

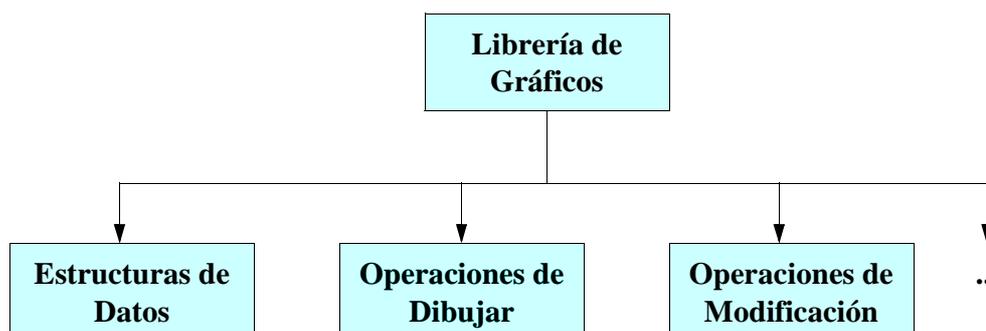
1. Introducción a los computadores y su programación
2. Elementos básicos del lenguaje
- 3. Modularidad y programación orientada a objetos**
4. Estructuras de datos dinámicas
5. Tratamiento de errores
6. Abstracción de tipos mediante unidades genéricas
7. Entrada/salida con ficheros
8. Herencia y polimorfismo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

## 3.1. Diseño modular y orientado a objetos

El diseño modular es una técnica que viene usándose desde hace mucho tiempo.

El problema es cómo partir en módulos independientes

- antes solía hacerse de acuerdo a criterios funcionales (agrupar funciones similares)



# Partición con criterios funcionales

## Inconvenientes

- Cada figura (objeto del espacio problema) reside en varios módulos de programa
  - por tanto estos módulos no son independientes
- Un cambio en el objeto del espacio problema implica cambiar muchos módulos, y volverlos a probar
- Añadir nueva funcionalidad implica normalmente tocar varios módulos, y volverlos a probar
  - por ejemplo, añadir una figura nueva.
- En definitiva, el software es difícil de cambiar o extender y poco reutilizable

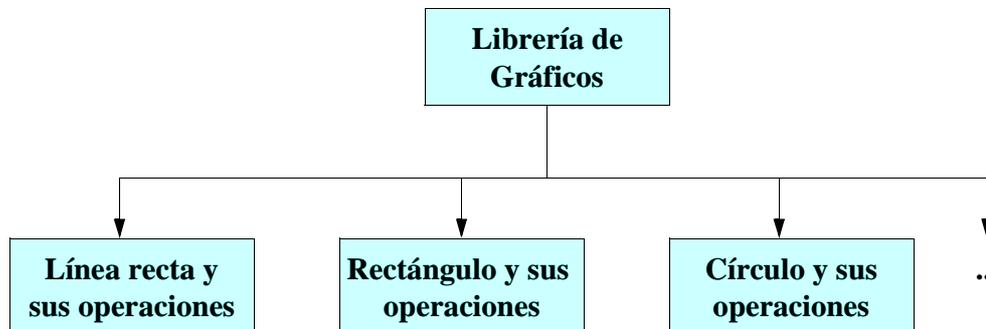
# Diseño Orientado a Objetos

Se puede utilizar otra aproximación a la partición en módulos:

- división orientada a objetos
- cada objeto del problema real que es preciso resolver es un módulo del programa
- se encapsulan juntas la definición del objeto y todas sus operaciones

El diseño de programas orientado a objetos pretende:

- simplificar la modificación y extensión del software, haciendo que la mayor parte del mismo sea reutilizable
- además es más fácil de entender ya que existe relación directa entre la estructura del software y la del problema



## Ventajas

Si el diseño es orientado al objeto:

- el cambio interno en un objeto afecta a un solo módulo
- en muchos casos, extender la funcionalidad no implica modificar el software existente, sino sólo añadir nuevos objetos
- el software es más reutilizable

## 3.2. Concepto de clase y objeto

Un **objeto** es un elemento de programa que se caracteriza por:

- **atributos o campos**: son datos contenidos en el objeto, y que determinan su estado
- **operaciones o métodos**: son operaciones con las que podemos solicitar información del objeto, o modificarla
  - Están compuestas por secuencias de instrucciones que operan con los atributos, y que pueden invocar operaciones de otros objetos

