

CLASES

EN

VISUAL FOXPRO 9

Walter R. Ojeda Valiente

Asunción, Paraguay, Julio de 2011

ÍNDICE

Introducción	3
Como se llegó a las clases	4
Beneficios al usar clases	6
Características de las cosas que existen	7
Propiedades, métodos y eventos	9
Clases y objetos	10
Clasificación de las clases	11
Ejemplo N° 1	12
Ejemplo N° 2	15
Simplificando la escritura	17
Clases base	19
Como crear subclases	23
Bibliotecas de clases	24
Ejemplo N° 4	25
Protegiendo y ocultando miembros de las clases	32
Modelando objetos	37
Creando formularios por programa	38
Comandos, funciones y métodos usados en la P.O.O.	40
Como acceder a una propiedad de la clase padre	43
Como ejecutar un método de la clase padre	45
Una clase label desplazante	48
Una clase label titilante	52
Una clase label sombreada	53
Una clase label para títulos	55
Una clase de botones	59
Una clase para mostrar fotografías	60
Varias clases para ingresar datos	66
Agregar un formulario a otro formulario	69
Métodos ACCESS y ASSIGN	71
El objeto _SCREEN	74
Manejo de errores en la P.O.O	76
Automatización de Word	78
Automatización de Excel	81
Una clase grilla mejorada	83
Creando un constructor	85
La función BINDEVENT()	88
¿Se puede convertir a VFP en un lenguaje fuertemente tipado?	89
Trucos y consejos	92
Conclusión	95

CLASES EN VISUAL FOXPRO 9

INTRODUCCIÓN

Aunque el Visual FoxPro permite manejar clases desde hace un montón de años, hay aún muchos programadores “foxeros” que no las utilizan, quizás porque no les encuentran utilidad o quizás porque les parece algo difícil de aprender. El objetivo de este documento es mostrar como se las utiliza y todas las ventajas que pueden obtenerse de ellas.

Desde luego que, como todo nuevo conocimiento complejo, no se aprenderá en un solo día a ser un experto; pero la intención del autor es que el proceso de aprendizaje de los estudiantes sea lo más rápido y sencillo posible.

Saber como utilizar las clases en la programación vale la pena, como se verá a lo largo de este documento.

Hay muchos ejemplos, muchos programas con sus respectivos códigos fuentes que acompañan a este documento. Es importante que a medida que los vayas encontrando en estas páginas también los vayas ejecutando en tu computadora para que entiendas mejor que es lo que hacen y como lo hacen.

Ejecuta todos los programas .PRG y todos los formularios .SCX que encontrarás en la misma carpeta donde se encuentra este documento.

Aunque este documento está basado en Visual FoxPro 9 y todos los ejemplos se han creado y verificado con esa versión del lenguaje, mucho de lo escrito es también aplicable a versiones anteriores. Si no usas la versión 9, no obtendrás el 100% de este documento, pero igualmente en algo te ayudará.

COMO SE LLEGÓ A LAS CLASES

Una pregunta muy común es: “¿por qué yo debería aprender a usar clases si durante años no las he usado y me ha ido muy bien sin ellas?”

Para responder a esta pregunta haremos un poco de Historia.

En los primeros lenguajes de programación (Cobol, Fortran, Basic, etc.) no se las usaba, es más ni siquiera se había pensado en inventarlas. En esos lenguajes cuando se quería ejecutar código que no se encontraba en la siguiente línea del programa se usaba la instrucción GO TO (ir a). Entonces el flujo del programa subía y bajaba de tal manera que era casi imposible seguirlo, cuando se quería modificar algo en el código fuente era muy complicado porque tantas subidas y bajadas hacían que uno fácilmente se perdiera. Para empeorar las cosas todas las variables eran públicas o sea que todas eran visibles en todas partes y en un programa grande frecuentemente ocurría que el programador pensaba que una variable tenía un cierto valor pero en realidad tenía otro. A esta forma de programar acertadamente se la denominó “programación spaghetti” porque cuando se escribía en un papel el flujo del programa realmente parecía un plato de comida con tallarines spaghetti. O sea, se imprimía el código fuente en varias hojas de papel y luego con un bolígrafo o lápiz se quería seguir las instrucciones que se iban ejecutando. Lo que se obtenía era un spaghetti.

El gran problema con la programación spaghetti era que se cometían muchos errores y era muy difícil corregirlos. Buscar donde estaba el error y corregirlo tomaba mucho tiempo, sólo para descubrir que al “corregirlo” se añadieron nuevos errores.

Es por eso que se pensó en como evitar el uso del muy mal afamado GO TO y así fue inventada lo que se llamó “programación estructurada”. En esta metodología para cambiar el orden de ejecución de las instrucciones se usan construcciones del tipo:

DO WHILE

...

...

ENDDO

REPEAT

...

...

UNTIL

y similares.

La “programación estructurada” fue un gran avance. La cantidad de errores que se cometían al programar y al modificar programas disminuyó muchísimo, aumentando por lo tanto la calidad de los programas y disminuyendo el tiempo para finalizar una aplicación. Y como tiempo es dinero, se ganó mucho dinero gracias a ella.

Pero la Historia no termina ahí. ¿Por qué? porque aunque la “programación estructurada” era mucho mejor que la “programación spaghetti” aún tenía un gran problema: no era fácil reemplazar un código fuente por otro y que el resto de la aplicación continuara igual. Entonces se pensó en como conseguir algo similar al hardware que tiene problemas. Si una pieza de hardware tiene problemas, un monitor por ejemplo ¿qué se puede hacer? Quitarlo y sustituirlo por otro. ¿No funciona tu teclado? Lo sustituyes por otro. ¿No funciona tu disco duro? Lo sustituyes por otro.

Si alguien ya escribió un código que realiza una determinada tarea: ¿acaso no es muchísimo más fácil y muchísimo más rápido insertar ese código en tu aplicación que volver a escribirlo? Además, ese código externo puede ser de muy buena calidad, estar muy bien probado y el autor conocer mucho más del tema que tú.

En programación muchas veces ocurre que se necesita escribir un código fuente muy similar a alguno que ya se había escrito con anterioridad. Quizás tú mismo o quizás otro programador ya tenía escrito ese código. Si lo vuelves a escribir a eso se le llama: “reinventar la rueda”. Y para ser productivo no deberías reinventar muchas ruedas.

La programación que utiliza clases se llama “orientada a los objetos” y se la abrevia como POO en castellano o como OOP (Object Oriented Programming) en inglés.

Por lo tanto, tenemos:

- Programación spaghetti
- Programación estructurada
- Programación orientada a los objetos

En la POO (Programación Orientada a los Objetos) las variables y las rutinas y las funciones que usan a esas variables se encuentran unidas, (se les suele decir: “encapsuladas”) en un solo lugar. Eso facilita agregar nuevas funcionalidades y encontrar y corregir errores de programación.

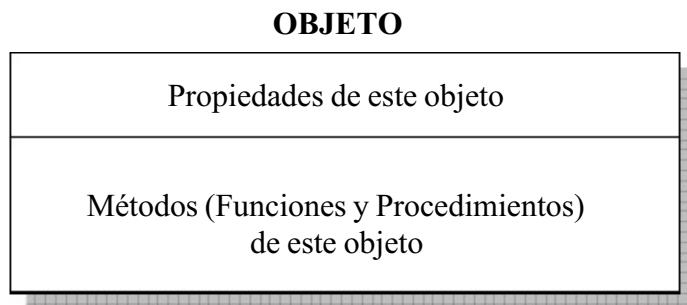


Diagrama N° 1

BENEFICIOS AL USAR CLASES

Cuando se usan clases se pueden desarrollar aplicaciones con mejor calidad y terminarlas más rápidamente que cuando no se las utiliza. Ya que obtendremos:

- **Código fuente reusable.** Eso significa que escribes tu código fuente una sola vez y luego lo puedes usar en muchas aplicaciones distintas.
- **Código más fácil de mantener.** Encontrar y corregir los errores es más sencillo.
- **Encapsulación.** Lo que hace cada porción de código puede ocultarse al resto de la aplicación para que no haya interferencias. O sea, el resto de la aplicación sólo ve lo que se quiere que vea y se le puede ocultar lo que no es necesario que vea.
- **Abstracción.** Que permite concentrarse en lo que realmente se necesita, sin preocuparse por los detalles de bajo nivel.
- **Creación de subclases.** Una subclase es una clase derivada de otra clase y que puede incluir todas (o solamente algunas) de las características de la *clase padre* más algunas características adicionales.
- **Más fácil aprendizaje de otros lenguajes de programación.** Muchos de los nuevos lenguajes de programación utilizan clases. Si aprendemos a utilizarlas con Visual FoxPro entonces aprender un nuevo lenguaje nos resultará más fácil.

CARACTERÍSTICAS DE LAS COSAS QUE EXISTEN

Todo lo que existe en el Universo tiene algunas características que ayudan a identificarlo y a diferenciarlo de las otras cosas existentes. Por ejemplo, pueden tener un color, una masa o peso, un tamaño, una velocidad, una ubicación, un material del que están constituidos, etc.

Tomemos por ejemplo una mesa:

- tiene un tamaño que se puede medir en metros
- tiene un color o una combinación de colores
- tiene una forma, que puede ser rectangular, cuadrada, ovalada, redonda, etc.
- está hecha de un material (o más de uno): madera, hierro, piedra, cemento, plástico, etc.
- tiene un peso que se puede medir en kilogramos
- tiene un precio
- etc.

Todas esas cosas que *tiene* o que *posee* la mesa se llaman **propiedades**.

Además, algunas de las cosas existentes también pueden realizar acciones, o sea que pueden hacer algo. Tomemos ahora como ejemplo un automóvil, tiene ciertas propiedades como:

- tamaño
- color
- forma
- material
- peso
- cilindrada del motor
- tipo de combustible que utiliza
- nombre del fabricante
- año de fabricación
- precio
- etc.

Pero además, un automóvil puede realizar acciones, tales como:

- acelerar
- desacelerar
- detenerse
- girar a la izquierda
- girar a la derecha
- encender los faros
- apagar los faros
- etc.

Todo lo que puede hacer el automóvil son sus **métodos**.

Un subgrupo dentro de los **métodos** son los **eventos**. ¿Qué es un **evento**? Es un **método** que se ejecuta sin que se pida su ejecución.

En el ejemplo del automóvil un **evento** es que al sentarte en el asiento del conductor y al cerrar la puerta el cinturón de seguridad se te coloque automáticamente. Algunos automóviles modernos tienen esa característica.

