Tecnología de Programación

Martín L. Larrea

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

Excepciones

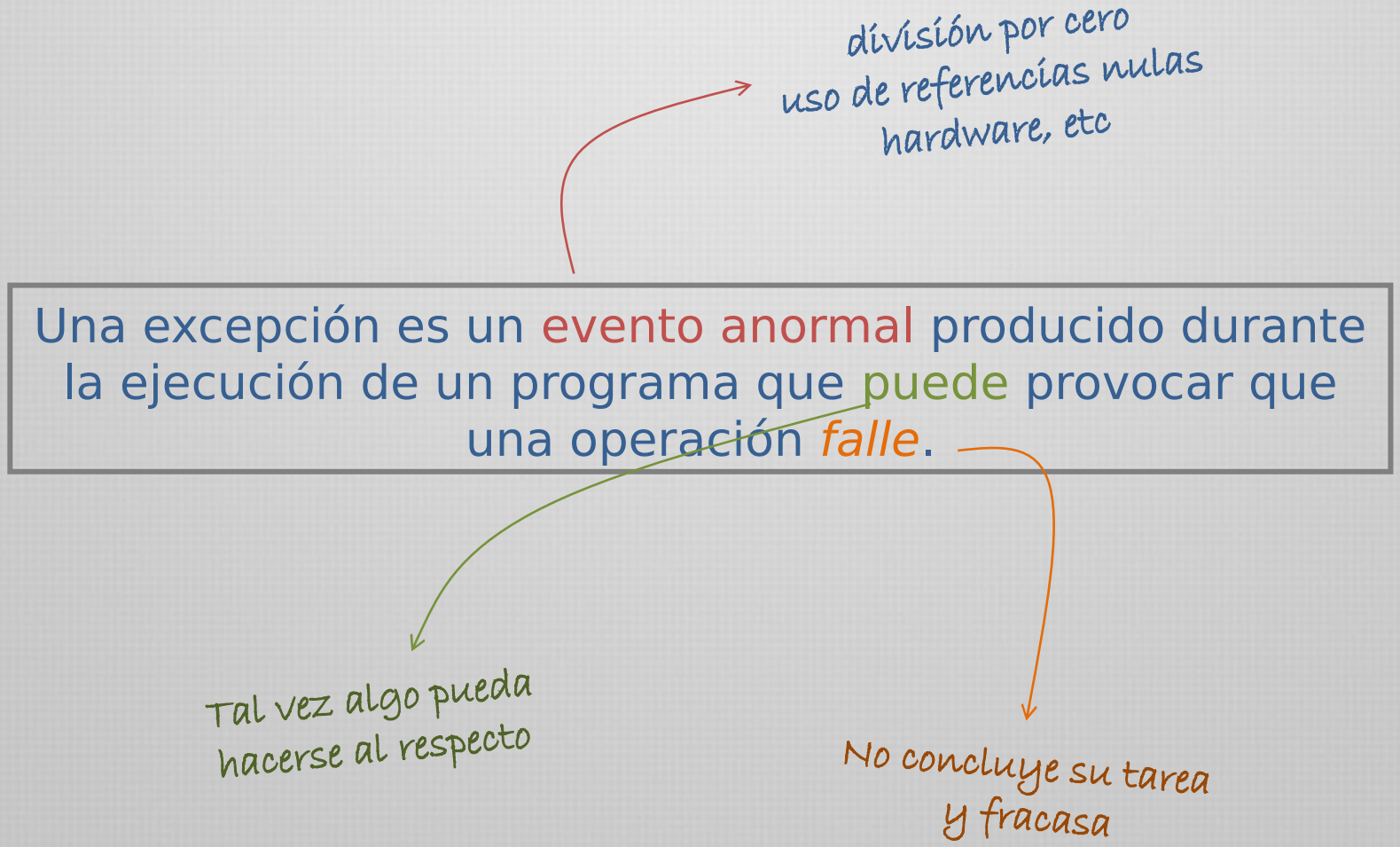
Existen ciertas operaciones que un programa no puede realizar

Pueden carecer de sentido real, como la división por cero
Pueden obtener datos incorrectos por parte del usuario
Pueden depender del hardware y el sistema operativo

Si se encuentra con una operación así, no puede continuar
Lo que sigue puede no tener sentido o verse afectado por la situación

