

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

8.1 Programación orientada a objetos

Un diseño orientado a objetos puede implementarse mediante un lenguaje convencional (como C o Pascal). Sin embargo:

- a) no hay un mecanismo para encapsular el objeto junto a sus operaciones:**
 - para expresarlas en un mismo módulo independiente
 - y especificar por separado la parte visible y los detalles internos o partes privadas de un objeto
- b) el software orientado al objeto sería más voluminoso, ya que si los objetos se parecen, repetiremos mucho código**
- c) operaciones similares de objetos parecidos (denominadas *polimórficas*) deben distinguirse en tiempo de compilación**

El problema de la repetición de código

Al crear un objeto parecido a otro debemos repetir los elementos comunes

Lo ideal sería que:

- el nuevo objeto herede lo que tuviese el viejo: *herencia*
- programar sólo las diferencias

El problema del polimorfismo

La palabra *polimorfismo* viene de “múltiples formas”

Las operaciones *polimórficas* son aquellas que hacen funciones similares con objetos diferentes:

```
mover círculo:  
  borrar círculo  
  desplazar el centro  
  dibujar círculo
```

```
mover cuadrado:  
  borrar cuadrado  
  desplazar el centro  
  dibujar cuadrado
```

El problema es que debemos tratarlas de forma diferente

```
mover figura:  
  case figura is  
    when círculo => mover círculo  
    when cuadrado => mover cuadrado  
    ...  
  end case
```

El problema del polimorfismo (cont.)

Para hacer una operación similar con diferentes objetos sería mejor expresarla de forma independiente del objeto usado:

- por ejemplo, para mover una figura sería más lógico programar de una sólo vez:

```
mover figura:  
  borrar figura  
  desplazar el centro  
  dibujar figura
```

- además, esta forma funcionaría también con figuras definidas en el futuro (es reutilizable)

Lenguajes para OOP

Estas necesidades e inconvenientes se resuelven utilizando lenguajes de programación orientada a objetos, que soportan:

- encapsulamiento** de los objetos y sus operaciones
- extensión de objetos y **herencia** de sus operaciones
 - se crean objetos a partir de otros, y se programan sólo las diferencias
- polimorfismo**, que significa invocar una operación de una familia de objetos parecidos; en tiempo de ejecución se elige la operación apropiada al objeto concreto utilizado
 - a esto se llama **enlace tardío** o **enlace dinámico** (“late binding” o “dynamic binding”)

Encapsulado de objetos: mediante paquetes

Clases de objetos: tipos de datos abstractos y *etiquetados*, declarados en un paquete, o en un paquete genérico

Objetos simples:

- variables de uno de los tipos anteriores, o un
- paquete que opere como máquina de estados abstracta

Operaciones de un objeto: subprogramas definidos en el paquete donde se declara el objeto o la clase de objetos

Notas:

Para poder hacer programación orientada al objeto es casi imprescindible contar con facilidades especiales en el lenguaje. El lenguaje Ada presenta un soporte completo para programación orientada a objetos.

Las clases de objetos pueden representarse como tipos abstractos de datos. Si el objeto se va a extender se declara como un tipo registro etiquetado. En Ada se pueden implementar como tipos de datos privados declarados en la especificación de un paquete, en donde también se declaran todas las operaciones asociadas a la clase de objetos.

Los tipos etiquetados se pueden extender para crear nuevos tipos de datos, que heredan los atributos y operaciones del tipo anterior. Permiten también el polimorfismo.

Un objeto se puede implementar como una instancia (una variable) de una clase de objetos (un tipo). También se puede implementar un objeto mediante un paquete que esté escrito en forma de máquina de estados abstracta, pero en este caso no se puede extender sin modificar. Un paquete de este tipo presenta en su especificación las operaciones (procedimientos y funciones) que el objeto puede sufrir. El estado del objeto se declara (mediante variables) en el cuerpo, o parte no visible. Las operaciones del objeto modifican el estado.