Entonces, un **método común** es lo que hace el automóvil porque el conductor le pidió que hiciera y un **evento** es lo que hace el automóvil por sí solo cuando ocurre una condición (a eso se le suele llamar: dispararse el evento).

El automóvil se mueve más rápido porque el conductor apretó el acelerador, se mueve más lentamente porque dejó de apretarlo, se detiene porque apretó el freno. Esos son **métodos comunes**. El conductor le está ordenando al automóvil lo que éste debe hacer.

Pero cuando el conductor se sienta y cierra la puerta él no le está ordenando al automóvil que le coloque el cinturón de seguridad. Eso lo hace el automóvil por sí solo cuando se cumplió una condición (en este caso, que se cerrara la puerta con el conductor sentado en el asiento). Eso es un **evento**.

Para escribir menos, generalmente se le llama solamente con **método** a los que en realidad son **métodos comunes**.

PROPIEDADES, MÉTODOS Y EVENTOS

Una **propiedad** es una característica que posee un objeto (un color, un tamaño, un peso, etc.)

Un **método** es una acción que realiza un objeto.

- Si se le pidió que la realice (subir, bajar, acelerar, frenar, girar, etc.) entonces es un **método común** o simplemente **método**.
- Si no se le pidió que la realice pero igualmente la realizó porque ocurrió un hecho que la desencadenó, entonces es un **evento**.

Todos los objetos tienen **propiedades** pero solamente algunos tienen **métodos o eventos**.

Una mesa, una silla, una casa, una calle, una hoja de papel, una baldosa, un zapato, tienen **propiedades**. Una vaca, una mosca, un automóvil, un televisor, una computadora, una persona, también tienen **métodos**.

En el caso de un programa de computadora, un **evento** es que el usuario presione una tecla del teclado. Otro evento es que presione el botón izquierdo del mouse o ratón. Y hay muchos **eventos** más, como se verá más adelante.

¿Cómo distinguir si es un **método común** o si es un **evento**? Si se ejecuta porque el programador le pidió que se ejecute entonces es un **método común**. Si se ejecuta sin que el programador lo haya pedido, entonces es un **evento**.

Recordemos que todos los **eventos** son **métodos** pero no todos los **métodos** son **eventos**.

Y ya que todos los **eventos** son **métodos**, la forma de programarlos es exactamente la misma, lo único que varía es cuando se ejecutarán.

Una **propiedad** es equivalente a una variable, pero solamente puede ser accedida dentro de la clase. Es inaccesible desde afuera de ella.

Un **método** es equivalente a un **procedimiento** o a una **función**, pero solamente puede ser accedido dentro de la clase. Es inaccesible desde afuera de ella.

Los programadores pueden crear todos los **métodos** que deseen pero no pueden crear nuevos **eventos**. Y en realidad no es necesario crear nuevos, ya que el Visual FoxPro tiene todos los que puedas necesitar.

CLASES Y OBJETOS

¿Qué es una clase?

Es una porción de código fuente que se utiliza como un modelo del cual se derivarán **objetos** que tendrán todas y cada una de las características de la **clase**.

Los programadores definimos **clases**, las cuales están compuestas por **propiedades** y por **métodos**.

Eso significa que se encuentran unidos (en la POO se dice: encapsulados) los datos y los procesos que manejan a esos datos.

Al hecho de especificar todas las características que tendrán algunos objetos se le llama: **definir la clase**.

¿Qué es un objeto?

Cuando se define una **clase** se especifican todas las características (propiedades y métodos) que tendrán los **objetos** de esa **clase** pero aún no existe algún **objeto** que posea esas características. Para que exista un **objeto** con todas las características de una **clase** se lo debe *instanciar*.

¿Qué significar instanciar un objeto?

Crear un **objeto** que tenga todas las características de una **clase**.

CLASIFICACIÓN DE LAS CLASES

Las **clases** pueden clasificarse según su:

- Tipo
- Visibilidad
- Origen

Según su Tipo

- Contenedora
- Control

Una **clase** es contenedora cuando puede tener a **objetos** de otras **clases** dentro suyo. Ejemplos de **clases** contenedoras son los formularios, los contenedores, los marcos de página, las grillas.

Una **clase** es de control cuando no puede contener a otra **clase** dentro suyo. Por lo tanto las **clases** de control no tienen el método AddObject(). No pueden agregarse un objeto.

Según su visibilidad

- Visible
- No visible

Las **clases** visibles (ejemplos: formularios, etiquetas, grillas, imágenes) pueden verse en el monitor de la computadora. Las no visibles (ejemplos: timer, custom) no pueden verse, no hay forma de verlas.

Según su origen

- Basada en una **clase** que no es del usuario
- Basada en una **clase** del usuario

EJEMPLO N° 1

En este ejemplo definiremos una **clase** llamada PERSONA (por claridad y legibilidad, los nombres de las **clases** se escriben en singular, nunca en plural)

EJEMPLO1.PRG

```
Local loNovial, loNovia2

CLEAR

loNovial = CreateObject ("PERSONA")

loNovial.cNombre      = "Marcela"
loNovial.dFechaNacimiento = CtoD("07/10/1980")

loNovia2 = CreateObject ("PERSONA")

loNovia2.cNombre      = "Silvia"
loNovia2.dFechaNacimiento = CtoD("28/09/1988")

loNovial.MostrarDatos()
loNovia2.MostrarDatos()

Release loNovial, loNovia2

Return
*
*
DEFINE CLASS PERSONA AS CUSTOM
  cNombre      = Space(20)
  cApellido     = Space(20)
  dFechaNacimiento = {}
  PROCEDURE MostrarDatos
    ? This.cNombre, This.dFechaNacimiento
    RETURN
ENDDEFINE
*
*
```

Explicación

Primero, se declararon dos variables locales llamadas loNovia1 y loNovia2 respectivamente. La letra “l” inicial significa que es una variable local. La letra “o” que le sigue significa que es un objeto. No es necesario llamar así a las variables, también se las podría haber llamado Novia1 y Novia2 y hubiera funcionado perfectamente, pero es una convención que se utiliza para hacer más entendible al código fuente.

Luego se crea un **objeto** basado en la **clase** PERSONA que está definida al final del programa. A eso se le llama *instanciar el objeto*. Es importante recordar que las definiciones de **clases** siempre deben escribirse al final del programa. ¿Por qué? porque el código que se escriba después de las definiciones de las **clases** nunca será ejecutado.

Después, se le asignan valores a las **propiedades** cNombre y dFechaNacimiento. Para referirse a una **propiedad** siempre hay que antecederla con el nombre del **objeto** al cual pertenece ¿por qué eso? Porque puede haber muchos **objetos** y **clases** distintos que tengan **propiedades** llamadas igual y el Visual FoxPro necesita saber a cual de ellas nos estamos refiriendo. Por ese motivo, siempre hay que escribirlas de la forma:

Objeto.propiedad

A continuación se crea otro **objeto** basado en la **clase** PERSONA. Eso significa que con este **objeto** se pueden acceder a todas las **propiedades** y a todos los **métodos** que posee la **clase** PERSONA.

Luego se le asignan valores a las **propiedades** del **objeto** recién creado.

Seguidamente se ejecuta un **método** del **objeto**. Ese **método** lo que hace es mostrar en la pantalla el contenido de dos **propiedades** del **objeto**. Al igual que con las **propiedades**, cuando queremos referirnos a un **método** debemos hacerlo escribiendo:

Objeto.método()

NOTA: Los paréntesis son opcionales, pero es mejor escribirlos para que el código sea más legible.

Alternativamente, en lugar de escribir:

```
loNovial.MostrarDatos()
```

se pudo haber escrito:

```
? loNovial.cNombre, loNovial.dFechaNacimiento
```

Sin embargo, en una correcta POO la primera forma es mucho más preferible ¿por qué? porque todo lo que pueda realizarse dentro de un **objeto** debe realizarse dentro de ese **objeto**.

Después se liberan los **objetos** de la memoria con la instrucción RELEASE. Cada vez que se crea un **objeto** éste ocupa un cierto espacio en la memoria. Si se lo terminó de usar y no se libera ese espacio entonces se está desperdiciando memoria. Y un buen programador no debería desperdiciarla.

Más abajo se define una **clase** llamada PERSONA. Veamos su definición con más detenimiento.

DEFINE CLASS

le dice al Visual FoxPro que queremos definir (crear) una nueva **clase**.

PERSONA

es el nombre que tendrá la nueva **clase**.

AS CUSTOM

todas las **clases** deben tener una **clase base**. O sea que deben estar basadas en otra **clase**. No se puede tener una **clase** que no esté basada en otra, no puede venir del aire, sí o sí debe provenir de otra **clase**.

cNombre

Es una **propiedad** de tipo carácter, que tiene 20 de ellos

cApellido

Es una **propiedad** de tipo carácter, que tiene 20 de ellos

dFechaNacimiento

Es una **propiedad** de tipo fecha

PROCEDURE MostrarDatos

Es un procedimiento interno a la **clase PERSONA** y solamente puede ser llamado por un **objeto** de dicha **clase**. Eso significa que si en el programa principal escribimos MostrarDatos() obtendremos un mensaje de error. Sí o sí debemos llamarlo usando un **objeto** de la **clase PERSONA**.

MostrarDatos() ← Es un error, no funcionará
loNovia1.MostrarDatos() ← Es correcto, funciona perfectamente

? This.cNombre, This.dFechaNacimiento

Cuando estamos dentro de una **clase** y queremos referirnos a una **propiedad** de esa misma **clase** debemos anteceder su nombre con la palabra THIS (este). Si no lo hacemos, obtendremos un mensaje de error.

? cNombre ← Es un error, no funcionará
? This.cNombre ← Es correcto, funciona perfectamente

RETURN

Finaliza el procedimiento llamado MostrarDatos. También puede escribirse ENDPROC si se prefiere, es indistinto, ya que cualquiera de ellos funciona igual.

ENDDEFINE

Finaliza la definición de la **clase PERSONA**.

EJEMPLO N° 2

Basándonos en la **clase PERSONA** creamos una nueva **clase** llamada SOCIO, la cual tendrá los datos de los socios de un Club.

