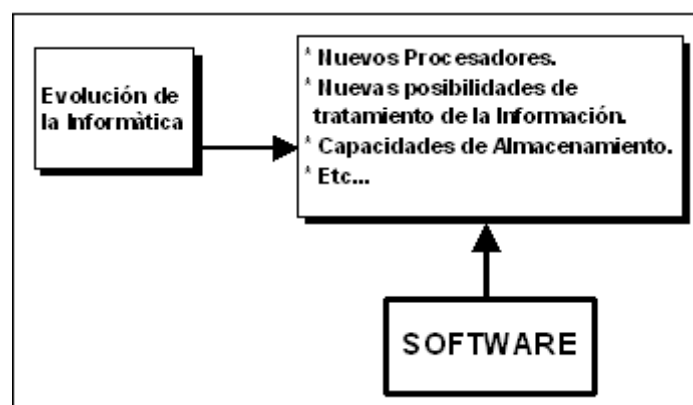


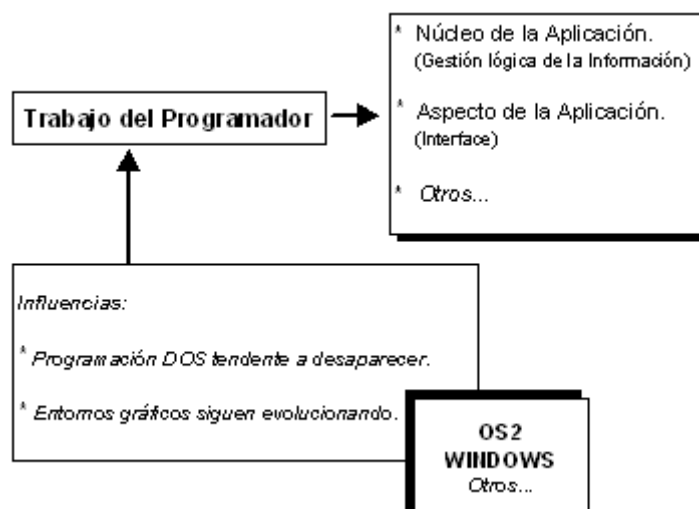
Comenzando en Delphi.

- 1. INTRODUCCIÓN.**
- 2. INTRODUCCIÓN A OBJECT PASCAL.**
- 3. INTRODUCCIÓN A DELPHI.**
- 4. EL ENTORNO DE TRABAJO.**
- 5. PROGRAMACIÓN ORIENTADA A OBJETOS**
- 6. INICIANDO CON COMPONENTES BASICOS.**
- 7. FUNCIONES Y PROCEDIMIENTOS ÚTILES DE DELPHI**

1. INTRODUCCIÓN.



(*) El Software debe adaptarse al Usuario y no al revés.



(*) El diseño del Aspecto de la Aplicación supone un trabajo adicional para el programador.

PROGRAMACIÓN EN WINDOWS.

Entorno de trabajo orientado a **EVENTOS**. El flujo de ejecución viene determinado por acciones del Usuario (o externas).

Evolución...

1	Lenguaje 'C'
2	Compiladores C++ Con jerarquías de clases específicas para Windows, como : <ul style="list-style-type: none">● Object Windows <i>Borland.</i>● Fundation classes <i>Microsoft</i>

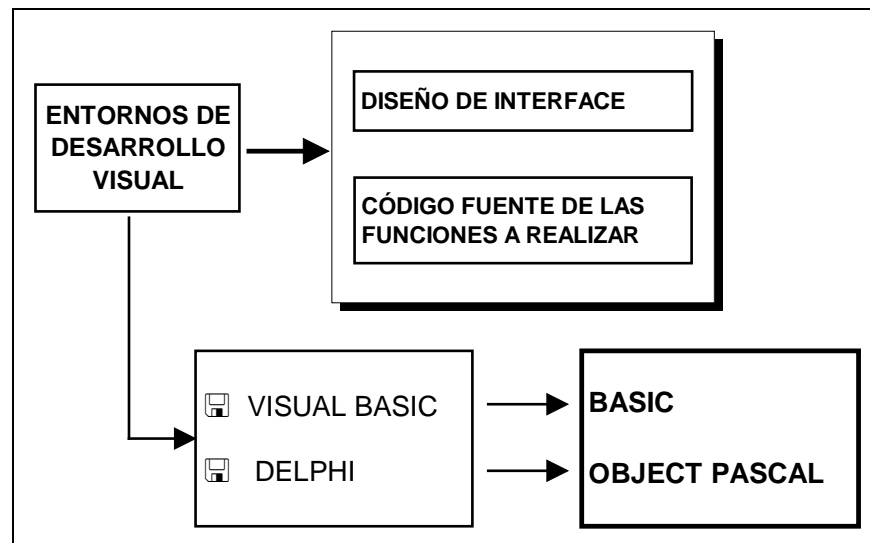
O.O.P.

3	Herramientas de desarrollo Visual <i>Ventajas :</i> <ul style="list-style-type: none">■ Simplifican la creación de Interface.■ Facilitan la comunicación con otras aplicaiones.■ Cuadros de Diálogo comunes.■ Gestión de Bases de Datos.
---	---

Las Herramientas de Desarrollo Visual se basan en el uso de **COMPONENTES** “prefabricados” que son dispuestos en **VENTANAS** y personalizados mediante **PROPIEDADES**.

El entorno más conocido hasta la fecha es **VISUAL BASIC**, aunque actualmente van apareciendo otros.

OJO !! A pesar de todas las ventajas siempre será necesario escribir “algo” de código para que las aplicaciones realicen alguna función útil. No sólo se diseña el Interface y basta.



DELPHI ENTORNO DE DESARROLLO VISUAL DE BORLAND.

- Integra un gran número de componentes “prefabricados”.
- Utiliza una variante de Borland Pascal como lenguaje, llamado OBJECT PASCAL. (Por el uso que hace de la Orientación a Objetos).

¿ QUÉ APORTA DELPHI (Que no ofrezcan los demás) ?

- ◆ Es un Compilador real.
- ◆ Genera código directamente ejecutable, sin necesidad de archivos adicionales (RunTime).

Consecuencias :

- Rapidez de ejecución.
- Tamaño menor de la aplicación.

¿ QUÉ DEBEMOS ESTUDIAR ?

- ASPECTOS DE LA PROGRAMACIÓN WINDOWS.
- EL GENERADOR DE INTERFACE.
- LA ESTRUCTURA GRAL. DEL LENGUAJE OBJECT PASCAL.

2. INTRODUCCIÓN A OBJECT PASCAL.

Pascal usa "begin" y "end" para encerrar sentencias en bloques, en lugar de llaves ("{}") como C o Java.

Las llaves ("{}") encierran comentarios. Se pueden usar "(*" y "*)" en su lugar (¡sin las comillas!). También se puede comentar hasta el final de la línea con "///" como en C o Java.

El punto y coma (";") es un separador de sentencias, no un terminador de las mismas como en C, de modo que no es necesario antes de un "end".

Se puede dividir una sentencia en muchas líneas como en C o Java dado que el retorno de carro no es un separador (el punto y coma lo es).

Los comentarios que comienzan con "\$" son directivas del compilador.

Los valores numéricos hexadecimales se preceden por un "\$", así como en C se preceden por "0x".

Un "#" precediendo un valor numérico denota un valor de tipo carácter, siendo ese carácter el correspondiente a ese valor numérico.

Los caracteres son compatibles en asignación con las cadenas: son tratados como cadenas de un carácter.

Las variables no son automáticamente inicializadas.

Las sentencias de asignación usan ":", mientras que "=" es un operador de comparación (en C y Java, "=" es el operador de asignación e "==" es el operador de comparación).

La sintaxis tipo C `a := b := c := 0;` no se admite.

Los operadores And, Or y Not tienen una precedencia más alta que los operadores de comparación, así que las operaciones de comparación se suelen encerrar entre paréntesis (a menos que use And, Or o Not como operadores de bits en vez de operadores booleanos). Por ejemplo, `a > b and a > c` se interpretaría como `a > (b and a) > c`, lo que generará un error de tipos. La expresión debería ser `(a > b) and (a > c)`.

Los identificadores no distinguen mayúsculas de minúsculas.

Las cadenas y los caracteres se delimitan por comillas simples.

ESTRUCTURA DEL CÓDIGO FUENTE DE UNA APLICACIÓN

Normalmente una aplicación se compone de un fichero de programa (".dpr") y muchos ficheros de "unidades" (".pas"). Estas unidades son módulos como los ficheros ".c".

Un fichero de un programa mínimo comienza con la palabra "program" seguida del nombre interno del programa, y un bloque begin...end. Por ejemplo:

```
program programa1;  
begin  
  WriteLn('Hola mundo!');  
end.
```

WriteLn es un procedimiento que escribe una cadena en el dispositivo estándar de salida del sistema (normalmente la consola).

Un programa puede declarar constantes, tipos, variables, procedimientos y funciones. El siguiente programa no hará nada útil, pero sirve como ejemplo:

```
program programa2;  
const  
  MIN: integer = 1;  
  MAX: integer = 20;  
type  
  RangoValores = 1..20;  
  ArregloValores = array[RangoValores] of integer;  
var  
  Valor: integer;  
  Valores: ArregloValores;  
function EstaEnRango(n: integer): boolean;  
begin  
  if (n >= MIN) and (n <= MAX) then  
    Result := True  
  else  
    EstaEnRango := False;  
end;  
  
procedure IngresarValores();  
procedure InicializarValores();  
var i: RangoValores;  
begin  
  for i := MIN to MAX do Valores[i] := 0;  
end;  
  
begin  
  InicializarValores;  
  repeat  
    WriteLn('Ingrese un valor entre 1 y 20 (0 p/salir): ');  
    ReadLn(valor);  
    if Valor <> 0 then  
      if EstaEnRange(Valor) then  
        Inc(Valores[Valor])  
      else  
        WriteLn('Valor inválido. Por favor intente de nuevo');  
    until Valor = 0;  
  end;
```

```
function MaxValor(): RangoValores;
var
  i: RangoValores;
begin
  Result := MIN;
  for i := MIN + 1 to MAX do
    if Valores[i] > Valores[Result] then
      Result := i;
  end;
begin
  IngresarValores;
  WriteLn('El valor más ingresado es: ', MaxValor());
end.
```

La palabra "const" precede la declaración de constantes. En el ejemplo se declaran dos constantes enteras llamadas "MIN" y "MAX", representando los valores 1 y 20 respectivamente.

La palabra "type" precede la declaración de tipos definidos por el usuario. En el ejemplo declaramos un rango de valores enteros y un arreglo de 20 enteros indexados del 1 al 20.

La palabra "var" precede la declaración de variables. En el ejemplo declaramos una variable entera y un arreglo (de 20 enteros indexados del 1 al 20).

Las declaraciones de funciones se preceden por la palabra "function". Las funciones tienen un nombre, parámetros (opcionales) y un tipo de retorno. Por ejemplo, la función "EstaEnRango" toma un argumento entero y devuelve un valor booleano. Para devolver un valor, éste debe asignarse a la variable implícita "Result" o a la variable implícita llamada igual que la función de modo que "Result := ..." o "EstaEnRango := ..." tienen el mismo efecto.

La declaración de procedimientos se precede por la palabra "procedure" en vez de "function", y no tienen tipo de retorno dado que los procedimientos no devuelven valores.

Para llamar un procedimiento o una función, simplemente escriba su nombre (seguido de los parámetros reales encerrados entre paréntesis si es que el procedimiento o función toma parámetros). Por ejemplo, "EstaEnRango(Valor)" llama a la función EstaEnRango pasando la variable "Valor" como el argumento. "IngresarValores" en el bloque principal llama al procedimiento IngresarValores. "Inc(Valores[Valor])" (en C o Java se escribiría Valores[Valor]++ en su lugar) llama al procedimiento incorporado que incrementa su argumento. En realidad no se trata de un procedimiento dado que el compilador generará el código de máquina INC...

Los procedimientos y funciones pueden tener constantes, tipos y variables locales, así como también otros procedimientos y funciones. Se declaran igual que sus contrapartes a nivel de programa, excepto que las declaraciones se ponen entre la cabecera del procedimiento o función y el "begin" de su bloque de código. Por ejemplo la función "MaxValor" declara una variable local "i" y el

procedimiento "IngresarValores" declara un procedimiento local "InicializarValores" que a su vez declara una variable local "i".

La ventaja de las funciones y procedimientos locales es que pueden acceder las variables locales y los parámetros de los procedimientos y/o funciones que los contienen, de modo que no se necesita pasarlos como argumentos, permitiendo un código más claro y eficiente.

Los procedimientos usados hasta ahora (ReadLn, que lee de la entrada estándar, WriteLn e Inc) son declarados en la unidad System que es "incluida" por el compilador de modo predeterminado, de manera tal que no necesitamos decirle que lo haga. La unidad System contiene muchas constantes, tipos, variables, procedimientos y funciones útiles, pero hay muchas más disponibles en otras unidades que vienen con Delphi. Por ejemplo, la mayoría de los elementos para trabajar con ficheros vienen en la unidad SysUtils y la mayoría de las API de Windows vienen en la unidad Windows. Para "incluir" estas unidades en un programa empleamos la cláusula "uses". Por ejemplo:

```
program programa3;  
uses  
  SysUtils, Windows;  
...
```

Y por supuesto, podemos crear nuestras propias unidades. Las unidades comienzan con la palabra "unit" en vez de "program" y tienen cuatro secciones:

interface

La sección interfaz contiene todas las declaraciones públicas (es decir, constantes, tipos, variables, procedimientos y funciones que estarán disponibles a programas y otras unidades)

implementation

La sección implementación contiene todas las declaraciones a nivel de módulo (unidad) y la definición de los procedimientos y funciones declaradas en la sección de interfaz.

initialization

Código para inicializar las variables públicas y modulares.

finalization

Código para liberar recursos asignados por la unidad.

Por ejemplo:

```
unit Unidad1;  
interface  
var globalvar: integer;  
procedure IncGlobalVar;  
implementation  
procedure IncGlobalVar;  
begin  
  Inc(globalvar);  
end;  
initialization
```



```
globalvar := 10;  
end.
```

Esta unidad declara una variable pública llamada globalvar y un procedimiento público llamado IncGlobalVar. En la sección de implementación este procedimiento es definido y en la sección de inicialización se establece el valor inicial de la variable. No hay sección de finalización porque no hay recursos que liberar.

Este programa usa la unidad declarada arriba:

```
program programa4;  
uses  
  Unidad1 in 'Unidad1.pas';  
  
begin // La unidad se inicializa aquí  
  WriteLn(globalvar); // 10  
  IncGlobalVar;  
  WriteLn(globalvar); // 11  
end.
```

SENTENCIAS DE CONTROL DE FLUJO

La sentencia "if"

if <condición> then <sentencia>;	end else
	<sentencia4>;
if <condición> then	
<sentencia1>	if <condición> then begin
else	<sentencia1>;
<sentencia2>;	<sentencia2>;
	<sentencia3>;
if <condición> then begin	end else begin
<sentencia1>;	<sentencia4>;
<sentencia2>;	<sentencia5>;
<sentencia3>;	end;

Note que no hay punto y coma antes de un "else". Necesita usar "begin...end" cuando las sentencias dentro de la estructura son más de una. <condición> es una expresión booleana (debe evaluarse como True o False).

La sentencia "case"

```
case <selector> of  
  
  <listacaso1>: <sentencia1>;  
  
  ...  
  
  <listacasoN>: <sentenciaN>;  
  
[else  
  
  <sentencia>;]  
  
end;
```

<selector> debe ser una expresión ordinal (no puede ser una cadena por ejemplo). Las listas de casos son listas de valores separados por comas. Listas de casos válidas son por ejemplo:

1:

5..10:

20, 30..40, 50..60, 67:

Cuando se necesita más de una sentencia en un case se deben encerrar mediante un bloque begin...end.

La sentencia "for"

```
for <variable> := <ini> to/downto <fin> do <sentencia>;
```

Si usa "to" el incremento (o "step") es 1, y si usa "downto", el incremento es -1. Si necesita más sentencias dentro del bucle for debe encerrarlas en un bloque begin...end. Tenga cuidado: a la salida de un bucle for, el valor de la variable de control queda indefinido!

La sentencia "while"

```
while <condición> do <sentencia>;
```

<condición> es una expresión booleana y si necesita más sentencias dentro del bucle while, ya sabe: begin...end.

La sentencia "repeat...until"

```
repeat  
  
    <sentencia1>;  
  
    <sentencia2>;  
  
    ...  
  
    <sentenciaN>[:];  
  
until <condición>;
```

Las sentencias se repiten mientras la condición NO es satisfecha. La condición es evaluada al final, así que las sentencias de adentro se ejecutan al menos una vez. No se necesita begin...end.

La sentencia "goto"

Su uso se desaconseja en la programación estructurada. Si embargo es útil para salir de ciclos (bucles). Antes de usar goto, primero debe declarar una etiqueta con label y preceder la sentencia donde desea saltar por esa etiqueta seguida de dos puntos (":"), como en el siguiente ejemplo:

```
function EstaCaracterEnCadena(char c, string s): boolean;  
  
var i, n: Integer;  
  
label CaracterEncontrado;  
  
begin  
  
    n := Length(s);  
  
    for i := 1 to n do  
  
        if s[i] = c then goto CaracterEncontrado;  
  
    Result := False;  
  
    Exit; // Sale de un procedimiento o función  
  
CaracterEncontrado:  
  
    Result := True;  
  
end;
```

Nunca use goto para saltar hacia adentro de un bucle u otra sentencia estructurada debido a que puede producir efectos impredecibles. No se permiten saltos de dentro hacia afuera, o de afuera hacia adentro de bloques try...except o try...finally. Tampoco se permiten saltos de y hacia otros procedimientos y funciones.

