

Introducción a Ada

Juan Antonio de la Puente
DIT/UPM

Índice

- ◆ Introducción
 - cómo escribir programas sencillos
- ◆ Datos e instrucciones
 - aspectos básicos del lenguaje
- ◆ Abstracción de datos
 - tipos de datos abstractos
- ◆ Programación con objetos
 - herencia, polimorfismo, clases e interfaces
- ◆ Otros temas

Introducción

Ada

- ◆ Es un lenguaje imperativo, descendiente de Pascal
 - estructura en bloques
 - fuertemente tipado
 - orientado a objetos

- ◆ Es un lenguaje pensado para realizar sistemas empotrados de gran dimensión
 - concurrencia y tiempo real incluidos en el lenguaje
 - módulos (paquetes) que se compilan por separado

- ◆ Tres versiones normalizadas
 - Ada 83 (ISO 8652:1987)
 - Ada 95 (ISO 8652:1995)
 - Ada 2005 (ISO 8652:1995 /Amd 1:2007)

Estructura de un programa en Ada

- ◆ Un programa en Ada se compone de una o más **unidades de programa**
 - **subprogramas** (procedimientos y funciones)
 - **paquetes** (módulos)
 - tareas y objetos protegidos (ejecución concurrente)
- ◆ Los dos primeros se pueden **compilar por separado**
 - un programa se hace a base de **componentes**
 - hay un **procedimiento principal** que se ejecuta inicialmente
 - » a partir de ahí se pueden ejecutar otras unidades de programa
 - normalmente se encapsula todo lo demás en **paquetes**
 - » hay una **biblioteca** de paquetes predefinidos
 - » se pueden añadir otros para cada programa concreto
 - el compilador comprueba todas las interfaces

Procedimientos

- ◆ Una abstracción básica que representa una acción:

```
procedure <nombre> [<parámetros>] is
    <declaraciones>
begin
    <instrucciones>
end <nombre>;
```

- ◆ Las declaraciones se **elaboran** al comenzar la ejecución
 - reservar memoria, asignar valor inicial, etc.
- ◆ Las instrucciones se **ejecutan** después

Ejemplo

```
with Ada.Text_IO;           -- paquete de biblioteca
procedure Hello is
  use Ada.Text_IO;          -- acceso al paquete
begin
  Put("Hello"); New_Line;
end Hello;
```

Compilación con GNAT

◆ Compilación y montaje:

```
$ gcc -c hello.adb  # compila el fichero fuente  
$ gnatbind hello    # genera código de elaboración  
$ gnatlink hello     # monta los módulos objeto
```

◆ Se puede hacer todo de una vez:

```
$ gnatmake hello    # compila todo lo que haga falta
```

◆ Ejecución:

```
$ ./hello
```


Paquetes

- ◆ Un **paquete** es un módulo donde se declaran datos, tipos de datos, operaciones, etc.
- ◆ Tiene dos partes (que se compilan por separado)
 - **especificación**: define la interfaz visible del paquete
 - » declaraciones de tipos (y a veces objetos) de datos
 - » declaraciones de operaciones (subprogramas)
 - **cuerpo**: contiene los detalles de la implementación
 - » tipos, objetos y subprogramas adicionales (para uso local)
 - » cuerpos de subprogramas declarados en la especificación

Todo lo que aparece en el cuerpo es invisible para el resto del programa

Especificación de un paquete

```
package <nombre> is  
  <declaraciones>  
end <nombre>;
```

Ejemplo

```
package Simple_IO is

    procedure Get (F : out Float);
    procedure Put (F : in  Float);
    procedure Put (S : in  String);
    procedure New_Line;

end Simple_IO;
```

Utilización de paquetes

```
with Simple_IO, Ada.Numerics.Elementary_Functions;  
procedure Root is  
  use Simple_IO, Ada.Numerics.Elementary_Functions;  
  X : Float;  
begin  
  Put("Enter a number :");  
  Get(X);  
  Put("The square root of "); Put(X); Put(" is ");  
  Put(Sqrt(X));  
  New_Line;  
end Root;
```

Cuerpo de un paquete

```
package body <nombre> is
  <declaraciones>
  [begin
    <instrucciones>]
end <nombre>;
```

Ejemplo (1)

```
with Ada.Text_IO, Ada.Float_Text_IO;  
package body Simple_IO is
```

```
    procedure Get (F : out Float) is  
    begin
```

```
        Ada.Float_Text_IO.Get(F);  
    end Get;
```

```
    procedure Put (F : in Float) is  
    begin
```

```
        Ada.Float_Text_IO.Put(F, Exp=>0);  
    end Put;
```

```
    -- (continúa)
```