EJEMPLO02.PRG

```
Local loNuevoSocio

CLEAR

loNuevoSocio = CreateObject ("SOCIO")

loNuevoSocio.cNombre      = "Juan"
loNuevoSocio.cApellido    = "Pérez"
loNuevoSocio.dFechaNacimiento = CtoD ("03/01/1980")
loNuevoSocio.dFechaIngreso = Date()

loNuevoSocio.MostrarDatos()
loNuevoSocio.MostrarDatos2()

Release loNuevoSocio

Return
*
*
*
DEFINE CLASS PERSONA AS CUSTOM
cNombre      = Space(20)
cApellido    = Space(20)
dFechaNacimiento = {}
PROCEDURE MostrarDatos
? This.cNombre, This.dFechaNacimiento
RETURN
ENDDEFINE
*
*
*
DEFINE CLASS SOCIO AS PERSONA
dFechaIngreso = {}
PROCEDURE MostrarDatos2
? This.cNombre, This.cApellido, This.dFechaIngreso
RETURN
ENDDEFINE
*
```

Si miramos la definición de la **clase SOCIO** veremos que:

1. Está basado en la **clase PERSONA** que habíamos definido anteriormente. Eso significa que posee todas las **propiedades** y todos los **métodos** de la **clase PERSONA**. En la terminología de la POO se dice que los *heredó*
2. Se le agregó una nueva **propiedad** (llamada: dFechaIngreso)
3. Se le agregó un nuevo **método** (llamado MostrarDatos2()).
4. Las **propiedades** cNombre y cApellido no fueron definidas en la **clase SOCIO** sino que fueron definidas en la **clase PERSONA**. Pero como SOCIO desciende de PERSONA entonces ha heredado esas **propiedades**.

5. El **método** MostrarDatos() no fue definido en la **clase SOCIO**, fue definido en la **clase PERSONA**, pero como SOCIO es un descendiente de PERSONA entonces ha heredado ese **método**.

Lo que todo esto significa es que un **objeto** creado a partir de la **clase SOCIO** tiene acceso a todas las **propiedades** y a todos los **métodos** de la clase SOCIO y también tiene acceso a todas las **propiedades** y a todos los **métodos** de la clase PERSONA. Interesante, ¿verdad?

En la POO (Programación Orientada a los Objetos) lo correcto es crear clases bases de las cuales se derivan subclases que tienen más funcionalidad, de las cuales se derivan subclases que tienen más funcionalidad y así sucesivamente. Se va formando una estructura de árbol, como se ve en el siguiente diagrama:

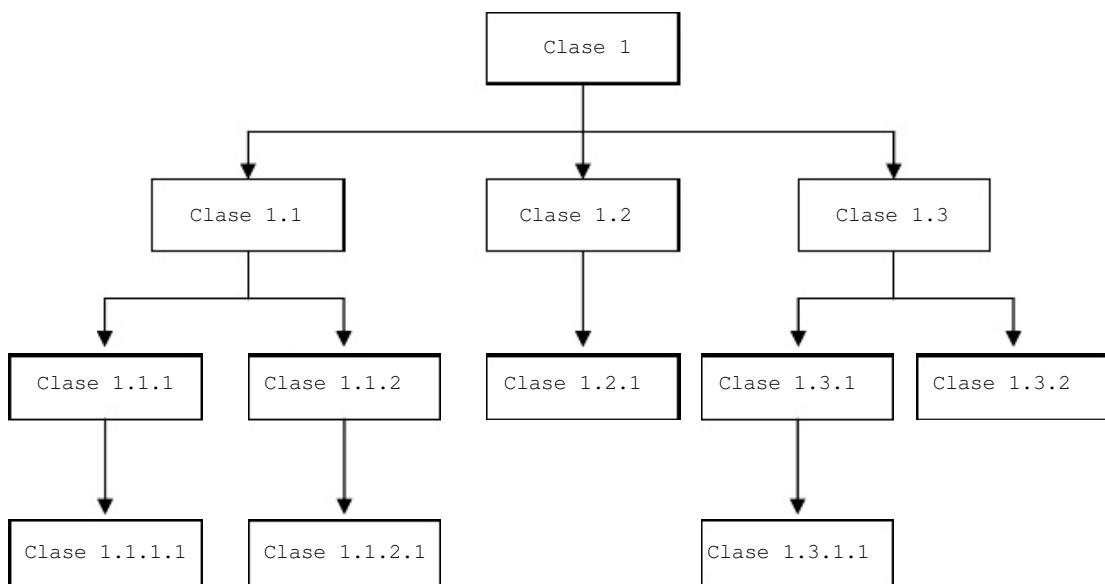


Diagrama Nº 2

En este diagrama podemos ver que cada clase *hereda* todas las funcionalidades (**propiedades** y **métodos**) de la clase superior. Así por ejemplo, la clase 1.1.2.1 ha heredado la funcionalidad de la clase 1.1.2 la cual ha heredado la funcionalidad de la clase 1.1, la cual ha heredado la funcionalidad de la clase 1.

Gracias a la *herencia* se puede escribir mucho menos código fuente y hacerlo más fácilmente mantenible. Si hay un error en una clase, ese error se propaga a todas sus subclases, pero al corregirlo, también se corrige en todas sus subclases.

SIMPLIFICANDO LA ESCRITURA

Cuando trabajamos con **clases** muchas veces tenemos que escribir repetidamente los nombres de los **objetos**. El Visual FoxPro tiene una instrucción que no tienen otros lenguajes y que facilita y simplifica mucho la escritura del código fuente, haciéndolo más legible. Es la construcción WITH ... ENDWITH.

Si usamos WITH ... ENDWITH, entonces:

1. Escribimos menos
2. Ahorramos tiempo al escribir
3. Hacemos nuestro código fuente más legible
4. Nuestro programa se ejecuta más rápido

¿Cuándo debemos usar WITH ... ENDWITH? Cuando hay dos o más líneas de código seguidas que se refieren al mismo objeto.

Veamos como funciona: el programa EJEMPLO03.PRG es idéntico al programa EJEMPLO02.PRG, hacen exactamente lo mismo. La única diferencia es que ahora se usa la construcción WITH ... ENDWITH.

EJEMPLO03.PRG

```
Local loNuevoSocio

CLEAR

loNuevoSocio = CreateObject ("SOCIO")

with loNuevoSocio
    .cNombre      = "Juan"
    .cApellido     = "Pérez"
    .dFechaNacimiento = CtoD("03/01/1980")
    .dFechaIngreso = Date()
    .MostrarDatos()
    .MostrarDatos2()
endwith

Release loNuevoSocio

Return
*
*

DEFINE CLASS PERSONA AS CUSTOM
    cNombre      = Space(20)
    cApellido     = Space(20)
    dFechaNacimiento = {}
PROCEDURE MostrarDatos
    ? This.cNombre, This.dFechaNacimiento
RETURN
ENDDFINE
*
*
```

```
DEFINE CLASS SOCIO AS PERSONA
    dFechaIngreso = {}
PROCEDURE MostrarDatos2
    ? This.cNombre, This.cApellido, This.dFechaIngreso
RETURN
ENDDEFINE
```

Si miramos los códigos fuente de EJEMPLO02.PRG y de EJEMPLO03.PRG veremos que en este último se han escrito menos caracteres.

Al escribir menos caracteres, se ha ganado tiempo.

Es más fácil entender EJEMPLO03.PRG cuando nos acostumbramos. Quizás al principio no lo sea, pero en unos días lo será.

Si usamos la construcción WITH ... ENDWITH entonces el Visual FoxPro evalúa al **objeto una sola vez**. Si no la usamos, entonces lo evalúa en cada línea. Eso lleva su tiempo que aunque es mínimo siempre es algo.

¿Conclusión? Siempre deberíamos usar WITH ... ENDWITH cuando dos o más líneas de código se estén refiriendo al mismo **objeto**.

A partir de este momento, en todos los ejemplos de este documento se los usará.

CLASES BASE

En el Visual FoxPro todas las clases deben derivarse de otra clase, no podemos tener una clase que apareció de la nada, debe obligatoriamente derivarse de otra clase. La clase de la cual se deriva se llama su **clase base**.

Y como ya habíamos visto, cada **clase** que nosotros creamos *hereda* todas las funcionalidades de su **clase base**.

Las **clases base** del Visual FoxPro son las siguientes:

CheckBox

Permite cambiar entre tres estados: Verdadero, Falso, Nulo.

Collection

Agrupa ítems relacionados, generalmente **objetos**, que pueden ser de cualquier tipo.

Column

Crea una columna en una grilla (grid en inglés). Puede contener datos de un campo, un valor o controles.

ComboBox

Inicialmente muestra un solo ítem, pero cuando se lo selecciona muestra una lista de todos los ítems que pueden ser seleccionados.

CommandButton

Ejecuta una acción.

CommandGroup

Crea un grupo (o sea: agrupa) a varios botones de comando.

Container

Crea un **objeto** que puede contener a otros **objetos**. Se puede acceder a los **objetos** contenidos tanto en tiempo de diseño como en tiempo de ejecución.

Control

Crea un **objeto** que puede contener a otros **objetos**. Es similar al Container pero los **objetos** contenidos son inaccesibles tanto en tiempo de diseño como en tiempo de ejecución.

Cursor

Se crea cuando una tabla o una vista es agregada al entorno de datos de un formulario, un conjunto de formularios, o un informe.

CursorAdapter

Permite manejar muchos tipos de datos externos como si fueran datos nativos de Visual FoxPro. Incluye ODBC, ADO, XML.

Custom

Crea un **objeto** definido por el usuario (o sea por los programadores).

DataEnvironment

Se crea automáticamente en el momento de crear un formulario, conjunto de formularios o informe. Funciona como un contenedor de las clases Cursor, CursorAdapter y Relation.

EditBox

Crea un campo de edición, el cual puede contener texto de gran longitud. Ese texto se puede cortar, copiar y pegar.

Empty

Crea una clase vacía que no contiene **propiedades ni métodos ni eventos**. No se pueden derivar clases de ella ni agregarle **métodos**. Sin embargo, en tiempo de ejecución se le pueden agregar **propiedades** con la función AddProperty().

Exception

Es creada automáticamente cuando ocurre un error en el bloque TRY de una construcción TRY... CATCH ... FINALLY. El comando THROW también crea un objeto de tipo *Exception*.

Form

Crea un formulario al cual se le pueden agregar controles.

FormSet

Crea un conjunto de formularios.

Grid

Crea una grilla. Es un **objeto** que puede mostrar datos en filas y en columnas. Es también un contenedor que contiene **objetos** de tipo *Column*.

Header

Crea una cabecera para una columna de una grilla. Puede mostrar un texto y una imagen en la parte superior de la columna y también puede responder a eventos.

Hyperlink

Permite navegar en Internet Explorer.

Image

Muestra archivos gráficos.

