# Conceptos y Paradigmas de Lenguajes de Programación - Práctica 9

EJERCICIO 1

**Excepción**: es una condición inesperada o inusual que surge durante la ejecución del programa y no puede ser manejada en el contexto local. Condición anómala e inesperada que necesita ser controlada.

EJERCICIO 2

Para que un lenguaje trate excepciones debe proveer mínimamente:
- Un modo de definirlas
- Una forma de reconocerlas
- Una forma de lanzarlas y capturarlas
- Una forma de manejarlas especificando el código y respuestas
- Un criterio de continuación

No, no todos los lenguajes lo proveen, por ejemplo **C** estándar **no provee** manejo de excepciones.

EJERCICIO 3

Cuando un lenguaje de programación **no proporciona un mecanismo de manejo de excepciones**, puede ser más difícil y menos estructurado manejar errores y situaciones excepcionales en el código. Sin un manejo adecuado de excepciones, los errores pueden propagarse sin control, lo que puede llevar a un comportamiento inesperado y a una mayor dificultad para depurar el código.

Aunque no se disponga de un mecanismo de manejo de excepciones incorporado en el lenguaje, es posible simularlo utilizando técnicas alternativas. Una forma común de simular el manejo de excepciones es mediante el **uso de estructuras de control condicionales y funciones de retorno de errores.**

Es importante tener en cuenta que este enfoque simulado del manejo de excepciones puede resultar más propenso a errores y puede llevar a un código **más complicado y difícil de mantener** en comparación con un lenguaje que ofrece un manejo de excepciones nativo. Además, la simulación no proporcionará todas las funcionalidades avanzadas de los mecanismos de manejo de excepciones, como la capacidad de capturar excepciones específicas o realizar acciones de limpieza antes de propagar el error.

EJERCICIO 4 - 01

**Reasunción**: cuando se produce una excepción, se maneja y al terminar de ser manejada se devuelve el control a la sentencia siguiente de donde se levantó la excepción. Lenguajes: PL/1.

**Terminación**: cuando se produce una excepción, el bloque donde se levantó la excepción es terminado y se ejecuta el manejador asociado a la excepción. Lenguajes: ADA, CLU, C++, Java, Python o PHP.

EJERCICIO 4 - 02

## Reasunción

| | PL/I |
|---|---|
| **¿Cómo se define?** | Con instrucciones como **'ON condition DO; ... END;'** |
| **¿Cómo se lanza?** | Explícitamente **'SIGNAL condition'.** Automáticamente cuando ocurre una condición predefinida (por ej. división por cero activa ON ZERODIVIDE). |
| **¿Cómo se maneja?** | El bloque **'DO/BEGIN ... END'** se ejecuta al ocurrir la condición. |
| **Criterio de continuación** | **Reasunción**: Luego de ejecutarse el manejador, el control vuelve a la instrucción donde se generó la excepción. |

## Terminación

| | ADA | C++ | JAVA | PYTHON | PHP |
|---|---|---|---|---|---|
| **¿Cómo se define?** | **Nombre: exception;** Se define con bloques **'begin .. exception .. end'.** | Se define con bloques **try-catch.** | Se utiliza **try-catch-finally.** | Con bloques **try-except-finally.** | Con bloques **try-catch-finally.** |
| **¿Cómo se lanza?** | Con **'raise NombreDeLa Excepción;'.** | Con la instrucción **'throw'.** | Con **'throw new Exception();'.** | Con **'raise Exception'.** | Con **'throw new Exception();'.** |
| **¿Cómo se maneja?** | Dentro del bloque '**exception**' usando '**when**'. | Con bloques '**catch**' que capturan excepciones según su tipo. | Con bloques '**catch**' específicos para cada tipo de excepción. | Con bloques '**except**' que capturan excepciones por tipo. | Con bloques '**catch**'. |
| **Criterio de continuación** | **Terminación**: El bloque se termina y el control pasa a después del bloque que lanzó la excepción. | **Terminación**: El control salta al bloque catch y luego continúa después del try-catch. | **Terminación**: El flujo pasa al bloque catch y luego continúa después del bloque try-catch-finally. | **Terminación**: El control pasa al bloque except y luego continúa después del try. | **Terminación**: El flujo salta al catch y luego continúa normalmente. |

EJERCICIO 4 - 03

El modelo por **reasunción** es mas inseguro dado que cuando surge una excepción no termina, sigue ejecutando donde se quedo y puede que eso acarree errores en la ejecución.