Una **clase** representa una definición de un módulo de programa, a partir de la cual se pueden crear muchos objetos

- Cada uno de estos objetos es una **instancia** de la clase

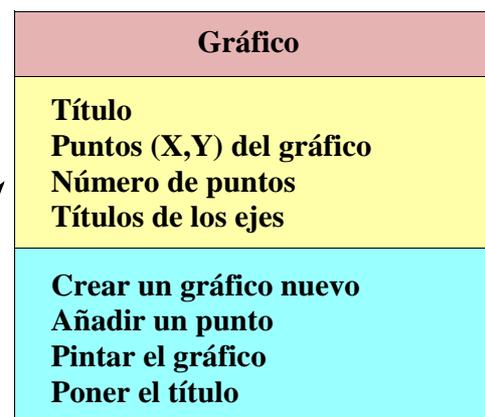
## Concepto de clase y objeto (cont.)

Se intenta siempre corresponder los objetos de un programa con objetos del problema que éste resuelve.

Ejemplo: mostrar uno o varios gráficos de funciones de una variable.

Los gráficos representan una clase de objetos.

Cada gráfica individual representa un objeto de la clase "Gráfico".



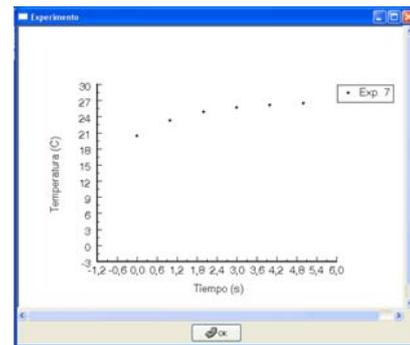
atributos

operaciones

# Ejemplo

```
with Plot_Windows; use Plot_Windows;

procedure Experimento is
  Plot : Plot_Window_Type :=
    Plot_Window("Experimento", "Tiempo (s)", "Temperatura (C)");
begin
  Set_Graph_Title(Plot, "Exp. 7");
  -- Anadir los puntos
  Add_Point(Plot, 0.0, 20.4);
  Add_Point(Plot, 1.0, 23.2);
  Add_Point(Plot, 2.0, 24.8);
  Add_Point(Plot, 3.0, 25.7);
  Add_Point(Plot, 4.0, 26.1);
  Add_Point(Plot, 5.0, 26.5);
  -- Pintar el gráfico
  Wait(Plot);
end Experimento;
```



## 3.2. Paquetes Ada

### Motivación:

- Encapsular juntas declaraciones de constantes, tipos y subprogramas
- **Programación orientada a objetos**: encapsulamiento de clases con especificación de una interfaz visible, y ocultando los detalles internos

### Los paquetes Ada tienen dos partes:

- parte visible o especificación
  - a su vez, tiene una parte no visible o privada
- cuerpo, con los detalles internos

# Declaración de paquetes

Se ponen en la parte de declaraciones de un programa (o en un fichero aparte como veremos más adelante)

## Especificación

```
package Nombre is
```

```
    declaraciones de datos y tipos;  
    declaraciones de tipos y constantes privadas;
```

```
    declaraciones de cabeceras de subprogramas;  
    declaraciones de especificaciones de paquetes;
```

```
private
```

```
    otras declaraciones necesarias más abajo;  
    declaraciones completas de los tipos y constantes privados
```

```
end Nombre;
```

# Declaración de paquetes (cont.)

## Cuerpo

```
package body Nombre is
```

```
    declaraciones de datos y subprogramas;  
    declaraciones de subprogramas y paquetes de la  
    especificación;
```

```
begin
```

```
    instrucciones de inicialización;
```

```
end Nombre;
```

## Las instrucciones de inicialización

- se ejecutan una sólo vez antes de que se use el paquete
- son opcionales (junto a su **begin**)
- sirven para inicializar variables declaradas en el paquete

# Uso de un paquete

Uso de objetos de la parte visible de un paquete:

```
Paquete.objeto
```

Con cláusula **use** se puede usar el objeto directamente

```
use Paquete;
```

Con una cláusula **use type** se pueden usar los operadores de un tipo directamente

```
use type Paquete.Tipo;
```