Label

Muestra texto que el usuario no puede cambiar directamente.

Line

Muestra una línea horizontal, vertical u oblícua.

ListBox

Muestra una lista de todos los ítems que pueden ser seleccionados.

OLE Bound

Permite mostrar el contenido de un **objeto** OLE (como una planilla Excel o un documento Word) en un campo general de una tabla.

OLE Container

Permite agregar **objetos** OLE a la aplicación. Incluye controles ActiveX (.ocx) y planillas Excel y documentos Word.

OptionButton

Crea un botón de opción.

OptionGroup

Crea un grupo de botones de opción. Al elegir uno de ellos todos los demás se quedan como no seleccionados.

Page

Crea una página en un marco de páginas.

PageFrame

Crea un marco de página, el cual contiene varias páginas.

ProjectHook

Se crea automáticamente cada vez que un proyecto es abierto, permitiendo acceso por programa a los **eventos** del proyecto.

Relation

Se crea automáticamente cuando se establecen relaciones dentro del entorno de datos para un formulario, conjunto de formularios o informe.

ReportListener

En un comando REPORT FORM o LABEL FORM antes de que el contenido sea enviado a su destino (pantalla, impresora, etc.) es atrapado por este **objeto** lo cual permite una comunicación ida y vuelta con el motor del informe.

Separator

Agrega espacios entre los controles de una barra de herramientas.

Session Object

Crea un objeto del usuario (o sea: programador) que posee y administra su propia sesión de datos.

Shape

Muestra un rectángulo, un cuadrado, un círculo o un óvalo.

Spinner

Permite que el usuario elija un número de entre varios valores numéricos consecutivos.

TextBox

Sirve para agregar, borrar, o modificar el texto de una variable de memoria, de un elemento de un vector o matriz, o de un campo. Se puede cortar, copiar y pegar texto.

Timer

Ejecuta un código cuando llega el momento predeterminado. Es invisible para el usuario, pero el control es visible en tiempo de diseño.

ToolBar

Es un contenedor que aparece siempre en la parte superior de la aplicación, y que puede contener cualquier tipo de control, por ejemplo: Label, TextBox, CommandButton, ComboBox, Image, etc.

XMLAdapter

Si se importa un XML se pueden crear tablas y también usando tablas se pueden crear objetos XML.

XMLField

En un objeto XMLTable la colección Fields contiene objetos XMLField. Y cada campo XMLField representa la estructura de un campo que es significativo para el Visual FoxPro.

XMLTable

La colección Tables en un objeto XMLAdapter contiene objetos XMLTable y describe XML como cursores de Visual FoxPro junto con la información de las relaciones.

CLASES QUE NO PUEDEN TENER CLASES DERIVADAS

Las siguientes **clases** no pueden ser usadas para crear **subclases**:

- Column
- Empty
- Header

PROPIEDADES QUE TODAS LAS CLASES POSEEN (excepto Empty)

- Class
- BaseClass
- ClassLibrary
- ParentClass

EVENTOS QUE TODAS LAS CLASES POSEEN (excepto Empty)

- Init
- Destroy
- Error

COMO CREAR SUBCLASES

Como ya habíamos visto, una **subclase** puede incluir todas o algunas de las características de la **clase** de la cual deriva, a la cual se le llama **clase padre**. La **clase** que hereda es la **clase hija**.

Creando y usando **subclases** provee los beneficios de *reusabilidad* (lo cual nos permite escribir menos) y de *herencia* (lo cual nos ahorra tiempo porque cuando una **clase padre** se actualiza también automáticamente se actualizan todas las **clases hijas**).

En Visual FoxPro hay dos formas de crear **subclases**:

1. Por programa (en un archivo .PRG)
2. Usando el Diseñador de Clases

CREANDO CLASES POR PROGRAMA

Se usa el comando **DEFINE CLASS**.

Es lo que hemos usado en los programas hasta ahora (EJEMPLO01.PRG, EJEMPLO02.PRG y EJEMPLO03.PRG)

CREANDO CLASES USANDO EL DISEÑADOR DE CLASES

Aquí podemos hacerlo de dos maneras:

1. Con File | New ... | Class | New File |
2. Con CREATE CLASS

En todos los ejemplos de este documento crearemos nuevas **clases** utilizando **DEFINE CLASS** o utilizando **CREATE CLASS**.

BIBLIOTECAS DE CLASES

Cuando tenemos muchas **clases**, en lugar de tenerlas todas esparcidas, desparramadas por ahí, es mucho más conveniente tenerlas agrupadas en un solo archivo. A ese archivo se le llama **biblioteca de clases**.

En Visual FoxPro todas las bibliotecas de clases tienen la extensión .VCX

Dentro de una biblioteca de clases podemos tener una o (usualmente) muchas **clases**. Lo más lógico es que todas esas **clases** estén relacionadas entre sí.

Por ejemplo, en una biblioteca de clases (a la cual podríamos llamarla como BOTONES.VCX) podemos tener los modelos predefinidos de nuestros botones de comando. Entonces, cuando necesitamos un nuevo botón simplemente lo derivamos de una de esas **clases** predefinidas.

¿Y por qué no usar simplemente el botón que viene incluido con el VFP?

Porque el botón que nosotros hacemos está *personalizado*, tiene todas las funcionalidades del botón estándar y además las nuevas funcionalidades que le hemos agregado.

Cuando programamos en Visual FoxPro lo mejor es usar nuestras propias clases, derivadas de las **clases base**. Eso quiere decir que en lugar de usar un control label es preferible usar un control *derivado* del control label. En lugar de usar un control form es mejor usar un control *derivado* del control form. Y así sucesivamente.

EJEMPLO N° 4

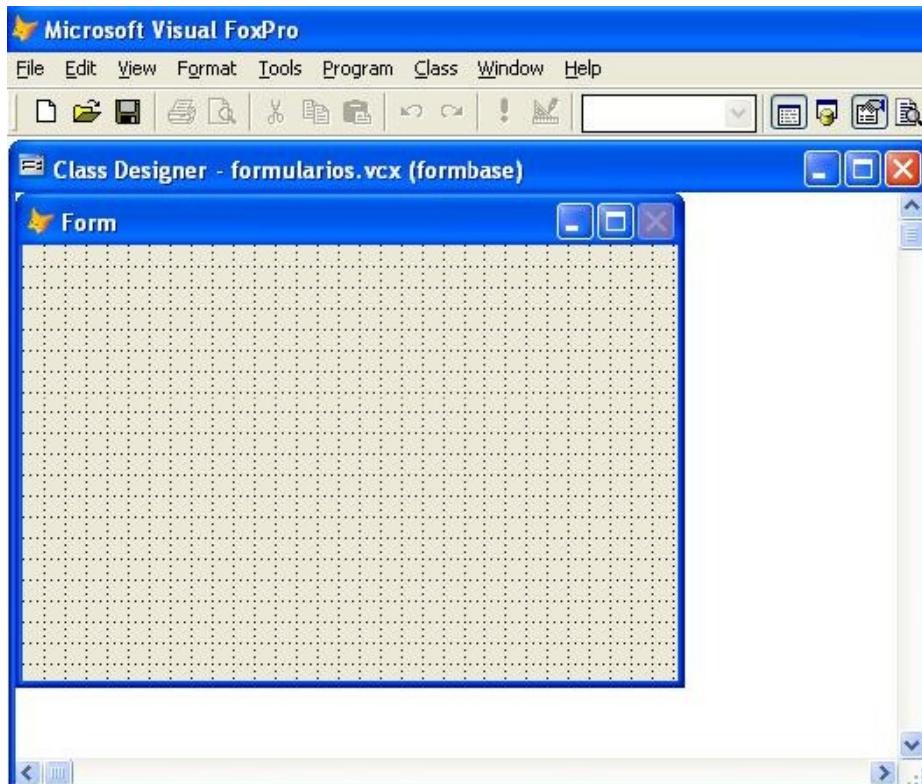
En este ejemplo crearemos un formulario base, o sea un formulario que utilizaremos como base de todos los demás formularios de nuestra aplicación.

Pero ahora lo haremos visualmente, eso significa que no usaremos la instrucción **DEFINE CLASS** sino que usaremos **CREATE CLASS**.

Al escribir en la ventana de comandos del Visual FoxPro:

```
CREATE CLASS FormBase OF Formularios AS Form
```

Veremos la siguiente pantalla:



Pantalla N° 1

¿Qué fue lo que hicimos?

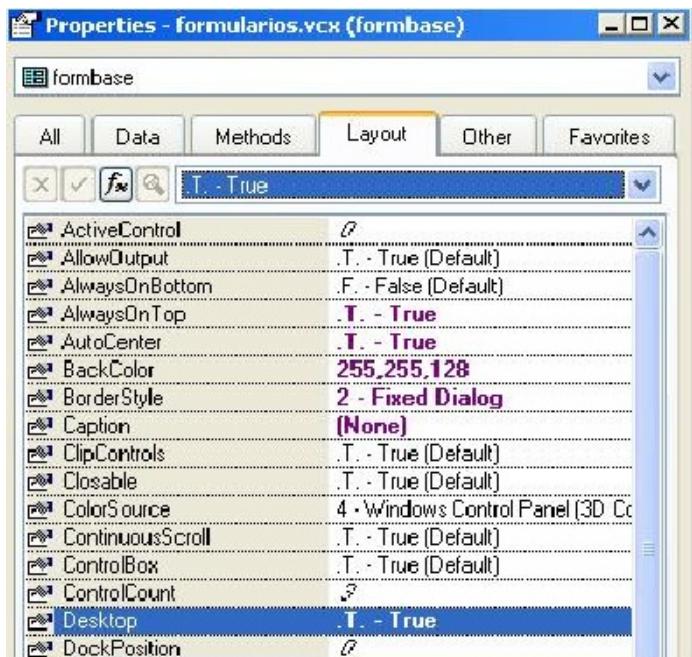
Creamos una **clase** llamada FormBase la cual fue derivada de otra **clase**, llamada Form y fue grabada en un archivo llamado FORMULARIOS.VCX.

Como FORMULARIOS.VCX no existía, también fue creado. Si ya hubiera existido entonces la **clase** FormBase se hubiera agregado a él.