Ejemplo (2)

```
procedure Put (S : in String) is
begin
    Ada.Text_IO.Put(S);
end Put;

procedure New_Line is
begin
    Ada.Text_IO.New_Line;
end New_Line;

end Simple_IO;
```

Compilación con GNAT

- ◆ Ahora tenemos varios ficheros fuente:

`root.adb, simple_io.ads, simple_io.adb`

- ◆ Hay que compilarlos todos:

`$ gcc -c simple_io.ads`

`$ gcc -c root.adb`

`$ gcc -c simple_io.adb`

- ◆ Montaje y enlace :

`$ gnatbind root.ali`

`$ gnatlink root.ali`

- ◆ Se puede hacer todo de una vez:

`$ gnatmake root`

Estructuras de control

Selección

```
if ... then ... else ... end if;
```

Bucles

```
while ... loop ... end loop;  
for i in 1..N loop ... end loop;  
loop ... end loop;
```

Salida de bucle

```
exit when ... ;
```

Ejemplo

```
with Simple_IO, Ada.Numerics.Elementary_Functions;
procedure Roots is
  use Simple_IO, Ada.Numerics.Elementary_Functions;
  X : Float;
begin
  loop
    Put("Enter a number :");
    Get(X);
    exit when X = 0.0;
    Put("The square root of "); Put(X); Put(" is ");
    if X > 0.0 then
      Put(Sqrt(X));
    else
      Put("not real");
    end if;
    New_Line;
  end loop;
end Roots;
```

Errores y excepciones

- ◆ Una *excepción* es una manifestación de un cierto tipo de error
 - las excepciones tienen nombre, pero no son objetos
 - cuando se produce un error, se *eleva* la excepción correspondiente
 - se abandona la ejecución normal y se pasa a ejecutar un *manejador* asociado a la excepción
 - se busca un manejador en el mismo cuerpo o bloque
 - » si no lo hay, la excepción se propaga al nivel superior
 - si no se encuentra ningún manejador, se termina el programa

Ejemplo

```
with Simple_IO, Ada.Numerics.Elementary_Functions;
procedure Roots is
  use Simple_IO, Ada.Numerics.Elementary_Functions;
  X : Float;
begin
  loop
    begin
      Put("Enter a number :");
      Get(X);
      exit when X = 0.0;
      Put("The square root of "); Put(X); Put(" is ");
      Put(Sqrt(X));
    exception
      when Ada.Numerics.Argument_Error =>
        Put ("not real");
    end;
    New_Line;
  end loop;
end Roots;
```

Biblioteca estándar

Paquetes predefinidos para:

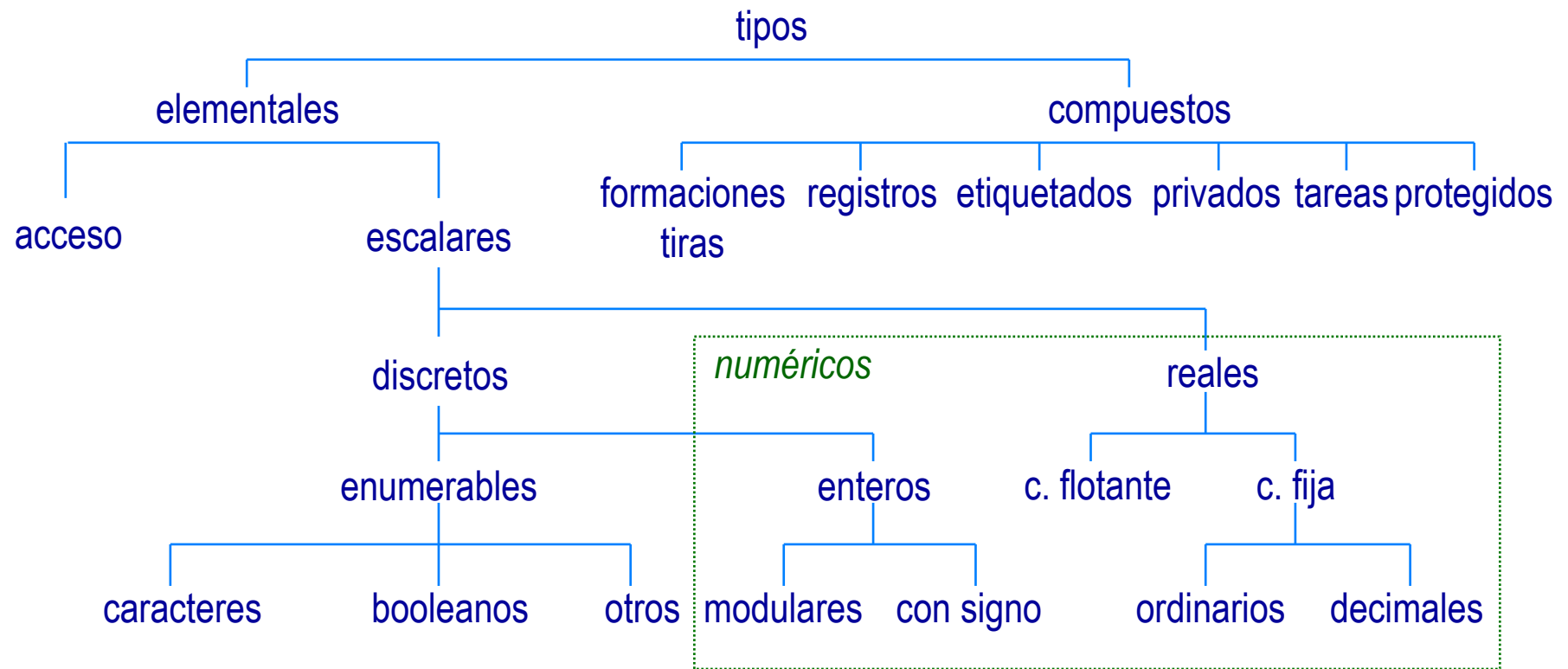
- ◆ Operaciones con caracteres y tiras
 - Ada.Characters, Ada.Strings, etc.
- ◆ Cálculo numérico
 - Ada.Numerics, Ada.Numerics.Generic_Elementary_Functions, etc.
 - » también números complejos, vectores y matrices
- ◆ Entrada y salida
 - Ada.Text_IO, Ada.Integer_Text_IO, Ada.Float_Text_IO, etc.
- ◆ Secuencias (*streams*)
- ◆ Contenedores (listas, conjuntos, etc.)
- ◆ Interfaz con el sistema operativo
- ◆ Interfaz con otros lenguajes (C, C++, Fortran, COBOL)
- ◆ Otros

Tipos de datos

Tipos de datos

- ◆ Un *tipo de datos* es un conjunto de *valores* con un conjunto de *operaciones primitivas* asociadas
- ◆ Ada es **estricto** con los tipos de datos
 - No se pueden usar valores de un tipo en operaciones de otro tipo sin efectuar una *conversión de tipo* explícita
 - Las operaciones dan siempre resultados del tipo correcto
- ◆ Una *clase* es la unión de varios tipos con características comunes

Clasificación de los tipos de datos



Tipos discretos

◆ Enumerables

```
Boolean           -- predefinido
Character         -- predefinido
Wide_Character    -- predefinido
type Mode is (Manual, Automatic); -- declarado
```

◆ Enteros

– Con signo

```
Integer           -- predefinido
type Index is range 1 .. 10; -- declarado
```

– Modulares

```
type Octet is mod 256; -- declarado
```

Tipos reales

◆ Coma flotante

```
Float                                     -- predefinido  
type Length is digits 5 range 0.0 .. 100.0; -- declarado
```

◆ Coma fija

– Ordinarios

```
Duration                                     -- predefinido  
type Voltage is delta 0.125 range 0.0 .. 5.25;
```

– Decimales

```
type Money is delta 0.01 digits 15;
```

Objetos

◆ Variables

```
X : Float;  
J : Integer := 1;
```

◆ Constantes

```
Zero : constant Float := 0.0;
```

Ejemplos

```
type Index is range 1 .. 100;           -- entero
type Length is digits 5 range 0.0 .. 100.0; -- coma flotante

First, Last : Index;
Front, Side : Length;

Last := First + 15;                      -- correcto
Side := 2.5*Front;                       -- correcto
Side := 2*Front;                         -- incorrecto
Side := Front + 2*First;                 -- incorrecto
Side := Front + 2.0*Length(First);       -- correcto
```