- Si no se hace, los operadores se tendrían que usar como funciones:

```
c:=Paquete."+"(a,b); -- en lugar de c:=a+b;
```

# Ejemplo: Números complejos

```
package Complejos is

  type Complejo is private;

  function Haz_Complejo (Re,Im : Float) return Complejo;

  function Real(C : Complejo) return Float;
  function Imag(C : Complejo) return Float;

  function "+" (C1,C2 : Complejo) return Complejo;

  function Image(C : Complejo) return String;

private

  type Complejo is record
    Re,Im : Float;
  end record;

end Complejos;
```

## Ejemplo (cont.)

```
package body Complejos is

    function "+" (C1,C2 : Complejo) return Complejo is
    begin
        return (C1.Re+C2.Re,C1.Im+C2.Im);
    end "+";

    function Haz_Complejo (Re,Im : Float) return Complejo is
    begin
        return (Re,Im);
    end Haz_Complejo;

    function Imag (C : Complejo) return Float is
    begin
        return C.Im;
    end Imag;
```

## Ejemplo (cont.)

```
function Image (C : Complejo) return String is
begin
    if C.Im>=0.0 then
        return Float'Image(C.Re)&" + "&Float'Image(C.Im)&" J";
    else
        return Float'Image(C.Re)&" - "&Float'Image(abs C.Im)&" J";
    end if;
end Image;

function Real (C : Complejo) return Float is
begin
    return C.Re;
end Real;

end Complejos;
```

# Ejemplo de uso

```
with Complejos, Ada.Text_IO;
use Ada.Text_IO;
use type Complejos.Complejo;

procedure Prueba_Complejos is
  C1,C2,C3 : Complejos.Complejo;
begin
  C1:=Complejos.Haz_Complejo(3.0,4.0);
  C2:=Complejos.Haz_Complejo(5.0,-6.0);
  Put_Line("C1=" & Complejos.Image(C1));
  Put_Line("C2=" & Complejos.Image(C2));
  C3:=C1+C2;
  Put_Line("Parte real C3=" & Float'Image(Complejos.Real(C3)));
  Put_Line("Parte imag C3=" & Float'Image(Complejos.Imag(C3)));
end Prueba_Complejos;
```

## 3.3. Compilación separada

En Ada se pueden compilar separadamente:

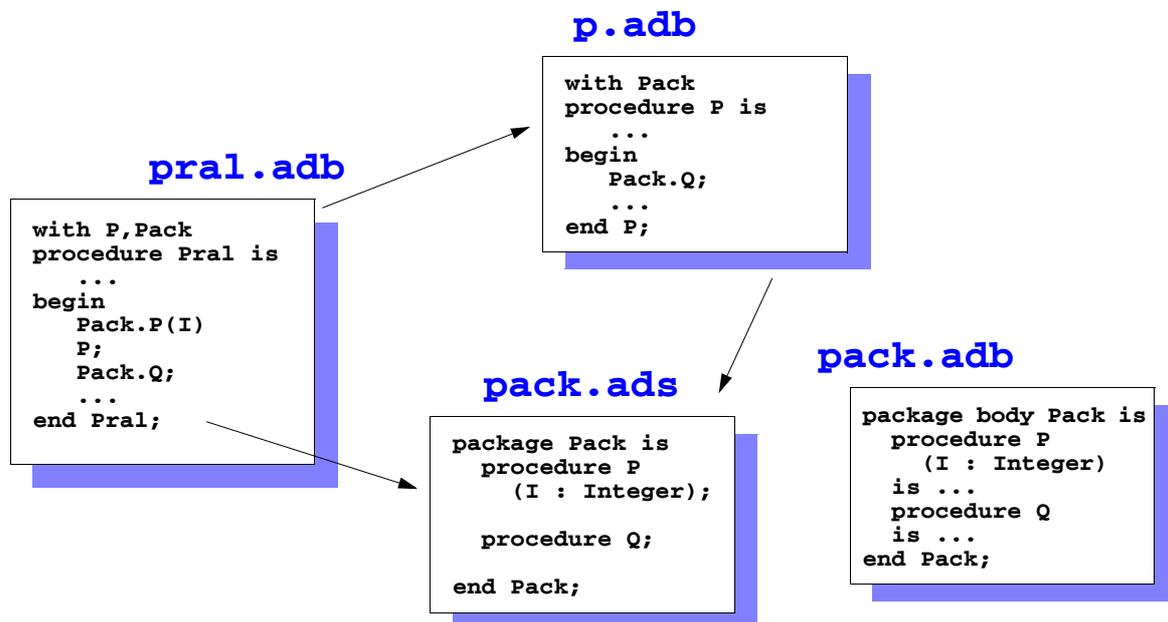
- subprogramas
- especificaciones de paquetes
- cuerpos de paquetes

Cada uno de ellos se llama “unidad de librería”

Para usar una unidad compilada separadamente se debe usar una cláusula **with**.