EJERCICIO 5

| Lenguaje | ¿Qué ocurre si no hay un manejador? |
|---|---|
| **JAVA** | La excepción se propaga por la pila de llamadas (call stack). Si no se encuentra un **catch**, el programa termina y se imprime un stack trace. |
| **PYTHON** | La excepción también se propaga por la pila. Si no es capturada por ningún **except**, se imprime un traceback y el programa finaliza con error. |
| **PL/1** | Si no hay un **ON condition** activo para manejar la excepción, el comportamiento depende del entorno de ejecución: puede abortar el programa o ejecutar una acción por defecto. |
| **JAVASCRIPT** | Si no hay un bloque **try...catch**, la excepción se propaga hacia arriba. Si no es manejada, aparece un error en la consola y puede interrumpirse la ejecución del script. En entornos como Node.js, puede provocar la terminación del proceso. |

EJERCICIO 6

El modelo de manejo de excepciones que está simulando es **reasunción**, ya que una vez que se maneja la excepción se continua con la sentencia siguiente de donde se levantó la excepción, es decir, no se termina el bloque. Para que encuadre con los lenguajes que no utilizan este modelo Procedure P debería no tener más código luego del if.

EJERCICIO 7A

**PL/1 - Reasunción**

```
Prog Principal;
DCL x INTEGER;
DCL b1,b2 BOOLEAN;
    PROC P (b1 BOOLEAN);
    DCL x INTEGER;
    BEGIN
        ON CONDITION Manejador1 BEGIN SET x = ARITH(x + 1); END;
        SET X = 1;
        IF b1=true THEN
            SIGNAL CONDITION Manejador1
        SET x = ARITH(x+4);
    END;

BEGIN
    ON CONDITION Manejador2 BEGIN SET x = ARITH(x * 100); END;
    SET X = 4;
    SET b2 = true;
    SET b1 = false;
    IF b1=false THEN SIGNAL CONDITION Manejador2
    CALL P(b1);
    WRITE (x);
END.
```

EJERCICIO 7B

| ¿Podría implementarlo en JAVA utilizando manejo de excepciones? | Implementación en Java - Terminación |
|---|---|
| Sí podría, pero la diferencia está en que Java utiliza la **terminación** y hay partes del código que no se ejecutarían si hay una excepción, por ejemplo en *PROC P SET x = ARITH(x+4);* si hay una excepción en Java, eso no se ejecutaría, ya que se terminaría el bloque. Una manera de solucionar esto seria haciendo uso de "**finally**" en donde debe estar el código que el programador desee que se ejecute siempre. |  |

```
public class Principal {
    public static void P (boolean b1){
        int x;
        try{
            x = 1;
            if (b1){
                throw new Manejador1Exception();
            }
        }
        catch (Manejador1Exception e){
            x = x + 1;
        }
        finally{
            x = x + 4;
        }

    }

    public static void main(String[] args) {
        int x = 4,
        boolean b1 = true, b2 = false;
        try {
            if (!b1){
                throw new Manejador2Exception();
            }
        }
        catch (Manejador2Excepction e){
            x = x * 100;
        }
        finally{
            P(b1);
            System.out.println(x);
        }
    }
}

public class Manejador1Exception extends Exception{
    public Manejador1Exception(){
        super();
    }
}

public class Manejador2Exception extends Exception{
    public Manejador2Exception(){
        super();
    }
}
```

EJERCICIO 8-A

int = 0 - "El elemento en 0 es 0" – "sentencia finally"
int = 25 - "El elemento en 25 es 175" – "sentencia finally"
int = 50 - "El elemento en 50 es 350" – "sentencia finally"
...
int = 475 - "El elemento en 475 es 3325" – "sentencia finally"
int = 500 - "El índice 500 no es una posición valida" – "sentencia finally"
int = 525 - "El índice 525 no es una posición valida" – "sentencia finally"
int = 550 - "El índice 550 no es una posición valida" – "sentencia finally"
int = 575 - "El índice 575 no es una posición valida" – "sentencia finally"

En los que se ejecuta un manejador (de int = 500 a int = 575), se ejecuta el manejador del **catch(Exception a)** del **main**.

EJERCICIO 8-B

La excepción se propaga dinámicamente del método "**acceso_por_indice**" al "**main**". Para propagarla se agrega la declaración **"throws Exception, ArrayIndexOutOfBoundsException"** en la firma del método "**acceso_por_indice**". Esto indica que el método puede lanzar esas excepciones y que cualquier llamada a ese

método debe manejar o propagar esas excepciones. En este caso, el método "**main**" captura y maneja las excepciones utilizando bloques **try-catch**.