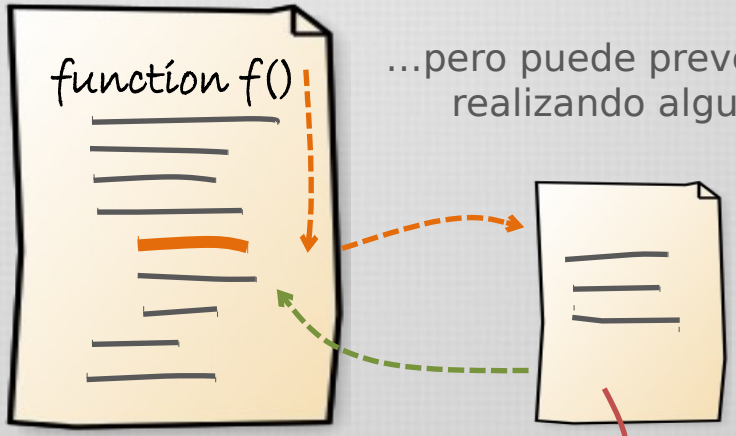
Requieren una acción especial…una excepción en el procesamiento

Excepciones



Excepciones

Una excepción *puede* provocar una falla de la rutina



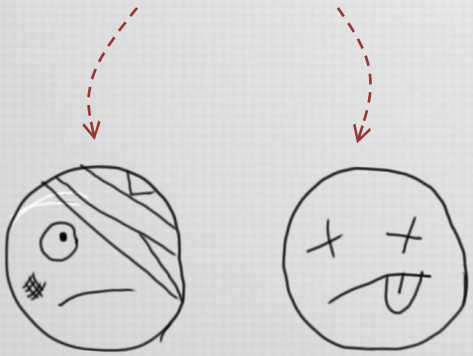
...pero puede prevenirse la falla realizando alguna acción

Exception Handler

segmento de código preparado para activarse ante la ocurrencia de una excepción

Exito, falla y excepción

excepción y falla son conceptos diferentes!



falla ⇒ excepción

excepción ⇏ falla

Para prevenir que una excepción provoque una falla, muchos lenguajes proveen mecanismos para el manejo de excepciones.

Existen diversas formas de tratar las excepciones,
aunque algunas muy similares entre si.
Por ejemplo, muchos lenguajes usan la estructura **`try-catch`**.

Causas de excepciones

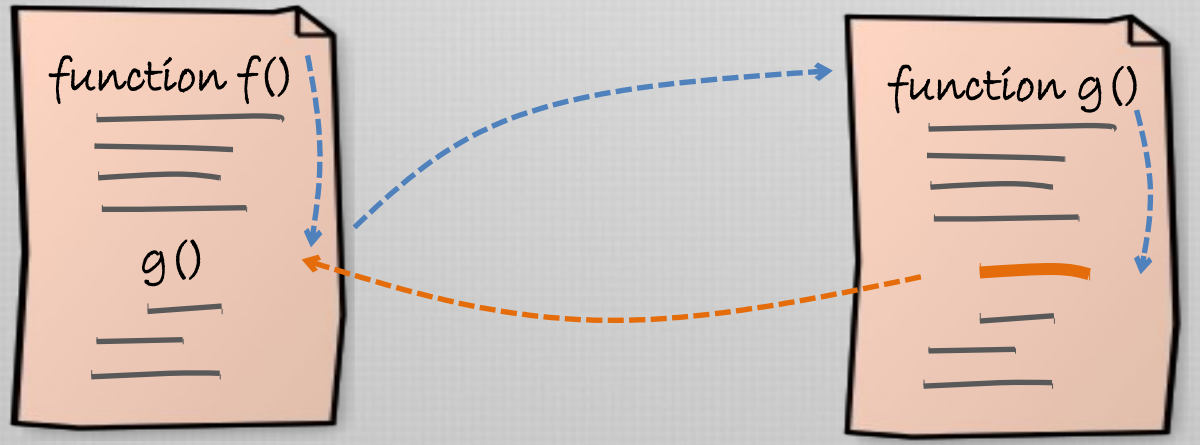
Algunas causas posibles de excepciones son las siguientes:

- Intentar realizar una llamada calificada sobre una referencia *nula.*
- Intentar asignar una referencia nula a una entidad expandida (subobjeto)
- Ejecutar una operación que produce una condición anormal detectada por el HW o Sistema Operativo:
  overflow (stack, arithmetic),
  accesos inválidos a memoria,
  división por cero,
  errores de I/O, etc.
- Invocar a una operación que falle (*"contagio"*).
- Incumplir alguna aserción.
- Ejecutar una operación que provoque explícitamente una excepción.