8.2. Tipos etiquetados

Contienen los atributos de la clase. Declaración:

```
type Una_Clase is tagged record
  atributo1 : tipo1;
  atributo2 : tipo2;
end record;
```

Es habitual hacer los atributos privados

```
package Nombre_Paquete is
  type Una_Clase is tagged private;
  -- operaciones de la clase
private
  type Una_Clase is tagged record
    atributo1 : tipo1;
    atributo2 : tipo2;
  end record;
end Nombre_Paquete;
```

Operaciones primitivas

Los métodos de la clase se llaman *operaciones primitivas*

- Tienen un parámetro del tipo etiquetado
- Están escritas *inmediatamente a continuación* del tipo etiquetado

8.3. Extensión de tipos etiquetados

Declaración de un tipo extendido:

```
type Clase_Nueva is new Una_Clase with record
  atributo3 : tipo3;
  atributo4 : tipo4;
end record;
```

El nuevo tipo

- también es etiquetado
- hereda todos los atributos del viejo, y añade otros

Si no se desea añadir nuevos:

```
type Clase_Nueva is new Una_Clase with null record;
```

Extensión de tipos etiquetados (cont.)

La extensión puede ser con atributos privados:

```
with Nombre_Paquete; use Nombre_Paquete;
package Nuevo_Paquete is
  type Clase_Nueva is new Una_Clase with private;
  -- operaciones de la clase nueva
private
  type Clase_Nueva is new Una_Clase with record
    atributo3 : tipo3;
    atributo4 : tipo4;
  end record;
end Nuevo_Paquete;
```

Herencia de operaciones primitivas

Al extender una clase

- se heredan todas las operaciones primitivas del padre
- se puede añadir nuevas operaciones primitivas

La nueva clase puede elegir:

- *redefinir* la operación: se vuelve a escribir
 - la nueva operación puede usar la del padre y hacer más cosas: *programación incremental*
 - o puede ser totalmente diferente
- dejarla como está
 - en este caso, no escribir nada

Invocar las operaciones del padre

Para usar una operación del padre del objeto, se hace un cambio de punto de vista sobre ese objeto

- Ejemplo: si definimos
`O : Clase_Nueva;`
- El cambio de punto de vista es como un cambio de tipo:
`Una_Clase(O)`

8.4. Ejemplo de Herencia: Figuras

```
package Figuras is

  type Coordenadas is record
    X,Y : Float;
  end record;

  type Figura is tagged record
    Centro : Coordenadas;
  end record;

  procedure Dibuja (F : Figura);
  procedure Borra (F : Figura);
  function Esta_Centrada (F : Figura) return Boolean;

end Figuras;
```

Figuras (cont.)

```
with Figuras; use Figuras;
package Circulos is

  type Circulo is new Figura with record
    Radio : Float;
  end record;

  procedure Dibuja (C : Circulo); -- redefinida
  procedure Borra (C : Circulo); -- redefinida
  -- hereda Esta_Centrada

end Circulos;
```

Figuras (cont.)

```
with Figuras; use Figuras;
package Rectangulos is

    type Rectangulo is new Figura with record
        Ancho, Alto : Float;
    end record;

    procedure Dibuja (R : Rectangulo); -- redefinida
    procedure Borra (R : Rectangulo);  -- redefinida
    -- hereda Esta_Centrada

end Rectangulos;
```

Figuras (cont.)

Ejemplo de invocación de las operaciones:

- si tenemos definido un rectángulo por ejemplo
With Rectangulos;
...
R : Rectangulos.Rectangulo;
- las dos invocaciones siguientes son equivalentes
Rectangulos.Dibuja(R); -- estilo Ada 95
R.Dibuja; -- añadido en Ada 2005

En este ejemplo el tipo “Figura” es un registro etiquetado, y por tanto extensible. Tiene definidas tres operaciones primitivas (las operaciones primitivas son las que se definen inmediatamente a continuación del tipo y que tienen al menos un parámetro de ese tipo): Dibuja, Borra, y Esta_Centrada.