EJERCICIO 8-C

Se agrega un **try-catch** para que "**acceso_por_indice**" maneje él mismo la excepción.

```
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(e.toString());
        return 0.0;
    } catch (Exception a) {
        System.out.println(a.toString());
        return 0.0;
    }
}
```

EJERCICIO 9

El manejo de excepciones en **Python** y **Java** comparte algunos conceptos fundamentales, pero también hay diferencias significativas. A continuación, se presentan las similitudes y diferencias clave entre Python y Java en cuanto al manejo de excepciones:

| Similitudes | Diferencias |
|---|---|
| **Estructura try:** Ambos lenguajes utilizan una estructura try para capturar y manejar excepciones. Se puede colocar código potencialmente problemático dentro de un bloque try, y las excepciones se capturan y manejan en bloques catch o except correspondientes. | **Declaración de excepciones**: En Java, se requiere que los métodos declaren las excepciones específicas que pueden lanzar utilizando la palabra clave throws en su firma. En Python, no se requiere una declaración explícita de excepciones en la firma de un método. |
| **Tipos de excepciones:** Tanto Python como Java tienen una jerarquía de clases de excepciones predefinidas. Ambos lenguajes proporcionan una serie de excepciones estándar, como NullPointerException, IOException, entre otras, que se utilizan para capturar errores comunes. | **Tipos de excepciones:** En Java, las excepciones se dividen en 2 categorías: comprobadas (checked exceptions) y no comprobadas (unchecked exceptions). Las excepciones comprobadas deben ser declaradas o manejadas explícitamente en el código. En Python, todas las excepciones son consideradas excepciones no comprobadas y no requieren una declaración explícita o manejo. |
| **Bloque finally**: Ambos lenguajes permiten el uso de un bloque finally opcional, que se ejecuta siempre, ya sea que se produzca o no una excepción. Se utiliza para realizar limpieza de recursos u otras acciones necesarias, independientemente de si se lanzó o no una excepción. | **Bloque except**: En Python, se utiliza la palabra clave except para capturar excepciones específicas y manejarlas en un bloque de código correspondiente. En Java, se utiliza la palabra clave catch para capturar y manejar excepciones. |

EJERCICIO 10

**Ruby** proporciona una serie de instrucciones específicas para el manejo de excepciones que permiten capturar, lanzar y manejar errores en el código. A continuación, se describen las instrucciones principales y su comportamiento en Ruby:

| Instrucciones principales | Comportamiento |
|---|---|
| **begin-rescue-end:** Esta es la estructura básica utilizada para capturar y manejar excepciones en Ruby. El código problemático se coloca dentro del bloque begin, y las excepciones se capturan y manejan en el bloque rescue. | `begin`<br>`  # Código problemático`<br>`rescate ExceptionType`<br>`  # Manejo de la excepción`<br>`end`<br><br>ExceptionType puede ser una clase de excepción específica o el tipo<br><br>StandardError para capturar cualquier excepción estándar. |
| **raise**: Esta instrucción se utiliza para lanzar una excepción explícitamente en un punto determinado del código. Puede lanzar una excepción predefinida o una excepción personalizada. | `raise ExceptionType, "Mensaje de error"`<br><br>ExceptionType es la clase de excepción que se lanza.<br><br>"Mensaje de error" es un mensaje opcional que proporciona información adicional sobre la excepción. |
| **rescue**: Esta instrucción se utiliza dentro de un bloque begin-rescue para capturar excepciones específicas. Puede capturar excepciones individuales o agrupar múltiples excepciones. | `begin`<br>`  # Código problemático`<br>`rescue ExceptionType1`<br>`  # Manejo de la excepción de tipo ExceptionType1`<br>`rescue ExceptionType2, ExceptionType3`<br>`  # Manejo de las excepciones de tipo ExceptionType2 y ExceptionType3`<br>`end`<br><br>Se pueden proporcionar múltiples cláusulas rescue para capturar diferentes tipos de excepciones. |
| **ensure**: Esta instrucción se utiliza dentro de un bloque begin-rescue para ejecutar código que se debe ejecutar siempre, independientemente de si se produce o no una excepción. | `begin`<br>`  # Código problemático`<br>`rescue ExceptionType`<br>`  # Manejo de la excepción`<br>`ensure`<br>`  # Código que se ejecuta siempre`<br>`end`<br><br>El bloque ensure se ejecutará incluso si no hay una coincidencia de excepción en el bloque rescue correspondiente. |
| **Excepciones predefinidas**: Ruby proporciona una serie de excepciones predefinidas que se pueden capturar y manejar según sea necesario. | Algunas de las excepciones comunes incluyen StandardError (base de las excepciones estándar), TypeError, ArgumentError, ZeroDivisionError, entre otras. Estas excepciones se pueden capturar y manejar individualmente utilizando la cláusula rescue. |

