Grado en Ingeniería de Computadores

Sistemas en tiempo real

L3. Programación a gran escala

Departamento de Automática

Universidad de Alcalá

23 de septiembre del 2019

Índice

Paquetes [1]

Los paquetes exportan, mediante una interfaz bien definida:

- **Tipos**
- **Objetos** ("variables")
- **Operaciones**

La utilización de paquetes permite ocultar su implementación

[1] https://es.wikibooks.org/wiki/Programación_en_Ada/Paquetes

Paquetes (Especificación y cuerpo) [2]

El paquete consta de **especificación** (parte visible) y **cuerpo** (implementación que se oculta) y pueden compilarse por separado.

- **Especificación** (parte visible, similar a los .h) (extension ads)

```
package nombre_de_la_unidad is
    --declaraciones
private
    --declaraciones privadas
end nombre_de_la_unidad;
```

- **Cuerpo** (implementación que se oculta, similar a un .c de una librería) (Extensión adb)

```
package body nombre_de_la_unidad is
    --desarrollo del paquete
end nombre_de_la_unidad;
```

[2] http://www.gedlc.ulpgc.es/docencia/mp_i/GuiaAda/ada14.html

Paquetes (Utilización) [3]

- Los paquetes se invocan con la claúsula **use**
- Podemos acceder a los tipos y objetos de su parte pública

Ejemplo

Crea un paquete que implemente varios procedimientos para sacar texto con colores por pantalla

[3] `http://www.gedlc.ulpgc.es/docencia/mp_i/GuiaAda/ada14.html`

Ejemplo: Paquete para sacar texto con color

Este paquete implementará procedimientos que cambian el color del texto.

Únicamente tiene un procedimiento al cual se puede acceder desde pintar.adb

colores.ads

```
package colores is
    procedure Put_Line_RED (line : string);
end colores;
```

colores.adb

```
with Ada.Text_IO;
use Ada.Text_IO;
with Ada.Characters.Latin_1;
use Ada.Characters.Latin_1;

package body colores is
    procedure Put_Line_RED (line : string) is
        RED : aliased constant String :=
                        ESC & "[91m";
        RESET : aliased constant String :=
                        ESC & "[0m";
    begin
        Put_Line(RED & line & RESET);
    end Put_Line_RED;
end colores;
```

pintar.adb

```
with colores;
use colores;

procedure pintar is
begin
    Put_line_RED ("--> Pintamos de color rojo");
end pintar;
```

Ejercicio

- Repite el ejercicio anterior modificándolo para que el procedimiento Put_line_color tome el color de una variable
- El tipo de esta variable será un enumerado y estará definido en el paquete
- Dentro del paquete implementa un procedimiento de nombre Get_color_string que devuelva el string del código ASCII necesario para cada color
- ¿Es necesario que desde fuera del paquete se pueda acceder a Get_color_string?

Solución: Paquete para sacar texto con color

Este paquete implementará un único procedimiento y una función.

El procedimiento y la función son accesibles desde pintar.adb

colores.ads

```
package colores is
    type color_t is (RED, GREEN, BLUE, YELOW, CYAN, MAGENTA);
    procedure Put_Line_color (line : string; color : color_t);
    function Get_color_string (color : color_t) return String;
end colores;
```

colores.adb

```
package body colores is
    procedure Put_Line_color(line : string; color : color_t) is
        RESET : aliased constant String := ESC & "[0m";
    begin
        Put_Line(Get_color_string(color) & line & RESET);
    end Put_Line_color;
    function Get_color_string(color : color_t) return String is
    begin
        case color is
            when Red => return ESC & "[91m";
            when Green => return ESC & "[92m";
            when Blue => return ESC & "[94m";
            when Yelow => return ESC & "[93m";
            when Magenta => return ESC & "[95m";
            when Cyan => return ESC & "[96m";
        end case;
    end Get_color_string;
end colores;
```

pintar.adb

```
with colores; use colores;

procedure pintar is
begin
    Put_line_color
        ("--> Pintamos de color rojo
              ", RED);
    Put_line_color
        ("--> Pintamos de color
              verde", GREEN);
    Put_line_color
        ("--> Pintamos de color
              amarillo", Yelow);
    Put_line_color
        ("--> Pintamos de color
              morado", Magenta);
    Put_line_color
        ("--> Pintamos de color azul
              ", BLUE);
    Put_line_color
        ("--> Pintamos de color cyan
              ", CYAN);
end pintar;
```

Paquetes (Tipos "private" y "limited private") [4]

Si se declara un tipo en la especificación de un paquete, cualquiera que utilice el paquete podrá acceder a los elementos internos de la implementación del mismo. Para evitar esto se declara el tipo como "private" en la parte pública del paquete y se pone su definición en la parte privada. Al declarar un tipo como "private" su definición queda oculta, y el usuario del paquete sólo podrá utilizar con él las operaciones que se hallan declarado en la parte pública del paquete, además de la asignación (:=), la comparación de igualdad (=) y la de desigualdad (/=).

