3- Excepciones y manejo de excepciones

1. Introducción

Dentro del software, existen 3 categorías en la ocurrencia de un error:

- **Errores de sintaxis**: violación de las reglas sintácticas del lenguaje de programación (omitir la puntuación, escribir mal una palabra clave, etc.). En los lenguajes compilados, el compilador detecta estos errores en tiempo de compilación. En lenguajes interpretados, el error se da al intentar ejecutar la sentencia inválida.
- **Errores de ejecución**: no se detectan al compilar. Son operaciones sintácticamente correctas pero que no se puede realizar (por ejemplo, dividir por 0).
- **Errores lógicos**: el código puede compilarse y ejecutarse, pero hay discrepancia entre resultado obtenido y esperado (no hace lo esperado).

2. Las excepciones y su manejo

Desde un punto de vista teórico, se pueden establecer características que debe cumplir un mecanismo de manejo de excepciones:

- Debe ser simple, fácil de entender y de usar.
- No debe oscurecer el flujo normal del programa en ausencia de error.
- No debe haber sobrecarga en las condiciones normales de operación.
- El tratamiento de las excepciones debe ser uniforme, con independencia de si son detectadas por el entorno, o por el programa.
- Debe permitir la programación de acciones de recuperación.

En ADA, las excepciones se definen como constantes que deben ser declaradas explícitamente, mientas que en JAVA son objetos de un tipo particular que pueden o no haber sido declarados explícitamente.

Dominio de una excepción

Dentro de un programa puede haber diferentes manejadores para una excepción en particular. El **dominio de una excepción** es la zona del código en la que será activado el manejador si se lanza la excepción. La precisión con la que se puede definir un dominio afecta a la detección de la fuente de error.

En un lenguaje estructurado en bloques como ADA, el dominio es normalmente el bloque. En JAVA no todos los bloques pueden tener manejador de excepciones, hay que definir explícitamente el bloque en el que se podrá lanzar una excepción (pueden actuar como dominios los procedimientos, funciones y otros tipos de bloques).

```
declare
   subtype Temperature is Integer range 0 .. 100;
begin
  -- read temperature sensor and calculate its value
exception
  -- handler for Constraint_Error
end;
```

```
try {
   // statements which may raise exceptions
} catch (ExceptionType e) {
   // handler for e
}
```

Propagación de las excepciones

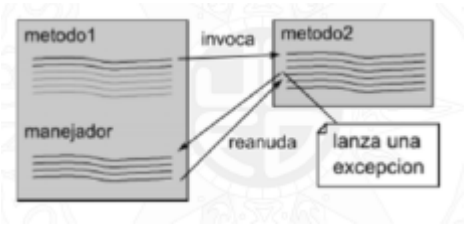
Tras la ocurrencia de una excepción, es necesario determinar qué manejador es el encargado de tratarla, acudiendo al concepto de dominio. Si no existe un manejador definido para la excepción actual, hay que determinar qué ocurre con la excepción. Hay dos posibles enfoques en relación a la propagación de excepciones:

- **Dinámico**: propagación de excepciones hacia atrás en la cadena de invocación. Esta búsqueda se realiza en tiempo de ejecución. Existe la posibilidad de que una excepción se propague fuera de su ámbito de visibilidad, por lo que se utilizan manejadores por defecto que atrapan cualquier tipo de excepción.
- **Estático**: los manejadores deben ser definidos en el contexto en que se lanza la excepción. Si no existe un manejador adecuado, se genera un error de compilación, ya que se considera que el programador debería

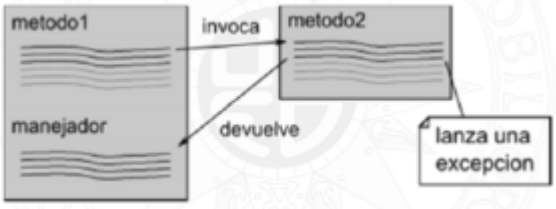
haberlo declarado. El problema es que una excepción sólo puede ser manejada en el contexto del procedimiento invocado.

Tras la ocurrencia de una excepción, debe saberse qué ocurre con el flujo de ejecución después de haber tratado la excepción. Hay diferentes modelos en el tratamiento de excepciones:

- **Modelo de continuación o reanudación**: se basa en asumir que el manejador de excepciones ha sido capaz de reparar el problema que causó la excepción. Restaurado el estado válido, el invocador puede continuar la ejecución. Este modelo se asimila a llamar a una función implícita que cuando termina, permite continuar la ejecución normal.
  La dificultad de implementación puede no compensar las ventajas de su uso, y que el flujo de control de un programa pueda ser difícil de seguir, y por tanto más propenso a errores.