Se han hecho dos extensiones del tipo Figura: Círculo y Rectángulo. Ambas extensiones heredan las tres operaciones primitivas de la Figura. Sin embargo las operaciones Dibuja y Borra no les sirven (porque son diferentes para las figuras, los círculos, y los rectángulos), y por ello las redefinen. La operación Esta_Centrada se hereda, lo que significa que, sin tener que declararlas, existen las dos nuevas funciones siguientes:

```
function Esta_Centrada (F : Círculo) return Boolean;  
function Esta_Centrada (F : Rectángulo) return Boolean;
```

Estas funciones tienen el mismo cuerpo (las mismas declaraciones e instrucciones) que la original.

Las extensiones (Círculo y Rectángulo) podrían declarar nuevas operaciones primitivas, y podrían a su vez extenderse.

8.5. Tipos etiquetados abstractos y privados

El tipo **Figura** del ejemplo anterior no representa ninguna figura concreta

- existe sólo como padre de la jerarquía

Se podría definir como *abstracto*

- no se permite crear objetos
- pueden tener *operaciones primitivas abstractas* (sin cuerpo),
 - será obligatorio redefinirlas en los hijos no abstractos

El tipo etiquetado puede tener sus atributos privados

- en ese caso hacen falta operaciones para cambiarlos y usarlos

Figuras abstractas

```
package Figuras_Abstractas is

  type Coordenadas is record
    X,Y : Float;
  end record;

  type Figura is abstract tagged record
    Centro : Coordenadas;
  end record;

  procedure Dibuja (F : Figura) is abstract;
  procedure Borra (F : Figura) is abstract;
  function Esta_Centrada (F : Figura) return Boolean;

end Figuras_Abstractas;
```

Figuras con atributos privados

```
package Figuras_Privadas is

  type Coordenadas is record
    X,Y : Float;
  end record;

  type Figura is abstract tagged private;

  procedure Cambia_Centro
    (F : in out Figura;
     Nuevo_Centro : Coordenadas);
  procedure Dibuja (F : Figura) is abstract;
  procedure Borra (F : Figura) is abstract;
  function Esta_Centrada (F : Figura) return Boolean;

private
  type Figura is abstract tagged record
    Centro : Coordenadas;
  end record;
end Figuras_Privadas;
```

Figuras con atributos privados (cont.)

```
package Figuras_Privadas.Circulos is

    type Circulo is new Figura with private;

    procedure Dibuja (C : Circulo); -- redefinida
    procedure Borra (C : Circulo);  -- redefinida
    procedure Cambia_Radio
        (C : in out Circulo;
         Nuevo_Radio : Float);
    -- hereda Esta_Centrada y Cambia_Centro

private
    type Circulo is new Figura with record
        Radio : Float;
    end record;
end Figuras_Privadas.Circulos;
```

8.6 Polimorfismo

Para cada tipo etiquetado existe un tipo de “ámbito de clase”:

- identifica los objetos de ese tipo etiquetado
- y todos sus descendientes

Declaración:

```
Figura'class
-- Incluye a Figura, Circulo, Rectangulo,
-- y los que se definan después
```

Existen también punteros de ámbito de clase

```
type Figura_Ref is access Figura'class;
```

Las llamadas a operaciones primitivas en las que se usan tipos o punteros de ámbito de clase son *polimórficas*

Ejemplo de polimorfismo: Figuras

```
with Figuras;  
use Figuras;  
procedure Mueve (F : in out Figura'Class; A : in Coordenadas) is  
begin  
    Borra(F);      -- no se sabe a priori a cuál de las  
                  -- operaciones de borrar se invocará  
    F.Centro:=A;  
    Dibuja(F);     -- no se sabe a priori a cuál de las  
                  -- operaciones de dibujar se invocará  
end Mueve;
```

8.7. Programación Incremental

El siguiente ejemplo muestra el concepto de la programación incremental

- al extender el programa, se programan sólo las diferencias
- añadir una nueva variante no implica modificación del código original