- **Especificación** (parte visible, similar a los .h) (extension ads)

```
package nombre_de_la_unidad is
    --declaraciones
private
    --declaraciones privadas
end nombre_de_la_unidad;
```

- **Cuerpo** (implementación que se oculta, similar a un .c de una librería) (Extensión adb)

```
package body nombre_de_la_unidad is
    --desarrollo del paquete
end nombre_de_la_unidad;
```

[4] http://www.adaspain.org/recursos/docencia/mp_i/GuiaAda/ada14.html

Ejemplo: Paquete público con coordenadas

Este paquete define un tipo de datos target_t. El procedimiento y la función son accesibles desde principal.adb

publico.ads

```
package publico is
    type target_t is record
        x : Integer;
        y : Integer;
    end record;
    function Asignar_blanco(x, y : Integer) return target_t;
    procedure Imprimir_blanco(T : target_t);
end publico;
```

publico.adb

```
package body publico is
    function Asignar_blanco(x , y: Integer) return target_t is
        T : target_t;
    begin
        T.x :=x;
        T.y :=y;
        return T;
    end;
    procedure Imprimir_blanco(T : target_t) is
    begin
        Put_Line("El blanco esta en " & Integer'image(T.x) & "
              " & Integer'image(T.y));
    end Imprimir_blanco;
end publico;
```

principal.adb

```
with publico; use publico;

procedure principal is
   b1 : target_t;
begin
    b1 := Asignar_blanco(4,7);

    Imprimir_blanco(b1);

    b1.y := -3; -- Cualquiera puede
            modificar el contenido!!

    Imprimir_blanco(b1);

end principal;
```

Ejemplo: Paquete privado con coordenadas

Este paquete define un tipo de datos target_t privado. Ese tipo no es accesible desde principal.adb, pero se puede copiar

privado.ads

```
package privado is
    type target_t is private; -- Anticipamos que habra un tipo
          privado
    function Asignar_blanco(x, y : Integer) return target_t;
    procedure Imprimir_blanco(T : target_t);
private      -- Esto no sera visible fuera del paquete
    type target_t is record
        x, y : Integer;
    end record;
end privado;
```

privado.adb

```
package body privado is
    function Asignar_blanco(x , y: Integer) return target_t is
        T : target_t;
    begin
        T.x :=x;          T.y :=y;
        return T;
    end;
    procedure Imprimir_blanco(T : target_t) is
    begin
        Put_Line("El blanco esta en " & Integer'image(T.x) & "
              " & Integer'image(T.y));
    end Imprimir_blanco;
end privado;
```

principal.adb

```
procedure principal is
    b1 : target_t;
    b2 : target_t;
begin
    b1 := Asignar_blanco(4,7);

    -- b1.x := -3; Error es privado
          !!!
    Imprimir_blanco(b1);

    b2:=b1; -- Podemos hacer copias
          de los objetos!!!

    Imprimir_blanco(b2);

end principal;
```

Ejemplo: Paquete privado limitado con coordenadas

Este paquete define un tipo de datos target_t privado limitado. Ese tipo no es accesible desde principal.adb, y no se puede copiar o comparar. No se puede asignar un valor inicial

privado_limitado.ads

```
package privado_limitado is
    type target_t is limited private;
    procedure Asignar_blanco(x, y : Integer; T : out target_t);
    procedure Imprimir_blanco(T : target_t);
private
    type target_t is record
        x , y : Integer;
    end record;
end privado_limitado;
```

privado_limitado.adb

```
package body privado_limitado is
    procedure Asignar_blanco(x,y: Integer; T : out target_t) is
    begin
        T.x :=x;          T.y :=y;
    end;

    procedure Imprimir_blanco(T : target_t) is
    begin
        Put_Line("El blanco esta en " & Integer'image(T.x) & "
              " & Integer'image(T.y));
    end Imprimir_blanco;
end privado_limitado;
```

principal.adb

```
procedure principal is
   b1 : target_t;
begin
    -- b1.y = -3;
    -- b1 := Asignar_blanco(4,7);
     Asignar_blanco(4,7, b1);
     Imprimir_blanco(b1);
end principal;
```

Unidades genéricas (definición) [5]

En Ada se pueden crear unidades genéricas (que dependen de un parámetro) símplemente anteponiendo la palabra *"generic"*

```
-- Creacion de un procedimiento generico
    generic              -- Esta es la parte generica, todo lo usado
         aqui es un parametro
        type TElemento is private;           -- El tipo TElemento no
             esta definido
    procedure Algo(Elemento:in TElemento) is
        --Declaraciones
    begin
        --Acciones
    end Algo;
```

[5] http://www.gedlc.ulpgc.es/docencia/mp_i/GuiaAda/ada15.html

Unidades genéricas (utilización) [6]

Se utilizan a modo de plantilla y tienen que ser **instanciados por el usuario**, es decir decirle qué va a ser la parte genérica

```
-- Ejemplo de instanciacion del procedimiento generico anterior:

    procedure Algo_entero is new Algo(TElemento=>integer);
    procedure Algo_caracter is new Algo(character);
```

[6] http://www.gedlc.ulpgc.es/docencia/mp_i/GuiaAda/ada15.html

1.- Paquetes genéricos [7]