Falla el servidor, falla en el cliente

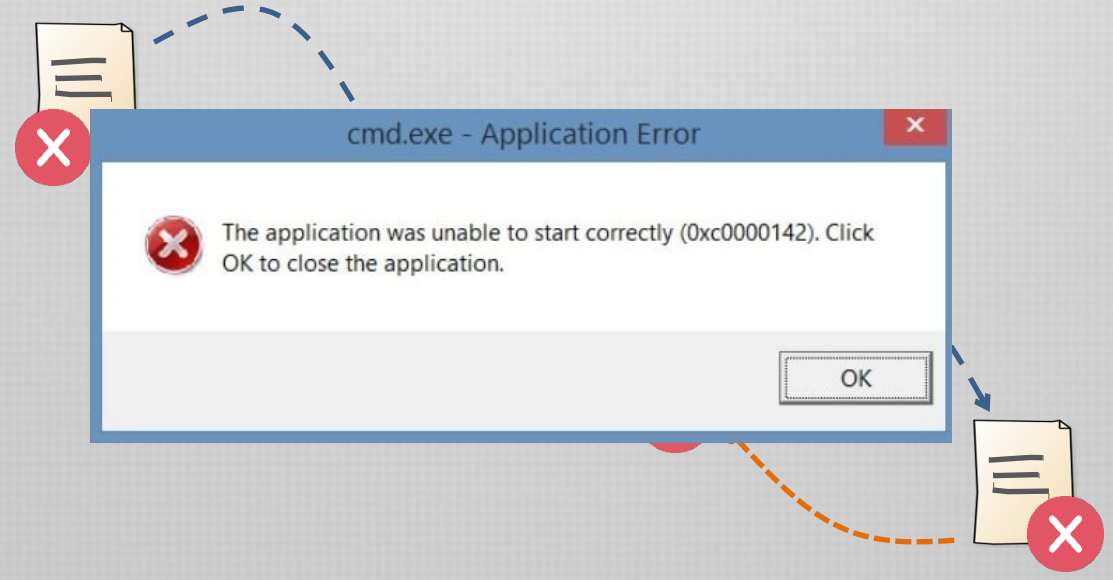
La llamada a una operación *fallará* si
una excepción ocurre durante su ejecución y
dicha operación no se *recupera* de la excepción.

Invocar a una operación que falle produce que quien
invocó a esa operación falle también..



Cadena de llamadas

Para entender los mecanismos de manejo de excepciones es necesario tener en claro el concepto de *cadena de llamadas*.



Uso de las excepciones

El tratamiento de excepciones tiene dos formas en general

Como estructura de control

Se usa una excepción para manejar casos particulares

Ejemplo

tope() con pila vacía

clave no encontrada en tabla hash

Como técnica de último recurso

"last resort"

