

1. Introducción a los computadores y su programación
2. Elementos básicos del lenguaje
3. Modularidad y programación orientada a objetos
4. Estructuras de datos dinámicas
- 5. Tratamiento de errores**
6. Abstracción de tipos mediante unidades genéricas
7. Entrada/salida con ficheros
8. Herencia y polimorfismo
9. Programación concurrente y de tiempo real

5.1. Excepciones

Representan circunstancias anormales o de error

Ventajas de este mecanismo:

- El código de la parte errónea del programa está separado del de la parte correcta
- Si se te olvida tratar una excepción, te das cuenta al probar el programa, porque éste se detiene
- Se puede pasar la gestión del error de un módulo a otro de modo automático
 - agrupar la gestión del error en el lugar más apropiado

Las excepciones pueden ser predefinidas, o definidas por el usuario

5.2. Excepciones predefinidas

En el lenguaje:

- **Constraint_Error** : variable fuera de rango, índice de array fuera de rango, uso de un puntero nulo para acceder a un dato, etc.
- **Program_Error**: programa erróneo o mal construido; por ejemplo, ocurre si una función se acaba sin haber ejecutado su instrucción **return**. No se trata.
- **Storage_Error**: Memoria agotada. Ocurre tras una operación **new**, o al llamar a un procedimiento si no hay espacio en el stack.
- **Tasking_Error**: Error con las tareas concurrentes

Excepciones predefinidas (cont.)

En librerías estándares hay muchas

Un par de ellas que son frecuentes:

- **Ada.Numerics.Argument_Error**: Error con un argumento de las funciones elementales (p.e., raíz cuadrada de un número negativo)
- **Ada.Text_IO.Data_Error**: Formato de dato leído es incorrecto
 - p.e., si se leen letras pero se pretende leer un entero

5.4. Declaración de excepciones propias

Las excepciones definidas por el usuario se declaran como variables especiales, del tipo `exception`

```
nombre : exception;
```

5.3. Elevar y tratar excepciones

Las excepciones se elevan, para indicar que ha habido un error:

- las predefinidas se elevan automáticamente
- las definidas por el usuario se elevan mediante la instrucción `raise`:

```
raise nombre_excepcion;
```

Una excepción que se ha elevado, se puede tratar

- Esto significa, ejecutar un conjunto de instrucciones apropiado a la gestión del error que la excepción representa

5.4. Tratamiento de excepciones

Se hace dentro de un bloque:

```
encabezamiento  
  declaraciones  
begin  
  instrucciones  
exception  
  manejadores  
end;
```

Los manejadores gestionan las excepciones que se producen en las instrucciones del bloque

Manejadores de excepción

Los manejadores se escriben así:

```
when Nombre_Excepcion =>  
  instrucciones;
```

Se pueden agrupar varias excepciones en el mismo manejador

```
when excepcion1 | excepcion2 | excepcion3 =>  
  instrucciones;
```

Se pueden agrupar todas las excepciones no tratadas en un único manejador final

```
when others =>  
  instrucciones;
```

- es peligroso ya que puede que el tratamiento no sea adecuado a la excepción que realmente ocurre

5.5. Funcionamiento de las excepciones

Cuando una excepción se eleva en un bloque:

a) Si hay un manejador:

- se abandonan las restantes instrucciones del bloque
- se ejecuta el manejador
- se continúa por el siguiente bloque en la secuencia de programa

b) Si no hay manejador

- se abandona el bloque
- se eleva la misma excepción en el siguiente bloque en la secuencia del programa

Funcionamiento de las excepciones (cont.)

El siguiente bloque en la secuencia es:

- para un subprograma, el lugar posterior a la llamada
- para un bloque interno (**declare-begin-end**) la instrucción siguiente al **end**
- para el programa principal, nadie
 - el programa se detiene con un mensaje de error

Creación de bloques para tratar excepciones

Si se desea continuar ejecutando las instrucciones de un bloque donde se lanza una excepción, es preciso crear un bloque más interno:

```
begin
  begin  -- del bloque interno
    instrucciones que pueden fallar
  exception
    manejadores
  end;   -- del bloque interno
  instrucciones que es preciso ejecutar, aunque haya fallo
end;
```

5.6. Formas más habituales de tratar excepciones

Dependen de la gravedad del error

- a) ignorar
- b) indicar el error y luego seguir
- c) reintentar la operación (sólo si el error es recuperable)
- d) abandonar la operación

A continuación veremos ejemplos de cada una

a) Ignorar

Escribir una operación, `Pon_Mensaje_Doble`, que pone un mensaje en el terminal normal, y en un terminal remoto.

Para el terminal remoto existe el paquete:

```
package Terminal_Remoto is
  procedure Pon_Mensaje (Str : String);
  No_Conectado : exception;
end Terminal_Remoto;
```

`Pon_Mensaje` eleva `No_Conectado` si el terminal no está conectado.