Los paquetes genéricos soportan los **tipos abstractos de datos** en Ada, es decir soportan la encapsulación y ocultación de las características y operaciones internas de los tipos

```
-- Ejemplo de paquete generico:
-- En la especificacion del paquete generico
    generic
        --Zona de declaracion de parametros genericos
    package nombre_del_paquete_generico is
        --Zona de uso de los parametros genericos
    end nombre_del_paquete_generico;

--  En la implementacion del paquete generico
    package body nombre_del_paquete_generico is
        --Zona de uso de los parametros genericos
    end nombre_del_paquete_generico;

--  Uso del paquete generico:
    package nombre_instancia is new nombre_del_paquete_generico(
         parametros_reales,...);
```

[7] http://www.gedlc.ulpgc.es/docencia/mp_i/GuiaAda/ada15.html

2.- Parámetros genéricos [8]

Los parámetros genéricos pueden ser **objetos, tipos o subprogramas**

- 2.1 - Tipos privados

```
-- Ejemplo de tipo generico:
        type Tipo is private;
```

- 2.2 - Tipos escalares

```
-- Discretos: Se utiliza el simbolo "<>"
        type TDiscreto is <>;
-- Enteros con signo: Se utiliza "range <>"
        type TEntero is range <>;
-- Modulares: Se utiliza "mod <>"
        type TModular is mod <>;
-- Reales en coma flotante: Se utiliza "digits <>"
        type TRealFlotante is digits <>;
-- Reales en coma fija: Se utiliza "delta <>"
        type TRealFijo is delta <>;
-- Decimales: Se utiliza "delta <> digits <>"
        type TDecimal is delta <> digits <>;
```

2.- Parámetros genéricos [9]

- 2.3 - Arrays

```
-- Hay que incluir como parametros, el tipo de los elementos del
     array, el tipo del indice del array y el tipo array
   generic
    type TElemento is private;
    type Indice is (<>);
    type Vector is array (Indice range <>) of TElemento;
   package P is
    ...
   end P;
```

- 2.4 - Punteros

```
-- Hay que especificar el tipo puntero y el tipo apuntado
   generic
    type TNodo is private;
    type TP_Nodo is access TNodo;
   package P is
    ...
   end P;
```

[9] http://www.gedlc.ulpgc.es/docencia/mp_i/GuiaAda/ada15.html

2.- Parámetros genéricos [10]

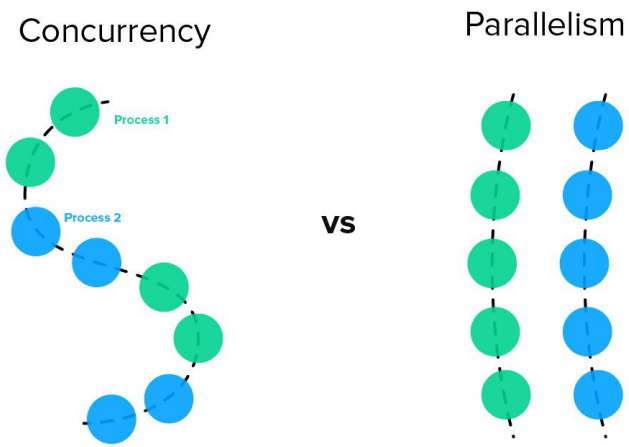
- 2.5 - Subprogramas

```
-- Se utiliza la palabra "with" precediendo al protocolo del
     subprograma que se espera
    generic
     type TElemento is private;
     with procedure Accion(X : in TElemento);
    procedure Iterar(Seq : in Secuencia_de_TElemento);
    ...
    procedure Asignar_Elemento(X : in Item);
    ...
    -- Es posible la siguiente instancia
    procedure Asignar_Lista is new Iterar(TElemento => Item,
         Accion => Asignar_Elemento);
```

[10] http://www.gedlc.ulpgc.es/docencia/mp_i/GuiaAda/ada15.html

Concurrencia

La concurrencia es la ejecución simultánea (aparente o real) de más de un proceso. En Ada la concurrencia se implementa a través de las tareas (**task**) y los **objetos protegidos**.



Tipo task [11]

Una tarea se puede entender como un proceso que se ejecuta concurrentemente con la aplicación principal (lo que habitualmente se conoce como hilo)

```
procedure Show_Simple_Task is
    task T;          -- Creamos la tarea como una variable tipo task

    task body T is   -- Definimos lo que hace la tarea en su cuerpo
    begin
          Put_Line ("In task T");
    end T;
begin        -- A partir del begin del procedimiento principal se ejecutan tanto la tarea como el proc
          Put_Line ("In main");
end Show_Simple_Task;
```

[11] https://learn.adacore.com/courses/intro-to-ada/chapters/tasking.html

Tipo task [12]

Podemos crear tantas tareas como queramos, símplemente creando más objetos

```
procedure Show_Simple_Task is
     task T1;      -- Creamos la tarea como una variable tipo task
     task T2;      -- Creamos la tarea como una variable tipo task

     task body T1 is  -- Definimos lo que hace la tarea en su cuerpo
     begin
          Put_Line ("In task T1");
     end T1;
     task body T2 is  -- Definimos lo que hace la tarea en su cuerpo
     begin
          Put_Line ("In task T2");
     end T1;
begin             -- A partir del begin del procedimiento principal se ejecutan tanto la tarea como el
       proc
     Put_Line ("In main");
end Show_Simple_Task;
```

[12] https://learn.adacore.com/courses/intro-to-ada/chapters/tasking.html

Sincronización

Como se ha comentado en los ejemplos, en el momento que la tarea principal comienza, todas las tareas que dependan de ese ámbito comienzan a la vez. Pero, ¿qué ocurre a su finalización?

