += 1
  raise "Error simulado" if intentos < 3
  puts "Éxito después de #{intentos} intentos"
rescue StandardError
  retry if intentos < 3
end

```

7. throw y catch:

- catch define un bloque etiquetado; throw interrumpe ese bloque con un valor opcional. Usado para control de flujo ligero, no para manejo de errores.

```

resultado = catch(:done) do
  throw(:done, "Terminado") if true
  "No se ejecuta"
end
puts resultado

```

Ejercicio 11: Indique el mecanismo de excepciones de JavaScript.

Se pueden lanzar excepciones usando la instrucción throw y manejarlas usando las declaraciones try...catch.

Tipos de excepciones:

Casi cualquier objeto se puede lanzar en JavaScript. Sin embargo, no todos los objetos lanzados son iguales.

- excepciones ECMAScript
- La interfaz DOMException representa un evento anormal (llamado excepción) que ocurre como resultado de llamar a un método o acceder a una propiedad de una API web y la interfaz DOMError describe un objeto de error que contiene un nombre de error.

Expresión throw:

Utiliza la expresión throw para lanzar una excepción. Una expresión throw especifica el valor que se lanzará:

throw expression;

Es posible lanzar cualquier expresión y especificar un objeto, no solo expresiones de un tipo específico.

Declaración try...catch:

La declaración try...catch consta de un bloque try, que contiene una o más declaraciones, y un bloque catch, que contiene declaraciones que especifican qué hacer si se lanza una excepción en el bloque try.

Si alguna instrucción dentro del bloque try (o en una función llamada desde dentro del bloque try) arroja una excepción, el control inmediatamente cambia al bloque catch. Si no se lanza ninguna excepción en el bloque try, se omite el bloque catch. El bloque finally se ejecuta después de que se ejecutan los bloques try y catch, pero antes de las declaraciones que siguen a la declaración try...catch.

El bloque catch:

Se puede usar un bloque catch para manejar todas las excepciones que se puedan generar en el bloque try.


```
catch (catchID) {
  instrucciones
}
```

El bloque catch especifica un identificador (catchID en la sintaxis anterior) que contiene el valor especificado por la expresión throw. Puedes usar este identificador para obtener información sobre la excepción que se lanzó.

JavaScript crea este identificador cuando se ingresa al bloque catch. El identificador dura solo la duración del bloque catch. Una vez que el bloque catch termina de ejecutarse, el identificador ya no existe.

El bloque finally:

El bloque finally contiene instrucciones que se ejecutarán después que se ejecuten los bloques try y catch. Además, el bloque finally ejecuta antes el código que sigue a la declaración try...catch...finally.

También es importante notar que el bloque finally se ejecutará independientemente de que se produzca una excepción. Sin embargo, si se lanza una excepción, las declaraciones en el bloque finally se ejecutan incluso si ningún bloque catch maneje la excepción que se lanzó.

Declaraciones try...catch anidadas:

Existe el anidamiento de una o más declaraciones try...catch.

Si un bloque try interno no tiene un bloque catch correspondiente:

1. debe contener un bloque finally, y
2. el bloque catch adjunto de la declaración try...catch se comprueba para una coincidencia.

Propagación de excepciones:

- Cuando se lanza una excepción con throw, JavaScript busca el bloque try/catch más cercano en la pila de llamadas.
- Si no hay un catch en el ámbito actual, la excepción se propaga al método o función que llamó al código, y así sucesivamente.
- Si no se captura en ningún nivel, el programa termina (en Node.js, el proceso termina con un error; en navegadores, se muestra en la consola).
- En contextos asíncronos (como con Promise o async/await), las excepciones no se propagan automáticamente a bloques try/catch tradicionales a menos que se usen dentro de una función async.

Ejercicio 12: Sea el siguiente programa escrito en PYTHON. Indique el camino de ejecución.

```
#!/usr/bin/env python
#calc.py

def dividir():
    x = a / b
    print (("Resultado"), (x))