`Pon_Mensaje_Doble` debe ignorar este error si se produce, pero siempre escribir el mensaje en el terminal principal

Implementación

```
with Ada.Text_IO, Terminal_Remoto;
procedure Pon_Mensaje_Doble (Str : String) is
begin
  Ada.Text_IO.Put_Line(Str);
  Terminal_Remoto.Pon_Mensaje(Str);
exception
  when Terminal_Remoto.No_Conectado =>
    null; -- para no hacer nada hay que decirlo
end Pon_Mensaje_Doble;
```

Observar que si no se pone manejador, en lugar de ignorar la excepción ésta se eleva en el siguiente bloque

b) Indicar el error

Se desea escribir una operación para mostrar los datos de un alumno, extrayéndolos de una lista

```
with Alumnos; -- lugar donde se declara el tipo Alumno
package Listas is
  type Lista is private;
  procedure Busca_Alumno(A1 : in out Alumnos.Alumno;
                        L : in Lista);
  procedure Inserta_Alumno(A1 : in Alumno; L : in out Lista);
  ...
  No_encontrado : exception;
  No_Cabe : exception;
end Listas;
```

Busca_Alumno busca el alumno por el nombre, y eleva **No_Encontrado** si no lo encuentra

Implementación

El tipo **Alumno** contiene el nombre y datos personales del alumno

```
with Ada.Text_IO, Listas, Alumnos;
procedure Muestra_Alumno(L : Lista) is
  A1 : Alumnos.Alumno;
begin
  Leer A1.nombre;
  Listas.Busca_Alumno(A1, L);
  Muestra datos del alumno;
exception
  when Listas.No_Encontrado =>
    Ada.Text_IO.Put_Line("Alumno no encontrado");
end Muestra_Alumno;
```

Las instrucciones correspondientes a “mostrar los datos del alumno” se saltan si hay un error

- eso es lo que se desea

En el código, quedan claramente separadas las instrucciones normales, de las que corresponden al caso con error

c) Reintentar

Un caso típico de operación que se reintentada es la lectura errónea de datos del teclado

Se desea escribir un procedimiento para leer un entero

- primero pone un mensaje en la pantalla
- si la lectura falla, se reintentada
- si el entero no está comprendido entre un límite inferior y un límite superior, se reintentada

Implementación

```
with Ada.Text_Io,Ada.Integer_Text_Io;
use Ada.Text_Io,Ada.Integer_Text_Io;
procedure Lee_Entero (Mensaje           : in String;
                     Num                : out Integer;
                     Lim_Inf, Lim_Sup  : in Integer) is
  I : Integer range Lim_Inf..Lim_Sup;
begin
  loop
    begin
      Put (Mensaje);
      Get (I);
      Skip_Line;
      Num:=I;
      exit;
    exception
      when Constraint_Error | Data_Error =>
        Skip_Line; Put_Line("Error al leer");
    end;
  end loop;
end Lee_Entero;
```

Programa de prueba

```
with Lee_Entero;
procedure Prueba_Entero is
  I : Integer range 2..20;
begin
  for J in 1..5 loop
    Lee_Entero("entero : ",I,2,20);
    -- no hace falta tratar Constraint_Error ni Data_Error,
    -- pues ya están tratadas dentro de Lee_Entero
  end loop;
end Prueba_Entero;
```

Comentarios

- creamos un tipo entero con el rango deseado, para que se eleve `Constraint_Error` automáticamente si estamos fuera de límites
- vale el mismo manejador para las dos excepciones

d) Abandonar

Creamos una operación para leer los datos de un alumno y luego insertarlo en una lista del paquete `Listas` anterior

`Inserta_Alumno` eleva `No_Cabe` si el alumno no cabe

La operación nueva debe abandonarse si ocurre el error, pero debe notificar este error al programa principal

- elevaremos para ello la misma excepción

Implementación

```
with Ada.Text_IO, Listas, Alumnos;
procedure Nuevo_Alumno(L : Lista) is
  Al : Alumnos.Alumno;
begin
  Leer todos los datos de Al;
  Listas.Inserta_Alumno(Al, L);
  Muestra confirmación de los datos insertados;
exception
  when Listas.No_Cabe =>
    Ada.Text_IO.Put_Line("Alumno no cabe");
    raise;
end Nuevo_Alumno;
```

Comentarios

- La instrucción `raise` “a secas” eleva la misma excepción que se produjo

5.7. Paquete Ada.Exceptions (1/2)

```
package Ada.Exceptions is

  type Exception_Id is private;
  Null_Id : constant Exception_Id;

  function Exception_Name
    (X : Exception_Id) return String;

  type Exception_Occurrence is
    limited private;

  function Exception_Message
    (X : Exception_Occurrence)
    return String;

  function Exception_Identity
    (X : Exception_Occurrence)
    return Exception_Id;
```

Paquete Ada.Exceptions (2/2)

```
function Exception_Name
  (X : Exception_Occurrence)
  return String;

function Exception_Information
  (X : Exception_Occurrence)
  return String;

...

private
  ...
end Ada.Exceptions;
```

Ejemplo de uso de Ada.Exceptions (1/2)

```
with Ada.Exceptions,Ada.Text_Io;
use Ada.Exceptions,Ada.Text_Io;
procedure Prueba_Exceptions is
  procedure Eleva_Excepcion is
    Opcion : Character;
  begin
    -- permite simular una excepción
    Put_Line("1-elevar Constraint_Error");
    Put_Line("2-elevar Data_Error");
    Put_Line("3-elevar Storage_Error");
    Put("Introduce opcion : ");
    Get(Opcion); Skip_Line;
    case Opcion is
      when '1'=> raise Constraint_Error;
      when '2'=> raise Data_Error;
      when '3'=> raise Storage_Error;
      when others => null;
    end case;
  end Eleva_Excepcion;
```

Ejemplo de uso de Ada.Exceptions (2/2)

```
Car : Character;

begin -- de prueba_exceptions
  loop
    begin
      -- acciones normales a repetir
      Eleva_Excepcion;
    exception
      when Error : others =>
        Put_Line("Se ha elevado "&Exception_Name(Error));
        Put_Line("Mensaje      : "&Exception_Message(Error));
        Put_Line("Información : "&Exception_Information(Error));
        New_Line;
        Put("¿Desea continuar el programa? (s/n)");
        Get(Car); Skip_Line;
        exit when Car/='S' and Car/='s';
      end;
    end loop;
  end Prueba_Exceptions;
```