Tareas y subtareas [13]

Una tarea "padre" esperará hasta que todas sus subtareas han terminado antes de terminar ella misma

```
procedure Show_Simple_Sync is
   task T;
   task body T is
   begin
      for I in 1 .. 10 loop
         Put_Line ("hello");
      end loop;
   end T;
begin              -- A partir de aqui comienzan a ejecutarse el padre o principal y la subtarea
   null            -- Esta instruccion no hace nada y continua
                   -- En este punto esperara a que termine la subtarea T
end Show_Simple_Sync;
```

[13] https://learn.adacore.com/courses/intro-to-ada/chapters/tasking.html

Sincronización

Esta espera también ocurre en otros **subprogramas** que contienen **subtareas**. El procedimiento o tarea principal esperará por las subtareas que se encuentren en subprogramas (como por ejemplo un paquete)

Subtareas en subprogramas [14]

Si declaramos una tarea en un paquete y utilizamos ese paquete, estamos creando una subtarea por la cual tendremos que esperar

simple_sync_pkg.ads

```
package Simple_Sync_Pkg is
    task T;
end Simple_Sync_Pkg;
```

simple_sync_pkg.adb

```
package body Simple_Sync_Pkg is
    task body T is
    begin
        for I in 1 .. 10 loop
            Put_Line ("hello");
        end loop;
    end T;
end Simple_Sync_Pkg;
```

test_simple_sync_pkg.adb

```
with Simple_Sync_Pkg;-- Incluyendo el
      paquete
procedure Test_Simple_Sync_Pkg is
begin
   null; -- Espera hasta que termina la
         tarea T del paq.
end Test_Simple_Sync_Pkg;
```

[14] https://learn.adacore.com/courses/intro-to-ada/chapters/tasking.html

Sincronización por cita (rendez-vous)

Hasta ahora hemos visto la sincronización que ocurre al final de los programas de forma automática. Podemos definir puntos de sincronización utilizando la palabra reservada **entry**

Entry y accept [15]

En una tarea podemos poner puntos de sincronización utilizando la palabra reservada **accept**. La tarea esperará en ese punto hasta que la tarea "padre" se sincronize con ella

```
procedure Show_Rendezvous is

   task T is
      entry Start; -- Se define un punto de entrada para T
   end T;

   task body T is
   begin
      accept Start; -- Esperamos que alguien llame a la entrada
      Put_Line ("In T");
   end T;

begin               -- Comienzan, T se queda en el accept
   Put_Line ("In Main");
   T.Start;         --  Se llama a la entrada. T pasa el accept
end Show_Rendezvous;
```

15 http://learn.adacore.com/courses/intro-to-ada/chapters/tasking.html

Sincronización por cita (rendez-vous) **select** [16]

Uno de los usos de la sentencia select es permitir a una tarea seleccionar entre varias posibles citas; en este caso, se permite su uso únicamente dentro del cuerpo de una tarea.

```
selecccion_aceptacion_cita ::=
select
    [ when condicion => ]
    ( acceptacion_cita | ( delay [ until ] expresion )
    [ secuencia_de_sentencias ] )
    | ( terminate ; ) )
{ or
    [ when condicion => ]
    ( acceptacion_cita | ( delay [ until ] expresion )
    [ secuencia_de_sentencias ] )
    | ( terminate ; ) ) }
[ else
    sequencia_sentencias ]
end select ;
```

[16] https://es.wikibooks.org/wiki/Programación_en_Ada/Tareas/Selección_de_citas

Sincronización por cita (rendez-vous) **select** [17]

Ejemplo

```
Task Servidor is
    entry Trabajar;
    entry Cerrar;
end;
task body Servidor is
begin
    loop
        select
            accept Trabajar do   -- Se acepta la llamada a trabajar.
            Trabajando := True;  -- Variable global.
        end;
        Trabajo_servidor;  -- Trabaja.
        or
            accept Cerrar;  -- Se cierra el servidor.
            exit;
        or
            delay (60.0);  -- Se han olvidado del servidor?
            Put ("Estoy esperando trabajar.");
        -- Otra opcion en vez de delay:
        -- or
        --      --Terminacion normal cuando se destruya el objeto tarea.
        -- terminate;
        end select;
    end loop;
end Servidor;
```

[17] https://es.wikibooks.org/wiki/Programación_en_Ada/Tareas/Selección_de_citas

Sincronización por cita (rendez-vous) **select** [18]