- **Modelo de terminación**: el invocador no retoma el control del flujo del programa, finalizando entonces la ejecución del código dentro del cual se produjo la excepción (Java, Ada o C++).



- **Modelo híbrido**: modelo que es posible definir para el tratamiento de excepciones, el cual comparte características de los dos modelos anteriores, donde el propio manejador será quien si continúa o se termina la operación que causó la excepción.

Ventajas y desventajas de usar excepciones

El uso de excepciones presenta ventajas:

- En los lenguajes de programación tradicionales, no hay diferenciación entre el código normal y de tratamiento de errores, dando lugar a código difícil de entender o erróneo. El uso de excepciones proporciona un método limpio y simple para separar el flujo normal de ejecución de situaciones anormales o erróneas.
- No es necesario maquinaria extra para propagar tanto los casos de éxito como de error al invocador. Sólo el método interesado de gestionar el error debe preocuparse de definir el código de recuperación.
- En lenguajes orientados a objetos, conlleva un agrupamiento jerárquico de los tipos de error, lo que permite una mejor representación del dominio del problema y el tratamiento de los errores.

Y también desventajas

- Para poder implementar el mecanismo de gestión de excepciones, el compilador introduce código extra que debe ser verificado durante las pruebas de la aplicación, lo que incrementa el esfuerzo y coste del desarrollo.
- Dificulta la escritura de código robusto, ya que altera el flujo de ejecución está determinado por la ocurrencia de excepciones, afectando a la comprensión del código.

3. Manejo de excepciones en ADA

ADA soporta declaración explícita de excepciones, modelo de terminación, propagación de excepciones y un paso de parámetros limitado.

- La excepción se declara con la palabra clave `exception`,
- con el paquete predefinido `Ada.Exceptions`, que define un tipo privado `Exception_Id`.
- Todas las excepciones declaradas con `exception` tienen asociado un `Exception_Id`, que se puede acceder con el atributo predefinido `Identity`.

```
package Ada.Exceptions is
  type Exception_Id is private;
  Null_Id : constant Exception_Id;

  function Exception_Name(Id : Exception_Id) return String;

  type Exception_Occurrence is limited private;
  Null_Occurrence : constant Exception_Occurrence;

  procedure Raise_Exception(E : in Exception_Id;
Message : in String := "");
  function Exception_Message(X :
 Exception_Occurrence) return String;
  procedure Reraise_Occurrence(X : in
 Exception_Occurrence);
  function Exception_Identity(X : Exception_Occurrence)
 return Exception_Id;
  function Exception_Name(X : Exception_Occurrence)
return String;
  function Exception_Information(X :
 Exception_Occurrence) return String;
  ....

private
  ... -- not specified by the language
end Ada.Exceptions;
```

Las excepciones también pueden lanzarse de forma explícita con `raise`. Así, si `IO_Error` es de tipo `Exception_Id`, se usa `Ada.Exceptions.Raise_Exception`, para pasar una cadena de texto como parámetro.

```
begin
  ...
  -- statements which request a device to
  -- perform some I/O
  if IO_Device_In_Error then
     raise IO_Error;
  end if; -- no else, as no return from raise
  ...
end;
```

Cada señalización de una excepción es una **ocurrencia**. El manejador puede obtener el valor de `Exception_Occurrence` y usarlo para conseguir más información sobre la causa de la excepción.

Los manejadores pueden declararse al final del bloque. Cada manejador es una secuencia de sentencias

```
declare
   Sensor_High, Sensor_Low, Sensor_Dead : exception;
begin
   -- statements which may cause the exceptions
exception
   when E: Sensor_High | Sensor_Low =>
    -- Take some corrective action
    -- if either sensor_high or sensor_low is raised.
    -- E contains the exception occurrence
   when Sensor_Dead =>
    -- sound an alarm if the exception
    -- sensor_dead is raised
end;
```

La palabra clave when_others se utiliza para no tener que enumerar todos los tipos posibles. Solo se permite como la última opción.

```
declare
  Sensor_High, Sensor_Low, Sensor_Dead: exception;
begin
  -- statements which may cause exceptions
exception
  when Sensor_High | Sensor_Low =>
    -- take some corrective action
  when E: others  =>  // last wishes
    Put(Exception_Name(E));
    Put_Line(" caught. Information is available is ");
    Put_Line(Exception_Information(E));
    -- sound an alarm
end;
```

Si no hay manejador en el bloque, la excepción se propaga de la siguiente forma:

- Para un bloque, se genera en el bloque que lo contiene o en el subprograma.
- Para un subprograma, en el punto de invocación al mismo.
- Para una sentencia `accept` tanto en el invocador como en la tarea que se invoca.