El ejemplo es parte del código necesario para escribir un simulador lógico sencillo para circuitos combinatoriales

Ejemplo: Paquete Nudos

```
package Nudos is

    Max_Nudos : constant Integer:=100;

    type Valor_Logico is range 0..1;

    -- El tipo Nudo representa un nudo lógico de un circuito
    type Nudo is range 1..Max_Nudos;

    procedure Cambia_Valor(N : Nudo; Nuevo : Valor_Logico);
    function Valor(N : Nudo) return Valor_Logico;

    -- Pone en pantalla el nudo y su valor lógico
    procedure Put_Nudo(N : Nudo);

    -- El paquete permite saber si ha habido algún cambio
    procedure Resetea_Cambios;
    function Hay_Cambios return Boolean;

end Nudos;
```

Ejemplo: Paquete Puertas

```
with Nudos; use Nudos;
package Puertas is
    subtype Nombre_Puerta is String(1..4);
    subtype Id_Puerta is String(1..9);

    -- Puerta abstracta
    type Puerta is abstract tagged private;

    procedure Pon_Nudo_Salida
        (P : in out Puerta;
         N : in Nudo);
    procedure Pon_Nombre
        (P : in out Puerta;
         Nombre : Nombre_Puerta);
    procedure Imprime (P : Puerta);
    function Identificador (P : Puerta) return Id_Puerta
        is abstract;
    procedure Opera (P : Puerta) is abstract;

    type Puerta_Ref is access all Puerta'Class;
```

Ejemplo: Puertas (cont.)

```
-- Inversor

type Inversor is new Puerta with private;

procedure Pon_Nudo_Entrada
  (P : in out Inversor;
   N : in Nudo);

procedure Imprime (P : Inversor);

function Identificador (P : Inversor) return Id_Puerta;

procedure Opera (P : Inversor);

type Inversor_Ref is access all Inversor'Class;
```

Ejemplo: Puertas (cont.)

```
-- Puerta de dos entradas, abstracta

type Puerta_Dos_Entradas is abstract new Puerta with private;

procedure Pon_Nudos_Entrada
  (P : in out Puerta_Dos_Entradas;
   N1,N2 : in Nudo);

procedure Imprime (P : Puerta_Dos_Entradas);

-- Puerta AND de dos entradas

type Puerta_And_2 is new Puerta_Dos_Entradas with private;

function Identificador (P : Puerta_And_2) return Id_Puerta;

procedure Opera (P : Puerta_And_2);

type Puerta_And_2_Ref is access all Puerta_And_2'class;
```

Ejemplo: Puertas (cont.)

```
-- Puerta OR de dos entradas  
  
type Puerta_Or_2 is new Puerta_Dos_Entradas with private;  
function Identificador (P : Puerta_Or_2) return Id_Puerta;  
procedure Opera (P : Puerta_Or_2);  
  
type Puerta_Or_2_Ref is access all Puerta_Or_2'class;
```

Ejemplo: Puertas (cont.)

```
private  
  
type Puerta is abstract tagged record  
  Nombre : Nombre_Puerta:= "  ";  
  Nudo_Salida : Nudo;  
end record;  
  
type Inversor is new Puerta with record  
  Nudo_Entrada : Nudo;  
end record;  
type Puerta_Dos_Entradas is abstract new Puerta with record  
  Nudo_Entrada_1, Nudo_Entrada_2 : Nudo;  
end record;  
  
type Puerta_And_2 is new Puerta_Dos_Entradas with null record;  
  
type Puerta_Or_2 is new Puerta_Dos_Entradas with null record;  
  
end Puertas;
```

Ejemplo: Puertas (cont.)

```
with Ada.Text_IO; use Ada.Text_IO;
package body Puertas is

  procedure Pon_Nudo_Salida
    (P : in out Puerta;
     N : in Nudo)
  is
  begin
    P.Nudo_Salida:=N;
  end Pon_Nudo_Salida;

  procedure Pon_Nombre
    (P : in out Puerta;
     Nombre : Nombre_Puerta)
  is
  begin
    P.Nombre:=Nombre;
  end Pon_Nombre;

end;
```