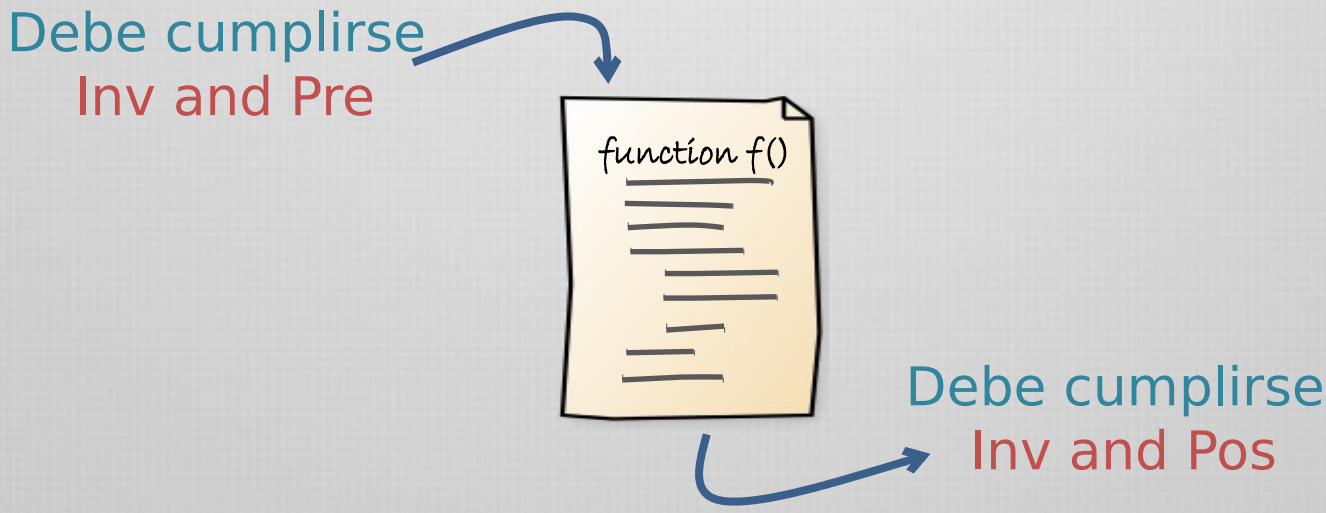
Ejemplo

Input/Output errors

IndexOutOfBounds

Contrato implícito

En el diseño por contrato, toda rutina debe finalizar en un estado que satisface el invariante y la poscondición.



Existe naturalmente una cláusula implícita en este contrato: que la rutina finalice exitosamente, sin interrumpir el flujo de ejecución del sistema

El Diseño por Contrato aporta una visión especial de la falla de una operación o rutina

Exito, falla y excepción

Desde el punto de vista del Diseño por Contrato,
una llamada a una operación
tiene éxito, si la ejecución
finaliza en un estado que satisface el contrato de la operación.

La ejecución de la operación falla si no tiene éxito
(no puede cumplir el contrato)

Por supuesto, una invocación
a una operación supone la finalización con éxito.
Es decir, esa parte del contrato está en todos lados :)

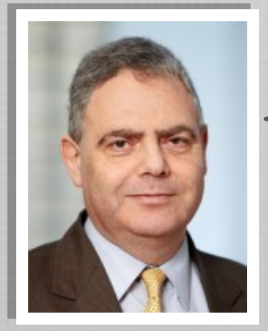
Exception handler

¿Qué puede hacer el manejador de excepciones?

Capturar y continuar una vez capturada la excepción, realizar algunas acciones de reparación y continuar.

Fallar (*pánico organizado*) *limpiar* el entorno, terminar la llamada y propagar la falla al llamador

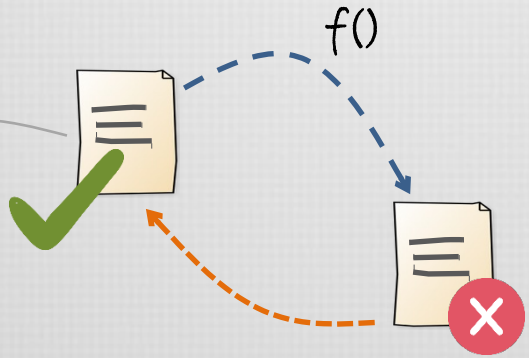
Reintentar tratar de cambiar las condiciones que llevaron a una excepción y ejecutar la rutina nuevamente desde el comienzo, tal vez con una estrategia diferente.



Bertrand Meyer

las dos únicas respuestas legítimas a una excepción en la política de manejo disciplinado de excepciones.

Exception handler



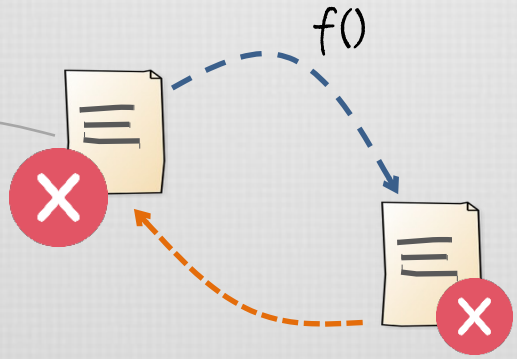
Una falla en un mensaje ocasiona el *disparo* de una excepción en el objeto cliente

Si existe un *manejador de excepciones*, entonces tomará el control tratando de solucionar el problema

Si puede solucionar los problemas generados por el evento ocurrido, puede intentar la recuperación.

intenta cambiar las condiciones que generaron la excepción, para luego terminar normalmente la rutina.