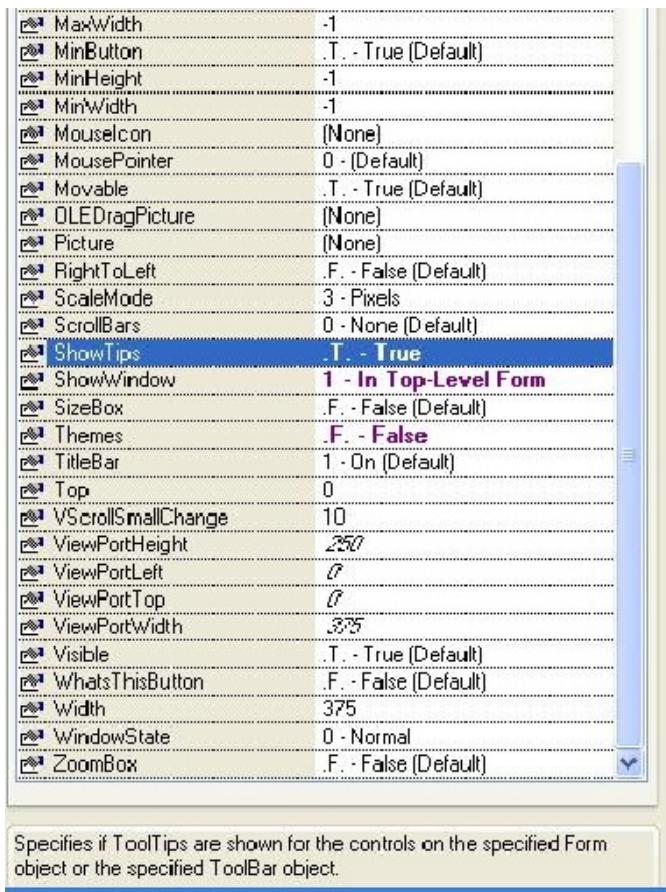
En otras palabras: creamos una **clase** llamada FormBase, la cual fue grabada en un archivo llamado FORMULARIOS.VCX y esta **clase** FormBase es descendiente (o hija) de la **clase** Form.

Por lo tanto, nuestra clase FormBase ha *heredado* todas las propiedades y todos los métodos de su **clase padre**, o sea de la **clase** Form.

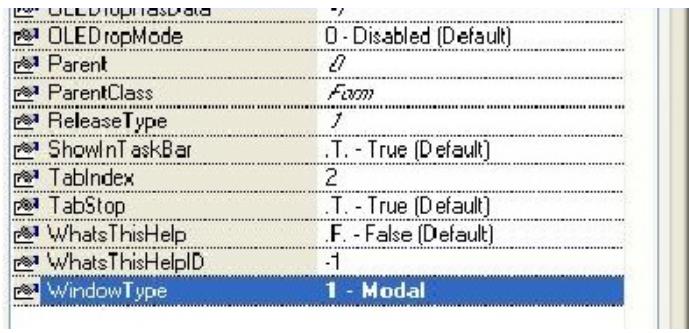
Ahora le cambiaremos algunas **propiedades** a nuestro formulario FormBase, como vemos en las siguientes pantallas.



Pantalla N° 2



Pantalla N° 3



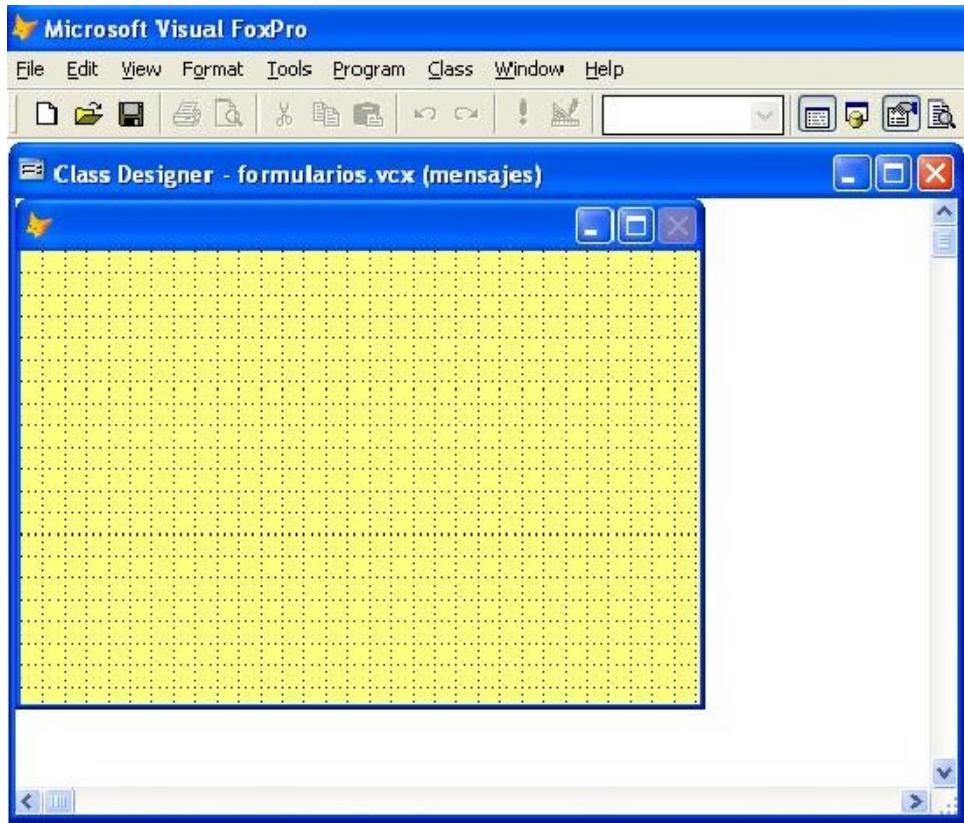
Pantalla N° 4

Grabamos nuestro formulario (presionando CTRL + W) o con File + Save o haciendo click en el icono del diskette y regresamos a la ventana de comandos del VFP.

Ahora, crearemos un nuevo formulario, pero esta vez estará basado en FormBase el cual acabamos de crear. ¿Cómo hacemos eso? Escribiendo:

```
CREATE CLASS MENSAJES OF FORMULARIOS AS FORMBASE FROM FORMULARIOS
```

Y el Visual FoxPro nos mostrará la siguiente pantalla:



Pantalla N° 5

¿Qué le hemos pedido que hiciera?

Que creara una nueva **clase**, basada en FormBase, la cual proviene de FORMULARIOS.VCX y la nueva **clase** que hemos creado, también queremos que sea grabada dentro de FORMULARIOS.VCX.

A veces queremos crear una **clase** derivada de otra **clase**, la cual se encuentra en una biblioteca de clases. Y a esta nueva **clase** que estamos creando queremos grabarla en otra biblioteca de clases.

O sea, la **clase padre** estará grabada en una biblioteca de clases y la **clase hija** estará grabada en otra biblioteca de clases.

Ya veremos algunos ejemplos más adelante.

Entonces, ¿qué tenemos hasta aquí?

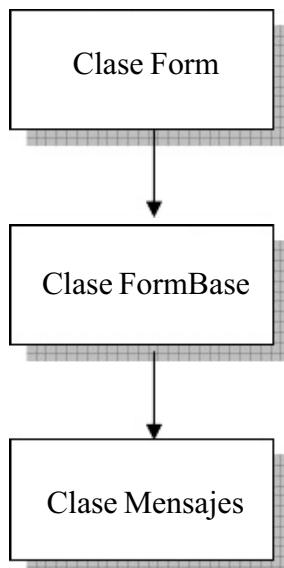


Diagrama N° 3

La **clase FormBase** es derivada (o hija) de la **clase Form**. Eso significa que todas las **propiedades** y todos los **métodos** de la **clase Form** están a su disposición. A su vez, la **clase Mensajes** es derivada (o hija) de la **clase FormBase**. Eso significa que todas las **propiedades** y todos los **métodos** de FormBase están a disposición de Mensajes y también todas las **propiedades** y todos los **métodos** de Form.

Si miramos los valores de las **propiedades** veremos que Mensajes ha *heredado* los de sus antecesores. O sea, los valores son los mismos, son idénticos.

Ahora, les haremos algunos cambios a la **clase Mensajes**.

1. Le asignamos el valor 540 a la propiedad Width
2. Le asignamos el valor 90 a la propiedad Height
3. Le agregamos un control *Label*, el cual tendrá:
 - a. BackStyle = 0 (transparente)
 - b. Left = 2
 - c. Top = 25
 - d. Width = 40

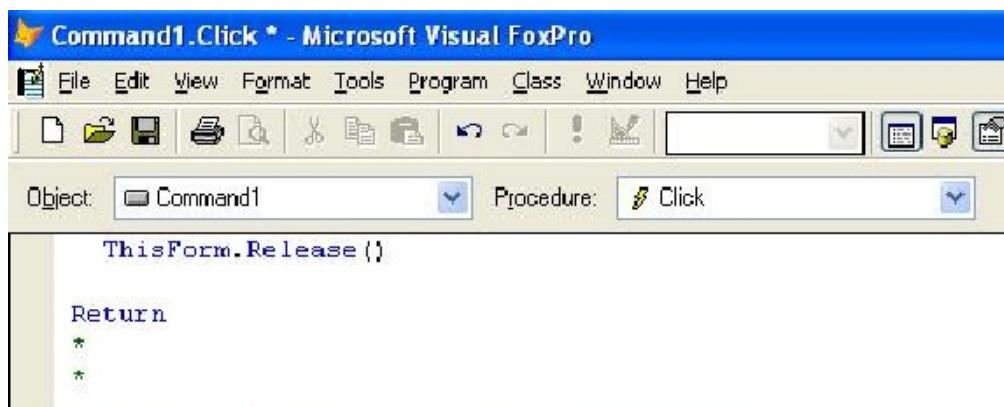
4. Le agregamos un botón de comandos, el cual tendrá:

- a. Caption = "OK"
- b. Height = 27
- c. Left = 228
- d. Top = 60
- e. Width = 84

Hacemos doble click sobre el botón de comandos y escribimos:

```
ThisForm.Release()
```

como se ve en la siguiente pantalla:



The screenshot shows the Microsoft Visual FoxPro IDE. The title bar says "Command1.Click * - Microsoft Visual FoxPro". The menu bar includes File, Edit, View, Format, Tools, Program, Class, Window, Help. The toolbar has various icons. The status bar shows "Object: Command1" and "Procedure: Click". The code window contains the following:

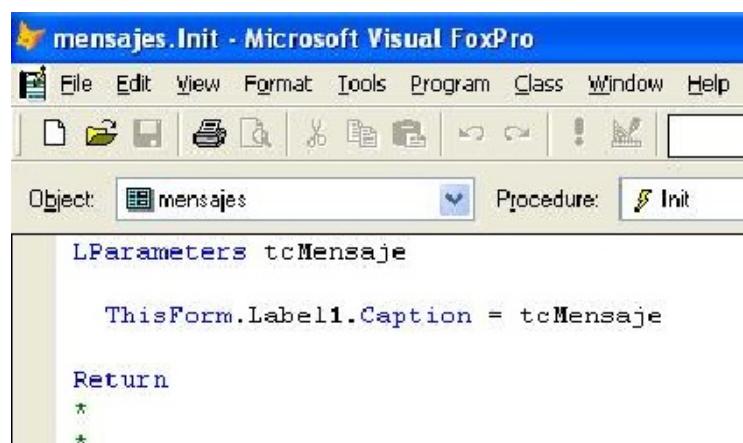
```
ThisForm.Release()  
  
Return  
*  
*
```

Pantalla N° 6

Lo que le estamos diciendo al Visual FoxPro es: "cuando el usuario haga click sobre este botón de comando, se debe ejecutar el **método** Release() de este formulario."