Números con nombre

◆ Reales

```
Pi : constant := 3.141_592_654;
```

◆ Enteros

```
Size : constant := 5;
```

Subtipos

- ◆ Un **subtipo** es un subconjunto de valores de un tipo, definido por una restricción

```
subtype Small_Index is Index    range 1 .. 5;  
subtype Big_Index   is Index    range 6 .. 10;  
subtype Low_Voltage is Voltage range 0.0 .. 2.0;
```

- La forma más simple de restricción es un intervalo de valores
- Hay dos subtipos predefinidos

```
subtype Natural  is Integer range 0 .. Integer'Last;  
subtype Positive is Integer range 1 .. Integer'Last;
```

- ◆ Las operaciones con valores de distintos subtipos de un mismo tipo están permitidas

Ejemplos

```
A : Small_Index := 1;  
B : Big_Index;  
C : Index;
```

```
A := 3;           -- correcto  
A := 6;           -- error  
A := B;           -- error  
A := C;           -- error si C > 5  
A := A + 1;       -- error si A > 4
```

Tipos compuestos: formaciones

◆ Formaciones o arrays

```
type voltages is array (Index) of voltage;  
type Matrix is array (1 .. 10, 1 .. 10) of Float;
```

◆ Elementos

```
v : voltages;                -- declaración  
  
v(5) := 1.0;                 -- elemento  
  
v      := (1.0, 0.0, 0.0, 0.0, 2.5,    -- agregado  
           0.0, 0.0, 0.0, 0.0, 0.0);  
  
v      := (1 => 1.0, 5 => 2.5, others => 0.0);
```


Tiras de caracteres

- ◆ Las **tiras** son formaciones de caracteres

```
type String is array (Positive range <>) of Character;  
-- predefinido
```

- ◆ **Objetos y operaciones**

```
A : String (1..10);  
B : String := "ejemplo";    -- los índices van de 1 a 7  
A := "abc";                 -- incorrecto  
A(1..3) := "abc";           -- correcto  
A(4) := 'X';                -- correcto  
A := B & " 1.";             -- concatenación
```

Tipos compuestos: registros

◆ Registros

```
type State is
  record
    Operating_Mode : Mode;
    Reference       : Voltage;
  end record;
```

◆ Objetos y componentes

```
X : State                                -- declaración

X.Reference := 0.0;                      -- componente
X := (Automatic, 0.0);                  -- agregado
X := (Operating_Mode => Automatic,
      Reference       => 0.0);
```

Registros con discriminantes

- ◆ Un discriminante es un componente de un registro que permite parametrizar los objetos del tipo

```
type Variable is (Temperature, Pressure);

type Measurement (Kind: Variable) is
  record
    value : voltage;
  end record;

T : Measurement(Temperature);
P : Measurement := (Kind => Pressure, value => 2.5);
```

- El discriminante tiene que ser de un tipo discreto
- No se puede cambiar una vez asignado

Tipos de acceso

- ◆ Los tipos de acceso apuntan a objetos de otros tipos

```
type State_Reference is access State;
```

- ◆ Objetos de acceso

```
Controller : State_Reference;    -- inicialmente null
```

Objetos dinámicos

- ◆ Los tipos de acceso permiten crear objetos dinámicamente

```
Controller : State_Reference := new State;
```

- ◆ Acceso a objetos dinámicos

```
Controller.Operating_Mode := Manual; -- componente  
Controller.all := (Manual, 0.0);      -- todo el objeto  
Controller      := new State'(Manual, 0.0);
```

Acceso a objetos estáticos

- ◆ En principio los tipos de acceso sólo permiten acceder a objetos dinámicos
- ◆ Para acceder a objetos estáticos hay que hacer dos cosas:

- declararlo en el tipo de acceso:

```
type State_Reference is access all State;
```

- permitir el acceso al objeto estático

```
Controller_State : aliased State;
```

- ◆ Ahora se puede dar acceso al objeto:

```
Controller : State_Reference :=  
    Controller_State'Access;
```

Instrucciones

Instrucciones simples

◆ Asignación

```
U := 2.0*v(5) + u0;
```

◆ Llamada a procedimiento

```
Get(V);
```

◆ Instrucción nula

```
null;
```


Bloque

- ◆ Agrupa una secuencia de instrucciones y puede incluir una zona declarativa
 - las declaraciones sólo son visibles en el bloque

```
declare
  V : Voltages;           -- variable local al bloque
begin
  Get(V);
  U := 2.0*V(5) + U0;
end;                       -- V deja de existir aquí
```