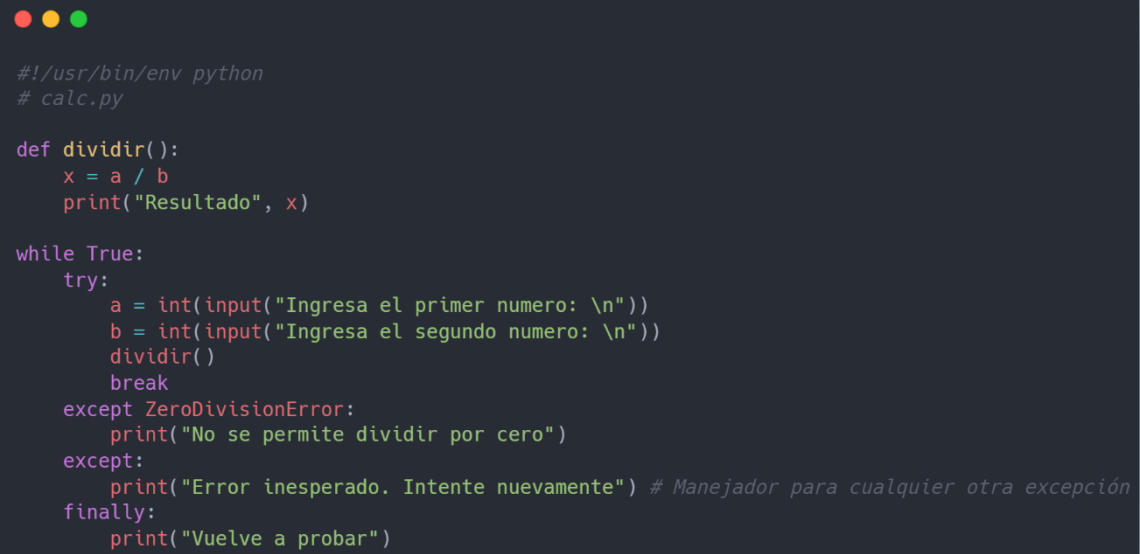
while True:
    try:
        a = int(input("Ingresa el primer numero: \n"))
        b = int(input("Ingresa el segundo numero: \n"))
        dividir()
        break
    except ZeroDivisionError:
        print ("No se permite dividir por cero")

finally:
    print ("Vuelve a probar")
```

a) Describa qué caminos ejecuta para diferentes valores de ingreso

1. Si hay cualquier valor numérico de a, y un valor de b != 0
 - No hay error de conversión ni división por cero
 - Se ejecuta dividir y muestra el resultado
 - Se ejecuta el finally
 - Finaliza el programa al ejecutar break
2. Si hay cualquier valor numérico de a y un valor de b = 0
 - Se ejecuta dividir
 - Ocurre un ZeroDivisionError
 - Se ejecuta el bloque except ZeroDivisionError, imprimiendo No se permite dividir por cero
 - Se ejecuta el bloque finally, imprimiendo Vuelve a probar
 - No se ejecuta break, vuelve el bucle
3. Si se ingresa un carácter o un valor que no sea numérico
 - Ocurre un ValueError
 - Se busca el except correspondiente, como no se encuentra finaliza el programa con error y muestra el Traceback

b) Agregar el uso de una excepción anónima

A screenshot of a terminal window with a dark background and light-colored text. The window has three colored window control buttons (red, yellow, green) in the top-left corner. The code is a Python script named 'calc.py' that defines a 'dividir()' function and a 'while True:' loop. The function calculates the division of two numbers 'a' and 'b' and prints the result. The loop prompts the user for two numbers, calls the 'dividir()' function, and handles exceptions: 'ZeroDivisionError' (division by zero) and a general 'except' block for other errors. The loop continues until the user enters a non-zero second number or the user chooses to try again.

```
#!/usr/bin/env python
# calc.py

def dividir():
    x = a / b
    print("Resultado", x)

while True:
    try:
        a = int(input("Ingresa el primer numero: \n"))
        b = int(input("Ingresa el segundo numero: \n"))
        dividir()
        break
    except ZeroDivisionError:
        print("No se permite dividir por cero")
    except:
        print("Error inesperado. Intente nuevamente") # Manejador para cualquier otra excepción
    finally:
        print("Vuelve a probar")
```

Ejercicio 13: Sea el siguiente código escrito en JAVA

```

public class ExcepcionUno extends Exception {

    public ExcepcionUno(){
        super(); // constructor por defecto de Exception
    }

    public ExcepcionUno( String cadena ){
        super( cadena ); // constructor param. de Exception
    }

}

public class ExcepcionDos extends Exception {

    public ExcepcionDos(){
        super(); // constructor por defecto de Exception
    }

    public ExcepcionDos( String cadena ){
        super( cadena ); // constructor param. de Exception
    }

}

public class ExcepcionTres extends Exception {

    public ExcepcionTres(){
        super(); // constructor por defecto de Exception
    }

    public ExcepcionTres( String cadena ){
        super( cadena ); // constructor param. de Exception
    }

}

public class Lanzadora {

    public void lanzaSiNegativo(int param) throws ExcepcionUno {
        if (param < 0)
            throw new ExcepcionUno("Numero negativo");
    }

    public void lanzaSiMayor100(int param) throws ExcepcionDos {
        if (param >100 and param<125)
            throw new ExcepcionDos("Numero mayor100");
    }

    public void lanzaSiMayor125(int param) throws ExcepcionTres {
        if (param >= 125)
            throw new ExcepcionTres("Numero mayor125");
    }