Siempre intente usar una alternativa estructurada, o por lo menos use el procedimiento Break:

```
function EstaCaracterEnCadena(char c, string s): boolean;  
  
var i, n: Integer;  
  
begin  
  
    Result := False;  
  
    n := Length(s);  
  
    for i := 1 to n do  
  
        if s[i] = c then begin  
  
            Result := True;  
  
            break; // Sale del bucle más interno  
  
        end;  
  
    end;  
  
end;
```

TIPOS BÁSICOS

Los tipos básicos de Object Pascal se pueden clasificar de la siguiente forma:

simples

ordinales

enteros

Shortint -128..127 8 bits con signo

Smallint -32768..32767 16 bits con signo

Longint -2147483648..2147483647 32 bits con signo

Integer -2147483648..2147483647 32 bits con signo

Int64	$-2^{63}..2^{63}-1$	64 bits con signo
Byte	0..255	8 bits sin signo
Word	0..65535	16 bits sin signo
Longword	0..4294967295	32 bits sin signo
Cardinal	0..4294967295	32 bits sin signo

caracteres

AnsiChar 1-bytes caracter ANSI
WideChar 2-bytes caracter UNICODE
Char AnsiChar

booleanos

Boolean 1-byte (valor booleano de 8 bits)
ByteBool 1-byte (valor booleano de 8 bits)
WordBool 2-byte (valor booleano de 16 bits)
LongBool 4-byte (valor booleano de 32 bits)

enumeraciones

type <tipo> = (valor1, ..., valorn);
Ejemplo: type Palos = (Oro, Espada, Copa, Basto);

subrangos

type <tipo> = valor1..valor2;
Ejemplo: type ExceptoOro = Espada..Basto;
Mayusculas = 'A'..'Z';
Horas = 0..23;

reales

Real48 $2.9 \times 10^{-39} \dots 1.7 \times 10^{38}$ 11-12 dígitos 6 bytes

Single $1.5 \times 10^{-45} \dots 3.4 \times 10^{38}$ 7-8 dígitos 4 bytes

Double $5.0 \times 10^{-324} \dots 1.7 \times 10^{308}$ 15-16 dígitos 8 bytes

Extended $3.6 \times 10^{-4951} \dots 1.1 \times 10^{4932}$ 19-20 dígitos 10 bytes

Comp $-2^{63}+1 \dots 2^{63}-1$ 19-20 dígitos 8 bytes

Currency -922337203685477.5808

.. 922337203685477.5807 19-20 dígitos 8 bytes

Real Double

cadenas

ShortString 255 caracteres Compatibilidad hacia atrás

AnsiString $\sim 2^{31}$ caracteres caracteres de 8 bits (ANSI)

WideString $\sim 2^{30}$ caracteres Caracteres Unicode

String AnsiString

estructurados

conjuntos

type <tipo> = set of <tipo-ordinal>;

arreglos

estáticos

type <tipo> = array [<tipo-ordinal>] of <tipobase>;

type <tipo> = array [<tipo-ordinal1>,
<tipo-ordinal2>,...] of <tipobase>;

dinámicos

type <tipo> = array of <tipobase>;

registros

```
type <tipo> = record  
    <listacampos1>: <tipo1>;  
    ...  
    <listacamposN>: <tipoN>;  
end;
```

ficheros

```
type <tipo> = file of <tipobase>;  
type <tipo> = file; // Fichero sin tipo
```

clases

// Las dejamos para el próximo newsletter

referencias a clases

```
type <tipo> = class of <tipo-class>;
```

interfaces

// Las dejamos para el próximo newsletter

punteros

```
type <tipo> = ^<tipobase>;
```

procedimientos (punteros a procedimientos o funciones)

```
type <tipo> = <declaración de procedimiento o función>;
```

```
type <tipo> = procedure(<parámetros>) of object;
```

Ejemplos: type TFuncionEntera = function: Integer;

```
TProcedure = procedure;
```

TStrProc = procedure(const S: string);

TMathFunc = function(X: Double): Double;

TMethod = procedure of object;

TNotifyEvent = procedure(Sender: TObject) of object;

variantes

Las variables tipo Variant pueden contener un valor de cualquier tipo, excepto los tipos estructurados, punteros e Int64. Sin embargo, sí pueden contener arreglos dinámicos y una tipo especial de arreglos estático llamado arreglo variante.

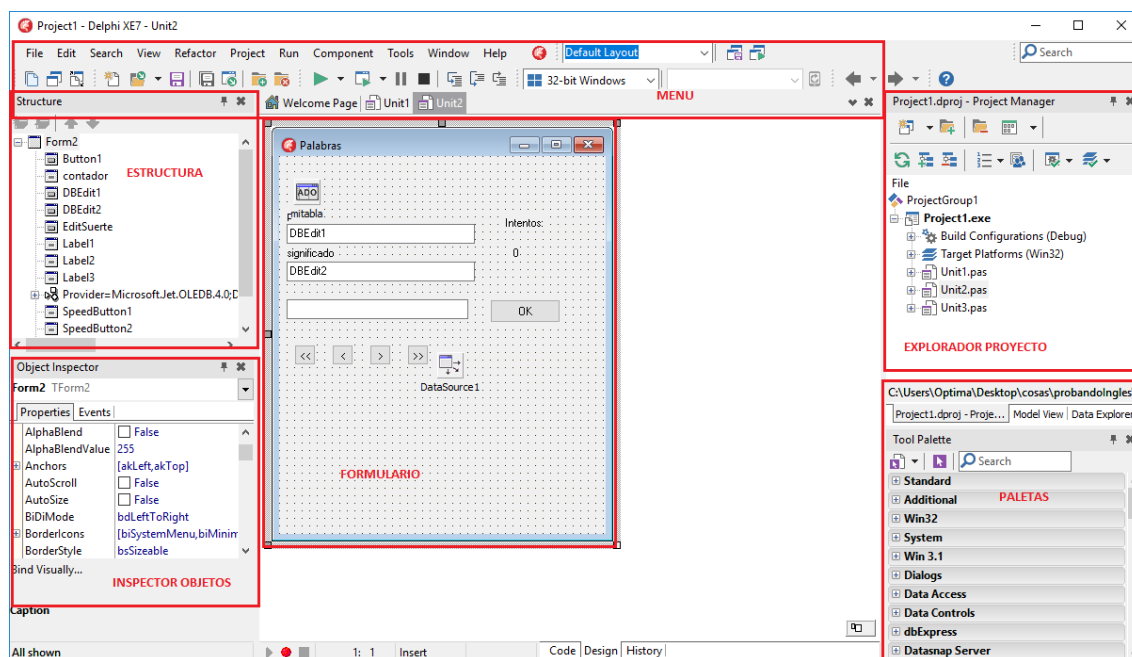
3. INTRODUCCIÓN A DELPHI.

Vamos a comenzar a familiarizarnos con **Delphi** a través de una pequeña demostración *-paso a paso-* sobre como se utiliza el **Entorno** y como se crean **Programas**; finalmente, nos introduciremos muy básicamente en el amplio mundo de los **Proyectos**.

Comencemos dando la **‘Bienvenida’** a **Delphi** a todos los futuros usuarios de esta **Herramienta**.

COMO CREAR PROGRAMAS.

Al entrar en **Delphi**, nos encontraremos un **Entorno de Desarrollo** gráfico muy estructurado, esquemático y fácilmente legible; en la **Pantalla** se han dispuesto **diferentes Ventanas** claramente identificables:



Cada una de estas **Ventanas** se utilizará para realizar una **Función** específica en el **Entorno**.

Ya hemos mencionado alguna vez que el **Entorno Visual Delphi** utiliza “**Objetos prefabricados**” (alguien en alguna parte de este “*mundillo informático*”, ha estado programando muchas horas). Desde este punto de partida, es fácil deducir que existirá un **ciclo o secuencia de operaciones** que debemos seguir para crear una **Unidad (programa)**:



- 1.- Selección del Componente (Objeto “*prefabricado*”).
- 2.- Ubicación del Objeto.
- 3.- Personalización del Objeto (Aspecto y Comportamiento).



- 4.- Compilación.
- 5.- Depuración.
- 6.- Ejecución.

Bien, veamos que tal lo hacemos...

DEMO.

“Saludos al Mundo”.

En este ejercicio sólo deseamos escribir en la Ficha que nos va a presentar el programa un mensaje de saludo, y esperar la pulsación de un ‘Click’ sobre un

botón para finalizar la ejecución del programa o la pulsación 'Click' sobre otro botón que presentará al **Autor**.



Deduciendo de lo dicho anteriormente, lo primero que tendré que hacer será averiguar si existe un **Objeto** "prefabricado" que me facilite la tarea de escribir mensajes.

Existe !! (luego "Pienso").



Hay un **Objeto (Componente)** en la **Paleta Standard** llamado **TLabel** que puedo emplear a tal efecto.

La operativa es simple :

1.- Pulso 'Click' sobre el componente.

Selección

2.- Me sitúo sobre la Ficha.

3.- Vuelvo a pulsar 'Click'.

Ubicación

Ahora ya tengo situado el **Componente** sobre la **Ficha**; según el ciclo que hemos expuesto, el tercer paso a realizar será : **Personalizar el aspecto y funcionamiento** de dicho **Objeto**.

La tarea de **personalizar** el **Objeto** se realiza desde la **Ventana** llamada: **Inspector de Objetos**.

Utilizando la "pestaña" **Propiedades** podemos diseñar el Aspecto, y desde **Eventos**, definir su Funcionamiento.

La tarea asignada a este **Componente** en este ejercicio, es la de presentar un mensaje, no deseamos que desde él se realice ninguna acción determinada; consecuentemente, sólo personalizaremos sus Propiedades.

Propiedades y sus valores:

Caption “Saludos al Mundo !!!”

Font Arial, 14p. (Cuadro de diálogo)

Align alTop

Alignment taCenter

Al asignar estos valores a las Propiedades del Objeto TLabel seleccionado, hemos conseguido el aspecto que se propuso en la pantalla del enunciado del ejercicio.

A continuación emplearemos un Objeto del tipo Tbutton para gestionar la “salida” del programa.

El **Objeto (Componente) TButton** también se encuentra en la **Paleta Standard**.



La operativa que emplearemos sobre este Componente es idéntica a la empleada con TLabel : **Seleccionar**, **Ubicar** y finalmente **Personalizar**.

Personalizamos ...

Propiedades:

Caption “&Salir”

Fonts Ms Sans Serif, 12p., Neg.

Eventos:

OnClick (Doble ‘Click’)

Escribir código fuente:

Close;

Como podemos comprobar, en este **Componente** si que hemos personalizado su comportamiento, el **Evento OnClick** lo utilizaremos para lanzar el proceso que se encargará de finalizar el programa.

A estas alturas de ejercicio, ya podemos probar a ver que tal funciona, pulsaremos <F9>, y ...

Hasta aquí hemos conseguido que la presentación y las operaciones más elementales funcionen, ahora debemos seguir completando el ejercicio.



En la Paleta **Additional**, existe un nuevo tipo de botón **TBitBtn** que utilizaremos para permitir al Autor que se presente. Este tipo de botón, permite al programador insertar imágenes (***.BMP**) ilustrativas del contenido y/o finalidad del componente, así como asignarle una función predeterminada (Ejm.: Abort, Ok, etc...) sin necesidad de programar nada.

Propiedades:

Caption "Autor"

Font..... Ms Sans Serif, 12p., Neg./Cur.

Glyph Load

\Delphi\Images\Splash\16Color\Athena

Eventos :

OnClick Antes de escribir el código fuente correspondiente al Evento, tendremos que diseñar la Ficha que presentará al Autor ; para ello utilizaremos un nuevo Form.

Luego volveremos a este Evento.

Antes que se vaya la luz, mejor será que guardemos los archivos.

Los ARCHIVOS ? ?

El Archivo del Proyecto..... **.DPR**

El Archivo del Código Fuente. ... **.PAS**

El Archivo de los Componentes. **.DFM**

Guardaremos estos archivos con los nombres que les asigna Delphi por defecto:

 **Project1.dpr**

 **Unit1.pas**

 **Unit1.dfm**

Para crear un nuevo Formulario iremos al menú **File** y seleccionaremos la opción **New > VCL Form**. Una vez hecho esto, aparecerá en pantalla una nueva Ficha en blanco cuyo **título (Caption)** será **Form2**.

El contenido del **Form** que vamos a diseñar será el siguiente :



A simple vista, se pueden observar algunos Componentes ya utilizados (**TLabel**, **TBitBtn**), y otros nuevos: **TPanel** y **TImage**.



El Componente **TPanel** hemos de considerarlo de manera muy amplia ya que por su funcionalidad lo utilizaremos a menudo en el diseño de Formularios. Su función principal será la de **ser contenedor de otros componentes**, agrupándolos en su interior y haciendo que sean considerados como un conjunto integrado.

Propiedades:

Caption..... Eliminar el Título.

BevelWidth..... 3

Color..... clAqua



Utilizaremos 3 Componentes del tipo **TLabel** para escribir los mensajes que deben aparecer en el interior del componente **TPanel**.

Propiedades:

①

Caption..... "I.F.P. Alacant - 3"

Alignement..... taCenter

Font..... Book Antiqua, 14 Bold.

②

Caption “Departament d’Informàtica de Gestió”

Alignement taCenter

Font Windsor, 9 Neg.

③

Caption “Desenvolupat && Copyright 1996”

Alignement taCenter

Font Calisto MT, 9 Bold.



Utilizaremos el Componente **Timage** -situado en la **Paleta Additional**- para insertar imágenes en nuestros Formularios. Permite la utilización de varios tipos de **Archivos Gráficos**:

Bitmaps **.BMP**

Icons **.ICO**

Enhanced Metafiles **.EMF**

Metafiles **.WMF**

En este ejercicio, lo utilizaremos para insertar un Icono que se encuentra en un Archivo en nuestro directorio de trabajo.

Propiedades:

Hint “El nostre departament es

Curs”
dessitja que tingues un bon

Picture (Cuadro de Diálogo)

Load

\Delphi\Curso\lcofp3

Abrir

OK

—————→ (TIcon)

ShowHint..... True

Situaremos el Icono en el centro del Objeto TPanel.



A continuación situaremos en el **Formulario**, fuera del Objeto TPanel un **Componente** botón de tipo **TBitBtn**. Lo utilizaremos asignándole una función predeterminada (de las muchas que contiene el entorno); Lo forzaremos a que tome el valor **Ok** en la salida del Formulario.

Propiedades:

Caption..... OK

Font Arial, 12 Neg.

Kind bkOk (*)

Layout blGlyphLeft

ModalResult..... mrOk

(*) Es interesante observar como, al asignarle valor a la propiedad Kind, muchas otras propiedades los toman -por defecto-.

¿ Qué tendríamos que haber hecho si no hubiera existido la posibilidad de asignarle una Función determinada a este Componente ?.

La respuesta es sencilla ... **Escribir tal Función** (Evento *OnClick*).

...*Si no recuerdo mal, nos queda una tarea pendiente...*

Todavía hemos de escribir el *Evento OnClick* correspondiente a la activación del botón (tipo **TBitBtn**) del **Form1** que servirá para presentar el Form2.

Volviendo a activar **Form1**, pulsamos un **doble “Click” sobre el Objeto TBitBtn** (esta es otra manera de presentar la Ventana de Código Fuente correspondiente a la Función que hemos de escribir para describir la respuesta del Evento).

Escribiremos:

Form2.ShowModal ;

Finalmente, añadiremos en el módulo Uses del Form1 la entrada correspondiente a la utilización por parte de éste de Form2:

Uses

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls, Buttons, Unit2;

Ahora debemos comprobar el funcionamiento del Proyecto que acabamos de realizar.

Recordemos :

Presentará un Formulario en Pantalla, cuya finalidad es la de “Saludar al Mundo”, además podemos :

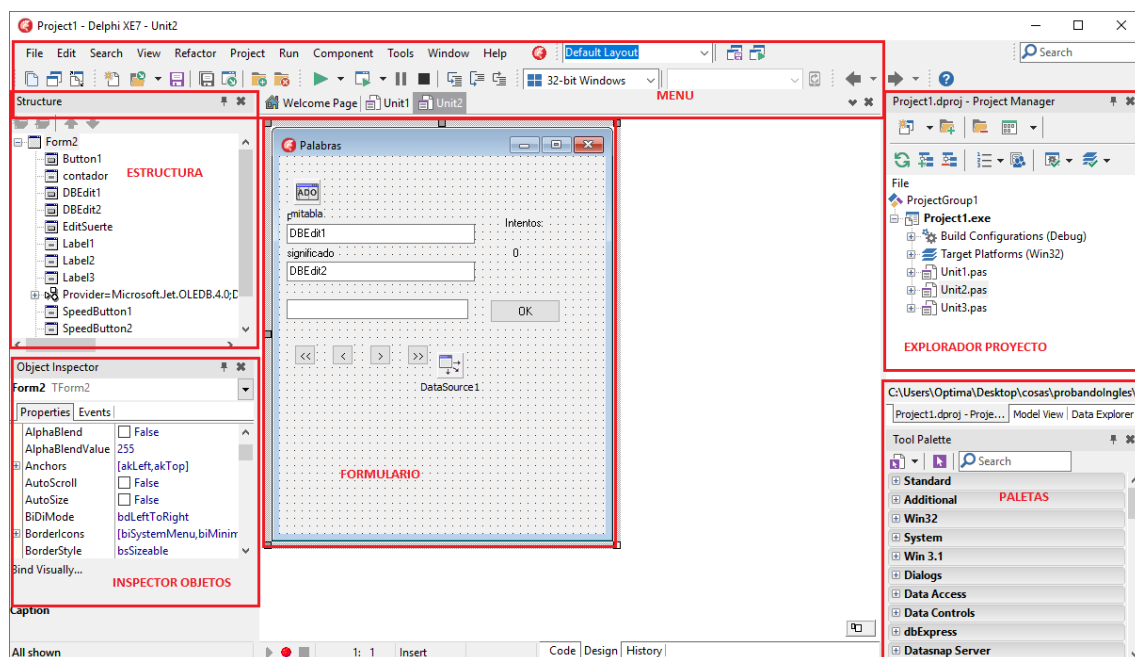
1.- Finalizar el programa.