- Esta alternativa de sentencia select permite una combinación de espera y selección entre varias aceptaciones de puntos de entrada a la tarea alternativas.
- Además, la selección puede depender de condiciones asociadas a cada alternativa.
- La sentencia **delay** sirve para indicar que, si en un determinado intervalo de tiempo no se produce ninguna llamada que corresponda con las selecciones anteriores, se ejecuten las sentencias posteriores
- La sentencia **terminate** se elige en la sentencia **select** si la unidad de la que la tarea depende ha llegado al final y todas las tareas hermanas y dependientes han terminado. Es una terminación controlada. Esta alternativa no puede aparecer si hay una alternativa **delay** o **else**

[18] https://es.wikibooks.org/wiki/Programación_en_Ada/Tareas/Selección_de_citas

Sincronización por cita (redez-vou) select y terminate

No hay un límite al número de veces que una entrada puede ser aceptada (se puede crear un bucle infinito de accepts). Sin embargo esto provocaría que la tarea principal nunca terminaría esperando a la subtarea que está atrapada en un bucle infinito. En su lugar esto se hace con **select** y **terminate**

```
procedure Show_Rendezvous_Loop is
   task T is
      entry Reset;     -- entrada Reset
      entry Increment; -- entrada Increment
   end T;
   task body T is
      Cnt : Integer := 0;
   begin
      loop
         select
            accept Reset do
               Cnt := 0;
            end Reset;
            Put_Line ("Reset");
         or
            accept Increment do
               Cnt := Cnt + 1;
            end Increment;
            Put_Line ("In T's loop (" &
                  Integer'Image (Cnt) & ")");
         or
            terminate;
         end select;
      end loop;
   end T;
```

```
begin
   Put_Line ("In Main");

   for I in 1 .. 4 loop
      T.Increment; -- Calling T's entry
                   multiple times
   end loop;

   T.Reset;
   for I in 1 .. 4 loop
      T.Increment; -- Calling T's entry
                   multiple times
   end loop;
end Show_Rendezvous_Loop;
```

- Cuando la tarea T está dentro del bloque **do .. end** la tarea principal espera a que se complete el bloque (como si llamara a un procedimiento)
- En el lazo infinito de T se puede llamar todas las veces que se quiera a las entradas **Reset** e **Increment**
- Si la tarea principal termina, comprobará el estado de T y finalizará a la tarea T por la rama **or terminate** del **select**

Objetos protegidos

Cuando múltiples tareas acceden a datos compartidos, hay que evitar accesos simultáneos para lectura y/o escritura para evitar inconsistencias en los datos. En Ada para asegurar el acceso coordinado se utilizan los objetos protegidos.

```
procedure Show_Protected_Objects is
   protected Obj is   -- Declaracion
      procedure Set (V : Integer);
      function Get return Integer;
   private
      Local : Integer := 0; -- aqui los datos
   end Obj;
   protected body Obj is    -- Cuerpo
--  los procedimientos pueden modificar los datos
      procedure Set (V : Integer) is
      begin
         Local := V;
      end Set;
--  las funciones NO pueden modificar los datos
      function Get return Integer is
      begin
         return Local;
      end Get;
   end Obj;
begin
   Obj.Set (5);
   Put_Line ("Number is: " & Integer'Image (Obj.Get
       ));
end Show_Protected_Objects;
```

Los objetos protegidos encapsulan datos y dan acceso a ellos a través de operaciones protegidas, que pueden ser **subprogramas** o **entradas**:

- En el ejemplo tenemos dos operaciones para el objeto protegido, **Set** para asignarle un valor y **Get** para leer el valor del objeto.
- Es el propio Ada el que se encarga de que el acceso de procedimientos a un objeto protegido se haga en **exclusión mutua** con otros procedimientos y funciones (obviamente pertenecientes al objeto).
- El acceso de funciones es en exclusión mutua con procedimientos, ya que las funciones protegidas no pueden modificar el objeto protegido.
- Esto implica que un único procedimiento de un objeto protegido puede estar ejecutándose en un instante determinado.
- Se pueden ejecutar tantas funciones de un objeto protegido como se quiera de forma concurrente ya que únicamente implican operaciones de lectura.

[18] https://learn.adacore.com/courses/intro-to-ada/chapters/tasking.html

Entradas protegidas

Además de procedimientos y funciones, también se pueden definir **puntos de entrada protegidos**. Los puntos de entrada protegidos te permiten definir barreras (o condiciones) con la palabra protegida **when** que tendrán que cumplirse antes de que se pueda ejecutar la entrada

```
procedure Show_Protected_Objects_Entries is
   protected Obj is
      procedure Set (V : Integer);
      entry Get (V : out Integer);
   private
      Local  : Integer;
      Is_Set : Boolean := False;
   end Obj;

   protected body Obj is
      procedure Set (V : Integer) is
      begin
         Local := V;
         Is_Set := True;
      end Set;

      entry Get (V : out Integer) -- Esta es
            la entrada
        when Is_Set is  -- Esta es la barrera.
      begin  -- Si no se cumple la tarea que
            la llama se duerme
         V := Local;
         Is_Set := False;
      end Get;
   end Obj;
   N : Integer := 0;
   task T;
   task body T is
   begin
      Put_Line ("Task T will delay 4s");
      delay 4.0;
      Put_Line ("Task T will set Obj");
      Obj.Set (5);
      Put_Line ("Task T has just set Obj");
   end T;
begin
   Put_Line ("Main app will get Obj");
   Obj.Get (N);
   Put_Line ("Main app has just retrieved
         Obj");
   Put_Line ("Number is: " & Integer'Image
         (N));
end Show_Protected_Objects_Entries;
```