El orden de compilación, en algunos compiladores, es importante. En *Gnat* no importa

# Ejemplo:



# Cláusulas use

Observar que no tiene nada que ver la compilación separada (**with**) con la cláusula **use**

Las cláusulas **use** se usan para no tener que poner el nombre del paquete delante del objeto

Criterios para las cláusulas **use**

- Poner **use** para paquetes muy conocidos y usados (**Ada.Text\_IO**)
- No poner cláusula **use** para paquetes nuestros
- Poner cláusula **use type** para poder usar los operadores de un tipo directamente

## 3.4. Reglas de visibilidad

Quedan modificadas en los siguientes aspectos:

- Las declaraciones de la especificación (visibles y privadas) son visibles en el cuerpo
- Las declaraciones de la parte visible de la especificación son visibles en el bloque en que se declara el paquete:
  - si se antepone el nombre del paquete
  - o mediante cláusula use
- Las declaraciones del cuerpo sólo son visibles en ese cuerpo.

## Paquetes hijos

Es posible crear paquetes que son “*hijos*” de otros, formando una estructura jerárquica.

Lo que caracteriza a un paquete hijo es que tiene visibilidad sobre la parte privada del padre, pero no sobre el cuerpo

Para declararlo se usa la notación “.” en el nombre

```
package Padre is
    ...
end Padre;

package Padre.Detalle is
    ...-- puede usar la parte privada de Padre
end Padre.Detalle;
```

En *Gnat* el fichero debe tener el nombre `padre-detalle.ads`

## 3.5. Tipos de paquetes

1. Paquetes sin cuerpo
2. Librerías:
  - engloban operaciones relacionadas entre sí
3. Paquetes con objetos o máquinas de estados abstractas
  - encapsulan un objeto (un dato oculto en el cuerpo) y operaciones para manipularlo
4. Paquetes con clases o tipos de datos abstractos
  - encapsulan un tipo de datos y operaciones para manipular objetos de ese tipo
  - el tipo de datos es privado
  - el usuario puede crear todos los objetos de ese tipo de datos que desee

## Paquete sin cuerpo

```
package Constantes_Atomicas is

    Carga_Electron : constant := 1.602E-19;    -- Coulombios
    Masa_Electron   : constant := 0.9108E-30;  -- Kg
    Masa_Neutron    : constant := 1674.7E-30;  -- Kg
    Masa_Proton     : constant := 1672.4E-30;  -- Kg

    -- Este package no tiene "body"

end Constantes_Atomicas;
```

# Paquete con un objeto o máquina de estados (1/3)

```
with Elementos; -- define el tipo Elemento, que es discreto
use Elementos;
package Un_Conjunto is

    procedure Inserta (E : Elemento);

    procedure Extrae (E : Elemento);

    function Pertenece (E: Elemento) return Boolean;

end Un_Conjunto;
```

# Paquete con un objeto o máquina de estados (2/3)

```
package body Un_Conjunto is

    type Conjunto is array (Elemento) of Boolean;

    C : Conjunto; -- aquí está la variable que contiene el estado

    procedure Inserta (E : Elemento) is
    begin
        C(E) := True;
    end Inserta;

    procedure Extrae (E : Elemento) is
    begin
        C(E) := False;
    end Extrae;
```

# Paquete con un objeto o máquina de estados (3/3)

```
function Pertenece (E: Elemento) return Boolean is
begin
    return C(E);
end Pertenece;

begin -- Inicialización del estado del paquete

    C:= Conjunto'(Conjunto'range=>False);

end Un_Conjunto;
```