Ejemplo: Puertas (cont.)

```
procedure Imprime (P : Puerta) is
begin
  -- Escribimos la parte común a todas las puertas
  -- Empezamos por el identificador
  Put(Identificador(Puerta'Class(P)));
  -- el ámbito de clase hace que la llamada sea polimórfica
  Set_Col(11);
  -- Ahora el nudo de salida
  Put_Nudo(P.Nudo_Salida);
  Set_Col(16);
end Imprime;

procedure Pon_Nudo_Entrada
  (P : in out Inversor;
   N : in Nudo)
is
begin
  P.Nudo_Entrada:=N;
end Pon_Nudo_Entrada;
```

Ejemplo: Puertas (cont.)

```
procedure Imprime (P : Inversor) is
begin
  Imprime(Puerta(P)); -- hemos invocado la operación del padre
  Put_Nudo(P.Nudo_Entrada);
  New_Line;
end Imprime;

function Identificador (P : Inversor) return Id_Puerta is
begin
  return "INV "&P.Nombre;
end Identificador;

procedure Opera (P : Inversor) is
begin
  if Valor(P.Nudo_Entrada)=0 then
    Cambia_Valor(P.Nudo_Salida,1);
  else
    Cambia_Valor(P.Nudo_Salida,0);
  end if;
end Opera;
```

Ejemplo: Puertas (cont.)

```
procedure Imprime (P : Puerta_Dos_Entradas) is
begin
  Imprime(Puerta(P)); -- hemos invocado la operación del padre
  -- Ahora escribimos los dos nudos de entrada
  Put_Nudo(P.Nudo_Entrada_1);
  Set_Col(22);
  Put_Nudo(P.Nudo_Entrada_2);
  New_Line;
end Imprime;

procedure Pon_Nudos_Entrada
(P : in out Puerta_Dos_Entradas;
 N1,N2 : in Nudo)
is
begin
  P.Nudo_Entrada_1:=N1;
  P.Nudo_Entrada_2:=N2;
end Pon_Nudos_Entrada;
```

Ejemplo: Puertas (cont.)

```
function Identificador (P : Puerta_And_2) return Id_Puerta is
begin
    return "AND2 "&P.Nombre;
end Identificador;

procedure Opera (P : Puerta_And_2) is
begin
    if Valor(P.Nudo_Entrada_1)=1 and then
        Valor(P.Nudo_Entrada_2)=1
    then
        Cambia_Valor(P.Nudo_Salida,1);
    else
        Cambia_Valor(P.Nudo_Salida,0);
    end if;
end Opera;
```

Ejemplo: Puertas (cont.)

```
function Identificador (P : Puerta_Or_2) return Id_Puerta is
begin
    return "OR2 "&P.Nombre;
end Identificador;

procedure Opera (P : Puerta_Or_2) is
begin
    if Valor(P.Nudo_Entrada_1)=1 or else
        Valor(P.Nudo_Entrada_2)=1
    then
        Cambia_Valor(P.Nudo_Salida,1);
    else
        Cambia_Valor(P.Nudo_Salida,0);
    end if;
end Opera;

end Puertas;
```

Ejemplo: Paquete Circuitos

```
with Puertas; use Puertas;

package Circuitos is

    Max_Puertas : Integer:=100;

    type Circuito is tagged private;

    procedure Anade_Puerta
        (C : in out Circuito;
         P : Puerta_Ref);
    -- puede elevar No_Cabe

    procedure Imprime (C : Circuito);
    -- Imprime todas las puertas del circuito

    procedure Opera (C : Circuito);
    -- Opera con todas las puertas; puede elevar No_Converge

    No_Cabe, No_Converge : exception;
```

Ejemplo: Circuitos (cont.)

```
private

    type Lista_Puertas is array(1..Max_Puertas) of Puerta_Ref;

    type Circuito is tagged record
        Puertas : Lista_Puertas;
        Num : Integer range 0..Max_Puertas:=0;
    end record;

end Circuitos;
```