- ◆ Puede tener un manejador de excepciones

Selección

```
if T <= 100.0 then
    P := Max_Power;
elsif T >= 200.0 then
    P := Min_Power;
else
    P := Control(R,t);
end if;
```

Selección por casos

```
case Day is
  when Monday           => Start_Week;
  when Tuesday .. Thursday => Continue_Work;
  when Friday | Saturday => End_Week;
  when others           => Relax;
end case;
```

Iteración

◆ Iteración en un intervalo de valores

```
for I in 1..10 loop  
  Get(V(I));  
end loop;
```

◆ Iteración mientras se cumple una condición

```
while T <= 50.0 loop  
  T := Interpolation(T);  
end loop;
```

◆ Iteración indefinida

```
loop  
  Get(T);  
  P := Control(R,T);  
  Put(T);  
end loop;
```

Bucles generalizados

- ◆ Se puede salir del bucle con una instrucción **exit**

```
loop
  Get(U);
  exit when U > 80.0;
  V(I) := U;
  I := I+1;
end loop;
```

Subprogramas

- ◆ Dos tipos:
 - **procedimiento**: abstracción de acción
 - **función**: abstracción de valorAmbos pueden tener parámetros

- ◆ Un subprograma tiene dos partes
 - **especificación** o **declaración**
 - » define la interfaz (nombre y parámetros)
 - **cuerpo**
 - » define la acción o el algoritmo que se ejecuta cuando se invoca el subprograma

- ◆ A veces se puede omitir la especificación
 - En este caso la interfaz se define al declarar el cuerpo

Declaración de subprograma

La especificación se declara en una zona declarativa

```
procedure Reset;                      -- sin parámetros

procedure Increment(Value : in out Integer;
                    Step  : in      Integer := 1);

function Minimum (X,Y : Integer) return Integer;
```

Modos de parámetros

- ◆ Los parámetros de los procedimientos pueden tener alguno de estos tres *modos*:

in : no se modifican al ejecutar el subprograma

- » si no se dice se aplica este modo
- » pueden tener un valor por omisión

out : el subprograma debe asignar un valor al parámetro

in out : el subprograma usa el valor del parámetro y lo puede modificar

- ◆ Los parámetros de las funciones son siempre de modo **in**
- ◆ Los modos no están ligados al mecanismo de paso de parámetros

Cuerpo de subprograma

- ◆ Se coloca en una zona declarativa

```
procedure Increment (Value : in out Integer;  
                    Step  : in      Integer := 1)  
is  
begin  
    Value := Value + Step;  
end Increment;
```

```
function Minimum(X,Y : Integer) return Integer is  
begin  
    if X <= Y then  
        return X;  
    else  
        return Y;  
    end if;  
end Minimum;
```

Llamada a subprograma

- ◆ La llamada a procedimiento es una instrucción simple, que puede formar parte de cualquier secuencia

```
Increment(X,2);  
-- asociación de parámetros por posición  
  
Increment(Value => X, Step => 2);  
-- asociación de parámetros por nombre  
  
Increment(X);  
-- Step => 1 (valor por omisión)
```

- ◆ La llamada a función puede formar parte de cualquier expresión del tipo correspondiente

```
W := 2*Minimum(U,V);
```

Abstracción de datos

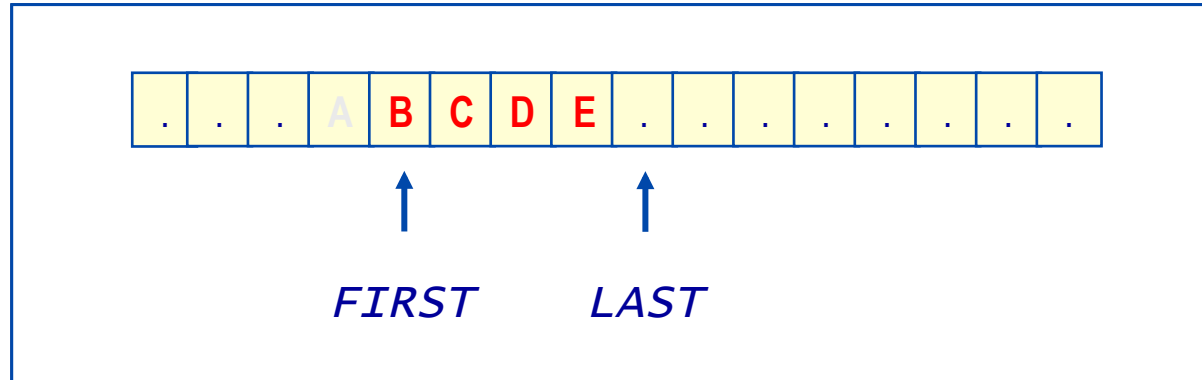
Tipos de datos y operaciones

- ◆ Podemos usar un paquete para declarar un tipo de datos y un conjunto operaciones que se pueden efectuar con los objetos del tipo:

```
package Q is
  type T is ...;
  procedure P (X: T; ...);
  function F (X: T; ...) return ...;
  function G (X: ...) return T;
end Q;
```