Además de estas instrucciones y conceptos principales, **Ruby** también ofrece funcionalidades adicionales como la definición de excepciones personalizadas mediante la creación de clases, el acceso a la traza de la pila de excepciones (backtrace), y la posibilidad de propagar excepciones a través de múltiples niveles de llamadas de métodos.

En el caso de **Ruby**, encaja más con el enfoque de "**Terminación**" cuando se produce una excepción. Cuando se lanza una excepción, el bloque donde se produjo la excepción es terminado y se captura en un bloque **rescue**.

EJERCICIO 11

**JavaScript** utiliza un mecanismo de excepciones similar a otros lenguajes de programación. Proporciona una estructura de control try-catch-finally para capturar y manejar excepciones. A continuación se describe el mecanismo de excepciones de JavaScript:

| Instrucciones principales | Comportamiento |
|---|---|
| **try-catch-finally**: Esta estructura se utiliza para capturar y manejar excepciones en JavaScript. El código problemático se coloca dentro del bloque try, y las excepciones se capturan y manejan en el bloque catch. El bloque finally es opcional y se utiliza para ejecutar código que siempre debe ejecutarse, independientemente de si se produce una excepción o no. | `try { // Código problemático } catch (error) { // Manejo de la excepción } finally { // Código que se ejecuta siempre }`<br><br>El bloque catch captura la excepción y la asigna a la variable error (que puede tener cualquier nombre que elijas).<br>El bloque finally se ejecutará incluso si no hay una coincidencia de excepción en el bloque catch correspondiente. |
| **throw**: Esta instrucción se utiliza para lanzar una excepción explícitamente en un punto determinado del código. Puede lanzar una excepción predefinida o una excepción personalizada. | `throw new Error('Mensaje de error');`<br><br>Error es una de las excepciones predefinidas en JavaScript, pero también puedes crear tus propias excepciones personalizadas. |
| **Excepciones predefinidas:** JavaScript proporciona un conjunto de excepciones predefinidas, como Error, TypeError, ReferenceError, SyntaxError, entre otras. | Estas excepciones se pueden capturar y manejar utilizando la estructura try-catch. |
| **Error object**: En JavaScript, todas las excepciones son objetos que heredan de la clase Error. Estos objetos contienen información sobre el error, como el nombre (name: por defecto, “Error"), mensaje de error (message), la traza de la pila (stack) y otros detalles relevantes. | `function dividir(a, b) { if (b === 0) { throw new Error("No se puede dividir por cero"); } return a / b; }`<br><br>`try { console.log(dividir(10, 0)); } catch (err) { console.error("Se atrapó un error:"); console.error("Nombre:", err.name); // "Error" console.error("Mensaje:", err.message); // "No se puede dividir por cero" console.error("Pila:", err.stack); // Traza completa del error }` |

Es importante tener en cuenta que **JavaScript** también proporciona otros mecanismos adicionales para manejar excepciones, como el uso de bloques catch anidados para capturar excepciones específicas. Además, las excepciones en JavaScript pueden propagarse a través de las llamadas de función hasta que se capturen o lleguen al ámbito global, donde pueden generar un error no capturado que detiene la ejecución del programa.

EJERCICIO 12-A

- Si se ingresa 4 y 2: “Resultado 2” (Luego de **dividir**() se ejecuta el **break** y termina)
- Si se ingresa 4 y 0: “No se permite dividir por cero” – “Vuelve a probar” (Al intentar dividir por cerro, se busca la excepción dinámicamente y se ejecuta **except ZeroDivisionError**; y luego **finally**)
- Si se ingresa x e y: se produce un **ValueError** al intentar convertir el valor ingresado a entero, el programa lanzará la excepción sin capturarla y mostrará el rastreo de la pila (traceback) junto con un mensaje de error estándar. En este caso, el bloque finally no se ejecutará y el programa se detendrá.

EJERCICIO 12-B

Se agrega el uso de una excepción anónima utilizando **except Exception**.

```
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 Exception:
        print("Ha ocurrido una excepcion")
    finally:
        print("Vuelve a probar")
```

EJERCICIO 13
EJERCICIO 14