Ejemplo: Circuitos (cont.)

```
with Nudos; use Nudos;
package body Circuitos is

  procedure Anade_Puerta
    (C : in out Circuito; P : Puerta_Ref)
  is
  begin
    if C.Num=Max_Puertas then
      raise No_Cabe;
    end if;
    C.Num:=C.Num+1;
    C.Puertas(C.Num):=P;
  end Anade_Puerta;

  procedure Imprime (C : Circuito) is
  begin
    for I in 1..C.Num loop
      Imprime(C.Puertas(I).all);
    end loop;
  end Imprime;
end;
```

Ejemplo: Circuitos (cont.)

```
procedure Opera (C : Circuito) is
begin
  -- limitamos el número de veces por si hay realimentación
  for Veces in 1..Max_Nudos loop
    Resetea_Cambios;
    -- operamos con todas las puertas
    for I in 1..C.Num loop
      Opera(C.Puertas(I).all);
    end loop;
    -- terminamos si no ha habido cambios de valores lógicos
    if not Hay_Cambios then
      return;
    end if;
  end loop;
  -- se ha excedido el máximo número de veces
  raise No_Converge;
end Opera;

end Circuitos;
```

Ejemplo: Programa principal

```
with Puertas, Circuitos, Nudos;
use Puertas, Circuitos, Nudos;

procedure Simulador_Logico is
  C : Circuito;
  Inv : Inversor_Ref;
  And2 : Puerta_And_2_Ref;
  Or2 : Puerta_Or_2_Ref;
begin
  -- Crear una puerta AND2
  And2:=new Puerta_And_2;
  Pon_Nombre(And2.all, "A  ");
  Pon_Nudo_Salida(And2.all,5);
  Pon_Nudos_Entrada(And2.all,1,2);
  Anade_Puerta(C,Puerta_Ref(And2));

  -- Crear otras puertas de forma similar
```

Ejemplo: Programa principal

```
-- Poner los valores de las entradas
Cambia_Valor(1,0);
Cambia_Valor(2,1);
Cambia_Valor(3,1);
Cambia_Valor(4,0);

-- Calcular las salidas e imprimir resultados
Opera(C);
Imprime(C);

end Simulador_Logico;
```

Ejemplo: Añadir una nueva variante

```
package Puertas.Nand2 is
    -- Puerta NAND de dos entradas --
    type Puerta_Nand_2 is new Puerta_Dos_Entradas with private;
    function Identificador (P : Puerta_Nand_2) return Id_Puerta;
    procedure Opera (P : Puerta_Nand_2);
    type Puerta_Nand_2_Ref is access all Puerta_Nand_2'class;
private
    type Puerta_Nand_2 is new Puerta_Dos_Entradas with null record;
end Puertas.Nand2;
```

Ejemplo: Añadir una nueva variante (cont.)

```
package body Puertas.Nand2 is
    function Identificador (P : Puerta_Nand_2) return Id_Puerta is
    begin
        return "NAND2 "&P.Nombre;
    end Identificador;

    procedure Opera (P : Puerta_Nand_2) is
    begin
        if Valor(P.Nudo_Entrada_1)=1 and then
            Valor(P.Nudo_Entrada_2)=1
        then
            Cambia_Valor(P.Nudo_Salida,0);
        else
            Cambia_Valor(P.Nudo_Salida,1);
        end if;
    end Opera;
end Puertas.Nand2;
```

Ventajas de la Programación Incremental:

- es más simple de entender (ya que cada variante está aparte)
- la estructura del código es mucho más simple (no hay variantes o instrucciones *case* anidadas)
- es extensible sin modificar el código existente, sin recompilarlo, y sin posibilidad de añadirle errores

Desventajas

- es necesario pensar siempre en la extensibilidad: hay que acordarse de hacer enlazado dinámico si es necesario
- el código fuente está distribuido en diferentes módulos y subprogramas