# Paquete con una clase o tipo abstracto de datos (1/3)

```
with Elementos; -- define el tipo Elemento, que es discreto
use Elementos;
package Conjuntos_Privados is
    type Conjunto is private;

    function Vacio return Conjunto;
    procedure Inserta (E : Elemento;
                      C : in out Conjunto);
    procedure Extrae (E : Elemento;
                     C : in out Conjunto);

    -- pertenencia
    function "<" (E: Elemento; C: Conjunto) return Boolean;

private
    type Conjunto is array (Elemento) of Boolean;
end Conjuntos_Privados;
```

# Paquete con una clase o tipo abstracto de datos (2/3)

```
package body Conjuntos_Privados is

    function Vacio return Conjunto is
    begin
        return Conjunto'(Conjunto'range=>False);
    end Vacio;

    procedure Inserta (E : Elemento;
                      C : in out Conjunto) is
    begin
        C(E):=True;
    end Inserta;

    procedure Extrae (E : Elemento;
                     C : in out Conjunto) is
    begin
        C(E):=False;
    end Extrae;

end Conjuntos_Privados;
```

# Paquete con una clase o tipo abstracto de datos (3/3)

```
-- pertenencia

function "<" (E: Elemento; C: Conjunto) return Boolean is
begin
    return C(E);
end "<";

end Conjuntos_Privados;
```

# Algunos paquetes de librerías: Números aleatorios

```
package Ada.Numerics.Float_Random is

  -- Basic facilities

  type Generator is limited private;

  subtype Uniformly_Distributed is
    Float range 0.0 .. 1.0;

  function Random (Gen : Generator)
    return Uniformly_Distributed;

  procedure Reset (Gen      : Generator);
  procedure Reset (Gen      : Generator;
                  Initiator : Integer);

  -- Advanced facilities
  ...
end Ada.Numerics.Float_Random;
```

## Ejemplo de uso de números aleatorios:

```
with Ada.Numerics.Float_Random, Text_Io, Ada.Integer_Text_Io;
use Ada.Numerics.Float_Random, Text_Io, Ada.Integer_Text_Io;
procedure Dado is

  Gen : Generator;
  Pulsada : Boolean;
  C : Character;

begin
  Put_Line("Pulsa una tecla para parar");
  Reset(Gen);
  loop
    Put(Integer(6.0*Random(Gen)+0.5));
    New_Line;
    delay 0.5;
    Get_Immediate(C,Pulsada);
    exit when Pulsada;
  end loop;
end Dado;
```

# Librería de strings variables (1/3)

```
with Ada.Text_IO; use Ada;
package Var_Strings is

    Max_Length : constant Integer := 80;
    type Var_String is private;

    Null_Var_String : constant Var_String;

    function To_String      (V : Var_String) return String;
    function To_Var_String (S : String)    return Var_String;

    function Length (V : Var_String) return Natural;

    procedure Get_Line (V : out Var_String);
    procedure Put_Line (V : in  Var_String);
    procedure Put      (V : in  Var_String);

    procedure Get_Line (F : in out Text_IO.File_Type;
                       V : out  Var_String);
```

# Librería de strings variables (2/3)

```
procedure Put_Line (F : in out Text_IO.File_Type;
                   V : in  Var_String);
procedure Put      (F : in out Text_IO.File_Type;
                   V : in  Var_String);

function "&" (V1,V2 : Var_String) return Var_String;
function "&" (V : Var_String; S : String) return Var_String;
function "&" (S : String; V : Var_String) return Var_String;
function "&" (V : Var_String; C : Character) return Var_String;
function "=" (V1,V2 : Var_String) return Boolean;
function ">" (V1,V2 : Var_String) return Boolean;
function "<" (V1,V2 : Var_String) return Boolean;
function ">=" (V1,V2 : Var_String) return Boolean;
function "<=" (V1,V2 : Var_String) return Boolean;

function Element (V : Var_String; Index : Positive)
                return Character;
function Slice (V : Var_String; Index1 : Positive;
               Index2 : Natural) return Var_String;
```

# Librería de strings variables (3/3)

```
function To_Upper (V : Var_String) return Var_String;
function To_Lower (V : Var_String) return Var_String;

procedure Translate_To_Upper (V : in out Var_String);
procedure Translate_To_Lower (V : in out Var_String);

private

type Var_String is record
  Str : String (1..Max_Length);
  Num : Integer range 0..Max_Length:=0;
end record;

Null_Var_String : constant Var_String :=
  (Str => (others => ' '), Num => 0);

end Var_Strings;
```