2.- Activar un segundo Formulario que presenta al Autor.

El ciclo del proyecto queda cerrado al existir la posibilidad de ir hacia adelante y/o volver hacia atrás, hasta que el usuario decida finalizar su ejecución.



4. EL ENTORNO DE TRABAJO.



Es momento de ir a **Inicio**, **Todos los programas**, y después buscar la carpeta llamada: " **Embarcadero Rad Studio XE**", y dar clic sobre de ella, para mostrar su contenido. Tal y como se muestra en la figura siguiente.

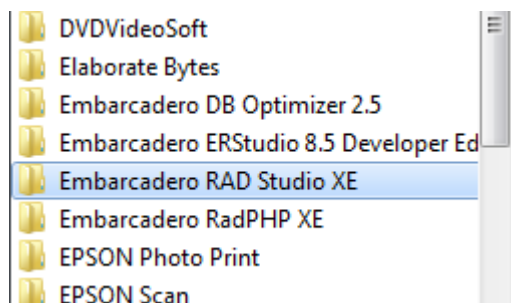
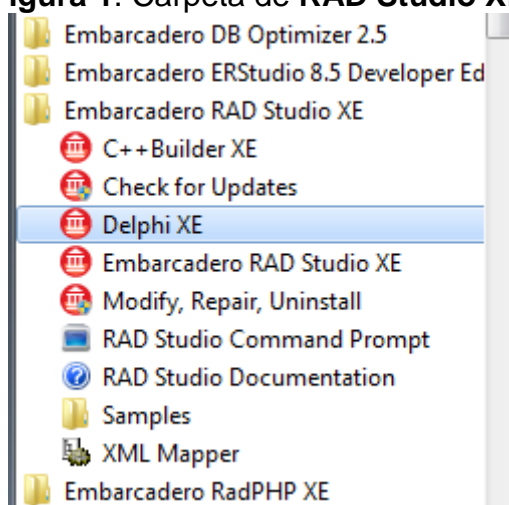
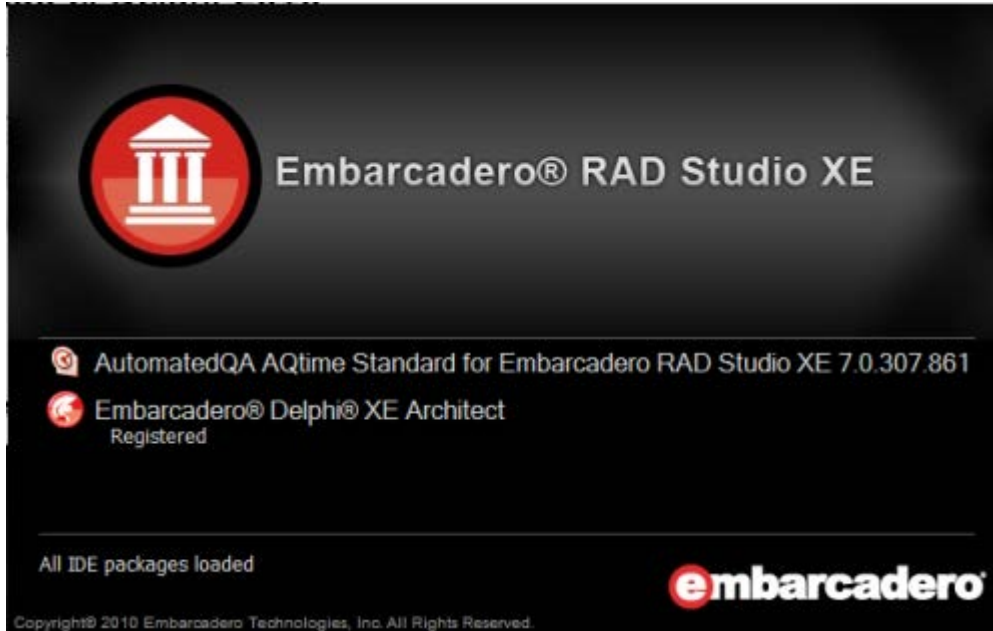


Figura 1. Carpeta de RAD Studio XE.



Hemos de dar un clic sobre la opción **Delphi XE**, la cual está sombreada en color azul en la figura anterior. Inmediatamente después empezara la carga del **Delphi XE**, mostrando la siguiente pantalla de inicio (SplashScreen).



Después de unos cuantos segundos, dependiendo de tu equipo de cómputo, se mostrara el **IDE** (siglas en ingles de Entorno de Desarrollo Integrado), de **Delphi XE**, tal y como se puede ver en la figura siguiente.



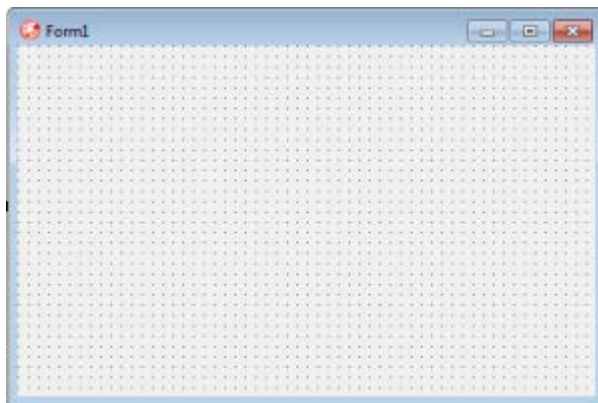
Las herramientas disponibles en el **IDE**, dependen de la edición del **RAD Studio** que tu estes usando, esta versión incluye lo siguiente:

- **Página de Bienvenida (Welcome Page)**



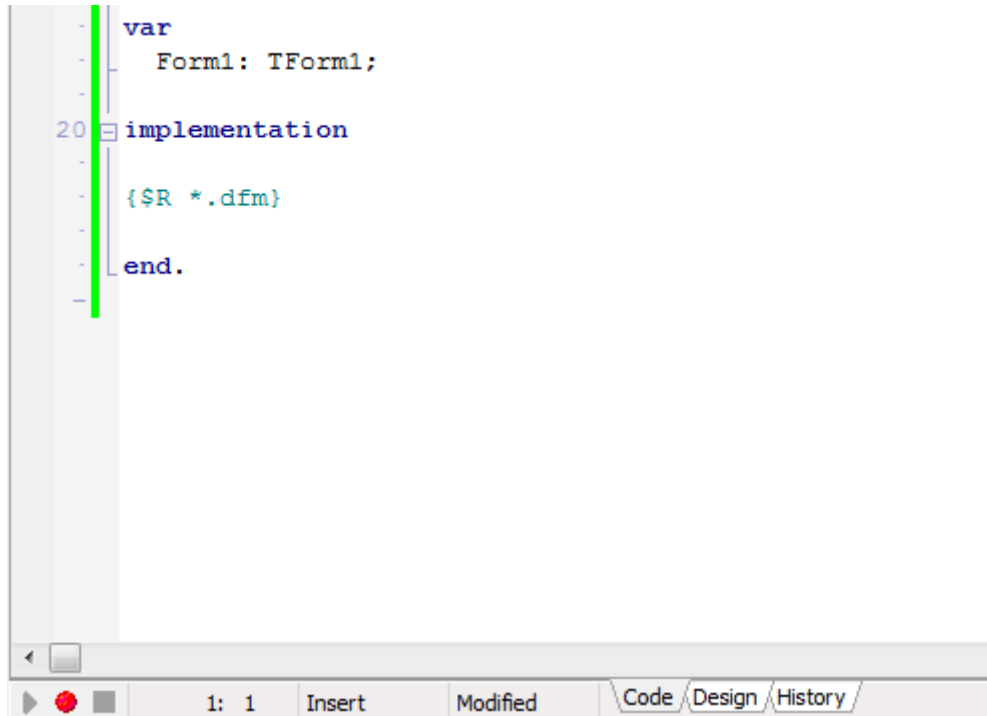
Cuando tu abres **RAD Studio**, la **Página de Bienvenida (Welcome Page)** aparece con un número de ligas para desarrollar recursos, como lo son; artículos, entrenamiento, y ayuda en línea. Así como tu puedes desarrollar proyectos, tu puedes acceder rápidamente a ellos ,de la lista d eproyectos recientes en la parte superior de la página. Si tu cierras la **Página de Bienvenida (Welcome Page)**, tu puedes reabirla al seleccionar **View > Welcome Page**.

- **Las Formas (Form)**



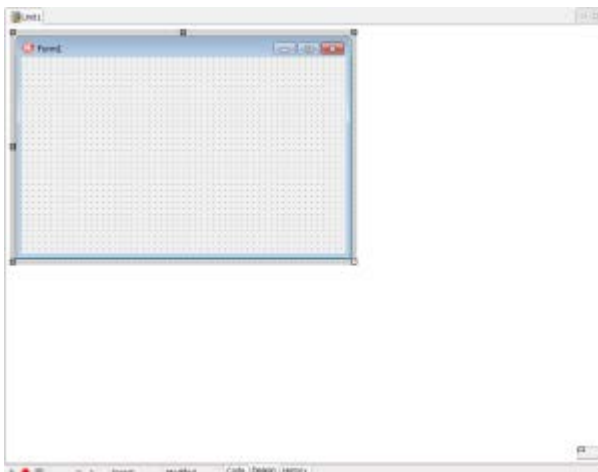
Típicamente, una forma representa a una ventana o página **HTML**, en la interfaz gráfica que tú estás diseñando. En tiempo de diseño, una forma es mostrada en la *Superficie del Diseñador*. Tú agregas componentes desde la **Paleta de Herramientas (Tool Palette)** a la forma para crear la interfaz de usuario.

RAD Studio provee una rica librería de formas. Selecciona la forma que mejor se adecue al diseño de tu aplicación. Para cambiar entre el **Diseñador** y el **Editor de Código**, solo basta un click sobre los tabs debajo del IDE.



Para agregar formas, selecciona **File > New > Other**. (Esto lo veremos mas adelante en práctica)

- **Diseñador de Formularios (Form Designer)**



El **Diseñador de Formularios (Form Designer)**, es mostrado automáticamente en el panel central cuando tu estas usando una forma. La apariencia y funcionalidad del **Diseñador de Formularios** depende de el tipo de foma que tu estes usando. Por ejemplo, si tu estas usando

una **FormaWEB (WebForm)**, el Diseñador mostrara un editor de **Tags** (etiquetas) **HTML**. Para acceder al **Diseñador**, solo da click sobre la pestaña (tab) **Design** (Diseño) en la parte baja del **IDE**.

En este punto es necesario aclarar, que: En **Delphi** encontraremos **Componentes Visuales** y **No Visuales**, así que hablemos de los primeros.

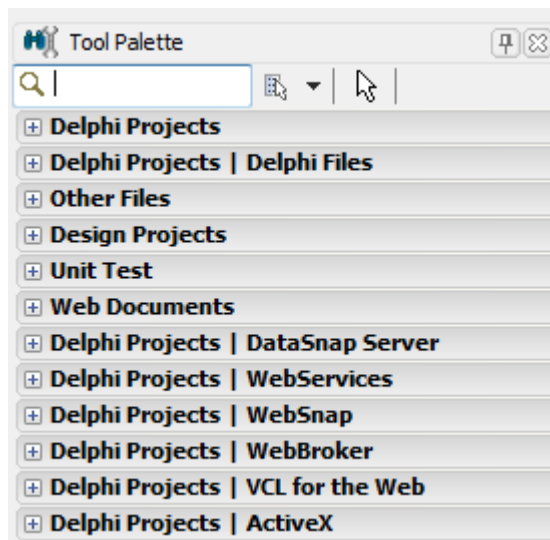
- **Componentes Visuales**

Los componentes Visuales aparecen sobre el form en tiempo de diseño y son visibles para el usuario final en tiempo de ejecución. Estos incluyen muchas cosas, como Botones, Etiquetas, Barras de Herramientas y Cajas de Lista.

Componentes No Visuales

Son aquellos que se colocan sobre el formulario pero, solo son vistos por el desarrollador, ya que solo realizar funciones a nivel de sistema y no de manera gráfica. Esto quiere decir que el usuario final no los puede ver, literalmente, pero si puede comprobar su funcionamiento.

- **Paleta de Herramientas (Tool Palette)**

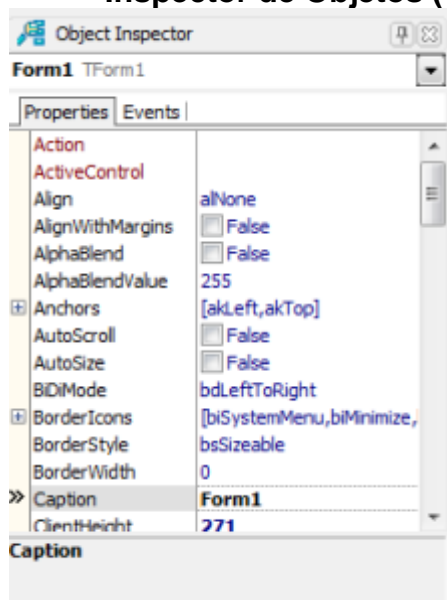


La **Paleta de Herramientas (Tool Palette)**, localizada sobre la columna derecha, contiene elementos para ayudarte en el desarrollo de tu aplicación. Los elementos mostrados dependen de la vista actual.

Por ejemplo, si tu estas viendo una forma en el Diseñador, la Paleta de Herramientas muestra componentes que son apropiados para ese formulario. Tu puedes dar doble click a un componente para agregarlo a tu forma. Tambien

puedesarrastrar a la posicion que tu desidas en tu forma. Si tu estas viendo el codigo en el Editor deCodigo, la Paleta de Herramientas muestra segmentos de codigo que tu puedes agregar a tu aplicaci3n.

• Inspector de Objetos (Object Inspector)



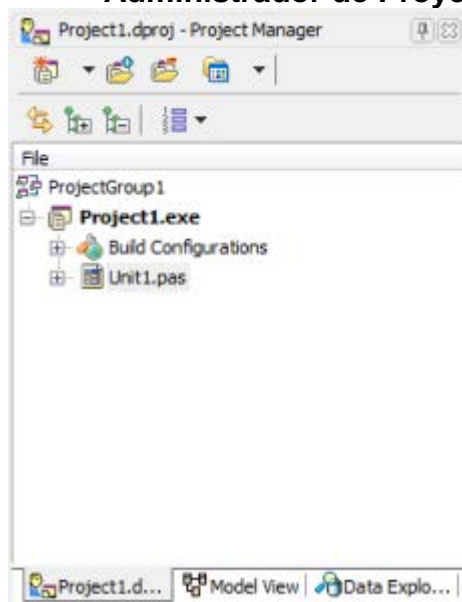
El **Inspector de Objetos (Object Inspector)**, localizado a la izquierda, te permite asignar propiedades en tiempo de dise1o y crear manejadores de eventos para componentes. Esto provee la conexi3n entre la apariencia visual de tu aplicaci3n y el codigo que hace que tu aplicaci3n funcione. El **Inspector de Objetos (Object Inspector)** contiene 2 etiquetas (tabs): **Propiedades (Properties)** y **Eventos (Events)**.

Usa la pesta1a **Propiedades** para cambiar los atributos f3sicos de tus componentes. Dependiendo de tu selecci3n, algunas opciones de categoria te permitir3n introducir valores en una caja de texto, mientras otros requerir3n que selecciones valores de una caja desplegable. Para operaciones **Booleanas (SI/NO)**, t3 puedes intercambiar entre **True (Verdadero)** y **False (Falso)**. Despues de que t3 cambies atributos f3sicos a tus componentes, t3 crearas manejadores de eventos que te permitir3n administrar o manipular los componentes.

Usa la pesta1a **Eventos (Events)** para especificar el evento para un objeto seleccionado por t3. Si el manejador de evento ya existe, usa la caja

desplegable y seccionalo. Por defecto, algunas opciones del Inspector de Onjetos (Object Inspector) están colapsadas (cerradas). Para expandir o abrir las opciones, click en el simbolo de (+) para mostrar la siguiente categoria.

- **Administrador de Proyectos (Project Manager)**



Los proyectos están hechos por muchos archivos de aplicación. El Administrador de Proyectos, localizado en la parte superior derecha, te permite ver y organizar tus archivos de proyecto como Formas, ejecutables, ensamblados, objetos, y archivos de librerías. Además con el Administrador de Proyectos puedes, remover, agregar y renombrar archivos usando la barra de herramientas o el menú contextual. Tu podrás invocar siempre la ventana de Dialogo de Opciones del Proyecto desde el menú contextual, en el Administrador de Proyectos.

- **EL EDITOR DE CÓDIGO FUENTE.**



El Editor de Código Fuente utiliza un entorno que intenta asumir la todas las tareas que se pueden realizar desde cualquiera de los Editores de Texto profesionales. Su funcionamiento se basa en la utilización de un Administrador de Páginas, donde cada página contiene el texto correspondiente a una Unidad fuente.

EL PRINCIPIO BASADO EN LOS MÓDULOS.

**BIBLIOTECA DE CLASES
ORINTADA A OBJETOS
VCL (Delphi)**

+

PROGRAMABILIDAD VISUAL

Página 32

- El Sistema Operativo.
- El Administrador de Almacenaje.
- El Administrador de Recursos.
- Con Contextos de Dispositivos.
- Con Controladores.