- las operaciones P, F y G son *operaciones primitivas* del tipo T
 - » T puede tener otras operaciones predefinidas

Ejemplo: tampón de caracteres



- ◆ Almacén temporal de caracteres
 - se insertan caracteres en el tampón, se extraen en el mismo orden
- ◆ Implementación típica: formación circular
 - se añadir un carácter en el elemento de índice LAST
 - se extrae un carácter del elemento de índice FIRST
 - en ambos casos se incrementa el índice correspondiente
 - cuando se llega al final se da la vuelta

Ejemplo

```
package Buffers is

  type Store is array (1..80) of Character;
  type Buffer is
    record
      Data:   Store;
      First:  Integer;
      Last:   Integer;
    end record;

  procedure Put (B: in out Buffer;
                 C: in   Character);
  procedure Get (B: in out Buffer;
                 C: out   Character);

end Buffers;
```

Tipos privados

```
package Buffers is
  type Buffer is private;
  procedure Put (B: in out Buffer;
                 C: in      Character);
  procedure Get (B: in out Buffer;
                 C: out     Character);
  function Is_Empty(B: Buffer) return Boolean;
  Error : exception;
private
  Size : constant Integer := 80;
  type Index is mod Size;      -- valores 0..Size-1
  type Store is array (Index) of Character;
  type Buffer is
    record
      Data:      Store;
      First:     Index := 0;
      Last:      Index := 0;
      Count:     Natural := 0;
    end record;
end Buffers;
```

Tipos limitados

```
package Buffers is
  type Buffer is limited private;
  procedure Put (B: in out Buffer;
                 C: in      Character);
  procedure Get (B: in out Buffer;
                 C: out     Character);
  function Is_Empty(B: Buffer) return Boolean;
  Error : exception;
private
  Size : constant Integer := 80;
  type Index is mod Size;
  type Store is array (Index) of Character;
  type Buffer is
    record
      Data:      Store;
      First:     Index := 0;
      Last:      Index := 0;
      Count:     Natural := 0;
    end record;
end Buffers;
```


Implementación (1)

```
package body Buffers is

  procedure Put (B: in out Buffer;
                 C: in      Character) is
  begin
    if B.Count = Size then           -- tampón lleno
      raise Error;
    end if;
    B.Data(B.Last) := C;
    B.Last := B.Last + 1;           -- módulo size
    B.Count := B.Count + 1;
  end Put;

  -- continúa
```

Implementación (2)

```
-- continuación

procedure Get (B: in out Buffer;
              C: out   Character) is
begin
  if B.Count = 0 then
    raise Error;
  end if;
  C := B.Data(B.First);
  B.First := B.First + 1;  -- módulo Size
  B.Count := B.Count - 1;
end Get;

function Is_Empty(B: Buffer) return Boolean is
begin
  return B.Count = 0;
end Is_Empty;

end Buffers;
```

Ejemplo de uso

```
with Buffers, Ada.Text_IO;
procedure Test_Buffers is
  use Buffers, Ada.Text_IO;
  My_Buffer: Buffer;
  C         : Character;
begin
  while not End_of_File loop           -- llenar el tampón
    Get(C);
    Put(My_Buffer,C);
  end loop;
  while not Is_Empty(My_Buffer) loop   -- vaciar el tampón
    Get(My_Buffer,C);
    Put(C);
  end loop;
  New_Line;
exception
  when Error => Put_Line("--- buffer error ---");
end Test_Buffers;
```

Programación mediante objetos

Programación mediante objetos

- ◆ Además de la posibilidad de definir tipos de datos abstractos, hacen falta más cosas:
 - *extensión de tipos*
 - *herencia*
 - *polimorfismo*
- ◆ Todo esto se consiguen en Ada mediante los tipos derivados y los **tipos etiquetados**