# Otras Librerías Estándares

Nombre	Descripción
<code>Ada.Numerics.Complex_Types</code>	Tipos para números complejos
<code>Ada.Numerics.Complex_Elementary_Functions</code>	Operaciones matemáticas para números complejos
<code>Ada.Calendar</code>	Fecha y hora
<code>Ada.Characters.Handling</code>	Conversión y clasificación de caracteres y strings
<code>Ada.Characters.Latin_1</code>	Conjunto de caracteres estándares
<code>Ada.Command_Line</code>	Operaciones para recibir los argumentos expresados en la orden de ejecución del programa

## 3.6. Ejemplo de programa con paquetes

### Programa para la gestión de una lista de alumnos

#### Fases:

- Análisis de requerimientos
- Especificación funcional
- Diseño arquitectónico
- Diseño detallado
- Codificación
- Prueba

## Análisis de requerimientos

Queremos manejar datos personales de una clase de alumnos

Cada alumno tiene los siguientes datos:

- Nombre
- Teléfono
- Escuela a la que pertenece (dentro de un conjunto fijo)

La clase contiene un número variable de alumnos

- Podemos acotar el número máximo a 100
- No es necesario que estén ordenados

El programa debe permitir insertar alumnos nuevos y ver los datos del alumno indicado por el usuario

El programa presentará al usuario un menú con tres opciones:

- Insertar un alumno nuevo
  - comprueba si cabe; si no cabe presenta un error
  - pedirá los datos del alumno por teclado
  - insertará el nuevo alumno en la clase
- Mirar los datos de un alumno
  - pedirá el número del alumno
  - si es incorrecto muestra un mensaje de error
  - si es correcto muestra los datos de ese alumno
- Salir del programa

Partimos el programa en las siguientes partes

- clase *alumno*
  - datos: datos personales del alumno
  - operaciones: para leer datos de teclado y escribirlos en pantalla
- clase "*clase*"
  - datos: la lista de los alumnos
  - operaciones: número de alumnos, saber si está llena, obtener el alumno "*n*", insertar un nuevo alumno
- librería "*menu*"
  - con operaciones para pedir una opción del menú, pedir el número de un alumno, y mostrar un mensaje de error

# Diseño arquitectónico (cont.)

- *programa principal* que gestiona las llamadas a las operaciones de los diferentes paquetes

Las interfaces de cada parte se muestran a continuación:

```
package Alumnos is
    type Tipo_Escuela is (Teleco, Caminos, Fisicas, Medicina);
    type Alumno is private;
    procedure Lee (Alu : out Alumno);
    procedure Escribe (Alu : in Alumno);
private
    ...
end Alumnos;
```

# Diseño arquitectónico (cont.)

```
with Alumnos;
package Clases is
    Max_Alumnos : constant Integer := 100;
    subtype Num_Alumno is Integer range 1..Max_Alumnos;
    type Clase is private;
    procedure Inserta_Alumno
        (Alu : in Alumnos.Alumno; La_Clase : in out Clase);
    function Dame_Alumno
        (Num : in Num_Alumno;
         La_Clase : in Clase)
        return Alumnos.Alumno;
    function Numero_Alumnos (La_Clase : in Clase) return Natural;
    function Llena (La_Clase : in Clase) return Boolean;
private
    ...
end Clases;
```

`with Clases;`

`package Menu is`

`type Opcion is (Insertar,Mirar,Salir);`

`procedure Pide_Opcion (La_Opcion : out Opcion);`

`procedure Lee_Num_Alumno (Num : out Clases.Num_Alumno);`

`procedure Mensaje_Error (Mensaje : in String);`

`end Menu;`

## Diseño detallado

### Clase `Alumno`

- **datos: registro con los siguientes campos:**

`nombre (string de hasta 20 caracteres)`

`número de caracteres del nombre`

`teléfono (string fijo de 9 caracteres)`

`escuela: dato enumerado`

- **leer datos de teclado**

`mostrar el nombre de cada dato y leerlo`

- **escribir datos en pantalla**

`mostrar el nombre de cada dato y su valor`

## Diseño detallado (cont.)

### Clase `Clase`

- Estructura de la lista de los alumnos
  - usaremos un array de alumnos y una variable para saber cuántos alumnos hay
- número de alumnos
  - `retornar el número almacenado`
- saber si está llena
  - `comparar el número de alumnos con el máximo`
- obtener el alumno "`n`"
  - `devolver la casilla n del array`
- insertar un nuevo alumno
  - `añadirle al final del array e incrementar el n de alumnos`