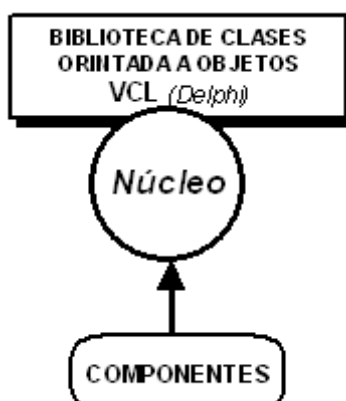
Se aprovecha el principio denominado **CASE** (**C**omputed **A**ided **S**oftware **E**ngineering - **D**esarrollo de Software Asistido por Ordenador) :

“ Desarrollo de Programas a partir de Módulos ‘prefabricados’, que deben enlazarse de una manera determinada. El tipo de enlace determina el desarrollo del programa “.

Una vez seleccionados y enlazados los Módulos que componen el programa; éste se puede ejecutar :

- Modo Intérprete.
- Código Intermedio + RunTime.....Visual Basic
- Código legible por la máquina.Delphi

COMPONENTES : ELEMENTOS DE CONSTRUCCIÓN.



○ **Standard** Contiene los elementos de Control de Windows.

Ejm. : Cuadros de Lista, Botón de Alternar, etc...

○ **Additional** Contiene los elementos que Windows generalmente no pone a disposición.

Ejm. : Botones gráficos, Componentes para mostrar Archivos Gráficos de diversos formatos, etc...

○ **Data Access** Componentes desarrollados para la captura rápida y cómoda de Bases de Datos.

Se pueden tratar Bases de Datos Locales como sistemas repartidos.

○ **Data Controls** Componentes variados para el manejo de Datos.

Ejm. : Elementos de Entrada, Botones de Opciones, etc...

○ **Dialogs** Contiene todos los Diálogos estándar de Windows.

○ **System** Varios componentes que ofrecen determinados servicios de Sistema de Windows.

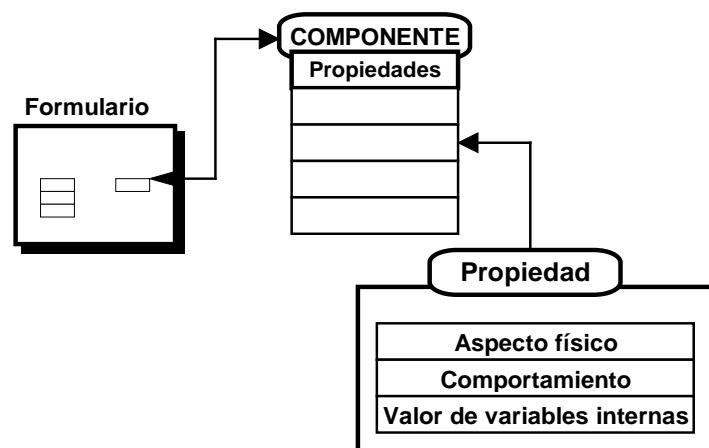
Ejm. : Reloj, Barras de Archivo, Funciones multimedia, DDE, OLE 2.0, etc...

○ **VBX** Contiene algunos elementos de VBX.

Ejm. : Botones de alternar nuevos, Observadores de Gráficos, Control para la representación de Diagramas.

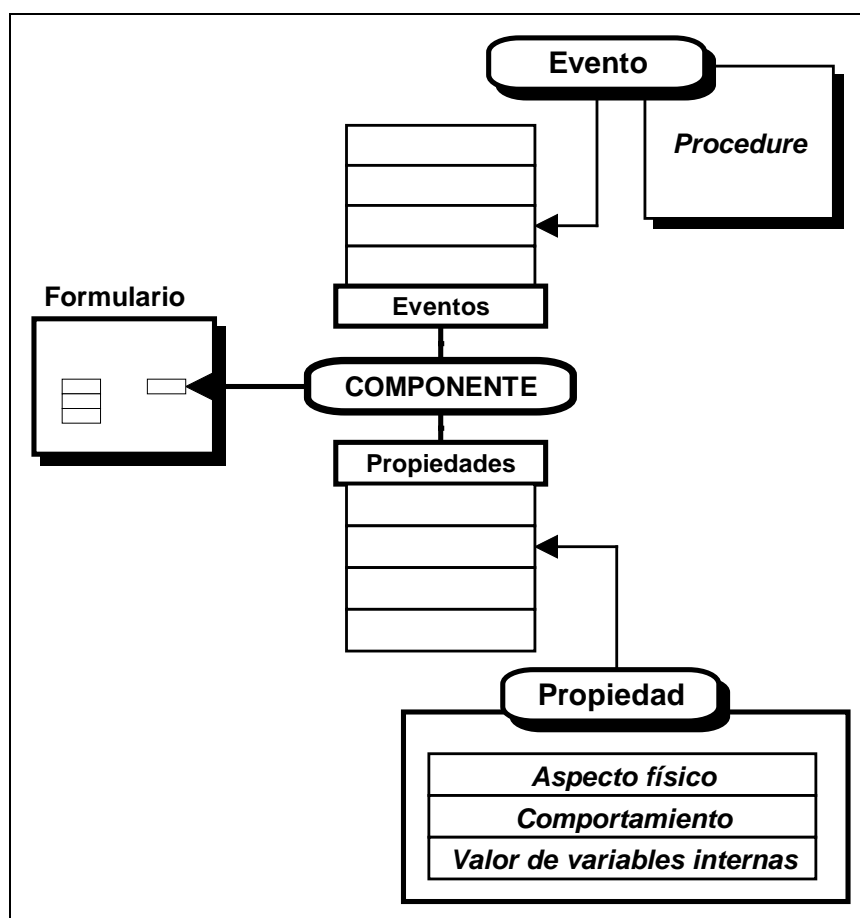
○ **Samples** Contiene algunos componentes útiles (Documentos con Código Fuente).

Ejm. : Barras de progresión, un Calendario, Cuadros de Texto especiales, etc...



Categorías de los Componentes :

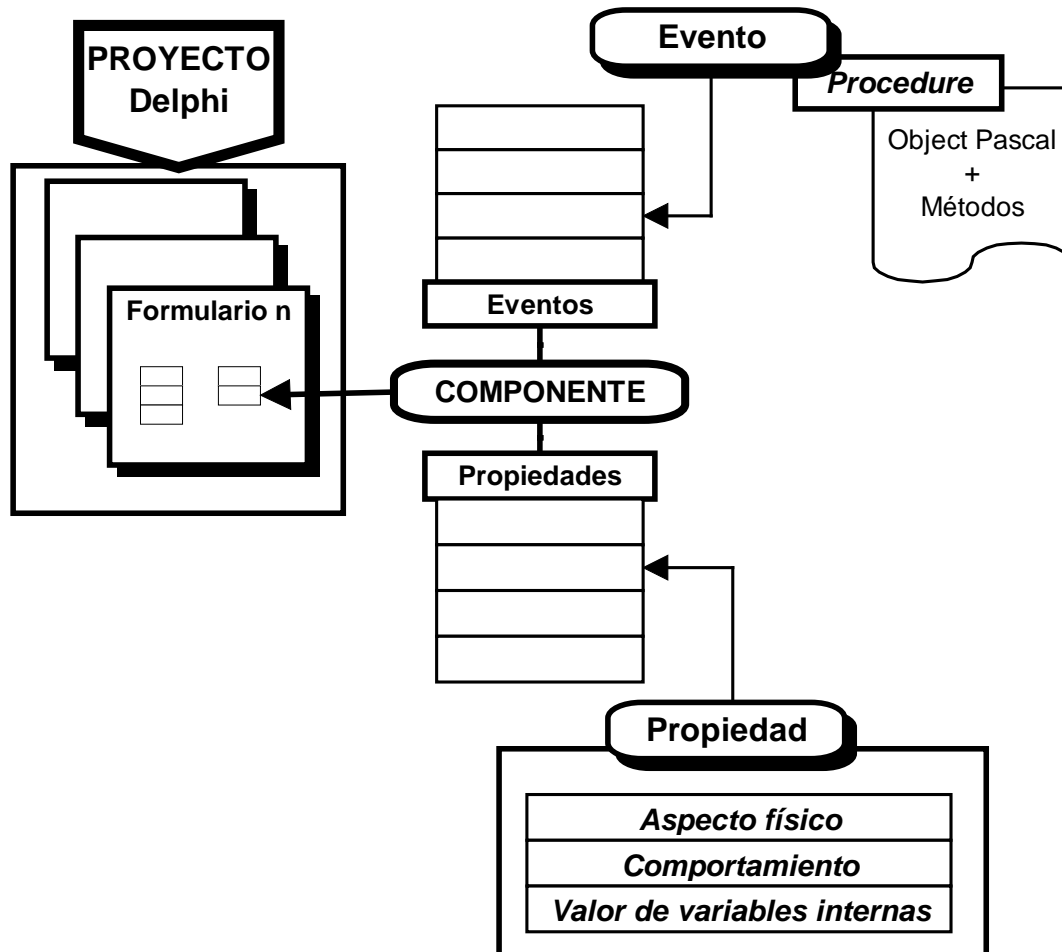
Completando aún más...



Los **EVENTOS** son tan importantes como las **Propiedades**, con ellos se pueden describir procesos que se ejecutarán según las especificaciones desarrolladas en el **Código Fuentes** que ha escrito.

En su **Código Fuente** se pueden incluir llamadas a procedimientos o funciones pertenecientes a la clase de la cual deriva el objeto que nos permiten realizar determinadas acciones sobre el componente. A estos procedimientos o funciones los llamaremos **MÉTODOS**.

Finalmente...



CONCEPTOS BÁSICOS - DEFINICIONES.

FICHA

FORM

FORMULARIO Es una **Ventana Windows** que nos servirá para comunicarnos con el **Usuario**.

En la **Ficha** podremos situar **Componentes** para realizar funciones determinadas.

Podemos partir de : ☐ Fichas vacías.

☐ Fichas “prefabricadas”.

Cualquier **Aplicación** (*Proyecto*) constará de :

☐ Una Ficha principal. (mínimo)

☐ Otras fichas. (opcionalmente)

OBJETO Todos los elementos del **Entorno de Delphi** son Objetos definidos.

Todo el desarrollo de Aplicaciones en **Delphi** está íntimamente ligado con la definición y uso de **Objetos**, por lo que es fundamental conocer la mecánica que **Object Pascal** utiliza para :

- Describir un Objeto.
- Definir sus características.
- Definir su funcionamiento.

Ejm. : La Ficha es un Objeto sobre el cual se sitúan otros Objetos ; la definición de sus características determinarán -por ejemplo- el Tipo de Formulario.

COMPONENTE Un **Componente** es cualquiera de los elementos que podemos insertar en una **Ficha**, tanto si su función es Visual o no lo es.

(*Función Visual* : se realiza interactuando con el Usuario).

A los Componentes que cuentan con una parte visual se les denomina : **CONTROLES**.

Características interesantes :

- Delphi permite la creación de nuevos **Controles** :
 - Desarrollados por el Usuario.
 - Desarrollados a partir de los existentes en su Librería (VCL).
- Los componentes son **Reutilizables**.
- Identificación de los Componentes :

NombreObjeto.NombreComponente

Ejm. : *TForm1.Button2*

TForm1.CheckBox1

PROPIEDAD Todos los Objetos Delphi son de uso general (tienen unas características “*por defecto*”), sin embargo, nos serviremos de sus **Propiedades** para personalizar su utilización y adecuarlos a nuestras necesidades.

Como programadores tradicionales es fácil pensar en las Propiedades como “**Variables**” pertenecientes al Objeto, que utilizamos para definir sus características -en realidad resulta cómodo pensar así-.

Desde el punto de vista del programador, las propiedades pueden parecer simples Variables, pero en muchas ocasiones esto no es así.

Ejm. : Imagina una Propiedad a la que llamamos **Envía** perteneciente a un Componente que facilite las comunicaciones. Su tarea consiste en “Lanzar” un proceso de transferencia de información, y esto no significa Asignación de Valores.

Tipos de Propiedades :

- Accesibles siempre.
- Accesible en tiempo de ejecución (*no en diseño*).
- De sólo Lectura (*sólo permite consultar su valor*).
- De sólo Escritura.

EVENTO Recordemos que la característica principal de la Programación en **Entornos Gráficos** es que está orientada a Eventos.

Los **Eventos** son “señales” que el **Entorno** recibe desde distintos elementos que interactúan son éste (*Ratón, Teclado, etc...*). Estos **Eventos** son redirigidos a las **Aplicaciones**, que en caso de aceptarlos, deben responder adecuadamente a ellos.

Gestión de Eventos:

- Desde Windows directamente.
- Desde el Lenguaje.
- Por el Programa en ejecución.

MÉTODO Es un **Procedimiento** o **Función** que nos permite realizar una determinada acción en el **Componente**, necesitando o no el pase de algún parámetro.

Identificación :

NomObjeto.NomCompon.NomMetodo(Parám)

5. PROGRAMACIÓN ORIENTADA A OBJETOS

¿QUÉ ES UN OBJETO?

Como su nombre indica, la programación orientada a objetos (POO) se basa fuertemente en el concepto de objeto. En nuestras vidas estamos acostumbrados a tratar con objetos -televisores, lámparas, libros, etc...- pero cuando encendemos la televisión, no distinguimos entre sus elementos físicos (selectos de canal, antena, tubo de imagen) y su comportamiento (proporcionar imagen y sonido). Simplemente la encendemos y seleccionamos un canal.

Viéndolo de la forma más simple un objeto es un registro, que se caracteriza porque además de contener miembros de distintos tipos, como números, cadenas, punteros, etc., también es capaz de contener definiciones de procedimientos y funciones.

Antes de poder crear un objeto debemos crear un tipo. Para definir un tipo de objeto utilizaremos una estructura similar a la de un registro, con la diferencia de que cambiaremos la palabra **Record** por **Class**. Tras esta cabecera

dispondremos los miembros que formarán parte del objeto, que además de ser variables de distintos tipos, también pueden ser funciones y procedimientos. La definición de un tipo de objeto finalizará con la palabra **End**. A los procedimientos y funciones que forman parte del objeto se les llama genéricamente métodos.

Con el fin de ir trabajando sobre un ejemplo práctico partamos del diseño de un tipo de objeto simple al que vamos a llamar **TPersona**. La definición podría ser la siguiente:

```
TPersona = Class { Tipo de objeto TPersona}

    Apellido: string[30];

    Nombre: string[20];

    Edad: integer;

    Procedure InicDatos; {Método que se encargará de inicializar el
objeto}

End;
```

La implementación del método **InicDatos** podría ser:

```
Procedure TPersona.InicDatos;

Begin

Nombre:= ' ';

Apellido:= ' ';

Edad:=0;

End;
```

ENCAPSULACIÓN

Generalmente, partiendo de un correcto diseño de los tipos de objetos, los datos que se declaren en el interior del objeto sólo serán manipulados por los métodos de ese objeto y recíprocamente, los métodos del objeto estarán especializados en la manipulación de los datos de ese objeto. Esta característica de los objetos se denomina encapsulación, ya que en cierta forma aísla a los datos y métodos de un objeto como si se tratase de una cápsula, impidiendo que sean usados unos parámetros erróneos al llamar a un método, o que un determinado procedimiento manipule las variables que no es.

USO DE UN OBJETO

Una vez definido el tipo de objeto, lo primero que hay que hacer es declarar una variable del tipo de objeto, en nuestro caso **TPersona**:

Var

UnaPersona : TPersona;

Sin embargo esto no basta, ya que al crear una variable de este tipo disponemos de lo que se llama una referencia a un objeto del tipo **TPersona** pero el objeto en sí aún no existe, hace falta crear lo que habitualmente se conoce como un instancia del objeto. Una de las formas en que podemos crear una instancia del objeto consiste en usar el procedimiento **New** utilizando posteriormente **Dispose** para liberar la memoria asignada.

Var

Una Persona : Tpersona;

Begin

New(UnaPersona); {Creamos una instancia del objeto}

.

. {Resto del programa}

.

Dispose(UnaPersona); {Liberamos la memoria asignada}

End.

PARTES PUBLICAS, PRIVADAS Y PROTEGIDAS

Las declaraciones realizadas en la definición de un objeto, tanto variables como métodos, pueden agruparse en tres apartados distintos, que se iniciarán con las palabras reservadas **Public**, **Private** y **Protected**. Las declaraciones existentes en el grupo **Public** serán públicas y por lo tanto otros programas o módulos podrán acceder a estos miembros. En el apartado **Private** podemos incluir la declaración de todos aquellos miembros que no deseemos poner a disposición de terceros. Todas las variables y métodos que incluyamos en esta parte serán accesibles sólo desde los métodos del objeto.

Por último, en el apartado **Protected** tendremos que los identificadores no serán accesibles por otros programas o módulos, pero sí tendrán acceso a ellos los objetos derivados del nuestro. El concepto de derivación y herencia lo

veremos más adelante y comprenderemos mejor el concepto de la parte protegida.

CONSTRUCTORES Y DESTRUCTORES

Aunque anteriormente hemos dicho que para crear una instancia del objeto utilizamos el procedimiento **New** y para eliminarla el procedimiento **Dispose**, los procesos de inicialización y destrucción de objetos se realizan en POO mediante los llamados constructores y destructores.

El constructor es el encargado de construir un objeto, asignarle la memoria necesaria y darle valores iniciales a los distintos miembros. Cualquier método que vaya a actuar como constructor cambiará la palabra **Procedure**, tanto en la cabecera como en la implementación del método, por la palabra **Constructor**. Aunque un constructor pueda recibir parámetros, no puede devolver un valor.

El método cuya finalidad es destruir el objeto, liberando memoria se le conoce como destructor. Cualquier método que vaya a actuar como destructor cambiará la palabra **Procedure**, tanto en la cabecera como en la implementación del método, por la palabra **Destructor**. Un destructor no puede ni recibir ni devolver parámetros.