¿Y qué hace el **método** Release()? Libera el formulario de la memoria y al hacer eso se cierra el formulario.

Ahora, en el **método** INIT del formulario escribimos:



The screenshot shows the Microsoft Visual FoxPro IDE. The title bar says "mensajes.Init - Microsoft Visual FoxPro". The menu bar includes File, Edit, View, Format, Tools, Program, Class, Window, Help. The toolbar has various icons. The status bar shows "Object: mensajes" and "Procedure: Init". The code window contains the following:

```
LParameters tcMensaje  
  
ThisForm.Label1.Caption = tcMensaje  
  
Return  
*  
*
```

Pantalla N° 7

¿Qué estamos haciendo aquí?

Diciéndole que el formulario debe recibir un parámetro, al cual llamamos `tcMensaje` (“`t`” significa: “parámetro”, y “`c`” significa: “de tipo carácter”). Esta forma de nombrar a los parámetros es opcional, pero conveniente por legibilidad.

A continuación asignamos el parámetro recibido a la **propiedad *Caption*** del control *Label1* el cual está contenido en una **clase contenedora** derivada de Form.

¿Cómo se ejecuta la clase Mensaje que acabamos de crear?

Primero, hay que decirle en cual biblioteca de clases se encuentra la clase, eso lo hacemos escribiendo:

```
SET CLASSLIB TO FORMULARIOS
```

Seguidamente, creamos un **objeto** de la **clase Mensaje**.

```
oMiFormMensaje = CreateObject("MENSAJES", "Este es mi primer mensaje")
```

Ya se ha creado el **objeto** `oMiFormMensaje` (o sea, ya está *instanciado*) pero aún no lo vemos. ¿Por qué no? Porque no le hemos dicho al Visual FoxPro que queremos verlo. Para hacerlo escribimos:

```
oMiFormMensaje.Show()
```

Y entonces ¡¡¡voilá!!! Aparece el formulario, como podemos ver en la siguiente pantalla.



Pantalla N° 8

Resumiendo, lo que hemos escrito hasta aquí fue:

```
SET CLASSLIB TO FORMULARIOS  
oMiFormMensaje = CREATEOBJECT("MENSAJES", "Este es mi primer mensaje")  
oMiFormMensaje.Show()
```

Con la instrucción SET CLASSLIB TO FORMULARIOS le estamos diciendo al Visual FoxPro que abra la biblioteca de clases llamada FORMULARIOS para que podamos *instanciar* sus **clases** (en otras palabras: para que podamos crear **objetos** de las **clases** que se encuentran dentro del archivo FORMULARIOS.VCX).

En la siguiente línea estamos creando un **objeto** llamado `oMiFormMensaje`, el cual es de la **clase MENSAJES**. Además, le estamos enviando un parámetro, que corresponde al texto que deseamos ver en el formulario.

En la tercera línea le decimos que muestre el formulario porque aunque en la segunda línea lo habíamos creado todavía no era visible.

Ahora, creamos otro **objeto** también basado en la **clase MENSAJE**.

```
oMiOtroForm = CREATEOBJECT("MENSAJES", "Este es otro mensaje")
oMiOtroForm.Show()
```



Pantalla N° 9

Aquí no fue necesario escribir SET CLASSLIB TO FORMULARIOS, ¿por qué no fue necesario? Porque ya lo habíamos escrito con anterioridad y la biblioteca continuaba abierta.

Si deseamos cerrar una biblioteca de clases lo hacemos escribiendo:

SET CLASSLIB TO

sin ningún parámetro.

PROTEGIENDO Y OCULTANDO MIEMBROS DE LAS CLASES

Los miembros de las clases son las **propiedades** y los **métodos**. Y tienen tres visibilidades posibles:

- Pública
- Protegida
- Oculta

Visibilidad pública significa que la **propiedad** o el **método** pueden ser accedidos desde cualquier programa o clase.

Visibilidad protegida significa que la **propiedad** o el **método** pueden ser accedidos desde la clase y desde cualquiera de sus subclases.

Visibilidad oculta significa que la **propiedad** o el **método** pueden ser accedidos solamente desde la clase.

EJEMPLO05.PRG

```
Local loNuevoEmpleado, loNuevoGerente

CLEAR

loNuevoEmpleado = CreateObject ("EMPLEADO")

with loNuevoEmpleado
    .cNombre      = "Raquel"
    .cApellido     = "Martínez"
    .dFechaNacimiento = CtoD("12/02/1975")
    .dFechaIngreso   = CtoD("15/09/1994")
    .MostrarFechaIngreso()
    .MostrarFechaEgreso()
    .MostrarSueldo()
    .dFechaEgreso = CtoD("15/09/2010")      && Esto es un ERROR, no
funcionará porque dFechaEgreso es una propiedad PROTEGIDA
    .nSueldo       = 1234567                 && Esto es un ERROR, no
funcionará porque nSueldo es una propiedad OCULTA
    .PonerFechaEgreso(CtoD("15/10/2010")) && Esta es la forma
correcta de asignarle un valor a la propiedad dFechaEgreso
    .PonerSueldo(1234567)                  && Esta es la forma
correcta de asignarle un valor a la propiedad nSueldo
?
    .MostrarFechaIngreso()
    .MostrarFechaEgreso()
    .MostrarSueldo()
endwith
```

Walter R. Ojeda Valiente

```
loNuevoGerente = CreateObject("GERENTE")

with loNuevoGerente
    .cNombre      = "Mercedes"
    .cApellido     = "Candia"
    .dFechaIngreso = CtoD("21/03/2009")
    .PonerSueldoGerente(5678901)      && Esto es un ERROR
endwith

Release loNuevoEmpleado, loNuevoGerente

Return
*
*

DEFINE CLASS PERSONA AS CUSTOM

cNombre          = Space(20)
cApellido         = Space(20)
dFechaNacimiento = {}

PROCEDURE MostrarDatos
    ? This.cNombre, This.dFechaNacimiento
RETURN

ENDDEFINE
*
*

DEFINE CLASS EMPLEADO AS PERSONA
    PROTECTED dFechaEgreso      && dFechaEgreso es una propiedad
PROTEGIDA, por lo tanto es visible en esta clase y en todas sus
subclases.
    HIDDEN nSueldo           && nSueldo es una propiedad OCULTA, por
lo tanto solamente es visible en esta clase.

    dFechaIngreso = {}
    dFechaEgreso  = {}
    nSueldo       = 0
    cNombreJefe   = ""

PROCEDURE MostrarFechaIngreso
    with This
        ? .cApellido + ", " + .cNombre + " Ingresó el día: " +
DtoC(.dFechaIngreso)
    endwith
ENDPROC

PROCEDURE MostrarFechaEgreso
    with This
        ? .cApellido + ", " + .cNombre + " Egresó el día: " +
DtoC(.dFechaEgreso)
    endwith
ENDPROC

PROCEDURE MostrarSueldo
    with This
        ? .cApellido + ", " + .cNombre + " Sueldo = " +
Transform(.nSueldo, "999,999,999")
    endwith
ENDPROC
```

```
PROCEDURE PonerFechaEgreso
  LPARAMETERS tdFecha
  This.dFechaEgreso = tdFecha
ENDPROC

PROCEDURE PonerSueldo
  LPARAMETERS tnSueldo
  This.nSueldo = tnSueldo
ENDPROC

ENDDEFINE
*
*

DEFINE CLASS GERENTE AS EMPLEADO

PROCEDURE MostrarNombreGerente
  ? This.cNombre
ENDPROC

PROCEDURE MostrarFechaEgresoGerente
  ? This.dFechaEgreso
ENDPROC

PROCEDURE MostrarSueldoGerente
  ? This.nSueldo      && Esto es un ERROR
ENDPROC

PROCEDURE PonerSueldoGerente
  LPARAMETERS tnSueldo
  This.nSueldo = tnSueldo      && Esto es un ERROR, nSueldo es una
propiedad OCULTA, por lo tanto no se puede acceder a ella desde esta
subclase
ENDPROC

ENDDEFINE
*
*
```

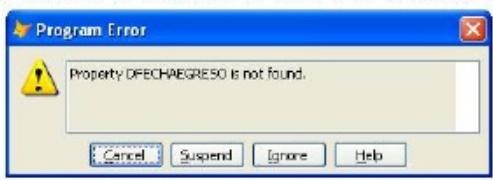
Analicemos este programa, el cual tiene varios errores escritos a propósito para mostrar lo que no debe hacerse.

Primero, creamos un **objeto** llamado `loNuevoEmpleado` y le asignamos algunos valores a sus propiedades. Pero cuando la ejecución del programa llega a esta línea:

```
.dFechaEgreso = CtoD("15/09/2010")
```

el Visual FoxPro se enoja y nos muestra este mensaje:

```
.MostrarSueldo()
.dFechaEgreso = CtoD("15/09/2010")    && Esto es un ERROR, no funcionará porque dFechaEgreso es una propiedad
.nSueldo = 1234567                      && Esto es un ERROR, no funcionará porque nSueldo es una propiedad OK
.PonerFechaEgreso(CtoD("15/10/2010"))   && Esta es la forma correcta de asignarle un valor a la propiedad dFe
.PonerSueldo(1234567)                  && Esta es la forma correcta de asignarle un valor a la propiedad nSu
?
.MostrarFechaIngreso()
.MostrarFechaEgreso()
.MostrarSueldo()
endwith
```



```
loNuevoGerente = CreateObject("GERENTE")

with loNuevoGerente
  .cNombre      = "Mercedes"
  .cApellido    = "Candia"
```

Pantalla N° 10

¿Por qué se enojó? Porque dFechaEgreso es una propiedad que está protegida dentro de la clase. Eso significa que puede accederse a ella solamente dentro de la **clase** o de una de sus **subclases**.