## Diseño detallado (cont.)

### Librería `Menu`

- pedir una opción del menú
  - `presentar todas las opciones con un número al lado`
  - `pedir un número`
  - `convertirlo al tipo enumerado Opcion`
- pedir el número de un alumno
  - `poner un texto y leer el número`
- mostrar un mensaje de error
  - `poner el texto en pantalla`

# Diseño detallado (cont.)

## Programa principal. Pseudocódigo:

```
procedure Principal is
  Tercero_B : Clase=Vacía
begin
  loop
    Pedir una opción
    case la opción es
      when insertar
        si no cabe: mostrar error
        si cabe:
          pedir datos del alumno
          insertar alumno
      when mirar
        pedir el número
        si es incorrecto mostrar error
        si no, mostrar el alumno
      when salir: finalizar
    end loop;
end Principal
```

# Codificación: Paquete Alumnos

```
package Alumnos is

  Tamano_Nombre : constant Integer := 20;

  type Tipo_Escuela is (Teleco, Caminos, Fisicas, Medicina);

  type Alumno is private;

  procedure Lee (Alu : out Alumno);
  procedure Escribe (Alu : in Alumno);

private

  type Alumno is record
    Nombre      : String(1..Tamano_Nombre);
    N_Nombre    : Integer range 0..Tamano_Nombre:=0;
    Telefono    : String(1..9):="          ";
    Escuela     : Tipo_Escuela:=Teleco;
  end record;
end Alumnos;
```

# Paquete Alumnos (cont.)

```
with Ada.Text_IO;
use Ada.Text_IO;

package body Alumnos is

    package Escuela_IO is new Enumeration_IO(Tipo_Escuela);

    procedure Escribe (Alu : in Alumno) is
    begin
        Put_Line("-----");
        Put("Nombre   : ");
        Put_Line(Alu.Nombre(1..Alu.N_Nombre));
        Put_Line("Telefono : "&Alu.Telefono);
        Put("Escuela   : ");
        Escuela_IO.Put(Alu.Escuela);
        New_Line;
        Put_Line("-----");
        New_Line;
    end Escribe;
```

# Paquete Alumnos (cont.)

```
procedure Lee (Alu : out Alumno) is
begin
    Put("Nombre del alumno: ");
    Get_Line(Alu.Nombre,Alu.N_Nombre);
    Put("Numero de Telefono: (9 cifras) ");
    Get(Alu.Telefono);
    Skip_Line;
    Put("Escuela: ");
    Escuela_IO.Get(Alu.Escuela);
    Skip_Line;
end Lee;

end Alumnos;
```

# Paquete Clases

```
with Alumnos;  
  
package Clases is  
  
    Max_Alumnos    : constant Integer := 100;  
  
    subtype Num_Alumno is Integer range 1..Max_Alumnos;  
  
    type Clase is private;  
  
    procedure Inserta_Alumno  
        (Alu : in Alumnos.Alumno;  
         La_Clase : in out Clase);  
  
    function Dame_Alumno  
        (Num : in Num_Alumno;  
         La_Clase : in Clase)  
        return Alumnos.Alumno;  
  
    function Numero_Alumnos (La_Clase : in Clase) return Natural;
```

# Paquete Clases (cont.)

```
    function Llena (La_Clase : in Clase) return Boolean;  
  
private  
  
    type Tipo_Alumnos is array (1..Max_Alumnos) of Alumnos.Alumno;  
  
    type Clase is record  
        Alumnos : Tipo_Alumnos;  
        Num      : Integer:=0;  
    end record;  
  
end Clases;
```

# Paquete Clases (cont.)

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

    function Dame_Alumno
        (Num : in Num_Alumno;
         La_Clase : in Clase)
        return Alumnos.Alumno
    is
    begin
        return La_Clase.Alumnos (Num);
    end Dame_Alumno;

    procedure Inserta_Alumno
        (Alu : in Alumnos.Alumno; La_Clase : in out Clase)
    is
    begin
        La_Clase.Num:=La_Clase.Num+1;
        La_Clase.Alumnos (La_Clase.Num) :=Alu;
    end Inserta_Alumno;

end Clases;
```

# Paquete Clases (cont.)