Redefinamos nuestro objeto **TPersona** para poder ver un ejemplo de lo dicho anteriormente:

```
TPersona = Class { Tipo de objeto Tpersona}
```

```
Private
```

```
    Apellido: string[30];
```

```
    Nombre: string[20];
```

```
    Edad: integer;
```

```
Public
```

```
    Constructor Inicializa;
```

```
    Destructor Libera;
```

```
    Procedure NuevoNombre(NuevoNom,Nuevo Apell : string);
```

```
End;
```

La implementación de los métodos será como sigue:

```
Constructor TPersona.Inicializa;
```

Begin

Apellido:= ' ';

Nombre:= ' ';

Edad:=0;

End;

Destructor TPersona.Libera;

Begin

Destroy;

End;

Procedure TPersona.NuevoNombre(NuevoNom,Nuevo Apell : string);

Begin

Nombre:=NuevoNom;

Apellido:=NuevoApell;

End;

HERENCIA

Aunque los objetos contienen sus propios métodos y datos, también pueden heredarlos de otros objetos. Supongamos que queremos crear un objeto **TAlumno**, lo primero que podemos pensar es que algunas de las variables del objeto ya existen en el objeto **TPersona**, por lo tanto aprovechemos esta nueva característica de la POO.

A la acción de crear un tipo de objeto a partir de otro tipo ya existente se le llama derivación. Al tipo original se le conoce como tipo base o ascendiente, mientras que al nuevo tipo, que hereda las características del anterior se le llama tipo derivado o descendiente. Cuando se desea derivar un tipo de objeto a partir de otro, detrás de la palabra **Class** y entre paréntesis habremos de facilitar el tipo del objeto base:

TAlumno = Class(TPersona) { Tipo de objeto TAlumno}

Private

```
Curso: string[4];
```

```
Profesor: string[30];
```

```
Public
```

```
Constructor Inicializa;
```

```
End;
```

LIMITACIONES DE ACCESO

Anteriormente vimos que los miembros de un objeto pueden definirse en tres apartados diferentes. Los miembros **Nombre** y **Apellidos** de **TPersona** son privados, lo que quiere decir que sólo podrán ser utilizados por los métodos de este objeto. Ni siquiera **TAlumno**, que es descendiente de **TPersona**, tiene acceso a esos miembros. Para poder ser accesibles basta con cambiar la palabra **Private** por **Protected**.

Por lo tanto, y si hiciéramos el cambio mencionado, la definición del método constructor del objeto **TAlumno** será:

```
Constructor TAlumno.Inicializa;
```

```
Begin
```

```
TPersona.Inicializa;
```

```
Curso:= ' ';
```

```
Profesor:= ' ';
```

```
End;
```

POLIMORFISMO

Es la característica más compleja de la POO, y como su propio nombre indica, se trata de algo capaz de presentarse o adoptar diversas formas. Es decir, que podremos manipular objetos de distintos tipos de forma genérica, ahorrando mucho código.

Supongamos que hemos creado un tipo **TVehículo**, que vamos a utilizar como base para derivar tres descendientes, **TBicicleta**, **TCiclomotor** y **TAutomovil**, de los cuales a su vez se dan a derivar otros tipos, construyendo la jerarquía que mostramos en la siguiente figura:

Analicemos esta jerarquía. Si el tipo **TVehículo** incorpora un cierto miembro, este será heredado por todos los demás. Imaginemos que queremos guardar una serie de objetos de los tipos anteriores en una matriz, lo que se puede hacer perfectamente definiéndola del tipo **TVehículo**, ya que una matriz podría

contener objetos de cualquiera de los tipos descendientes. Mediante esa matriz y un bucle podríamos llamar a un método **Nombre** que se encargara de devolver el nombre de cada objeto, o un método **Ruedas** que devolviera el número de ruedas. Lógicamente cada una de los tipos debería incorporar su propios métodos **Nombre** y **Ruedas**, ya que los valores a devolver serían distintos. En los listados siguientes podemos ver la implementación de la jerarquía de la figura anterior:

Program Polimorf;

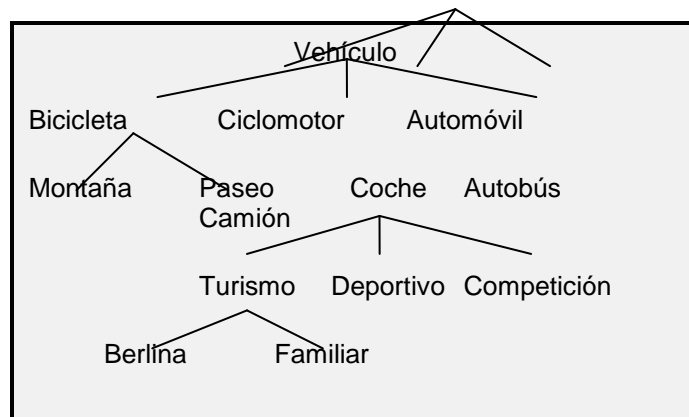
Uses

Vehiculo, WinCrt;

Var

X: Array[1..5] Of TVehiculo; { Matriz de objetos TVehiculo }

N: Integer;



Begin

{ Creamos una serie de objetos derivados de TVehiculo }

X[1] := TBicicletaMontana.Inicializa(18);

X[2] := TAutomovil.Inicializa(5, False, 110);

X[3] := TBicicleta.Inicializa;

```
X[4] := TBicicletaMontana.Inicializa(21);

X[5] := TAutomovil.Inicializa(4, True, 280);

WriteLn('Tipo de vehículo, Ejes, Ruedas, Motor, Características adicionales'); WriteLn;

For N := Low(X) To High(X) Do { Recorrer toda la matriz }

  With X[N] Do { Asumir la referencia X[N] }

    Begin

      Write(Nombre, ', ', NumeroDeEjes, ', ', NumeroDeRuedas, ', ',

FuncionaConMotor);

      If X[N] Is TBicicletaMontana Then{ Si el vehículo es bicicleta de montaña }

        { hacer moldeado de tipo para acceder al método NumeroDeCambios }

        Write(', con ', (X[N] As TBicicletaMontana).NumeroDeCambios, 'cambios');

      If X[N] Is TAutomovil Then { Si el vehículo es un automóvil }

        { hacer un moldeado de tipo para acceder al método Potencia }

        Write(', con ', (X[N] As TAutomovil).Potencia, ' caballos');

      WriteLn; { Saltar a la siguiente línea }

    End;

  End.

Unit Vehiculo;

Interface

Type

{ La clase TVehiculo será la base de todas las de más clases }

TVehiculo = Class

Private { Miembros de datos, privados }

  NEjes, NRuedas: Byte;
```

```
TieneMotor: Boolean;

Public { Métodos públicos }

    { El constructor toma tres parámetros }

Constructor Inicializa(Ejes, Ruedas: Byte; Motor: Boolean);

    { Esta función es virtual abstracta, lo que significa que tendrá que ser
necesariamente implementada por los descendientes de TVehiculo }

Function Nombre: String; Virtual; Abstract;

Function NumeroDeEjes: Byte;

Function NumeroDeRuedas: Byte;

Function FuncionaConMotor: Boolean;

End;

{ La clase TBicicleta está derivada de TVehiculo, heredando miembros y métodos }

TBicicleta = Class(TVehiculo)

Public

    Constructor Inicializa; { El constructor no necesita parámetros }

    Function Nombre: String; Override; { Se redefine el método Nombre }

End;

{ La clase TBicicletaMontana está derivada de TBicicleta }

TBicicletaMontana = Class(TBicicleta)

Private

    NCambios: Byte; { Número de cambios }

Public

    Constructor Inicializa(Cambios: Byte); { Constructor }

    Function Nombre: String; Override; { Redefinimos el método Nombre }

    Function NumeroDeCambios: Byte; { Devuelve el número de cambios }
```


End;

{ La clase TAutomovil está derivada de TVehiculo }

TAutomovil = Class(TVehiculo)

Private

NCambios: Byte; { Número de cambios }

CAutomatico: Boolean; { Cambio automático }

NPotencia: Integer; { Potencia en caballos }

Public

Constructor Inicializa(Cambios: Byte; Automatico: Boolean; Potencia: Integer);

Function Nombre: String; Override; { Redefinimos el método nombre }

Function NumeroDeCambios: Byte; { Y añadimos los métodos necesarios }

Function EsAutomatico: Boolean; { para devolver el resto de los datos }

Function Potencia: Integer;

End;

Implementation

{ Implementación del constructor Inicializa de la clase TVehiculo }

Constructor TVehiculo.Inicializa;

Begin

NEjes := Ejes; { Guardar los datos facilitados }

NRuedas := Ruedas;

TieneMotor := Motor;

End;

{ Implementación del método NumeroDeEjes }

Function TVehiculo.NumeroDeEjes;

Begin

Result := NEjes;

End;

{ Implementación del método NumeroDeRuedas }

Function TVehiculo.NumeroDeRuedas;

Begin

Result := NRuedas;

End;

{ Implementación del método FuncionaConMotor }

Function TVehiculo.FuncionaConMotor;

Begin

Result := TieneMotor;

End;

{ Implementación del constructor del tipo TBicicleta. }

Constructor TBicicleta.Inicializa;

Begin

Inherited Inicializa(2, 2, False);

End;

{ Implementación del método Nombre, que devuelve el nombre del vehículo. }

Function TBicicleta.Nombre;

Begin

Result := 'Bicicleta';

End;

{ Constructor del tipo TBicicletaMontana. }

Constructor TBicicletaMontana.Inicializa;

Begin

Inherited Inicializa; { Llamar al constructor de la clase base }

NCambios := Cambios; { Guardar el dato adicional recibido }

End;

{ Implementación del método Nombre }

Function TBicicletaMontana.Nombre;

Begin

Result := 'Bicicleta de montaña';

End;

{ Implementación del método NumeroDeCambios }

Function TBicicletaMontana.NumeroDeCambios;

Begin

Result := NCambios;

End;

{ Implementación del constructor del tipo TAutomovil }

Constructor TAutomovil.Inicializa;

Begin

Inherited Inicializa(2, 4, True); { Llamar al constructor base }

NCambios := Cambios; { Preservar el resto de los datos }

CAutomatico := Automatico;

```
NPotencia := Potencia;  
  
End;  
  
{ Implementación del método Nombre de TAutomovil }  
  
Function TAutomovil.Nombre;  
  
Begin  
  
    Result := 'Automóvil';  
  
End;  
  
{ Implementación del método NumeroDeCambios de TAutomovil }  
  
Function TAutomovil.NumeroDeCambios;  
  
Begin  
  
    Result := NCambios;  
  
End;  
  
{ Implementación del método EsAutomatico de TAutomovil }  
  
Function TAutomovil.EsAutomatico;  
  
Begin  
  
    Result := CAutomatico;  
  
End;  
  
{ Implementación del método Potencia de TAutomovil }  
  
Function TAutomovil.Potencia;  
  
Begin  
  
    Result := NPotencia;  
  
End;
```

End.

6. INICIANDO CON COMPONENTES BASICOS.

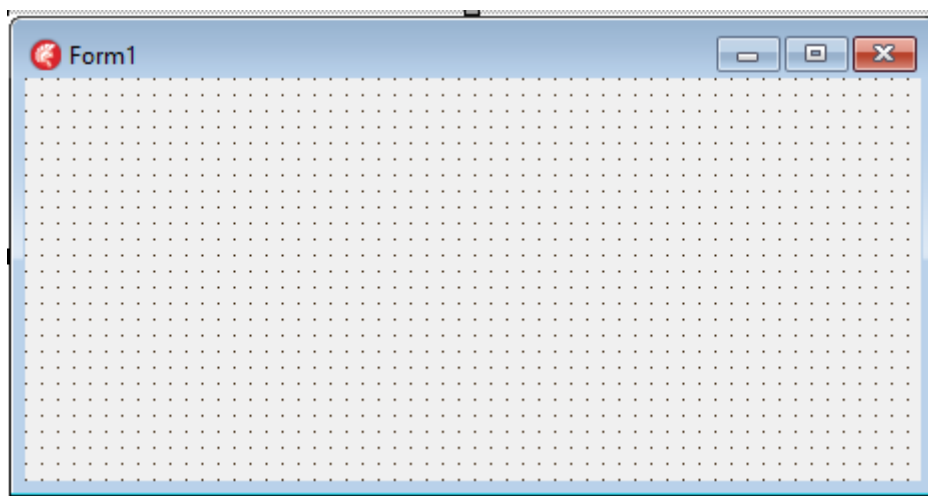
Una vez que conocemos el entorno de trabajo que vamos a utilizar, comenzaremos a introducirnos -poco a poco- en el funcionamiento de cada uno de los **Componentes** que integran la paleta **Standard**.

Con estos Componentes realizaremos las operaciones básicas de Entrada / Salida cualquier gestión de información.

FORMS, FORMULARIOS, FICHAS.

Como hemos visto en la pequeña demostración realizada en el *Tema 3*, y luego hemos explicado en el *Tema 4*; al utilizar un **Entorno Visual** como plataforma de desarrollo, todas las gestiones de la información que necesite de la interacción con el **Usuario** a través de cualquier dispositivo de **Entrada/Salida**, debe ser **incluida, diseñada y personalizada** en/sobre la **Ventana** correspondiente al **Formulario** o *Ficha*.

Al crear un proyecto VCL Form Applications, ya aparece una Ficha -en blanco- esperando ser **“diseñada”**. Desde este punto, el usuario podrá comenzar un nuevo **Proyecto** ó cargar uno ya creado.



Comenzaremos aportando una definición técnica sobre la Ficha:

Una Ficha es un Componente, y por tanto, es un Objeto perteneciente al tipo **TForm**. Siendo así, dispondrá de Propiedades, Eventos y Métodos.

Observemos y analicemos el siguiente código fuente :

```
unit Unit1;

interface

uses

    Windows, Messages, SysUtils, Classes,
    Graphics, Controls, Forms, Dialogs;

type

    TForm1 = class(TForm)

        Button1: TButton;

    private
        { Private declarations }
    public
        { Public declarations }
    end;

var

    Form1: TForm1;

implementation

{$R *.DFM}

end.

... ..
```



- En la cápsula Uses se hace referencia a un módulo llamado Form.
- Se define un nuevo tipo de Objeto: TForm1 (que deriva el tipo TForm).
- Se crea una variable Form1 perteneciente al tipo TForm1.

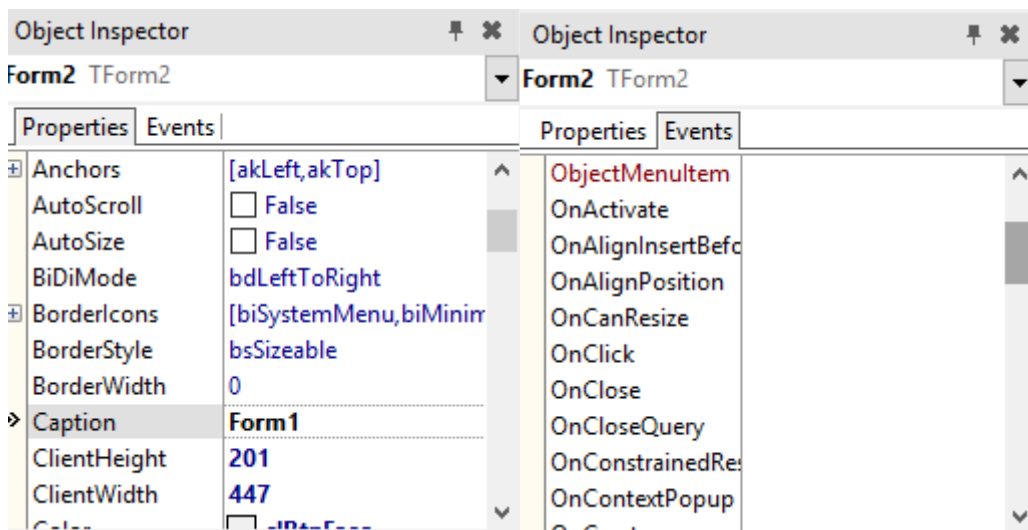
....

- Se ha incluido un Componente TButton ; se ha insertado como miembro de TForm1.

....

Esta información puede parecer un tanto confusa, ya que su función principal será la de **contener Objetos** que configuren el **Interface** del programa con el **Usuario**.

Partiendo de esta información, podemos concluir diciendo que **una Ficha es un Componente cuya funcionalidad consiste generalmente en contener otros Componentes, y que, como conjunto configuran el Interface con el Usuario final.**



Generalmente, las Aplicaciones incluirán múltiples Fichas en su entorno, una de ellas deberá actuar como Principal y aparecerá al iniciar la ejecución del programa, el resto se considerarán como Fichas Secundarias y serán mostradas cuando considere el usuario o el programa.

Tabla de **Propiedades**:

ActiveControl	ActiveMDIChild	AutoScroll
BorderIcons	BorderStyle	Caption
Canvas	ClientHeight	ClientWidth
Color	ComponentCount	Components
ControlCount	Controls	Ctl3D
Cursor	Enabled	Font
FormStyle	Handle	Height
HelpContext	Hint	HorzScrollBar
Icon	KeyPreview	Left



MDIChildCount	MDIChildren	Menu
Name	ObjectMenuItems	PixelsPerInch
PopupMenu	Position	PrintScale
Scaled	ShowHint	Tag
Top	VerScrollBar	Visible
Width	WindowMenu	WindowState

En la gestión y diseño de un Proyecto, desde el menú **File / New** podemos :

1.- Incluir nuevas **Fichas** Opción **VCL Form**.


2.- Reutilizar Fichas " **Open**.

Habitualmente, en la Pantalla sólo aparecerá la Ficha sobre la estemos trabajando, pero podemos mostrar otras fichas del Proyecto desde :

1.- La opción Forms del menú View.

