Resolución y explicación de concepto

INTEGRANTES

1. 1501321 - Allan Eduardo Pérez Ajanel
2. 2012421 - Francisco Javier Sánchez Tasej

Manejo de Excepciones

El manejo de excepciones es una técnica que permite detectar y controlar errores que ocurren durante la ejecución de un programa, sin que éste se detenga bruscamente. En lugar de devolver códigos de error, se lanzan excepciones que pueden ser capturadas y gestionadas de manera ordenada.

Generalmente, el manejo de excepciones involucra estructuras como los bloques try y catch (u otros equivalentes según el lenguaje), donde el código susceptible a errores se coloca dentro de un bloque try. Si ocurre un error (llamado excepción), el flujo de ejecución pasa automáticamente al bloque catch, donde se define cómo responder: por ejemplo, mostrando un mensaje, registrando el error o ejecutando acciones alternativas. De esta forma, el manejo de excepciones actúa como una "red de seguridad", aumentando la robustez, mantenibilidad y experiencia de usuario del software

Principios Clave del Manejo de Excepciones

- **Usar excepciones en lugar de códigos devueltos:**
  - En lugar de devolver un código de error que el programador debe verificar, se recomienda lanzar una excepción para separar la lógica del programa del control de errores.
  - Esto hace que el código sea más impio y legible

```python
# USAR EXCEPCIONES EN LUGAR DE CODIGOS DE ERROR
# USO INCORRECTO
def abrir_archivo(nombre):
    if nombre != "datos.txt":
        return -1 # código de error
        return 1 # éxito

resultado = abrir_archivo("archivo.txt")
if resultado == -1:
    print("Error: archivo no encontrado")

# USO CORRECTO
def abrir_archivo(nombre):
    if nombre != "datos.txt":
        raise FileNotFoundError("El archivo no existe.")
    return "Archivo abierto"

try:
    abrir_archivo("archivo.txt")
except FileNotFoundError as e:
    print("Error:", e)
```

- **Separar lógica del manejo de errores:**
  - La idea es escribir el código como si todo fuera a salir bien, y manejar los errores aparte usando try, catch y finally.

```python
# SEPARAR LÓGICA DEL MANEJO DE ERRORES
def dividir(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return "Error: división entre cero"

print(dividir(10, 2)) # 5.0
print(dividir(10, 0)) # Error: división entre cero
```

- **Usar excepciones no comprobadas (unchecked):**

- Las excepciones comprobadas (checked) en Java obligan a cambiar muchos métodos para declarar la excepción.
- Las no comprobadas (unchecked) permiten mantener el código más flexible.

```python
# USAR EXCEPCIONES NO COMPROBADAS
def obtener_precio(producto):
    if producto != "pan":
        raise ValueError("Producto no disponible")
    return 3.5

try:
    print(obtener_precio("leche"))
except ValueError as e:
    print("Error:", e)
```

En Python se puede lanzar errores cuando algo no tiene sentido, todas son no comprobadas

- **Incluir contexto en las excepciones:**
  - Las excepciones deben tener mensajes claros que expliquen el error y ayuden a encontrarlo fácilmente (ej. tipo de error, ubicación, datos).

```python
# INCLUIR CONTEXTO EN EL MENSAJE DE EXCEPCION
def cargar_usuario(nombre):
    if nombre == "":
        raise ValueError("No se puede cargar un usuario con nombre vacío.")
    return {"nombre": nombre}

try:
    cargar_usuario("")
except ValueError as e:
    print("Error:", e)
```

- **Evitar devolver o pasar null:**
  - En lugar de devolver null, es mejor lanzar una excepción o usar un “objeto especial” para evitar errores de tipo NullPointerException.

```python
# NO DEVOLVER NI PASAR NULL
# MAL USO
def buscar_producto(codigo):
    return None # Devolviendo Null

producto = buscar_producto("123")
if producto is not None:
    print(producto.nombre)

# USO CORRECTO
class ProductoNulo:
    nombre = "Producto desconocido"

def buscar_producto(codigo):
    return ProductoNulo()

producto = buscar_producto("123")
print(producto.nombre)
```