# Manejo de Errores

## La forma tradicional

Habitualmente cuando un programador hace algo, tal como llamar a una función, puede evaluar la validez del resultado devuelto por dicha función. Por ejemplo, si intentamos abrir un archivo que no existe, el resultado devuelto podría ser nulo. Hay dos estrategias comunes para manejar este tipo de situaciones:

1. Incluir el código del error en el resultado de la función, o
2. utilizar una variable global con el estatus del error.

En ambos casos, el programador debe revisar si ha ocurrido un error, y en ese caso predecir una acción apropiada que lo maneje.

En BASIC esto puede hacerse del siguiente modo:

```
OPEN "A:\DATA.TXT" FOR INPUT AS #1
IF ERR = 53 THEN
   CALL FileNotFoundError
ELSE
   REM CONTINUAR CON EL PROCESAMIENTO DEL ARCHIVO
END IF
```

Esto produce programas de gran calidad donde casi la mitad del código se ocupa en revisar si todas las acciones se desarrollan normalmente o si producen algún tipo de error. Esto es bastante complicado y vuelve al código bastante difícil de seguir (sin embargo, en la práctica, la mayor parte de los programas actuales hacen precisamente esto). Una estrategia consistente es tratar de evitar los errores considerados "tontos".

## El uso de excepciones

En los entornos de programación más recientes se ha desarrollado una forma alternativa de manejar los errores, conocida como *manejo de excepciones*, la cual funciona generando una excepción tan pronto aparece un error. El sistema fuerza un salto hacia el bloque de excepciones más cercano del código en el cual se toman las acciones apropiadas tendientes a solucionar o alertar acerca del error producido. El sistema provee un "manejador" estándar por defecto que toma todas las excepciones y que muestra los mensajes de error, deteniendo la ejecución del programa.

El bloque que maneja las excepciones se codifica de manera análoga a un bloque `if...then...else`:

```
try:
   # aquí va la instrucción que se desea ejecutar
except TipoDeExcepcion:
   # el proceso de la excepción llamada "TipoDeExcepción" va aquí
except OtroTipo:
   # el proceso de otro tipo de excepción va aquí
else:
   # llegamos aquí si no se ha producido ninguna excepción
```

Hay otro tipo de bloque de excepciones que nos permite "limpiar" todo después que se ha generado un error: `try...finally`. Este bloque se usa típicamente para cerrar archivos, vaciar búferes, etc. El bloque definido por `finally` se ejecuta siempre al final, sin importar lo que haya pasado en la sección `try`.

```
try:
   # lógica normal del programa
finally:
   # aquí "limpiamos" todo independientemente del éxito o la falla en el bloque del try
```

Tcl utiliza un mecanismo similar por medio de la instrucción `catch`:

```
set errorcode [catch {
    unset x
    } msg ]
if {$errorcode != 0} {
    # hacer algo con el error aquí
    }
```

En este caso `x` no existe, por lo cual es imposible aplicarle la función `unset`. Tcl entonces genera una excepción pero la instrucción `catch` evita que el programa se detenga y como resultado coloca un mensaje de error en la variable `msg` y devuelve un resultado distinto de cero (lo cual puede ser definido por el programador). Podemos evaluar el valor de retorno de la instrucción `catch` en `errorcode`. Si es distinto de cero, implica que ha ocurrido un error y deberemos examinar la variable `msg`.

BASIC no incluye funciones para manejar excepciones, pero incluye un constructo que ayuda bastante:

```
100 OPEN "A:\Temp.dat" FOR INPUT AS #1
110 ON ERROR GOTO 10010
120 REM EL CÓDIGO DEL PROGRAMA VA AQUÍ...
130 ...
```

```
10000 REM MANEJADORES DE ERRORES:
10010 IF ERR = 54 THEN....
```

Es importante notar el uso de los números de línea. Esto era muy común en los antiguos lenguajes de programación. Actualmente se puede hacer lo mismo utilizando *etiquetas*:

```
ON ERROR GOTO ErrorCero
REM Ahora creamos un error de "división por cero"
x = 5/0
ErrorCero:
   IF ERR = 23 THEN
      PRINT "No se pude dividir por cero"
      x = 0
      RESUME NEXT
   END IF
```

La instrucción `RESUME NEXT` nos permite volver al lugar del código siguiente al que produjo el error, y por lo tanto, continuar normalmente con el programa.

## Generando errores

¿Qué pasa si deseamos generar nuestras propias excepciones? Simplemente utilizamos la instrucción `raise` en Python:

```
numerador = 42
denominador = input("¿Por cuánto querés dividir a 42?")
if denominador == 0:
   raise "denominador es cero"
```

Esta instrucción genera una excepción del tipo objeto de cadena que puede ser manejada por un bloque `try ... except`.

En Tcl la instrucción `return` puede llevar un flag opcional `-code` que también puede ser tomado por `catch`:

```
proc spam {val} {
        set x $val
        return -code 3 [expr $x]
        }
set err [catch {
               set foo [spam 7]
               } msg]
```

`err` debe tener el valor 3 y `msg` el valor 7. Una vez más este es un caso en el cual la sintaxis de Tcl es menos intuitiva de lo que debería ser.

En BASIC es posible asignar la variable ERR con la instrucción ERROR:

```
ON ERROR GOTO ERRORES
INPUT "INGRESE UN CÓDIGO DE ERROR"; E
ERROR E

ERRORES:
IF ERR = 142 THEN
    PRINT "Se encontró el error número 142"
    STOP
ELSE
    PRINT "No se encontró ningún error"
    STOP
END IF
```

---



Si tenés sugerencias o dudas podés enviar un email en inglés a: alan.gauld@btinternet.com o en español a: manilio@xoommail.com