2.- Desde el botón ➔

PROPIEDADES DE TForm

Muchas de  estas propiedades son aplicables a otros componentes **Delphi**, por lo que su conocimiento nos ahorrará mucho trabajo a la hora de poder trabajar con dichos componentes.

♦ PROPIEDADES ACCESIBLES EN TIEMPO DE DISEÑO

• ACTIVECONTROL

Permite elegir cuál será el control activo en principio.

☐ AUTOSCROLL

Permite el uso de barras de desplazamiento cuando se modifican el tamaño de una ficha. activar o desactivar la ficha. El valor por defecto es **True** aunque puede tomar el valor **False**.

☐ BORDERICONS

Permite el uso de los controles de barra de las ventanas Windows.

☐ BORDERSTYLE

Permite la modificación en tiempo de ejecución de las dimensiones de la ficha si el valor es **bsSizeable**. Posee otros valores que realizan diferentes acciones sobre los estilos del borde de la ventana.

☐ CAPTION



Título en la parte superior de la ficha. En principio coincidirá con el valor de la propiedad **Name**. Es una cadena de caracteres, por lo que no puede contener espacios en blanco y otros caracteres especiales no permitidos en un identificador. Es una propiedad de lectura y escritura.

☐ **CLIENTHEIGHT**

Altura del área cliente.

☐ **CLIENTWIDTH**

Anchura del área cliente.

☐ **COLOR**

Establece el color del fondo de la ficha. Se puede elegir el color de una lista desplegable.

☐ **CTL3D**

Apariencia tridimensional. El valor por defecto es **True** aunque puede tomar el valor **False**.

☐ **CURSOR**

Permite cambiar el aspecto del cursor mediante a los valores de una lista desplegable.

☐ **ENABLED**

Permite activar o desactivar la ficha. El valor por defecto es **True** aunque puede tomar el valor **False**.

☐ **FONT**

Permite modificar la fuente del texto, que es en realidad un objeto del tipo **TFont**. Al modificar la propiedad generalmente no observaremos nada, ya que los efectos se producirán en el momento que insertemos algún control, cuya propiedad **Font** tomará los mismos valores de la ficha. Es una propiedad compuesta que puede modificar múltiples parámetros.

☐ **FORMSTYLE**

Permite elegir cuál será el estilo de la ficha.

☐ **HEIGHT**

Altura de la ficha.

☐ **HELPCONTEXT**

Permite asociar a una ficha una determinada página de ayuda contextual.

☐ **HINT**



Permite añadir una etiqueta de ayuda que aparece cerca del punto en que se tiene situado el cursor. Además debemos dar el valor **True** a la propiedad **ShowHint**.

☐ **HORZSCROLLBAR**

Permite modificar las características de la barra de desplazamiento horizontal.

☐ **ICON**

Permite asignar un icono distinto a la barra de título de una ficha.

☐ **KEYPREVIEW**

Por defecto vale **False**, de tal forma que la s pulsaciones van directamente al control de la ficha que esté en ese momento activo. Dándole el valor **True** permite interceptar el valor por parte de la ficha.

☐ **LEFT**

Coordenada horizontal de la ficha relativa a posiciones de pantalla.

☐ **MENU**

Permite mantener una referencia al menú asociado a la ficha.

☐ **NAME**

Nombre de la ficha. Es una propiedad de sólo lectura durante la ejecución. En el momento en que se da un valor a esta propiedad se actualiza automáticamente el código generado por Delphi.

☐ **OBJECTMENUITEM**

Permite establecer una opción de menú que se hará activa cuando en la ficha se esté trabajando con un campo **OLE**.

☐ **PIXELSPERINCH**

Permite modificar el número de puntos por cada pulgada. Dando el valor **False** a la propiedad **Scaled** desactivaremos el escalado, a pesar de que hayamos asignado cualquier valor a **PixelsPerInch**.

☐ **POPUPMENU**

Permite asociar un menú emergente a la ficha.

☐ **POSITION**

Permite cambiar la posición relativa de la ficha durante el tiempo de ejecución. Puede tomar cualquiera de los siguientes valores:



- a) **poDesigned**: Es el valor tomado por defecto, causando que la ficha aparezca en la misma posición y con el mismo tamaño que se fijó en el tiempo de diseño.
- b) **poDefault**: La posición y el tamaño irán cambiando en cada ejecución, tomando la posición que les corresponda por defecto y el mayor tamaño posible.
- c) **poDefaultPosOnly**: El tamaño de la ficha permanece fijo, mientras que la posición va cambiando con cada ejecución.
- d) **poDefaultSizeOnly**: Inversa a la anterior.
- e) **poScreenCenter**: La ficha aparece ocupando el centro de la pantalla, con su tamaño original.

☐ **PRINTSCALE**

Controla el escalado de la impresión de una ficha. Puede tomar los valores:

- a) **poNone**: No guarda proporción alguna.
- b) **poProportional**: Imprime el mismo número de puntos en impresora que en pantalla.
- c) **poPrinterToFit**: Ocupa el mayor espacio posible en la página manteniendo la proporción de la ficha.

☐ **TAG**

No tiene significado alguno. Permite guardar un valor entero.

☐ **TOP**

Coordenada vertical de la ficha relativa a posiciones de pantalla.

☐ **VERTSCROLLBAR**

Permite modificar las características de la barra de desplazamiento vertical.

☐ **VISIBLE**

Permite ocultar o visualizar la ficha. El valor por defecto es **True** aunque puede tomar el valor **False**.

☐ **WIDTH**

Anchura de la ficha.

☐ **WINDOWMENU**

Permite crear un menú con las distintas ventanas hijas existentes.

☐ **WINDOWSTATE**

Apariencia inicial de la ficha. Puede tomar cualquiera de los siguientes valores:

a) **wsNormal**

b) **wsMinimize**



c) *wsMaximize*

♦ PROPIEDADES ACCESIBLES EN TIEMPO DE EJECUCIÓN

☐ **ACTIVEMDICHILD**

Permite saber cuál es la ventana activa.

☐ **CANVAS**

Se trata de un objeto que contiene todo el espacio interior de la ventana y que cuenta con una serie de métodos que nos permiten escribir texto, dibujar, etc.

☐ **COMPONENTCOUNT**

Número de componentes que existen en una ficha. A partir de este número podemos acceder a la matriz **Components**. La base del índice es cero. El acceso a un elemento permite determinar su tipo, realizar conversiones y manipular sus propiedades.

☐ **CONTROLCOUNT**

Número de controles que existen en una ficha. A partir de este número podemos acceder a la matriz **Controls**. La base del índice es cero. El acceso a un elemento permite determinar su tipo, realizar conversiones y manipular sus propiedades.

☐ **HANDLE**

Identificador de la ventana Windows.

☐ **MDICHILDCOUNT**

Número de ventanas hijas existentes.

☐ **MDICHILDREN**

Matriz que contiene las ventanas hijas.

1.1. EVENTOS DE TForm

Al igual que ocurre con las propiedades, muchos de los eventos que puede recibir una ficha también son comunes a otros componentes. Los eventos los podemos asociar en varios grupos dependiendo del origen del que procedan.

♦ EVENTOS GENERADOS POR EL RATÓN

☐ **ONMOUSEMOVE**

Es recibido por un componente a medida que el cursor del ratón se mueve sobre él. Lleva asociados los parámetros **X** e **Y**, que expresan la posición actual del cursor y **Shift**, un conjunto que puede contener los valores siguientes:

Valor	Corresponde a ...
ssShift	Una de las teclas de mayúsculas



ssAlt	La tecla <Alt>
ssCtrl	La tecla <Control>
ssRight	El botón derecho del ratón
ssLeft	El botón izquierdo del ratón
ssMiddle	El botón central del ratón
ssDouble	Los botones izquierdo y derecho del ratón

• ONMOUSEDOWN

Se genera cuando se pulsa cualquiera de los botones del ratón. Lleva asociados aparte de los parámetros **X**, **Y** y **Shift**, el parámetro **Button** cuyos valores pueden ser **mbRight**, **mbLeft** y **mbMiddle**.

☐ ONMOUSEUP

Se genera cuando se libera cualquiera de los botones del ratón. Lleva asociados los mismos parámetros que **OnMouseDown**.

☐ ONCLICK

Indica la pulsación del botón izquierdo del ratón, en ocasiones se produce por la barra espaciadora, <Intro> o <Escape>.

☐ ONDBLCLICK

Doble pulsación con el botón izquierdo del ratón.

◆ EVENTOS DE TECLADO

☐ ONKEYDOWN

Se genera al pulsar cualquier tecla cuyo código es facilitado por el parámetro **Key**. También recibo el parámetro **Shift**. No es un código ASCII, sino uno de los valores de la siguiente tabla:

Constante	Tecla que representa
VK_0 . . . VK_9	Los dígitos 0 a 9
VK_NUMPAD0...VK_NUMPAD9	Los dígitos 0 al 9 del teclado numérico
VK_A ... VK_Z	Las teclas A a Z
VK_BACK	Borrado hacia atrás
VK_TAB	Tabulador
VK_RETURN	Intro
VK_SHIFT	Mayúsculas
VK_CONTROL	Control
VK_MENU	Alt
VK_PAUSE	Pausa
VK_CAPITAL	Fija mayúsculas



VK_ESCAPE	Escape
VK_SPACE	Barra espaciadora
VK_PRIOR	RePág
VK_NEXT	AvPág
VK_END	Fin
VK_HOME	Inicio
VK_LEFT	Flecha izquierda
VK_RIGHT	Flecha derecha
VK_UP	Flecha arriba
VK_DOWN	Flecha abajo
VK_INSERT	Insert
VK_DELETE	Supr
VK_MULTIPLY	* en teclado numérico
VK_ADD	+ en teclado numérico
VK_SUBTRACT	- en teclado numérico
VK_DECIMAL	. en teclado numérico
VK_DIVIDE	/ en teclado numérico
VK_F1 . . . VK_F12	Teclas de función
VK_NUMLOCK	BloqNum
VK_SCROLL	BloqDespl

☐ **ONKEYUP**

Se genera al liberar una tecla.

☐ **ONKEYPRESS**

Se genera al pulsar una de las teclas “normales”. El parámetro **Key** corresponde con el código ASCII de la tecla pulsada.

♦ **OTROS EVENTOS**

☐ **ONACTIVATE**

Se produce en el momento en que la ficha se convierte en la ventana activa del entorno.

☐ **ONCLOSE**

Se produce cuando se cierra una ficha. Antes de realizar el cierre de la ventana se genera este evento que recibe como parámetro una variable llamada **Action** a la cual deberemos asignar uno de los valores mostrados en la siguiente tabla:

Valor	Significado
caFree	La ficha es cerrada



caMinimize	La ficha se minimiza
caHide	La ficha se oculta
caNone	No se permite el cierre de la ficha

• ONCLOSEQUERY

Al igual que el anterior, este evento se genera cuando se quiere cerrar una ficha, recibándose como parámetro la variable **CanClose** a la que daremos en valor **True** o **False**.

☐ ONCREATE

Se genera cuando la ficha va a ser creada, permitiendo así inicializar valores.

☐ ONDEACTIVATE

Se produce cuando la ficha se vuelve inactiva.

☐ ONDESTROY

Se produce cuando destruimos una ficha, liberando memoria.

☐ ONDRAGDROP

Se produce cuando sobre un determinado control se “suelta” un elemento que está siendo “arrastrado”.

☐ ONDRAGOVER

Se produce cuando sobre un determinado control se desplaza un elemento que está siendo “arrastrado”.

☐ ONENTER

Se produce cuando se activa un componente de la ficha.

☐ ONEXIT

Se produce cuando se inactiva un componente de la ficha.

☐ ONHIDE

Se produce justo antes de que una ficha se oculte.

☐ ONPAINT

Se produce cuando la ficha es dibujada.

☐ ONRESIZE

Se produce cuando se modifica el tamaño de la ficha.



☐ ONSHOW

Se produce antes de la visualización de una ficha.

1.2. MÉTODOS DE TFORM

☐ BRINGTOFRONT

Sitúa una ficha visible por encima de todas las demás.



☐ CLOSE

Cierra una ficha. Previamente generará el evento **OnCloseQuery** para asegurarse de que puede cerrarla, dependiendo del valor del evento **OnClose**.

☐ DESTROY

Destruye una ficha.

☐ GETFORMIMAGE

Se almacena la ficha en formato BitMap.

☐ HIDE

Oculto una ficha.

☐ PRINT

Se obtiene una copia impresa de la ficha.

☐ SENDTOBACK

Sitúa una ficha visible por debajo de las demás.

☐ SHOW

Visualiza la ficha haciéndola visible por encima de las demás ventanas.

ETIQUETAS DE TEXTO (Label)

Mediante este control podemos mostrar un texto estático en la ficha, fijando su posición, color, tipo de letra y tamaño.

Tras insertar la etiqueta lo primero que tendremos que hacer es asignarle un nombre, modificando el valor de la propiedad **Name**. El valor almacenado en la propiedad **Caption** será el texto que se mostrará en la etiqueta.

• POSICIÓN, TAMAÑO Y ALINEACIÓN

Si asignamos el valor **True** a la propiedad **Autosize** provocaremos que la dimensiones de la etiqueta se ajusten al texto. Mediante la propiedad **Align** podemos controlar la posición



y las dimensiones del control. Puede contener cualquiera de los valores de la siguiente tabla:

Valor	El control se ajusta a ...
alNone	Conserva la posición y dimensiones
alTop	Margen superior del contenedor
alBottom	Margen inferior del contenedor
alLeft	Margen izquierdo del contenedor
alRight	Margen derecho del contenedor
alClient	Todo el espacio disponible en el

La propiedad **Alignment** justifica el texto respecto al control según los valores:

Valor	Ajuste del texto
taLeftJustify	A la izquierda
taRightJustify	A la derecha
taCenter	Centrado

Si asignamos el valor **True** a la propiedad **WordWrap** conseguiremos que el texto se pueda dividir en las líneas que sean necesarias.

En cualquier momento podemos obtener las coordenadas correspondientes a las cuatro esquinas de la etiqueta de texto mediante la propiedad **BoundsRect**.

• FAMILIA, ESTILO Y TAMAÑO DE LETRA

La propiedad **Font** contiene un objeto del tipo **TFont** con todos los estilos de letra de Windows. Su edición se puede realizar mediante una ventana o desplegando las propiedades de **Font**

• COLOR DE FONDO

Mediante la propiedad **Color** se puede seleccionar un color sobre el que aparecerá dibujado el texto sobre toda la extensión del control. Asignando el valor **True** a la propiedad **Transparent** podemos hacer que el texto se dibuje directamente sobre la ficha sin llenar toda la extensión del control.

• ASOCIACIÓN A OTROS CONTROLES

Podemos utilizar una etiqueta para servir de título a otro control, pudiendo incluir una tecla de acceso rápido de tal forma que al pulsarla se activa el control asociado. Para ello es necesaria asignar el valor **True** a la propiedad **ShowAccelChar** y crear la asociación entre la etiqueta y el control mediante la propiedad **FocusControl**.

• OTRAS PROPIEDADES

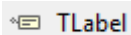
Al igual que para Fichas, la visualización de una etiqueta de ayuda se llevará a cabo si el valor de la propiedad **ParentShowHint** es **True**.



Mediante la propiedad **Cursor** podemos fijar la apariencia del cursor cuando desplazamos el ratón sobre la etiqueta. En la siguiente tabla podemos ver algunos de los valores posibles que puede tomar:

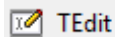
Valor	Forma del cursor
crDefault	La que tenga en ese momento
crArrow	Puntero habitual
crCross	Una cruz
crIBeam	Cursor de texto
crSize	Cuatro flechas unidas
crUpArrow	Flecha hacia arriba
crHourGlass	Reloj de arena
crNoDrop	Señal de prohibido

• METODOS DE LABEL



Aparte de los métodos ya conocidos en el tema de Fichas, el método **SetBounds** nos permite modificar la dimensión y posición de la etiqueta mediante cuatro parámetros correspondientes a las propiedades **Left**, **Top**, **Width** y **Height**.

ENTRADA DE DATOS (Edit)



Mediante este control podemos introducir o modificar texto en tiempo de ejecución.

• CONTENIDO DEL CAMPO DE EDICIÓN

La propiedad **Text** contendrá el texto, pudiéndose asignar cualquier valor inicial. La longitud máxima es de 255 caracteres, pudiendo establecer un límite menor en la propiedad **MaxLenght**. Aunque se trate de una sola línea de texto podemos usarlo como cualquier editor de texto.

• CONTROL DE ENTRADA

La propiedad **CharCase** permite realizar la conversión del texto introducido según los valores de la siguiente tabla:

Valor	Conversión realizada
ecNormal	No se realiza conversión alguna
ecLowerCase	Todas las mayúsculas a minúsculas

Mediante la propiedad **Modified** podemos saber si el campo ha sido editado. Mediante la propiedad **ReadOnly** podemos visualizar un texto permitiendo desplazarse por él pero no editarlo, aunque el contenido de **Text** podrá ser modificado mediante código. Podemos introducir texto de manera confidencial asignando un carácter máscara a la propiedad **PasswordChar**.

• SELECCIÓN DE TEXTO

Mediante la combinación de la tecla **<Mayúscula>** y las teclas de cursor es posible marcar texto contenido en el control. La propiedad **SelStart** contiene la posición del primer carácter marcado, teniendo en cuenta que empieza por 0; **SelLength** contiene el número de caracteres marcados y **SelText** contendrá el texto marcado. **HideSelection** controla que el texto marcado se oculte cuando se cambia de control. **AutoSelect** controla que se marque todo el texto de un control cuando se activa dicho control.



• OTRAS PROPIEDADES

ParentColor, **ParentCtl3D** y **ParentFont** permiten que el control tome las características del componente en que se haya incluido.