```
function Llena (La_Clase : in Clase) return Boolean is
begin
    return La_Clase.Num=Max_Alumnos;
end Llena;

function Numero_Alumnos (La_Clase : in Clase) return Natural is
begin
    return La_Clase.Num;
end Numero_Alumnos;

end Clases;
```

# Paquete Menu

```
with Clases;  
  
package Menu is  
  
    type Opcion is (Insertar,Mirar,Salir);  
  
    procedure Pide_Opcion (La_Opcion : out Opcion);  
  
    procedure Lee_Num_Alumno (Num : out Clases.Num_Alumno);  
  
    procedure Mensaje_Error (Mensaje : in String);  
  
end Menu;
```

# Paquete Menu (cont.)

```
with Ada.Text_IO, Ada.Integer_Text_IO;  
use Ada.Text_IO, Ada.Integer_Text_IO;  
  
package body Menu is  
  
    procedure Lee_Num_Alumno (Num : out Clases.Num_Alumno) is  
    begin  
        Put("Numero del alumno: ");  
        Get(Num);  
        Skip_Line;  
    end Lee_Num_Alumno;  
  
    procedure Mensaje_Error (Mensaje : in String) is  
    begin  
        Put_Line(Mensaje);  
    end Mensaje_Error;
```

# Paquete Menu (cont.)

```
procedure Pide_Opcion (La_Opcion : out Opcion) is
  Num_Opcion : Integer range 1..3;
begin
  New_Line;
  Put_Line("LISTA DE ALUMNOS:");
  Put_Line("1-Insertar alumno");
  Put_Line("2-Mirar un alumno");
  Put_Line("3-Salir");
  Put("Elige una opcion:");
  Get(Num_Opcion);
  Skip_Line;
  La_Opcion:=Opcion'Val(Num_Opcion-1);
end Pide_Opcion;
```

```
end Menu;
```

# Programa principal

```
with Menu,Alumnos,Clases;
```

```
procedure Lista_Alumnos is
```

```
  Tercero_B : Clases.Clase;
  Eleccion  : Menu.Opcion;
  Alu       : Alumnos.Alumno;
  Num       : Clases.Num_Alumno;
```

```
begin
```

```
  loop
```

```
    Menu.Pide_Opcion (Eleccion);
```

```
    case Eleccion is
```

```
      when Menu.Insertar =>
```

```
        if Clases.Llena(Tercero_B) then
```

```
          Menu.Mensaje_Error("No caben mas alumnos");
```

```
        else
```

```
          Alumnos.Lee(Alu);
```

```
          Clases.Inserta_Alumno(Alu,Tercero_B);
```

```
        end if;
```

# Programa principal (cont.)

```
when Menu.Mirar =>
  Menu.Lee_Num_Alumno(Num);
  if Num>Clases.Numero_Alumnos(Tercero_B) then
    Menu.Mensaje_Error("Alumno no existe");
  else
    Alu:=Clases.Dame_Alumno(Num,Tercero_B);
    Alumnos.Escribe(Alu);
  end if;
when Menu.Salir => exit;
end case;
end loop;
end Lista_Alumnos;
```

# Prueba

La prueba se puede hacer en las siguientes fases:

- Paquete **Alumnos**: hacer un programa principal de prueba que lea un alumno y lo muestre por la pantalla
- Paquete **Menu**: hacer un programa principal de prueba que invoque cada operación y muestre el resultado obtenido
- Paquete **Clases** y programa principal: usar el programa completo para probar las clases y el funcionamiento completo
  - introducir varios alumnos
  - mirarlos
  - probar los dos errores definidos: llena (reduciendo temporalmente el tamaño máximo) y número de alumno incorrecto

# Criterios para la prueba de una unidad

---

El programa de prueba de una unidad debe:

- probar todas las operaciones, con todos sus casos, comparando los resultados con algún resultado conocido
- probar las operaciones en diferente orden (p.e. insertar, eliminar, volver a insertar, etc.)
- probar los casos límite: p.e. estructura de datos llena, vacía, insertar al principio o al final, etc.
- probar todos los casos de error que se pueda