Hacemos click en el botón “Ignore” y ... el Visual FoxPro vuelve a enojarse, esta vez en la siguiente línea, como vemos en la siguiente pantalla:

```
.dFechaEgreso = CtoD("15/09/2010")    // Esto es un ERROR, no funcionará porque dFechaEgreso es una propiedad
.nSueldo      = 1234567                 // Esto es un ERROR, no funcionará porque nSueldo es una propiedad
.PonerFechaEgreso(CtoD("15/10/2010"))   // Esta es la forma correcta de asignarle un valor a la propiedad dFechaEgreso
.PonerSueldo(1234567)                  // Esta es la forma correcta de asignarle un valor a la propiedad nSueldo
?
.MostrarFechaIngreso()
.MostrarFechaEgreso()
.MostrarSueldo()
endwith

LoNuevoGerente = CreateObject("GERENTE")

with LoNuevoGerente
.cNombre      = "Mercedes"
.cApellido     = "Candia"
.dFechaIngreso = CtoD("21/03/2009")
```



Pantalla N° 11

¿Y ahora por qué se enojó? Porque la propiedad nSueldo está oculta dentro de la clase. Eso significa que solamente se puede acceder a ella desde su propia clase y desde ningún otro lado, ni siquiera puede accederse a ella desde una subclase.

Y entonces, ¿cómo le asignamos un valor a una propiedad que está protegida o que está oculta? Lo hacemos mediante un procedimiento o una función, como se ve en la siguiente línea:

```
.PonerFechaEgreso( CtoD("15/10/2010") )
```

Esta es la forma. Ejecutamos un procedimiento llamado “PonerFechaEgreso()” el cual recibe como parámetro la fecha del egreso.

Para poner o cambiar el sueldo de un empleado usamos el procedimiento PonerSueldo(), el cual recibe como parámetro el sueldo del empleado, tal como vemos en la siguiente línea:

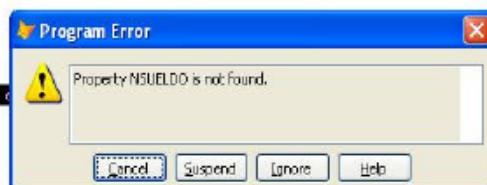
```
.PonerSueldo(1234567)
```

Eso sí acepta el Visual FoxPro, así que el programa continúa su ejecución hasta que ... otra vez se enoja ¿y ahora por qué?

```
PROCEDURE MostrarSueldoGerente
? This.nSueldo
ENDPROC

PROCEDURE PonerSueldoGerente
!Parameters tnSueldo
This.nSueldo = tnSueldo    // Esto es un ERROR, nSueldo es una propiedad
ENDPROC

ENDDEFINE
+
```



Pantalla 12

Ahora se enojó porque en la **clase GERENTE** se intentó cambiar el valor de la propiedad nSueldo. Pero la propiedad nSueldo pertenece a la **clase EMPLEADO** y está oculta. Por lo tanto, solamente es visible para la **clase EMPLEADO** y para nadie más. Ni siquiera es visible para las **subclases** de la **clase EMPLEADO**. Por lo tanto no es visible para la **clase GERENTE**.

Resumiendo:

Si una propiedad es **pública** entonces se le puede cambiar su valor u obtener su valor desde cualquier programa o clase. La forma de hacerlo es escribiendo:

Objeto.propiedad

Si una propiedad es **protegida** entonces se le puede cambiar su valor u obtener su valor solamente desde la clase que la creó o desde una de sus subclases. Para asignarle un valor o para obtener su valor se necesita escribir un procedimiento o una función que realice esa tarea. La forma de hacerlo es escribiendo:

MiMétodo(parámetro 1, parámetro 2, ..., parámetro n)

Si una propiedad es **oculta** entonces se le puede cambiar su valor u obtener su valor solamente desde la clase que la creó. Para asignarle un valor o para obtener su valor se necesita escribir un procedimiento o una función que realice esa tarea. La forma de hacerlo es escribiendo:

MiMétodo(parámetro 1, parámetro 2, ..., parámetro n)

Con los métodos (procedimientos y funciones) se aplica la mismísima idea.

MODELANDO OBJETOS

Se le llama **modelar objetos** al hecho de definir las propiedades y los métodos (funciones y procedimientos) que tendrá una **clase** (y por consiguiente, todos los **objetos** pertenecientes a esa **clase**) para representar la realidad (o una parte de la realidad).

Por ejemplo, necesitamos definir una **clase** que represente a los clientes de la Empresa que nos contrató.

¿Cuáles propiedades tendrá esa **clase**? ¿Y cuáles métodos?

Después de pensar un rato en el asunto y basándonos en lo que necesitaremos para nuestra aplicación podríamos decidirnos por lo siguiente:

Propiedades

cCódigo	Carácter	5
cNombre	Carácter	20
cApellido	Carácter	20
cDirección	Carácter	50
cTeléfono/s	Carácter	30
cCelular/es (móvil)	Carácter	30
cLocalidad	Carácter	30
cPaís	Carácter	25
cEmail	Carácter	30
cNº del documento de identidad	Carácter	15
nSexo	Numérico	1
nEstado Civil	Numérico	1
dFecha de Nacimiento	Fecha	8

Métodos

MostrarDatos()	&& Código, Nombre, Dirección, Teléfono, E-mail
NombreYApellido()	&& Muestra el Nombre y después el Apellido
ApellidoYNOMBRE()	&& Muestra el Apellido, una coma, y el Nombre

Por lo tanto, al hecho de definir todas las propiedades y todos los métodos que tendrá nuestra **clase CLIENTE** lo llamamos *modelar el objeto CLIENTE*.

CREANDO FORMULARIOS POR PROGRAMA

Normalmente cuando queremos crear un formulario lo hacemos con los comandos CREATE FORM o MODIFY FORM. Sin embargo, también podemos crear una **clase** basada en formulario ... desde un .PRG

EJEMPLO06.PRG

```
Local loMiForm

*--- Primero, creamos una instancia de FORM y la llamamos loMiForm

loMiForm = CreateObject ("FORM")

with loMiForm
    *--- Segundo, asignamos valores a las propiedades
    .BackColor = RGB(64, 255, 255)
    .Caption   = "Mi primer formulario usando clases"
    .Height    = 600
    .Width     = 800
    *--- Tercero, le agregamos botones de comando
    .AddObject ("BotonOK" , "MiBoton1")
    .AddObject ("BotonSALIR", "MiBoton2")
    *--- Cuarto, hacemos a los botones visibles
    .BotonOK.Visible = .T.
    .BotonSALIR.Visible = .T.
    *--- Quinto, mostramos el formulario y los botones
    .Show()
endwith

*--- Sexto, leemos los eventos

READ EVENTS

*--- Séptimo, liberamos al objeto de la memoria

Release loMiForm

Return
*
*

DEFINE CLASS MiBoton1 AS COMMANDBUTTON

    Caption = "OK"
    Height  = 30
    Left    = 50
    Top     = 570
    Width   = 100

PROCEDURE Click
    WAIT WINDOW "Has presionado el botón OK"
ENDPROC

ENDDEFINE
*
```

```
*  
DEFINE CLASS MiBoton2 AS COMMANDBUTTON  
  
    Caption = "SALIR"  
    Height = 30  
    Left = 650  
    Top = 570  
    Width = 100  
  
    PROCEDURE Click  
        CLEAR EVENTS  
    ENDPROC  
  
ENDDEFINE  
*  
*
```

¿Qué fue lo que hicimos?

1. Crear una variable llamada `loMiForm`, la cual es una instancia de la **clase FORM**. Eso significa que tiene todas sus propiedades y todos sus métodos.
2. Asignar valores a algunas de las propiedades de ese **objeto**.
3. Agregar dos botones de comando, uno llamado `BotonOK` y el otro llamado `BotonSALIR`
4. Ponerle el valor verdadero (.T.) a las propiedades “Visible” de ambos botones
5. Mostrar el formulario (con sus botones, claro)
6. Leer los eventos
7. Liberar al objeto de la memoria
8. Crear una subclase, derivada de `CommandButton`. Eso significa que posee todas las propiedades y todos los métodos de la **clase CommandButton** que son **públicos** o **protegidos**.
 - a. Le asignamos valores a algunas propiedades
 - b. Escribimos el código que se ejecutará cuando el usuario haga click con el botón izquierdo del mouse sobre el botón. En este caso mostrará un mensaje.
9. Crear otra subclase, también derivada de `CommandButton`.
 - a. Le asignamos valores a algunas propiedades
 - b. Escribimos el código que se ejecutará cuando el usuario haga click con el botón izquierdo del mouse sobre el botón. En este caso limpiará los eventos para detener el procesamiento de los mismos.

COMANDOS, FUNCIONES Y MÉTODOS USADOS EN LA PROGRAMACIÓN ORIENTADA A LOS OBJETOS

ACLASS()

Guarda el nombre de una clase y de todos sus antecesores dentro de un array

ADD CLASS

Agrega una definición de clase a una biblioteca de clases .VCX

ADDOBJECT()

Agrega un objeto a un contenedor de objetos en tiempo de ejecución.

AEVENTS()

Recupera el número de eventos enlazados

AINSTANCE()

Guarda las instancias de una clase en un array y devuelve la cantidad de instancias que hay en el array

AMEMBERS()

Guarda los nombres de las propiedades, los procedimientos, y los miembros de un objeto dentro de un array

AMOUSEOBJ()

Crea un array que contiene información sobre la posición del puntero del mouse y el objeto sobre el cual está posicionado el puntero.

ASELOBJ()

Localiza una referencia de objeto para el control seleccionado dentro de un array

AVCXCLASSES()

Guarda la información de las clases de una biblioteca de clases dentro de un array

COMPOBJ()

Compara las propiedades de dos objetos y devuelve .T. si las propiedades y los valores de ellas son idénticos

CREATE CLASS

Abre el Diseñador de Clases, para que se pueda crear una nueva definición de clase

CREATE CLASSLIB

Crea una nueva (y vacía) biblioteca de clases

CREATEOBJECT()

Crea un objeto derivado de una definición de clase

DEFINE CLASS

Crea una clase o una subclase definida por el programador y especifica sus propiedades y métodos.