• MÉTODOS DEL CONTROL EDIT

SelectAll selecciona todo el texto del control y **ClearSelection** elimina el texto seleccionado en ese momento. Para vaciar el control se usa el método **Clear**. Podemos realizar operaciones con el portapapeles mediante los métodos **CopyToClipboard**, **CutToClipboard** y **PasteFromClipboard**.

ENTRADA DE TEXTO (Memo)

Cuando las posibilidades del control **Edit** no son suficientes, podemos usar el control **Memo** que a diferencia del anterior puede trabajar en múltiples líneas.

• CONTENIDOS DEL CONTROL Y VISUALIZACIÓN DE TEXTO

El texto en un control **Memo** se estructura en múltiples cadenas que son almacenadas en la propiedad **Lines**. Es posible dar un valor inicial mediante el editor de cadenas de **Delphi**. En tiempo de ejecución podemos tratar la propiedad **Lines** como una matriz, teniendo en cuenta que la primera tiene el índice cero. Realmente la propiedad **Lines** es un objeto del tipo **TStrings** y podemos añadir texto al final mediante el método **Add**, o en cierta posición mediante el método **Insert**, borrarlo con **Delete**, moverlo mediante **Move** o intercambiarlo mediante **Exchange**. Podemos tomar el contenido de un archivo de texto mediante el método **LoadFromFile** o salvarlo mediante **SaveToFile**.

Podemos hacer aparecer barras de desplazamiento vertical u horizontal mediante la propiedad **ScrollBars**. Si deseamos usar las teclas **<Tabulador>** e **<Intro>** de forma normal daremos el valor **True** a las propiedades **WantTabs** y **WantReturns**.

• MÉTODOS DEL CONTROL MEMO

Además de los métodos conocidos anteriormente para el control **Edit** que son válidos tenemos el método **GetTextBuf**, que es una función que toma dos parámetros: un puntero a la dirección en la que se almacena el texto y un entero indicando la longitud máxima del espacio asignado. La longitud real del texto la obtenemos mediante **GetTextLen**. Para modificar el contenido de **Memo** asignando un nuevo texto utilizaremos **SetTextBuf**, pasándole la dirección del nuevo texto., Los punteros son de tipo **Pchar**.

BOTONES (Button)

El botón es uno de los elementos más comunes del entorno gráfico y permite llevar a cabo una determinada acción cuando se pulsa.



• TÍTULO DEL BOTÓN

El título de un botón corresponde al valor asignado a la propiedad **Caption**. Si la propiedad **Enabled** toma el valor **False**, el título aparecerá difuminado.

• BOTÓN POR DEFECTO Y DE CANCELACIÓN

En toda ficha podemos tener un botón por defecto que se active cuando pulsemos la tecla **<Intro>** para ello debemos asignar el valor **True** a la propiedad **Default**, en consecuencia se generará el evento **OnClick**. Si deseamos tener un botón de cancelación debemos asignar **True** a la propiedad **Cancel**.

• CUADROS DE DIÁLOGO

Un cuadro de diálogo es una ventana que se caracteriza por no permitir el acceso a otra ventana mientras él se encuentre abierto y al cerrarlo se devuelve un valor indicando la causa de la salida. Habitualmente un cuadro de diálogo se cierra asignando a la propiedad **ModalResult** el valor de salida, el cual puede tomar cualquier valor, aunque existen una serie de valores predefinidos que son los más usados:

Valor	Valor devuelto
mrNone	0
mrYes	idYes
mrNo	idNo
mrOk	idOk
mrCancel	idCancel
mrAbort	idAbort
mrRetry	idRetry
mrIgnore	idIgnore

CAJAS DE SELECCION (CheckBox) TCheckBox

Este control permite activar o desactivar una cierta opción, sin necesidad de escribir nada, simplemente habrá que pulsar sobre el control.


El título que aparecerá junto a la caja de selección será el que asignemos a la propiedad **Caption**, pudiendo existir una tecla de acceso rápido.

El control puede aparecer en dos estados que se controlan mediante la propiedad **Checked**, tomando el valor **True** si está marcado, o el valor **False** si no lo está. Opcionalmente se puede activar un tercer estado indeterminado, para permitir esto daremos el valor **True** a la propiedad **AllowGrayed** y usaremos el valor de la propiedad **State**, que podrá tomar los siguientes valores:

Valor	Estado del control
cbChecked	Marcado
cbUnchecked	Desmarcado
cbGrayed	Indeterminado

Los eventos y métodos disponibles son los mismos que para la mayoría de los controles.

BOTONES DE RADIO (RadioButton)

 TRadioButton


Este control permite activar una cierta opción de entre varias disponibles de manera exclusiva, de modo que al elegir una se desmarca automáticamente la que estuviera elegida.

El título que aparecerá a la derecha del control será el que asignemos a la propiedad **Caption**.

El control puede aparecer en dos estados que se controlan mediante la propiedad **Checked**, tomando el valor **True** si está marcado, o el valor **False** si no lo está.

1.3. GRUPOS DE CONTROLES (GroupBox)


 TGroupBox

Se trata de un control que sirve para agrupar una serie de controles y no dispone de ninguna propiedad que no conozcamos ya.

Al insertar un **GroupBox** aparecerá un rectángulo con un título en la parte superior izquierda, modificable con la propiedad **Caption**.

Para que un control pertenezca a un grupo necesitamos insertarlo en él. Desplazando un componente de la ficha dentro de un **GroupBox**, sólo conseguimos modificar su posición, pero el control no estará contenido en el grupo. Para incluirlos, podemos cortarlos al portapapeles, activar el grupo y pegarlos.

GRUPOS DE BOTONES DE RADIO (RadioGroup)

 TRadioGroup

Permite agrupar los botones de radio como el control anterior, pero cuenta con una serie de propiedades que permite disponer los botones de radio de una manera más cómoda.



Para indicar los botones de radio que existirán en el grupo editaremos la propiedad **Items**, la cual es una lista de cadenas, por lo tanto en tiempo de ejecución podremos modificar su contenido.

Mediante la propiedad **Columns** podemos variar el número de columnas en que queremos que se repartan los botones. La propiedad **ItemIndex** nos permite saber qué botón está seleccionado y modificar dicha selección, su valor va desde 0 y el número de botones menos 1. El valor -1 indica que no hay seleccionado ningún botón.

EL CONTROL PANEL (Panel)

Permite agrupar una serie de controles de la misma manera que **GroupBox** pero posee unas propiedades de presentación distintas.

Mediante las propiedades **BevelInner** y **BevelOuter** fijaremos el estilo del panel que puede aparecer resaltado o hundido. La propiedad **BevelWidth** permite modificar el efecto generado por **BevelInner** y **BevelOuter**, mientras que **BorderOuter** establece la anchura del borde.

LISTAS DE DATOS(ListBox)

Es un control que contiene cadenas de caracteres, cada una de las cuales aparece como un elemento de la lista. Se utiliza para seleccionar datos de unos valores dados.

• CONTENIDO DE LA LISTA

Los elementos se gestionan mediante la propiedad **Items**. En tiempo de ejecución se puede modificar como los controles **Memo** y **RadioGroup**.

La visualización de la lista se dividirá en tantas columnas como indique la propiedad **Columns**. Se pueden mostrar los elementos ordenados dando el valor **True** a la propiedad **Sorted**.

Para conseguir que el tamaño del control sea proporcional a la altura de un elemento de la lista daremos el valor **True** a la propiedad **IntegralHeight**.

• SELECCIÓN DE ELEMENTOS

En tiempo de ejecución normalmente sólo podemos seleccionar un elemento, que se puede conocer mediante la propiedad **ItemIndex**; si no hay ninguno seleccionado contendrá el valor -1.



Dando el valor **True** a la propiedad **MultiSelect** conseguiremos seleccionar múltiples elementos. Dependiendo del valor de **ExtendedSelect**, cuyo valor por defecto es **True**, podemos seleccionarlos mediante las teclas **<Mayúsculas>** , **<Control>** y el botón izquierdo del ratón; si el valor es **False** podremos seleccionar varios elementos marcándolos con el ratón.

Cuando hay varios elementos marcados la propiedad **SelCount** indica cuántos hay, y la propiedad **Selected** para conocer qué elementos lo están. Se trata de una matriz con los elementos y los valores **True** o **False**.

• LISTAS ESPECIALES

Podemos utilizar este control para presentar objetos distintos a cadenas. Para ello modificaremos el valor de **Style** según los valores siguientes:

Valor	Estilo de la lista
ibStandard	Todos los elementos son cadenas
ibOwnerDrawFixed	Cada elemento es distinto pero con altura fija
ibOwnerDrawVariable	Cada elemento tiene una altura variable

Si el estilo es **ibOwnerDraw**, la altura de cada elemento vendrá dada por el valor de **ItemHeight** en puntos. Si el estilo es **ibOwnerDrawVariable**, cada vez que se necesite dibujar un elemento se generará un evento **OnMeasureItem**. El método de respuesta recibirá tres parámetros: la lista que ha generado el evento, el número de elemento a dibujar y una variable **Height** con la altura del elemento.

Si el estilo de una lista no es **ibStandard** cada vez que dibujemos un elemento se generará un evento **OnDrawItem**, recibiendo los siguientes parámetros:

- **ListBox**: Una referencia a la lista.
- **Index**: Un entero indicando el elemento.
- **Rect**: Un registro **TRect** indicando las coordenadas del recuadro donde se dibujará el elemento.
- **State**: Estado del elemento que podrá tener los valores de la siguiente tabla:

Valor	Estado actual del elemento
odSelecte <i>d</i>	Seleccionado
odDisable <i>d</i>	La lista no está activa
odFocuse <i>d</i>	El elemento es el elemento actual de la lista

Para dibujar el elemento utilizaremos la propiedad **Canvas** que representa el área cliente de la lista.

LISTAS COMBINADAS (ComboBox) TComboBox

Es una combinación de los controles **ListBox** y **Edit**, cuya finalidad es tener asociado un campo **Edit** a la lista para mostrar el elemento seleccionado o para facilitar alguna otra operación.

• ESTILO DE LA LISTA COMBINADA

El control puede adoptar varias formas dependiendo de propiedad **Style**, que será uno de los mostrados a continuación:

Valor	Estilo
csDropDown	Lista con campo de edición asociado
csSimple	Campo de edición sin lista
csDropDownList	Lista sin campo de edición
csOwnerDrawFixed	Elementos distintos con altura fija
csOwnerDrawVariable	Elementos distintos con altura variable

El estilo por defecto es **csDropDown**, que aparece como un control **Edit** con la flecha en la parte derecha. Pulsando sobre la flecha se despliega la lista, pudiéndose seleccionar un elemento que se mostrará en el campo **Edit**. A diferencia de las listas normales no se pueden seleccionar varios elementos.

• DIMENSIONES DE LA LISTA

La extensión de la lista al desplegarse depende de la propiedad **DropDownCount**, que contendrá el número de líneas que se mostrarán simultáneamente. Si hay menos elementos que **DropDownCount**, se ajustará para mostrar los existentes.

BARRAS DE DESPLAZAMIENTO (ScrollBar) TScrollBar

Visualiza una barra de desplazamiento. Usada de forma independiente, podría utilizarse para simular el control de volumen en un programa capaz de reproducir un CD.

El tipo de barra lo establece la propiedad **Kind**, que puede ser **sbHorizontal** o **sbVertical**. El cursor puede desplazarse en pequeños incrementos, cuyo factor viene dado por la propiedad **SmallChange** o por incrementos mayores según la propiedad **LargeChange**. Los valores extremos de la barra vienen dados por **Min** y **Max**. La posición actual del cursor la proporciona la propiedad **Position**. La posición actual en la barra, los valores extremos, mínimo y máximo pueden fijarse mediante el método **SetParams**.

Cada vez que se modifica el valor de la propiedad **Position** se genera un evento **OnChange**. Al manipular la barra también se genera un evento **OnScroll**, cuyo método de respuesta recibirá un parámetro que represente a la barra, un segundo para conocer la acción que se ha efectuado sobre la barra según los valores de la siguiente tabla y un tercero con la nueva posición en la barra.

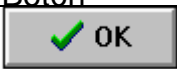

Valor	Operación de desplazamiento
scLineUp	Ir al paso anterior, arriba o a la izquierda
scLineDown	Ir al paso siguiente, abajo o a la derecha
scPageUp	Ir a la página anterior
scPageDown	Ir a la pagina siguiente
scTop	Ir a la posición de inicio
scBottom	Ir a la posición de fin
scPosition	Se ha desplazado el cursor a una posición
scTrack	Se está desplazando el cursor
scEndScroll	Se ha terminado de desplazar el cursor

GRÁFICOS EN BOTONES (Bitbtn) TBitBtn

Mediante este control podemos mostrar una pequeña imagen (mapa de bits) en el botón, además del título, para que pueda dar un significado más intuitivo al mismo (está en la paleta Additional).

• TIPOS PREDEFINIDOS

Al seleccionar cualquiera de los botones predefinidos se establecerán automáticamente los valores apropiados para **Caption**, **Glyph**, **ModalResult**, **Default**, **Cancel**, **NumGlyphs**, **Margin**, **Layout**. y **Spacing**. La selección del tipo de botón se realiza asignando uno de los valores de la siguiente tabla a la propiedad **Kind**.

Valor	Botón
bkOK	
bkCancel	

<i>bkYes</i>	 <u>Y</u> es
<i>bkNo</i>	 <u>N</u> o
<i>bkHelp</i>	 <u>H</u> elp
<i>bkClose</i>	 <u>C</u> lose
<i>bkAbort</i>	 <u>A</u> bort
<i>bkRetry</i>	 <u>R</u> etry
<i>bkIgnore</i>	 <u>I</u> gnore
<i>bkAll</i>	 <u>A</u> ll



• DISEÑAR EL BOTÓN

Si los estilos predefinidos no se ajustan a nuestras necesidades, podemos modificar algunas propiedades. Podemos diseñar el gráfico que aparece en el botón mediante el Editor de imágenes. Para asociar la imagen al botón asignaremos el mapa de bits a la propiedad **Glyph**. En tiempo de ejecución se podrá realizar mediante el método **LoadFromFile**.

El gráfico asociado puede tener hasta cuatro imágenes dispuestas horizontalmente y de las mismas dimensiones. La primera de ellas se utiliza cuando el botón está en estado normal, la segunda cuando este desactivado, la tercera cuando este pulsado y la cuarta cuando permanezca pulsado. Para ello asignaremos el número de imágenes a la propiedad **NumGlyphs**.

Mediante **Layout** determinamos la posición del gráfico dentro del botón. Mediante **Spacing** el número de puntos de separación entre el borde del botón y el texto, mientras que mediante **Margin** realizamos lo mismo pero referida al gráfico.

• USO DE BITBTN

El comportamiento es el mismo que para un control **Button**. Si lo utilizamos en un cuadro de diálogo, podemos asignar un valor a la propiedad **ModalResult**, de tal forma que cuando lo cerremos devolverá dicho valor. En general, podemos sustituir cualquier **Button** por un **Bitbtn** sin modificar nada, solamente seleccionar el gráfico a mostrar en el botón.

BARRAS DE BOTONES (SpeedButton) TSpeedButton

Este control permite crear barras de botones similares a las que Delphi tiene en la parte izquierda de la ventana principal, situando en el interior del botón un gráfico, no texto.

La propiedad **GroupIndex** permite crear grupos de botones de manera que al pulsar uno los demás se muestren liberados. El estado de cada botón lo podemos conocer mediante la propiedad **Down**, que vale **True** si está pulsado. Por defecto, uno de los botones ha de estar pulsado. Si damos el valor **True** a la propiedad **AllowAllUp** conseguiremos que todos los botones estén liberados.

FICHAS CON VARIAS PÁGINAS (Notebook) TNotebook

Permite estructurar la ventana en varias páginas, conteniendo cada una de ellas un grupo de controles relacionados. Este componente sólo es visible por su color de fondo. Podemos encontrarla en la paleta Win 3.1.



El componente contará con tantas páginas como elementos haya en la propiedad **Pages**. Las propiedades **PageIndex** y **ActivePage** permiten saber cuál es la página actual y activar otras páginas. Cuando se cambia de página se produce el evento **OnPageChanged** que nos permite actualizar cualquier información que dependa de la página activa en cada momento.

PESTAÑAS (TabSet)

Permite el acceso directo a una determinada página de un componente **Notebook** o similar, disponiendo una serie de pestañas, conteniendo los títulos de las páginas.

Los títulos los facilitamos mediante la propiedad **Tabs**, que es una lista de cadenas que contendrá un elemento por pestaña. En el caso de que lo usemos con un control **Notebook**, podemos asignar el valor **Pages** de este último a la propiedad **Tabs** del primero.

El color de la pestaña seleccionada coincide con el color de la pagina activa en el componente **Notebook** asociado. Dicho color se selecciona en la propiedad **SelectedColor** y el resto de pestañas tienen el color de la propiedad **UnselectedColor**. Mediante **BackgroundColor** y **DitherBackground** podemos controlar el color de fondo sobre el que aparecen las pestañas. **StartMargin** contiene el número de puntos de distancia desde el margen izquierdo del control hasta la primera pestaña, mientras que **EndMargin** contine la misma información para el margen derecho.

En el caso en que sea imposible la visualización de todas las pestañas , mediante la propiedad **AutoScroll** podemos hacer aparecer una pequeña barra de desplazamiento horizontal. **VisibleTabs** indica el número de pestañas visibles y **FirstIndex** indica la primera visible.