Tipos derivados

- ◆ Un **tipo derivado** es una copia de un tipo de datos, con los mismos valores y las mismas operaciones primitivas

```
type Colour is (Red, Blue, Green);  
type Light is new Colour; -- tipo derivado de Colour
```

- *Light* tiene las mismas operaciones primitivas que *Colour*
 - » son tipos distintos, no se pueden mezclar
 - » pero se pueden convertir valores de uno a otro:

```
C : Colour; L : Light;  
C := L;                -- ilegal  
C := Colour(L)         -- legal
```

Tipos etiquetados

- ◆ Son una variante de los tipos registro.
 - proporcionan todo lo necesario para programar mediante objetos

```
type R is tagged
  record
    ...      -- componentes de R
  end record;
```

```
type S is new T with
  record
    ...      -- componentes adicionales
  end record;
```

```
type T is new R with null record;
  -- sin componentes adicionales
```

Ejemplo: figuras geométricas

```
type Object is tagged
  record
    X_Coordinate: Float;
    Y_Coordinate: Float;
  end record;
```

```
type Circle is new Object with
  record
    Radius: Float;
  end record;
```

```
type Point is new Object with null record;
```


Componentes y agregados

```
O : Object;  
C : Circle;  
P : Point;  
S : Float;  
...  
S := Pi*C.Radius**2;  
  
O := (-1.0, 2,0);  
C := (0.0, 1.0, 2.5);  
  
C := (O with 3.2);  -- agregado con extensión  
O := object(C);     -- proyección
```

Operaciones y herencia

- ◆ Las **operaciones primitivas** de un tipo son las declaradas en el paquete junto con el tipo
- ◆ Un tipo extendido **hereda** las operaciones primitivas del padre
- ◆ Se pueden añadir operaciones primitivas al tipo extendido
- ◆ Se pueden redefinir las operaciones primitivas (*overriding*)
- ◆ Pero no se pueden quitar operaciones al definir el tipo extendido

Ejemplo

```
package Objects is
```

```
  type Object is tagged  
    record
```

```
      X_Coordinate: Float;
```

```
      Y_Coordinate: Float;
```

```
    end record;
```

```
  function Distance (O : Object) return Float;
```

```
end Objects;
```

Implementación

```
with Ada.Numerics.Elementary_Functions;  
package body Objects is  
  
    function Distance (O : Object) return Float is  
        use Ada.Numerics.Elementary_Functions;  
    begin  
        return Sqrt(O.X_Coordinate**2 + O.Y_Coordinate**2);  
    end Distance;  
  
end Objects;
```

Tipo extendido

```
package Objects.Circles is

  type Circle is new Object with
    record
      Radius: Float;
    end record;

  function Area (C : Circle) return Float;

end Objects.Circles;
```

Implementación

```
with Ada.Numerics;  
package body Objects.Circles is  
  
    function Area (C : Circle) return Float is  
        use Ada.Numerics;  
    begin  
        return Pi*C.Radius**2;  
    end Area;  
  
end Objects.Circles;
```

Redefinición de operaciones

```
package Objects is
  ...
  function Area (O : Object) return Float;
end Objects;
```

```
package Objects.Circles is
  ...
  function Area (C : Circle) return Float;
end Objects.Circles;
```

Implementación

```
...  
package body Objects is  
    ...  
    function Area (O : Object) return Float is  
    begin  
        return 0.0;  
    end Area;  
end Objects;
```

```
with Ada.Numerics;  
package body Objects.Circles is  
    function Area (C : Circle) return Float is  
        use Ada.Numerics;  
    begin  
        return Pi*C.Radius**2;  
    end Area;  
end Objects.Circles;
```


Ejemplos

```
with Objects, Objects.Circles;
procedure Test is
  use Objects, Objects.Circles;
  O : Object := (1.0, 1.0);
  C : Circle := (0.0, 0.0, 0.5);
  R : Circle := (O with 0.4);
  P, A : Float;
begin
  ...
  P := O.Distance;
  p := C.Distance;
  A := C.Area;
  A := R.Area;
  ...
end Test;
```

Clases y polimorfismo

- ◆ La unión de todos los tipos derivados de un mismo tipo es la **clase** de ese tipo
 - la clase de T es un tipo de datos llamado T'Class
- ◆ Se pueden declarar variables o parámetros pertenecientes a una clase
 - al ejecutarse el programa se determina el tipo concreto del objeto
 - las operaciones que se le aplican se determinan en el momento de la ejecución
 - » esto se llama **polimorfismo**
 - » el mecanismo por el que se resuelve la operación se llama *despacho dinámico*