En el ejemplo anterior se podía leer el objeto protegido antes de que se le hubiera asignado un valor. Esto se puede controlar con una barrera

- En el ejemplo tenemos dos procesos, el ppal que intenta leer el objeto y la tarea T, que tras 4 segundos lo rellena.
- La tarea principal esperará en la entrada hasta que la tarea T rellene el valor y active el flag

Introducción

Hasta ahora hemos utilizado tareas y objetos protegidos definiendo objetos de ese tipo. Pero se pueden declarar **tipos** que sean tareas u objetos protegidos. De esta forma luego se pueden "instanciar" objetos tarea/protegidos con características comunes

[18] `https://learn.adacore.com/courses/intro-to-ada/chapters/tasking.html`

Tareas como tipos

Un tipo tarea es una generalización de una tarea. La declaración es similar a una tarea utilizando **task type**. La diferencia es que no se ha creado ningún objeto tarea si no que tenemos que declarar uno.

```
procedure Show_Simple_Task_Type is
   task type TT;

   task body TT is
   begin
      Put_Line ("In task type TT");
   end TT;

   A_Task : TT; -- Aqui es donde creamos el
         objeto tarea
begin
   Put_Line ("In main");
end Show_Simple_Task_Type;
```

- Arriba, ejemplo de creación de un tipo tarea que imprime por pantalla.
- A la derecha, ejemplo de creación de un tipo tarea con una entrada que hace que cada tarea espere para empezar a ejecutar su cuerpo. Además se crea un array de objetos de tipo tarea

```
procedure Show_Task_Type_Array is
   task type TT is
      entry Start (N : Integer);
   end TT;

   task body TT is
      Task_N : Integer;
   begin
      accept Start (N : Integer) do
         Task_N := N;
      end Start;
      Put_Line ("In task T: " & Integer'
            Image (Task_N));
   end TT;

   My_Tasks : array (1 .. 5) of TT;
begin
   Put_Line ("In main");

   for I in My_Tasks'Range loop
      My_Tasks (I).Start (I);
   end loop;
end Show_Task_Type_Array;
```

[18] https://learn.adacore.com/courses/intro-to-ada/chapters/tasking.html

Objetos protegidos como tipos

Un tipo protegido es una generalización de un objeto protegido. Para declarar un tipo protegido se utiliza **protected type**. Cuando declaramos un tipo protegido necesitamos declarar un objeto de ese tipo para obtener un objeto.

```
procedure Show_Protected_Object_Type is
   protected type Obj_Type is
      procedure Set (V : Integer);
      function Get return Integer;
   private
      Local : Integer := 0;
   end Obj_Type;

   protected body Obj_Type is
      procedure Set (V : Integer) is
      begin
         Local := V;
      end Set;

      function Get return Integer is
      begin
         return Local;
      end Get;
   end Obj_Type;

   Obj : Obj_Type;
begin
   Obj.Set (5);
   Put_Line ("Number is: " & Integer'Image (
         Obj.Get));
end Show_Protected_Object_Type;
```

Podemos reescribir el ejemplo anterior sobre objetos protegidos para utilizarlo con un tipo

- En este ejemplo en lugar de definir el objeto protegido, definimos primero un tipo y entonces declaramos un objeto de ese tipo protegido

Gestión del tiempo

Introducción

En Ada podemos gestionar el tiempo de diversas formas:

1.- Paquete **Ada.Calendar** con funciones para fecha, hora (**absoluto**)

2.- Paquete **Ada.Real_Time** para un reloj **monótono creciente**

3.- Instrucciones **delay until** y **delay**, para dormir tareas

Paquete Ada.Calendar [19]

Permite consultar y modificar la fecha y hora

```
-- Imprimir en pantalla el dia, mes y ano
-- y luego las horas y minutos

with Ada.Calendar, Ada.Text_Io;
use Ada.Calendar; use Ada.Text_Io;

procedure Muestra_Dia_Y_Hora is
-- Con Clock leemos la hora actual del sistema
    Instante : Time:= Clock;
    Hora     : Integer := Integer(Seconds(Instante))/3600;
    Minuto   : Integer := (Integer(Seconds(Instante)) - Hora*3600)/60;
begin
    Put_Line("Hoy es " & Integer'Image(Day(Instante)) &
             " del " &  Integer'Image(Month(Instante)) &
             " de "  &  Integer'Image(Year(Instante)));

    Put_Line("La hora es : "&Integer'Image(Hora)
             & ":" & Integer'Image(Minuto));

end Muestra_Dia_Y_Hora;
```

La función Clock permite obtener la fecha y hora actual, del tipo Time. Para obtener a partir de este dato el año, mes, día, o segundos dentro del día, (y al revés) existen también funciones en Ada.Calendar que puedes ver en este ejemplo.