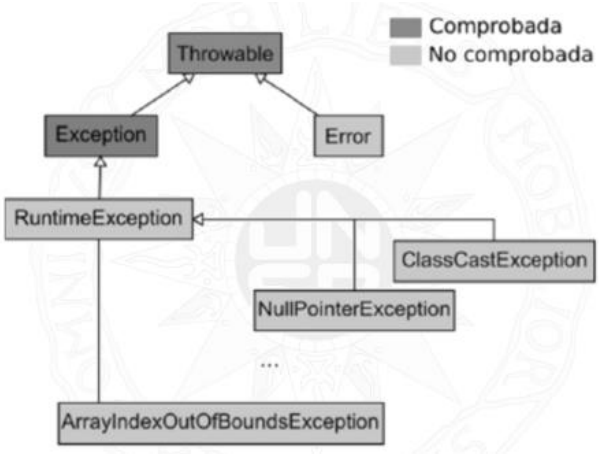
Inconvenientes del modelo de excepciones en ADA

En relación al uso de excepciones y paquetes, las excepciones que pueden ser generadas por un paquete se declaran en su especificación, por lo que no se sabe qué subprogramas pueden lanzar unas u otras. Además, el programador debe enumerar todos los tipos o usar `when_others`.

Para el paso de parámetros, ADA solo permite cadenas de texto. En cuanto a su visibilidad, las excepciones pueden propagarse fuera de su ámbito de declaración (solo pueden manejarse por `when_others`).

4. Manejo de excepciones en JAVA

Java soporta un modelo de terminación, integrado en el modelo de la programación OO, donde todas las excepciones son subclases de `java.lang.Throwable`, que se especializa a su vez en `Exception` y `Error`. Por tanto, en Java se consideran los siguientes tipos de excepciones de las que el programa podría querer recuperarse, dependiendo de di el compilador comprueba la existencia de manejadores asociados:



- **Excepciones comprobadas**: son condiciones excepcionales que están sujetas a la comprobación por parte del compilador, y cumplen con el requisito *try-catch*. Si el método contiene código que puede lanzar algún tipo de excepción se gestiona internamente en el método o se especifica la excepción en la definición del mismo mediante la cláusula throws. Son todas aquellas clases que heredan de `Throwable` y a su vez no sean ni `RuntimeException` ni `Error`.
- **Excepciones no comprobadas**: son condiciones excepcionales o de error que el compilador ni verifica ni se espera (queda en manos del programador que se atrapen). Entre ellas están los objetos `Error` o `RuntimeException` y sus subclases.

Encadenamiento de excepciones

Este mecanismo es usado para la gestión de excepciones, ya que un manejador puede responder a la ocurrencia de una excepción generando otra excepción, por lo que resulta útil mantener la información sobre qué excepción provocó la actual. Se utiliza para encapsular una excepción, aportando información adicional y añadiendo capas de abstracción.

De esta forma, en lugar de propagar a niveles superiores los errores producidos en las capas de menor nivel, se puede hacer una traducción y adaptar el significado a las abstracciones de la capa superior.

5. Conclusiones finales

La mayoría de los lenguajes utilizan un modelo de terminación, ya que los beneficios del uso del modelo de continuación no compensan las dificultades de implementación (aunque existen lenguajes que utilizan un modelo híbrido). Igualmente, la mayoría de los lenguajes utilizan un dominio de bloque.

Lenguaje	Dominio	Propagación	Modelo	Parámetros
Ada	Bloque	Si	Terminación	Limitado
Java	Bloque	Si	Terminación	Si
C++	Bloque	Si	Terminación	Si
CHILL	Sentencia	No	Terminación	No
CLU	Sentencia	No	Terminación	Si
MESA	Bloque	Si	Híbrido	Si

Aunque la mayoría de lenguajes utilizan un mecanismo de excepciones, no está uniformemente aceptado que el mecanismo de gestión deba ser proporcionado por el lenguaje (por ejemplo, C no lo tiene), ya que pueden asimilarse al uso de sentencias GO TO donde el destino no se puede determinar, y el origen es desconocido, por lo que puede ser considerado como la antítesis de la programación estructurada.