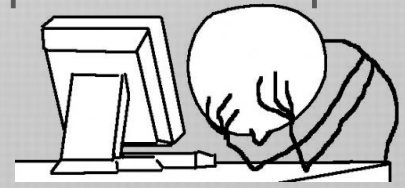
Exception handler



Una falla en un mensaje ocasiona el *disparo* de una excepción en el objeto cliente

Si no es posible la recuperación, entonces la operación falla, lo que genera una excepción en el objeto cliente
(pánico organizado)

Una operación falla si durante su ejecución se levanta una excepción y el manejador
no puede recuperarse.



Manejo de excepciones en lenguajes

Muchos lenguajes implementan manejadores de excepciones en forma similar

Lenguajes como *Java, PHP, C++, C#* ofrecen
una manera de agrupar sentencias riesgosas a ejecutar,
y
una forma de capturar posibles excepciones
que surjan de estas sentencias
e indicar las acciones a realizar (*handler*)

En este modelo una operación puede tener asociados
más de un manejador de excepciones.

Excepciones en Eiffel

Eiffel tiene un tratamiento diferente,
coherente con la noción de Diseño por Contrato
que se procura adherir.

Incluye dos formalismos especiales:

`rescue`
define un bloque de sentencias que describen cómo intentar recuperarse de un evento no deseado en tiempo de ejecución (handler).

`retry`
es la sentencia del manejador de excepciones que indica *reintentar* la operación desde la primer instrucción.

La zona del **`rescue`** es única y figura al finalizar el cuerpo del servicio.

Excepciones en Eiffel

```
connect_to_server (server: SOCKET)
      -- conectar a un servidor
    require
        server /= Void and then server.address /= Void
    local
        attempts: INTEGER
    do
        server.connect
    ensure
      connected: server.is_connected
    rescue
        if attempts < 10 then
            attempts := attempts + 1
            retry
        end
    end
```

Excepciones en Eiffel

La sentencia **retry** sólo puede aparecer en la zona de **rescue**

La ejecución del **retry** consiste en recomenzar la operación desde la primer sentencia,
sin realizar las inicializaciones nuevamente.

Cualquier ejecución del **rescue** que no lleva a una sentencia **retry** causa que la operación falle.

Si al ejecutar el **retry** no volvemos a *caer* en el **rescue**, entonces la operación termina exitosamente.

No pueden devolverse resultados en la cláusula **rescue**

Excepciones en Eiffel

Un ejemplo simple: solicitar un entero y reintentar si hubo un error.

```
get_integer is
        --   solicitar un entero. Si hubo error,
                  reintentar    indefinidamente.
do
        print ("Please enter an integer: ")
        read_one_integer
rescue
        retry
end
```

Excepciones en Eiffel

Solicitar un entero y reintentar una cierta cantidad de veces, en caso de haber un error.

```
get_integer is
        --   solicitar un entero. Si hubo error,
                  reintentar indefinidamente.
local
        attempts: INTEGER
do
        if attempts < Maximum_attempts then
                print ("Please enter an integer: ")
                read_one_integer
                integer_was_read := True
        else
                integer_was_read := False
                attempts := attempts + 1
        end
rescue
        retry
end
```