[19] https://www.ctr.unican.es/asignaturas/lenguajes_str/parte2-tiempo-real-ada-3en1.pdf

Paquete Ada.Calendar [20]

Ada.Calendar ofrece los siguientes tipos, funciones y procedimientos

```
package Ada.Calendar is

   type Time is private;
   subtype Year_Number is Integer range
         1901..2399;
   subtype Month_Number is Integer range
         1..12;
   subtype Day_Number is Integer range 1..
         31;
   subtype Day_Duration is Duration range
         0.0..86_400.0;

   function Clock return Time;
   function Year (Date : Time) return
         Year_Number;
   function Month (Date : Time) return
         Month_Number;
   function Day (Date : Time) return
         Day_Number;
   function Seconds (Date : Time) return
         Day_Duration;

   procedure Split ( Date : in Time;
           Year : out Year_Number;
           Month  : out Month_Number;
           Day  : out Day_Number;
           Seconds : out Day_Duration);
```

```
   function Time_Of (Year : Year_Number;
             Month : Month_Number;
             Day : Day_Number;
             Seconds : Day_Duration :=
                   0.0)
          return Time;

   function "+" (Left : Time; Right :
         Duration) return Time;
   function "+" (Left : Duration; Right :
         Time) return Time;
   function "-" (Left : Time; Right :
         Duration) return Time;
   function "-" (Left : Time; Right : Time
         ) return Duration;
   function "<" (Left : Time; Right : Time
         ) return Boolean;
   function "<=" (Left : Time; Right :
         Time) return Boolean;
   function ">" (Left : Time; Right : Time
         ) return Boolean;
   function ">=" (Left : Time; Right :
         Time) return Boolean;
   ...
   Time_Error : exception;
end Ada.Calendar;
```

[20] https://www.ctr.unican.es/asignaturas/lenguajes_str/parte2-tiempo-real-ada-3en1.pdf

Paquete Ada.Real_Time [21]

Permite hacer operaciones con tiempos con los tipos **Time**, **Time_Span** y **Duration**

```
with Ada.Real_Time;
use Ada.Real_Time;

procedure P is
-- Leemos el instante inicial
   T_Inicial : Time:=Clock;
   T_Final : Time;
-- Time_Span nos permite
-- definir intervalos de tiempo (pej 5ms)
   Plazo : Time_Span:=Milliseconds(5);

   begin
    -- Instrucciones;
    -- ...
    --  Leemos el instante actual
     T_Final:=Clock;

    -- Si hemos sobrepasado el plazo
     if T_Final-T_Inicial>Plazo then
         raise Plazo_Sobrepasado;
     end if;
end P;
```

- La función **Clock** permite obtener la fecha y hora actual, del tipo **Time**
- El tipo **Time_Span** nos permite utilizar **intervalos** de tiempo, en este caso 5 milisegundos
- La resta de dos tiempos absolutos (T_Final – T_Inicial). Tiene como resultado un **intervalo** de tiempo (el tipo **Time_Span**)

[21] https://www.ctr.unican.es/asignaturas/lenguajes_str/parte2-tiempo-real-ada-3en1.pdf

Paquete Ada.Real_Time [22]

Ada.Real_Time ofrece los siguientes tipos, funciones y procedimientos

```
package Ada.Real_Time is

 type Time is private;
 Time_First : constant Time;
 Time_Last : constant Time;
 Time_Unit : constant := implementation-defined-real-number;

 type Time_Span is private;
 Time_Span_First : constant Time_Span;
 Time_Span_Last : constant Time_Span;
 Time_Span_Zero : constant Time_Span;
 Time_Span_Unit : constant Time_Span;

 Tick : constant Time_Span;
 function Clock return Time;

 function "+" (Left : Time; Right : Time_Span) return Time;
 function "+" (Left : Time_Span; Right : Time) return Time;
 function "-" (Left : Time; Right : Time_Span) return Time;
 function "-" (Left : Time; Right : Time) return Time_Span;

 function "<" (Left, Right : Time) return Boolean;
 function "<="(Left, Right : Time) return Boolean;
 function ">" (Left, Right : Time) return Boolean;
 function ">="(Left, Right : Time) return Boolean;

 function "+" (Left, Right : Time_Span) return Time_Span;
 function "-" (Left, Right : Time_Span) return Time_Span;
 function "-" (Right : Time_Span) return Time_Span;
```

```
 function "*" (Left : Time_Span; Right : Integer) return
        Time_Span;
 function "*" (Left : Integer; Right : Time_Span) return
        Time_Span;
 function "/" (Left, Right : Time_Span) return Integer;
 function "/" (Left : Time_Span; Right : Integer) return
        Time_Span;

 function "abs"(Right : Time_Span) return Time_Span;
 function "<" (Left, Right : Time_Span) return Boolean;
 function "<="(Left, Right : Time_Span) return Boolean;
 function ">" (Left, Right : Time_Span) return Boolean;
 function ">="(Left, Right : Time_Span) return Boolean;

 function To_Duration (TS : Time_Span) return Duration;
 function To_Time_Span (D : Duration) return Time_Span;
 function Nanoseconds (NS : Integer) return Time_Span;
 function Microseconds (US : Integer) return Time_Span;
 function Milliseconds (MS : Integer) return Time_Span;
 function Seconds (S : Integer) return Time_Span;
 function Minutes (M : Integer) return Time_Span;
 type Seconds_Count is range implementation-defined;
 procedure Split(T : in Time; SC : out Seconds_Count; TS :
        out Time_Span);
 function Time_Of (SC : Seconds_Count; TS : Time_Span)
        return Time;

end Ada.Real_Time;
```