El valor de la propiedad **Style** determina el estilo de las pestañas que puede ser **tsStandard** para pestañas normales, o **tsOwnerDraw**, que permite incluir elementos gráficos. Si seleccionamos el segundo estilo indicaremos la altura de la pestaña con la propiedad **TabHeight**. El ancho será solicitado mediante el evento **OnMeasureTab**, cuyo método de respuesta recibirá el parámetro **Index**, indicando el númro de pestaña y el parámetro **TabWidth** que contendrá el númro de puntos de la anchura. Cuando haya que dibujar una pestaña se generará el evento **OnDrawTab**, pasando la siguiente información;

- **TabCanvas**: Un objeto del tipo **TCanvas**.
- **R**: Un objeto **TRect** con los límites de la pestaña.
- **Index**: Un entero indicando la pestaña a dibujar.
- **Selected**: Vale **True** si es la pestaña seleccionada.

TabIndex contiene cuál es la pestaña activa. Podemos usar el método **SelectNext** para activar la siguiente o anterior pestaña. Cada vez que se cambia de pestaña se genera un evento **OnChange**, indicando al método correspondiente el número de pestaña que se va a cambiar mediante le parámetro **NewTab** y el parámetro **AllowChange** si se permite la activación de dicha pestaña.

EL CONTROL TABBEDNOTEBOOK (TabbedNotebook)

Se trata de un control semejante a los anteriores que permite sustituir los controles **Notebook** y **TabSet** por uno solo con una apariencia algo distinta.



Se comporta igual que el control **Notebook**, de tal forma que podemos definir el número y el título de las páginas. Se puede modificar el número de pestañas por fila mediante la propiedad **TabsPerRow**. A diferencia del control **TabSet**, todas las pestañas son visibles siempre, adecuándose la anchura a las dimensiones del control.

MÁSCARA DE ENTRADA (MaskEdit) TMaskEdit

Este control permite aplicar un mayor control sobre los datos a introducir, sin necesidad de tener que escribir código para interceptar la pulsación de cada carácter.

Se comporta en gran parte igual que el control **Edit**, por lo tanto, la entrada de datos se realizará de idéntica manera a no ser que establezcamos una máscara de entrada. Esta máscara es una cadena de caracteres en la cual algunos tienen un significado especial. La signaremos en la propiedad **MaskEdit**. En tiempo de ejecución se condicionarán tanto la introducción de datos como la visualización.

Los caracteres con un significado especial son los siguientes:

Valor	Significado
C	Requiere la entrada de un carácter
c	Permite la entrada de un carácter
A	Requiere la entrada de un carácter alfanumérico
a	Permite la entrada de un carácter alfanumérico
L	Requiere la entrada de un carácter alfabético
l	Permite la entrada de un carácter alfabético
0	Requiere la entrada de un carácter numérico
9	Permite la entrada de un carácter numérico
#	Permite la entrada de un carácter numérico o de signo
<	Los caracteres siguientes aparecen en mayúsculas
>	Los caracteres siguientes aparecen en minúsculas
<>	Desactiva la función de los dos caracteres anteriores
\	Interpreta el carácter siguiente como no especial
_	Representa un espacio en blanco
:	Separador de horas y minutos
/	Separador de fechas
;	Separador interno de la máscara



En la propiedad **Text** podemos encontrar el contenido actual del campo. Se puede guardar con o sin la máscara. En la propiedad **EditText** tendremos el aspecto del campo en pantalla.

ENTIDADES GRÁFICAS SIMPLES (Shape) TShape

Este control permite crear algunas entidades gráficas simples como círculos o rectángulos, pudiendo modificar su tamaño, color, bordes, etc...

Left y **Top** determinarán la posición de la esquina superior izquierda, mientras que **Width** y **Height** se sumaran a las anteriores para dar su esquina opuesta.

La entidad gráfica depende del valor de la propiedad Shape que será uno de los valores mostrados en la siguiente tabla:

Valor	Entidad gráfica a dibujar
stSquare	Cuadrado
stRoundSquare	Cuadrado con esquinas redondeadas
stRectangle	Rectángulo
stRoundRectangle	Rectángulo con esquinas redondeadas
stCircle	Círculo
stEllipse	Elipse

PINCELES

La propiedad **Pen** es un objeto del tipo **TPen**, que cuenta con las propiedades necesarias para pintar el borde. mediante **Color** elegimos el color del pincel. El ancho del trazo se determina mediante **Width**. El tipo de trazo dependerá de **Style** que puede tomar los siguientes valores:

Valor	Entidad gráfica a dibujar
psSolid	Continuo
psDash	Discontinuo
psDot	Punteado
psDashDot	Discontinuo y punteado
psDashDotDot	Discontinuo y doble punteado
psClear	No se dibuja borde



El trazo del pincel se define mediante **Mode** tomando uno de los valores posibles siguientes, entre otros:

Valor	Trazo del pincel
<i>pmCopy</i>	Con el color del pincel
<i>pmBlack</i>	Siempre negro
<i>pmWhite</i>	Siempre en blanco
<i>pmNot</i>	El inverso del fondo
<i>pmNotCopy</i>	El inverso del pincel
<i>pmNop</i>	No se dibujará

• BROCHAS

El interior de la entidad gráfica depende de la propiedad **Brush**, que es un objeto de la clase **TBrush**, que cuenta con la propiedad **Color** y **Style** que puede tomar uno de los valores siguientes:

Valor	Estilo del relleno
bsClear	Sin relleno
bsSolid	Sólido
bsBDiagonal	Diagonal hacia la izquierda
bsFDiagonal	Diagonal hacia la derecha
bsCross	Cuadrícula
bsDiagCross	Cuadrícula diagonal
bsHorizontal	Líneas horizontales
bsVertical	Líneas verticales

Si deseamos utilizar otro relleno podemos asignar a la propiedad **Bitmap** una matriz de ocho por ocho puntos con el patrón que deseamos repetir.

MOSTRAR IMÁGENES (Image) TImage

Permite visualizar imágenes gráficas de los tipos **Bitmap**, **Meta-File** o **Icon** cargándolas en la propiedad **Picture**, o bien en tiempo de ejecución mediante el método **LoadFromFile**.

Mediante la propiedad **Stretch** podemos controlar que la imagen que vamos a cargar se ajustará al tamaño del control, mientras que con **AutoSize** será el control el que modifique el tamaño. Mediante **Center** podemos hacer que la imagen aparezca centrada en el interior del control.

DESPLAZAMIENTO DE ÁREAS (ScrollBar) TScrollBar

Permite el desplazamiento de sólo una determinada área de la ficha, cuando algunos de los controles de la ficha deben permanecer en su lugar.

La propiedad **AutoScroll** puede tomar los valores **True** o **False**, siendo el primero el valor por defecto, en este caso aparecen las barras de desplazamiento, horizontal y vertical, cuando sea necesario. En caso de que valga **False**, será el programador el que se ocupe de mostrar las barras cuando sea procedente mediante las propiedades **HorzScrollBar** y **VertScrollBar**.

REALZAR EL INTERFAZ (Bevel)

Mediante este control podemos insertar líneas, recuadros y bordes que pueden aparecer hundidos en la ficha o sobresalir de la misma.

Mediante la propiedad **Shape** podemos seleccionar cualquiera de las figuras que aparecen en la tabla de la página siguiente, pudiendo aparecer hundido o emergido según el valor de **Style**.

Valor	Forma
bsBox	Rectángulo
bsFrame	Borde de rectángulo
bsTopLine	Línea horizontal superior
bsBottomLine	Línea horizontal inferior
bsLeftLine	Línea vertical izquierda
bsRightLine	Línea vertical derecha

CABECERAS (Header)

Permite crear literales del mismo modo que el control **Label** pero de una manera que resulte más “atractiva” para el diseño de nuestro interfaz.

Puede dividirse en tantos apartados como sea necesario, conteniendo cada uno de ellos un título. Mediante **Sections** podemos mantener una lista con los distintos títulos.

El ancho y los títulos de las divisiones pueden ser establecidos tanto en tiempo de diseño como de ejecución. En tiempo de diseño usaremos el botón derecho del ratón para establecer el ancho. En tiempo de ejecución, si la propiedad **AllowResize** vale **True**, podemos usar la propiedad **SectionWidth**, una matriz de enteros en la que cada elemento contiene la anchura en puntos de cada división.

No tiene sentido modificar la anchura de los títulos si los datos no se adaptan en consecuencia. Por ello, en tiempo de ejecución, cuando se modifica la anchura de una división se producen dos eventos: **OnSizing**, mientras se está realizando la modificación, y **OnSized** cuando se finaliza. En ambos casos se reciben los parámetros **Sender**, **ASection**, un entero indicando la división a redimensionar y **AWidth** que contendrá la nueva anchura.

7. FUNCIONES Y PROCEDIMIENTOS ÚTILES DE DELPHI

No todo en Delphi son objetos. Además de la librería de componentes existe una librería de funciones y procedimientos de utilidad. Son las mismas que conformaban la batería de



funciones de las versiones anteriores de Turbo Pascal, con algunos agregados muy prácticos.

Repasaremos ahora algunas de las más utilizadas. La referencia completa, como siempre, se puede encontrar en la ayuda del producto.

IntToStr

```
function IntToStr(Value: Longint): string;
```

Esta función toma un valor numérico entero y lo transforma a string. Para transformar un valor real, use la función FloatToStr..

StrToInt

```
function StrToInt(const S: string): Longint;
```

Esta función toma una cadena compuesta por números y la transforma al valor entero correspondiente. Si no se puede convertir, se provoca una excepción EconvertError.

Para convertir un valor real, use la función StrToFloat o StrToFloatF

StrToIntDef

```
function StrToIntDef(const s: string; Default: integer): string;
```

Esta función toma una cadena compuesta por números y la transforma al valor entero correspondiente, al igual que StrToInt; pero si la cadena **s** no contiene un valor entero válido, devuelve el valor dado en **Default** en lugar de generar un error.

FloatToStr

```
function FloatToStr(Value: Extended): string;
```

Esta función toma un valor numérico real y lo transforma a string. No se puede controlar el formato del string resultante; para esto, utilice la función FloatToStrF.

Para transformar un valor entero, use la función IntToStr..

StrToFloat

```
function StrToFloat(const S: string): Extended;
```

Esta función toma una cadena y la transforma al valor real correspondiente. Si no se puede convertir, se provoca una excepción EconvertError.

Son válidos en el string los símbolos “+”, “-”, “E” (que indica notación científica y quiere decir “por diez a la...”) y el separador decimal. Este separador (normalmente el punto o la coma) se define en el Panel de Control de Windows. No obstante, es posible cambiarlo localmente, solamente para nuestro programa, asignando el valor deseado a la variable



DecimalSeparator.

Veamos un par de ejemplos:

! el String "5,32E2" se convierte al número 532.

! el String "-43.23" se convierte en el número -43.23

! para utilizar siempre la "coma" como separador decimal, al margen de lo que esté definido en el Panel de Control, podemos hacer

DecimalSeparator:=

al principio de la aplicación, por ejemplo en el evento OnCreate del form principal. Para convertir un valor entero, use la función StrToInt.

FloatToStrF

function FloatToStrF(Value: Extended; Format: TFloatFormat; Precisión, Digits: Integer): string;

Esta función convierte un número real a su representación en caracteres (string) con el formato especificado. Los parámetros son los siguientes:

! **Value**: el número real a convertir

! **Format**: especifica el formato del string resultante. Puede ser alguna de las constantes siguientes:

" ffGeneral: el valor es convertido a la forma más corta posible, usando notación científica si es necesario.

" ffExponent: el valor es convertido a notación científica

" ffFixed: notación normal, con al menos un dígito antes de la coma decimal y la cantidad de dígitos decimales después de la coma dada por **Digits**.

" ffNumber: el mismo formato que para ffFixed pero con separadores de miles.

" ffCurrency: un valor monetario

! **Precisión**: especifica la precisión del resultado; debe ser 7 o menos para variables de tipo *single*, 15

o menos para tipo *double* y 18 o menos para *extended*.

! **Digits**: indica la cantidad de dígitos decimales cuando no se usa la notación científica, o la cantidad de dígitos del exponente cuando se la utiliza.

StrToDate

function StrToDate(const S: string): TdateTime;



Convierte una cadena a Fecha. El formato se especifica en el Panel de Control de Windows, y dentro de la aplicación se puede consultar o modificar con las variables *DateSeparator* (separador: normalmente la barra inclinada “/”) y *ShortDateFormat* (indica el orden de los componentes; por ejemplo 'd/m/y' para día/mes/año o 'm/d/y' para mes/día/año).

DateToStr

```
function DateToStr(Date: TDateTime): string;
```

Convierte una fecha a cadena. El formato de la cadena resultante está dado por la variable *ShortDateFormat*, como se explica en la función **StrToDate** más arriba.

De la misma manera se definen dos funciones **StrToTime** y **TimeToStr** que convierten entre cadenas y variables de tipo *TDateTime* pero considerando la parte de la hora, no la fecha. El carácter que se utiliza aquí para separar las distintas partes de la especificación está almacenado en la variable *TimeSeparator* (usualmente “:”).

Format

```
function Format(const Format: string; const Args: array of const): string;
```

Esta función nos permite crear un string con datos de distintos tipos fácilmente. La cadena que queremos de resultado contiene caracteres especiales que indican dónde insertar los valores dados en el argumento **Args**, y con algún control sobre el formato de la conversión.

Los parámetros son los siguientes:

Args: un array de valores. Es una lista de valores a ser introducidos en la expresión final, separados por comas y encerrados entre corchetes. El reemplazo se hace en orden de aparición, es decir que el primer valor reemplaza al primer código, el segundo al segundo y así sucesivamente.

Format: el string a ser formateado, con códigos especiales que indican el tipo de dato que debe reemplazar. El string tiene el siguiente formato:

"%" [índice ":"] ["-"] [ancho] ["."] precisión] tipo

comienza con un signo de porcentaje “%”.

El único especificador obligatorio es el de tipo, que puede ser una letra de las siguientes:

d Decimal. El argumento correspondiente debe ser un valor entero, que se convierte a un string de dígitos decimales. Si la cadena de formato tiene una especificación de precisión, indica que el string resultante debe contener al menos la cantidad especificada de dígitos; si tiene menos, se rellena con espacios a la izquierda.



- e Notación científica. El argumento debe ser un número en punto flotante. El valor se convierte a un string de la forma “d.ddd...E+ddd”. Siempre hay un dígito precediendo al punto decimal, y si el número es negativo se coloca un signo menos delante. El número total de caracteres de la cadena resultante -incluyendo el punto decimal- es dado por el especificador de precisión de la cadena de formato. Si no se da este especificador, se toma una precisión de 15. La parte que sigue a la letra “E” (que indica 10 elevado a la cantidad que sigue) tiene siempre tres dígitos y un signo, “+” o “-”.
- f Precisión fija. El argumento debe ser un número de punto flotante, que es convertido a un string de la forma “ddd.ddd...”. Si el número es negativo se agrega el signo menos delante. El número de dígitos se especifica con la precisión, tomándose un valor por defecto de 2.
- g General. El argumento debe ser un número de punto flotante, que se convierte a la cadena decimal más corta posible, usando formato fijo o científico. El número de dígitos se especifica con la precisión, tomándose 15 por defecto.
- n Número. El argumento debe ser un número en punto flotante, que se convierte a un string de la forma “d.ddd.ddd.ddd,ddd...”. Es igual que el formato con “f” pero con separadores de miles.
- m Monetario. El argumento debe ser un número en punto flotante, que se convierte a un string representando un valor monetario. El formato de la cadena es controlado por las variables globales CurrencyString, CurrencyFormat, NegCurrFormat, ThousandSeparator, DecimalSeparator, y CurrencyDecimals que se inicializan con los valores del panel de control de Windows, sección Internacional. Si se especifica un valor de precisión, tiene prioridad sobre el valor de la variable CurrencyDecimals.
- p Puntero. El argumento debe ser un valor de puntero, que es convertido a la forma segmento:offset “XXXX:YYYY” con dígitos hexadecimales.
- s String. El argumento debe ser un carácter, un string o un valor pChar. La cadena se inserta en el lugar del identificador de formato. Si se especifica un valor de precisión, se toma como longitud máxima de la cadena resultante (si se pasa de esta longitud, se trunca).
- x Hexadecimal. El argumento debe ser un valor entero, que se convierte a una cadena de dígitos hexadecimales. Si se especifica una precisión, se toma como la cantidad mínima de dígitos que debe contener la cadena resultante, llenándose por la izquierda con ceros.

Los especificadores de índice, ancho y precisión pueden ser dados directamente con dígitos decimales, o a través de parámetros indicando en la cadena de formato un asterisco “*”.

Por ejemplo,

`format(“%*.f”, [8,2,123.456])` es lo mismo que

`format(“%8.2f”, [123.456])`



El especificador de ancho indica el mínimo ancho de un campo para la conversión. Si la cadena resultante es más corta, se llena con espacios a la izquierda; salvo que se especifique también el indicador de justificación a la izquierda (un signo menos antes del ancho) en cuyo caso los espacios se agregan a la derecha. El indicador de índice nos permite reutilizar los valores del array de parámetros. Este número especifica el índice del parámetro a usar en el próximo reemplazo. Por ejemplo, `format("%d %d %1:d,[1,2,3])` da como resultado la cadena '1 2 2'.

DateToStr

function DateToStr(Date: TDateTime): **string**;

Convierte una fecha dada en formato TdateTime a un string.

StrToDate

Convierte un string en TdateTime. Toma la hora como las 0:00. Si no puede convertir (el string contiene una fecha errónea) se genera una excepción EconvertError.

Si se ingresa el año con sólo dos dígitos, se determina el siglo a que pertenece utilizando la variable global TwoDigitYearCenturyWindow. Los detalles se pueden ver en la ayuda, con ejemplos.

IncMonth

function IncMonth(**const** Date: TDateTime; NumberOfMonths: Integer): TDateTime;

Incrementa la fecha pasada como parámetro (Date) en el número de meses dado por **NumberOfMonths**, tomando en cuenta el cambio de año y la diferencia de días entre los meses, si es necesario. El número de meses a incrementar puede ser negativo, dando como resultado una fecha anterior a la original.