Ejemplo

```
function Moment (O : Object'Class) is
begin
    return O.X_Coordinate*O.Area;
end Moment;
```

```
C : Circle;
M : Float;
...
M := Moment(C);
```

Tipos y operaciones abstractas

- ◆ Los tipos abstractos se usan como fundamento de una clase, pero sin que se puedan declarar objetos de ellos
- ◆ Las operaciones abstractas definen operaciones primitivas comunes para toda una clase, pero no se pueden invocar directamente
 - las operaciones abstractas no tienen cuerpo
 - es obligatorio redefinirlas en todos los tipos derivados

Ejemplo

```
package Objects is
```

```
  type Object is abstract tagged  
    record
```

```
      X_Coordinate: Float;
```

```
      Y_Coordinate: Float;
```

```
    end record;
```

```
  function Distance (O : Object) return Float;
```

```
  function Area (O: Object) return Float is abstract;
```

```
end Objects;
```

Ejemplo (cont.)

```
package Objects.Circles is
  type Circle is new Object with
    record
      Radius: Float;
    end record;
  function Area (C : Circle) return Float;
end Objects.Circles;
```

```
package Objects.Squares is
  type Square is new Object with
    record
      Side: Float;
    end record;
  function Area (S : Square) return Float;
end Objects.Squares;
```

Interfaces

- ◆ Una **interfaz** es un tipo extensible sin componentes ni operaciones concretas
 - puede tener operaciones abstractas y nulas
 - las operaciones nulas no tienen cuerpo, pero se comportan como si tuvieran un cuerpo nulo
 - » se las puede llamar, pero no hacen nada
- ◆ Un tipo puede derivarse de una o varias interfaces, además (en su caso) de un tipo extensible ordinario
 - las interfaces permiten hacer herencia múltiple

Ejemplos

```
type Printable is interface;  
procedure Put (Item : Printable'Class) is abstract;  
...  
type Printable_Object is new Object and Printable;  
procedure Put (Item : Printable_Object'Class);  
...  
P : Printable_Object;  
...  
Put(P);
```


Tipos controlados

- ◆ Son derivados de un tipo predefinido
`Ada.Finalization.Controlled`
- ◆ Se pueden definir subprogramas que se ejecutan automáticamente al
 - crear un objeto
 - destruir un objeto
 - asignar un valor a un objeto
 - Initialize
 - Finalize
 - Adjust

Unidades genéricas

Unidades genéricas

- ◆ Las **unidades genéricas** (paquetes y subprogramas) permiten definir *plantillas* de componentes en los que se dejan indefinidos algunos aspectos (*parámetros genéricos*)
 - tipos de datos, objetos, operaciones,
- ◆ Los componentes concretos (*ejemplares*) se crean a partir de la plantilla concretando los parámetros genéricos

Ejemplo

```
generic
  type Num is digits <>;
package Float_IO is
  ...
  procedure Get (Item : out Num);
  procedure Put (Item : out Num);
  ...
end Float_IO;
```

```
type Variable is digits 5 range 0.0..100.0;
package Variable_IO is new Float_IO (Variable);
...
V : Variable;
...
Variable_IO.Get(V);
```

Parámetros genéricos (1)

- ◆ Ada utiliza un **modelo de contrato** para los parámetros genéricos
- ◆ Los parámetros genéricos pueden tomar distintas formas
 - tipos de datos
 - » cualquier tipo `type T is limited private;`
 - » con `':='` y `'='` `type T is private;`
 - » discretos `type T is (<>);`
 - » enteros `type T is range <>;`
`type T is mod <>`
 - » reales `type T is digits <>;`
`type T is delta <>;`
 - » accesos `type T is access S;`
 - » otros: derivados, extensibles, abstractos, formaciones

Parámetros genéricos (2)

◆ Los parámetros genéricos también pueden ser:

- objetos

- » constantes
- » variables

C : in T;

X : in out T;

- subprogramas

with function F (....) return T;

with procedure P (...);

- paquetes

with package P(<>);

Resumen

- ◆ Ada es un lenguaje adecuado para programar sistemas complejos
 - tipado fuerte
 - modularidad
 - orientación a objetos
 - genericidad

- ◆ Hay otros aspectos y muchos detalles que no hemos visto
 - concurrencia y tiempo real
 - interacción con el hardware
 - restricciones para sistemas de alta integridad
 - ...