[22] https://www.ctr.unican.es/asignaturas/lenguajes_str/parte2-tiempo-real-ada-3en1.pdf

La instrucción **delay** [23]

- **Retardo relativo:** se duerme por lo menos el intervalo especificado (tipo **Duration**)

```
delay Intervalo;
```

- **Retardo absoluto:** se duerme por lo menos hasta que pase la hora indicada (puede ser tipo **Calendar.Time** o **Real_Time.Time**)

```
delay until Tiempo_Absoluto;
```

```
task body Periodica is
    Periodo : constant Time_Span:=Milliseconds
          (50);
    Proximo_Periodo : Time := Clock;
begin
    loop
        -- hace cosas
        Proximo_Periodo:=Proximo_Periodo+
              Periodo;
        delay until Proximo_Periodo;
    end loop;
end Periodica;
```

Son mejores los retardos absolutos porque no dependen del tiempo de ejecución ni la sobrecarga del sistema

[23] https://www.ctr.unican.es/asignaturas/lenguajes_str/parte2-tiempo-real-ada-3en1.pdf

Aceptación temporizada [24]

Es posible aceptar una o varias entradas con un tiempo máximo de espera. Para ello debemos usar **delay** o **delay until** en la orden **select**

```
task Watchdog is
    Entry Todo_Bien;
end Watchdog;

task body Watchdog is
begin
    loop
        select
            accept Todo_bien;
         or
            delay 1.0;
            Notificar_Error;
    end loop;
end Watchdog;
```

Ejemplo del Watchdog

- Los watchdogs son tareas que tienen que ser "re-seteadas" periódicamente como indicación de que todo está funcionando de forma adecuada. Si no son re-seteadas es indicativo de un error y, normalmente, comienzan a eliminar tareas y resetear el sistema
- Una vez lanzada esta tarea Watchdog si no se llama a la entrada Todo_bien al menos una vez por segundo, se llamará al procedimiento Notificar_Error

[24] https://www.ctr.unican.es/asignaturas/lenguajes_str/parte2-tiempo-real-ada-3en1.pdf

Acción con plazo [25]

Es posible asignar un tiempo máximo de espera para algunas acciones y abortar su ejecución si no se cumple el plazo. Para ello debemos usar **then abort** en la orden **select**

```
task Accion_con_plazo;

task body Accion_con_plazo is
    Plazo : Time:=Clock+Milliseconds(10);
begin
    select
        delay until Plazo;
        Acciones_de_Recuperacion;
    then abort
        Accion_normal;
    end select;
end Accion_con_plazo;
```

Ejemplo de acción con plazo

- Es posible que queramos ejecutar una acción con un tiempo máximo de espera y, de lo contrario, abortar su ejecución. Esto se puede hacer con un then abort en el select
- Una vez lanzada esta tarea Accion_con_plazo se ejecuta Accion_normal. Si no termina Accion_normal antes del delay until Plazo, se aborta la ejecución de Accion_normal y se ejecuta Acciones_de_Recuperacion

[25] https://www.ctr.unican.es/asignaturas/lenguajes_str/parte2-tiempo-real-ada-3en1.pdf

Medida del tiempo de ejecución de una tarea [26]

El paquete Execution_Time permite medir el tiempo de CPU empleado en la ejecución de una tarea y sus subtareas

```
package Ada.Execution_Time is
    type CPU_Time is private;
    CPU_Time_First : constant CPU_Time;
    CPU_Time_Last : constant CPU_Time;
    CPU_Time_Unit : constant := implementation-defined-real-number;
    CPU_Tick : constant Time_Span;

    function Clock
          (T : Ada.Task_Identification.Task_Id := Ada.Task_Identification.Current_Task)
          return CPU_Time;

    function  "+" (Left : CPU_Time; Right : Time_Span) return CPU_Time;
    function  "+" (Left : Time_Span; Right : CPU_Time) return CPU_Time;
    function  "-" (Left : CPU_Time; Right : Time_Span) return CPU_Time;
    function  "-" (Left : CPU_Time; Right : CPU_Time) return Time_Span;
    function  "<"  (Left, Right : CPU_Time) return Boolean;
    function  "<=" (Left, Right : CPU_Time) return Boolean;
    function  ">"  (Left, Right : CPU_Time) return Boolean;
    function  ">=" (Left, Right : CPU_Time) return Boolean;

    procedure Split (T : in CPU_Time; SC : out Seconds_Count; TS : out Time_Span);
    function Time_Of (SC : Seconds_Count; TS : Time_Span := Time_Span_Zero) return CPU_Time;
end Ada.Execution_Time;
```

En este paquete la función Clock se llama para la tarea que se ejecuta y **mide el tiempo ejecutando esa tarea, no el transcurrido**

Grado en Ingeniería de Computadores

Sistemas en tiempo real

L3. Programación a gran escala

23 de septiembre del 2019