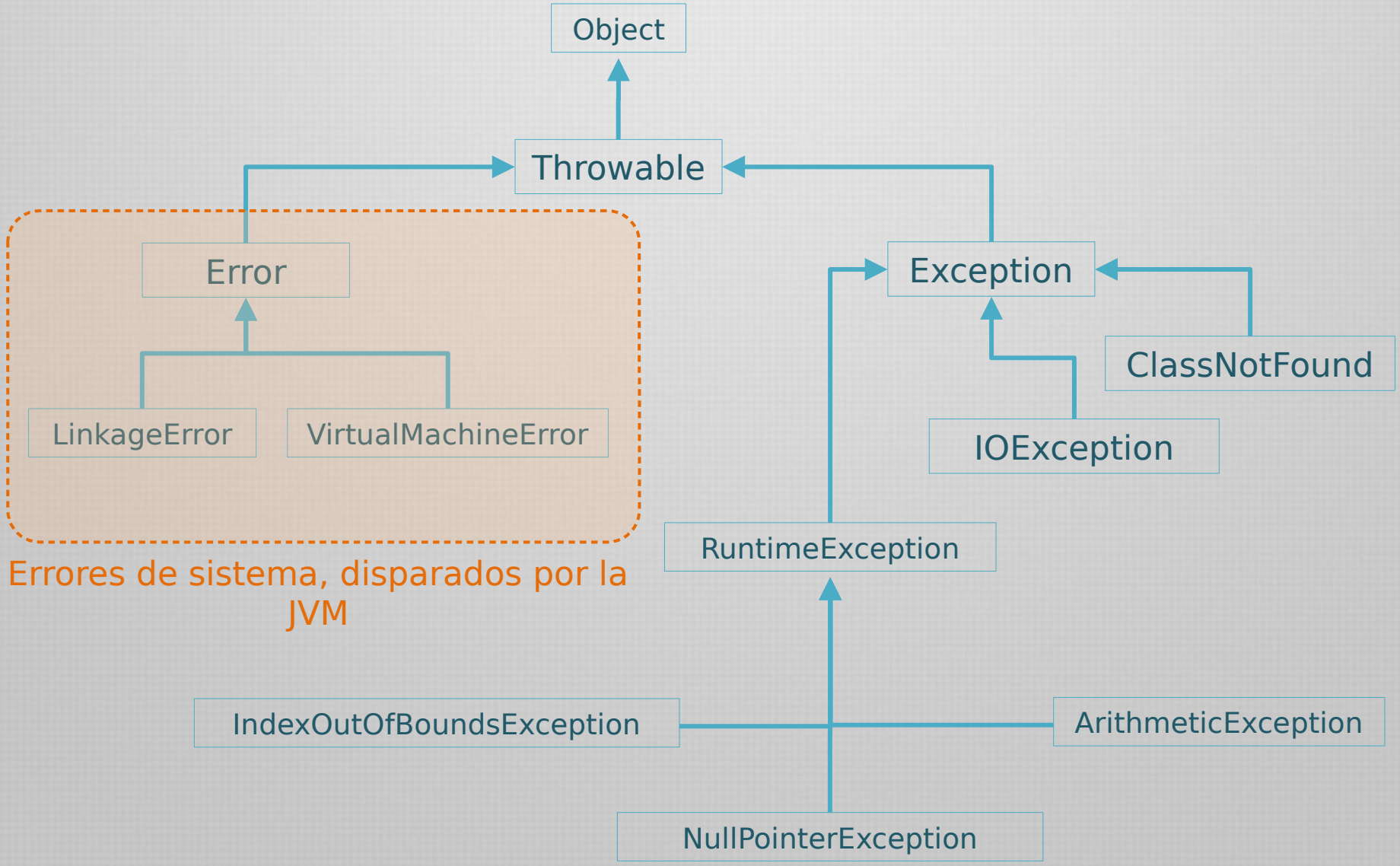
¿esta operación puede fallar?