}

```

```

import java.io.FileInputStream;
import java.io.IOException;

public class Excepciones {

    public static void main(String[] args) {
        // Para leer un fichero
        Lanzadora lanza = new Lanzadora();
        FileInputStream entrada = null;
        int leo;
        try {
            entrada = new FileInputStream("fich.txt");
            while ((leo = entrada.read()) != -1){
                if (leo < 0)
                    lanza.lanzaSiNegativo(leo);
                else if (leo > 100)
                    lanza.lanzaSimayor100(leo);
            }

            entrada.close();
            System.out.println("Todo fue bien");
        }
        catch (ExcepcionUno e) { // Personalizada
            System.out.println("Excepcion: " + e.getMessage());
        }
        catch (ExcepcionDos e) { // Personalizada
            System.out.println("Excepcion: " + e.getMessage());
        }
        catch (IOException e) { // Estándar
            System.out.println("Excepcion: " + e.getMessage());
        }
        finally {
            if (entrada != null)
                try {
                    entrada.close(); // Siempre queda cerrado
                }
                catch (Exception e) {
                    System.out.println("Excepcion: " + e.getMessage());
                }
            System.out.println("Fichero cerrado.");
        }
    }
}

```

a) Indique cómo se ejecuta el código. Debe quedar en claro los caminos posibles de ejecución, cuáles son los manejadores que se ejecutan y cómo se buscan los mismos y si en algún caso se produce algún error.

1. El archivo existe, el primer valor leído es 5 y el segundo es -1
 - No ocurren excepciones
 - Se cierra el archivo
 - Se imprime "Todo fue bien"
 - Se ejecuta el bloque finally
 - Imprime fichero cerrado
 - Finaliza la ejecución

2. El archivo existe y el primer valor leído es -2
 - Se llama a `lanza.lanzaSiNegativo(leo)`
 - Se lanza la excepción `"ExcepcionUno("Numero negativo")"`
 - Se busca un manejador dinámicamente y se encuentra `catch (ExcepcionUno e)`
 - Se imprime `"Excepción: Numero negativo"`
 - Se ejecuta el bloque `finally`
 - Imprime fichero cerrado
 - Finaliza la ejecución
3. El archivo existe y el primer valor leído es 110
 - Se llama a `lanza.lanzaSimayor100(leo)`
 - Se lanza la excepción `"ExcepcionDos("Numero mayor100")"`
 - Se busca un manejador dinámicamente y se encuentra `catch (ExcepcionDos e)`
 - Se imprime `"Excepcion: Numero mayor100"`
 - Se ejecuta el bloque `finally`
 - Imprime fichero cerrado
 - Finaliza la ejecución
4. El archivo existe y el primer valor leído es 200
 - Se llama a `lanza.lanzaSimayor125(leo)`
 - Se lanza la excepción `"ExcepcionTres("Numero mayor125")"`
 - Se busca un manejador dinámicamente y se encuentra `catch (ExcepcionTres e)`
 - Se imprime `"Excepcion: Numero mayor125"`
 - Se ejecuta el bloque `finally`
 - Imprime fichero cerrado
 - Finaliza la ejecución
5. El archivo no existe
 - Se lanza la excepción `"IOException"`
 - Se busca un manejador dinámicamente y se encuentra `catch (IOException e)`
 - Se imprime `"Excepcion: ..."`
 - Se ejecuta el bloque `finally`
 - Imprime fichero cerrado
 - Finaliza la ejecución
6. Falla cerrar el archivo en el bloque `finally`
 - Se lanza una excepción
 - Se busca un manejador dinámicamente y se encuentra `catch (Exception e)`
 - Se imprime `"Excepcion: ..."`
 - Imprime fichero cerrado
 - Finaliza la ejecución

Ejercicio 14. Dado el siguiente código en Java. Indique todos los posibles caminos de resolución, de acuerdo a los números que vaya leyendo del archivo.

```

class ExcepcionE1 extends Exception {

    public ExcepcionE1(){
        super(); // constructor por defecto de Exception
    }

    public ExcepcionE1( String cadena ){
        super( cadena ); // constructor param. de Exception
    }
}

class ExcepcionE2 extends Exception {

    Public ExcepcionE2(){
        super(); // constructor por defecto de Exception
    }
    Public ExcepcionE2( String cadena ){
        super( cadena ); // constructor param. de Exception
    }
}

// Esta clase lanzará la excepción
public class Evaluacion {
    void Evalua( int edad ) throws ExcepcionE1, ExcepcionE2 {
        if ( edad < 18 )
            throw new ExcepcionE1( "Es una persona menor de edad" );
        else if ( edad > 70 )
            throw new ExcepcionE2( "Es persona mayor de edad" );
    }

    void Segmenta( int edad ) throws ExcepcionE1, ExcepcionE2 {
        if ( edad < 35 )
            throw new ExcepcionE1( "Es una persona joven" );
    }
}

```

```

class AnalisisEdadPoblacion{
    public static void main( String[] args ){
        // Para leer un fichero
        Evaluacion Invoca = new Evaluacion();
        FileInputStream ream entrada = null;
        int leo;
        try{
            entrada = new FileInputStream( "fich.txt" );
            while ( ( leo = entrada.read() ) != -1 ) {
                try {
                    if (leo<0) {
                        throw new ExcepcionE1( "Edad inválida");
                    }
                    else{
                        if (leo>120){
                            throw new ExcepcionE1( "Edad inválida" );
                        }
                    }
                    invoca.evalua (leo);
                    invoca.segmenta( leo );
                    System.out.println( " ( "Es persona adulta, Todo fue bien" );
                }
                catch ( ExcepcionE2 e ){
                    System.out.println( "Excepcion: " + e.getMessage() );
                }
                catch ( ExcepcionE1 e ){
                    System.out.println( "Excepcion: " + e.getMessage() );
                }
            } catch (FileNotFoundException e1) {
                System.out.println("No se encontró el archivo");
            } catch (IOException e) {
                System.out.println("Problema para leer los datos");
            } finally {
                if (entrada != null)
                    try {
                        entrada.close();
                    } catch (Exception e) {
                        System.out.println("Excepcion: " + e.getMessage());
                    }
                System.out.println("Fichero cerrado.");
            }
        }
    }
}

```

Caminos según los números:

1. Si se lee -1
 - Se sale del while
 - Se ejecuta el bloque finally
 - Continúa leyendo el siguiente valor del archivo y finaliza si no hay más datos
2. Si se lee -2 o cualquier otro valor negativo
 - Se lanza ExcepcionE1("Edad inválida")
 - Se busca estáticamente un manejador y se encuentra
 - Imprime el mensaje de la excepción
 - Ejecuta finally

- Continúa leyendo el siguiente valor del archivo y finaliza si no hay más datos
- 3. Si se lee 121 o cualquier otro valor mayor
 - Se lanza ExcepcionE1("Edad inválida")
 - Se busca estáticamente un manejador y se encuentra
 - Imprime el mensaje de la excepción
 - Ejecuta finally
 - Continúa leyendo el siguiente valor del archivo y finaliza si no hay más datos
- 4. Si se lee cualquier valor entre 0 y 17
 - Se llama a "invoca.evalua(leo)"
 - Se lanza ExcepcionE1("Es una persona menor de edad")
 - Se busca un manejador estáticamente y se encuentra
 - Imprime el mensaje de la excepción
 - Ejecuta finally
 - Continúa leyendo el siguiente valor del archivo y finaliza si no hay más datos
- 5. Si se lee cualquier valor entre 18 y 34
 - Se llama a "invoca.evalua(leo)"
 - Se llama a "invoca.segmenta(leo)"
 - Se lanza la ExcepcionE1("Es una persona joven")
 - Se busca un manejador estáticamente y se encuentra
 - Imprime el mensaje de la excepción
 - Ejecuta finally
 - Continúa leyendo el siguiente valor del archivo y finaliza si no hay más datos
- 6. Si se lee cualquier valor entre 35 y 70
 - Se llama a "invoca.evalua(leo)"
 - Se llama a "invoca.segmenta(leo)"
 - Imprime "Es una persona adulta, todo fue bien"
 - Se ejecuta el bloque finally
 - Continúa leyendo el siguiente valor del archivo y finaliza si no hay más datos
- 7. Si se lee cualquier valor entre 71 y 120
 - Se llama a "invoca.evalua(leo)"
 - Se lanza ExcepcionE2("Es una persona mayor de edad")
 - Se busca un manejador estáticamente y se encuentra
 - Imprime el mensaje de la excepción
 - Ejecuta finally
 - Continúa leyendo el siguiente valor del archivo y finaliza si no hay más datos
- 8. El archivo no existe
- 9. Se lanza FileNotFoundException
- 10. Se busca un manejador y se encuentra
- 11. Imprime "No se encontró el archivo"
- 12. Ejecuta bloque finally
- 13. Continúa leyendo el siguiente valor del archivo y finaliza si no hay más datos
- 14. Problema para leer los datos
- 15. Se lanza IOException
- 16. Se busca un manejador estáticamente y se encuentra
- 17. Se imprime "Problema para leer los datos"
- 18. Ejecuta bloque finally
- 19. Continúa leyendo el siguiente valor del archivo y finaliza si no hay más datos

20. Problema para cerrar el archivo en el finally
21. Se lanza Exception
22. Se busca un manejador estáticamente y se encuentra
23. Imprime el mensaje de la excepción
24. Continúa leyendo el siguiente valor del archivo y finaliza si no hay más datos