Excepciones en Eiffel

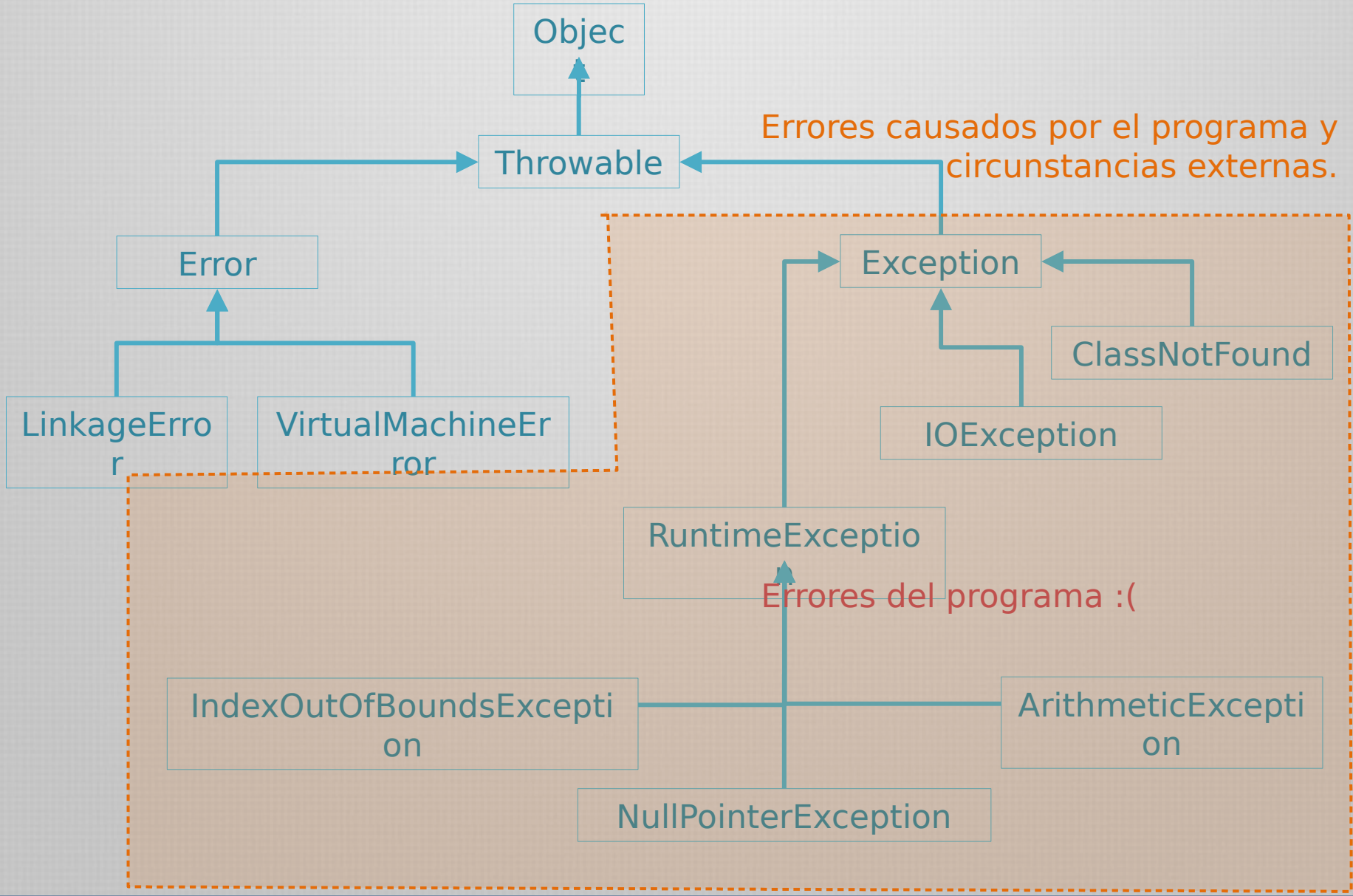
```
get_integer is
        --   solicitar un entero. Si hubo error,
                  reintentar indefinidamente.
local
        attempts: INTEGER
do
            print ("Please enter an integer: ")
            read_one_integer


rescue
        attempts := attempts + 1
        if attempts < Maximum_attempts then
            retry
end
```

Java - errores y excepciones



Java - errores y excepciones



Java - checked exceptions

RunTimeException y *Error* son excepciones *unchecked*.

usualmente errores en la lógica del programa.

El resto son *checked exceptions*.

El programador debe controlar su ocurrencia y actuar apropiadamente.

Cada método debe declarar qué excepciones *chequeadas* puede propagar:

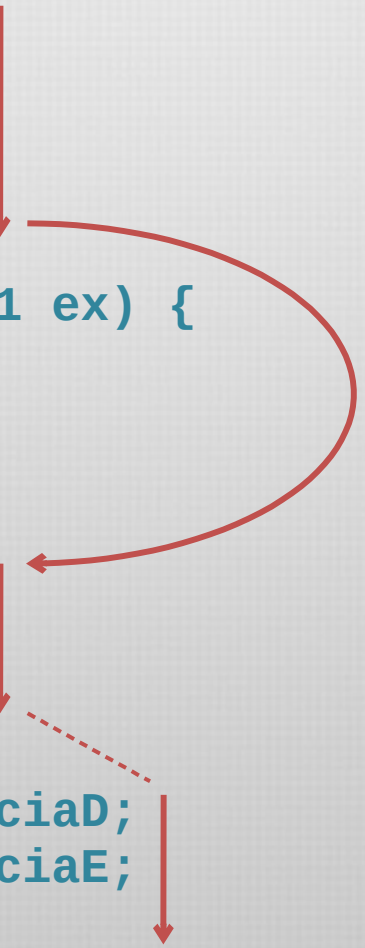
```
public void unMetodo() throws IOException, OtherException
{
...
}
```

Esto obliga al cliente del método a *capturar (catch)* dicha excepción, o *declararla* de manera similar.

Java - finally

```
try {
   sentenciaA;
   sentenciaB;
   sentenciaC;
}
catch(Exception1 ex) {
   sentenciasX;
   sentenciasX;
}
finally {
   sentenciasY;
   sentenciasY;
}

siguienteSentenciaD;
siguienteSentenciaE;
```



Java - finally