DISPLAY OBJECTS

Muestra información sobre un objeto o sobre un conjunto de objetos

DODEFAULT()

Ejecuta el método de la clase padre que tiene el mismo nombre que el método donde se escribió esta función

GETOBJECT()

Activa un objeto de automatización y crea una referencia a ese objeto

GETPEM

Devuelve el valor actual de una propiedad o código de programa para un método o un evento, en tiempo de diseño

LIST OBJECTS

Continuamente muestra información sobre un objeto o conjunto de objetos

NEWOBJECT()

Crea un nuevo objeto o clase desde una biblioteca de clases (.VCX) o desde un programa (.PRG) pero sin abrir el archivo.

NODEFAULT

Impide que se ejecute el método de la clase padre que tiene el mismo nombre

MODIFY CLASS

Abre el Diseñador de Clases, permitiendo modificar la definición de una clase existente o crear una nueva definición de clase

OBJTOCLIENT()

Devuelve la posición o dimensión de un objeto con relación a su formulario

PEMSTATUS()

Recupera un atributo para una propiedad, evento, método u objeto

REMOVEOBJECT

Borra un objeto de un objeto contenedor en tiempo de ejecución

RELEASE CLASSLIB

Cierra una biblioteca de clases

RENAME CLASS

Le cambia el nombre a una definición de clase que está contenida dentro de una biblioteca de clases

SET CLASSLIB

Abre una biblioteca de clases visuales, la cual contiene definiciones de clases

SYS(1269)

Devuelve un valor que indica si una propiedad ha cambiado su valor por defecto o si la propiedad es de solo lectura (read-only)

SYS(1270)

Devuelve una referencia para el objeto que se encuentra debajo del Mouse o en cualquier otra posición especificada.

SYS(1271)

Devuelve el nombre del archivo .SCX en el cual la instancia del objeto especificado está guardada

SYS(1272)

Devuelve la jerarquía de objeto para el objeto especificado.

THIS

Provee una referencia al objeto actual en un evento o en una definición de clase

WITH ... ENDWITH

Especifica múltiples propiedades para un objeto

COMO ACCEDER A UNA PROPIEDAD DE LA CLASE PADRE

Si necesitamos conocer el valor de una propiedad de la **clase padre** o si queremos cambiar ese valor, podemos hacerlo usando la propiedad PARENT, como en el siguiente ejemplo.

- Creamos un nuevo formulario
- Le agregamos un CommandButton
- En el método click del botón escribimos:
 This.Parent.Caption = "Este es el nuevo caption del formulario"
- Ejecutamos el formulario
- Hacemos click sobre el botón de comandos

EJEMPLO07.PRG

```
Local loForm

loForm = CreateObject ("FORM")

with loForm
    .Caption = "Este es el caption original del formulario"
    .AddObject ("Boton1", "BotonCAMBIA")
    .AddObject ("Boton2", "BotonSALIR")
    .Boton1.Visible = .T.
    .Boton2.Visible = .T.
    .Show()
endwith

READ EVENTS

Release loForm

Return
*
*

DEFINE CLASS BotonCAMBIA AS COMMANDBUTTON
    Caption = "Cambiar título"
    Left     = 20
    PROCEDURE Click
        This.Parent.Caption = "Este es el nuevo caption del formulario"
    ENDPROC
ENDDEFINE
*
*

DEFINE CLASS BotonSALIR AS COMMANDBUTTON
    Caption = "SALIR"
    Left     = 200
    PROCEDURE Click
        CLEAR EVENTS
    ENDPROC
ENDDEFINE
*
```

¿Qué hicimos?

Primero, creamos un **objeto** de tipo formulario (o sea, FORM) al cual llamamos loForm (ya vimos que la “l” inicial significa “local” y la “o” que le sigue significa “objeto”, o sea que se trata de un “objeto local”).

Segundo, cambiamos el caption del formulario

Tercero, le agregamos dos botones llamados Boton1 y Boton2. La **clase** del primero es BotonCAMBIA y la **clase** del segundo es BotonSALIR.

Cuarto, hicimos que ambos botones se tornen visibles.

Quinto, mostramos el formulario.

Sexto, procesamos los eventos.

Séptimo, liberamos al **objeto** loForm de la memoria, para que desaparezca totalmente.

En el procedimiento click de la **clase** BotonCAMBIA cambiamos el valor de una propiedad de la **clase padre** de BotonCAMBIA, más específicamente de la propiedad “Caption”.

Eso significa que estando en un botón de comando (BotonCAMBIA) pudimos cambiar el valor de una propiedad del formulario (el cual es su **clase padre**).

Esta funcionalidad es progresiva, o sea que es posible acceder a la propiedad del padre, del padre, del padre, etc. Algo como.

This.Parent.Parent.Parent.Caption = “Este es el nuevo caption”

This.Parent.Parent.Width = 200

This.Parent.Parent.Parent.Left = 150

COMO EJECUTAR UN MÉTODO DE LA CLASE PADRE

A veces cuando estamos dentro de un método necesitamos ejecutar un método de su **clase padre**, la forma de hacerlo es con el operador de resolución de alcance. La sintaxis es la siguiente:

ClasePadre::Método

El siguiente ejemplo es muy similar al EJEMPLO05.PRG, pero ahora utiliza al operador de resolución de alcance para ejecutar la función MostrarFechaIngreso() que pertenece a su **clase padre** (es decir, pertenece a la **clase EMPLEADO**).

EJEMPLO08.PRG

...

```
DEFINE CLASS GERENTE AS EMPLEADO

PROCEDURE MostrarNombreGerente
? This.cNombre
ENDPROC

PROCEDURE MostrarFechaIngresoGerente
EMPLEADO::MostrarFechaIngreso()      && Aquí se utilizó el operador
de resolución de alcance para ejecutar un método de la clase padre.
ENDPROC

ENDDEFINE
*
*
```

El siguiente ejemplo es muy similar al anterior, pero quiere asignarle el sueldo al Gerente desde su propio procedimiento, ese es un error que cometemos a propósito, para que se vea más claramente lo que NO DEBE HACERSE.

EJEMPLO09.PRG

```
Local loNuevoEmpleado, loNuevoGerente

CLEAR

loNuevoEmpleado = CreateObject ("EMPLEADO")

with loNuevoEmpleado
.cNombre      = "Raquel"
.cApellido     = "Martínez"
.dFechaNacimiento = CtoD("12/02/1975")
.dFechaIngreso   = CtoD("15/09/1994")
.MostrarFechaIngreso()
.MostrarFechaEgreso()
.MostrarSueldo()
```

Walter R. Ojeda Valiente

```
.PonerFechaEgreso( CtoD("15/10/2010")) && Esta es la forma
correcta de asignarle un valor a la propiedad dFechaEgreso
.PonerSueldo(1234567) && Esta es la forma
correcta de asignarle un valor a la propiedad nSueldo
?
.MostrarFechaIngreso()
.MostrarFechaEgreso()
.MostrarSueldo()
?
endwith

?

loNuevoGerente = CreateObject("GERENTE")

with loNuevoGerente
    .cNombre      = "Mercedes"
    .cApellido     = "Candia"
    .dFechaIngreso = CtoD("21/03/2009")
    MostrarNombreGerente()
    MostrarFechaIngresoGerente()
    PonerSueldoGerente(9876543) && Esto es un ERROR.
endwith

Release loNuevoEmpleado, loNuevoGerente

Return
*
*

DEFINE CLASS PERSONA AS CUSTOM

cNombre          = Space(20)
cApellido        = Space(20)
dFechaNacimiento = {}

PROCEDURE MostrarDatos
    ? This.cNombre, This.dFechaNacimiento
RETURN

ENDDEFINE
*
*

DEFINE CLASS EMPLEADO AS PERSONA
    PROTECTED dFechaEgreso && dFechaEgreso es una propiedad
PROTEGIDA, por lo tanto es visible en esta clase y en todas sus
subclases.
    HIDDEN nSueldo && nSueldo es una propiedad OCULTA, por
lo tanto solamente es visible en esta clase.

dFechaIngreso = {}
dFechaEgreso  = {}
nSueldo       = 0
cNombreJefe   = ""

PROCEDURE MostrarFechaIngreso
    with This
        ? .cApellido + ", " + .cNombre + " Ingresó el día: " +
DtoC(.dFechaIngreso)
    endwith
ENDPROC
```

```
PROCEDURE MostrarFechaEgreso
  with This
    ? .cApellido + ", " + .cNombre + " Egresó el día: " +
  DtoC(.dFechaEgreso)
  endwith
ENDPROC

PROCEDURE MostrarSueldo
  with This
    ? .cApellido + ", " + .cNombre + " Sueldo = " +
  Transform(.nSueldo, "999,999,999")
  endwith
ENDPROC

PROCEDURE PonerFechaEgreso
LParameters tdFecha
  This.dFechaEgreso = tdFecha
ENDPROC

PROCEDURE PonerSueldo
LParameters tnSueldo
=MESSAGEBOX (TRANSFORM(TNSUELDO))
  This.nSueldo = tnSueldo
ENDPROC

ENDDEFINE
*
*

DEFINE CLASS GERENTE AS EMPLEADO

PROCEDURE MostrarNombreGerente
  ? This.cNombre
ENDPROC

PROCEDURE MostrarFechaIngresoGerente
  EMPLEADO::MostrarFechaIngreso()      && Aquí se utilizó el operador
de resolución de alcance para ejecutar un método de la clase padre.
ENDPROC

PROCEDURE PonerSueldoGerente
LParameters tnSueldo
  EMPLEADO::PonerSueldo(tnSueldo)      && Esto es un ERROR, porque
tnSueldo es una propiedad oculta de la clase EMPLEADO
ENDPROC

ENDDEFINE
*
*
```

Como vemos, utilizar el procedimiento PonerSueldoGerente() de la **clase GERENTE** es un error, ¿por qué eso? Porque está llamando al procedimiento PonerSueldo() de la clase EMPLEADO, y ese procedimiento cambia el valor de la propiedad nSueldo. Pero resulta que nSueldo es una propiedad **oculta**, por lo tanto desde una subclase no se puede acceder a él, ni directamente ni indirectamente (como se intentó hacer aquí).

RECUÉRDALO: A una propiedad **oculta** solamente se puede acceder desde la clase que la definió, es imposible desde otro lado. Si necesitas acceder a ella desde otro lado entonces no la pongas como **oculta**.

UNA CLASE LABEL DESPLAZANTE

Ahora crearemos visualmente una **clase** que mostrará a un texto desplazándose de derecha a izquierda, para ello utilizaremos tres controles:

- Un container
- Una etiqueta
- Un timer

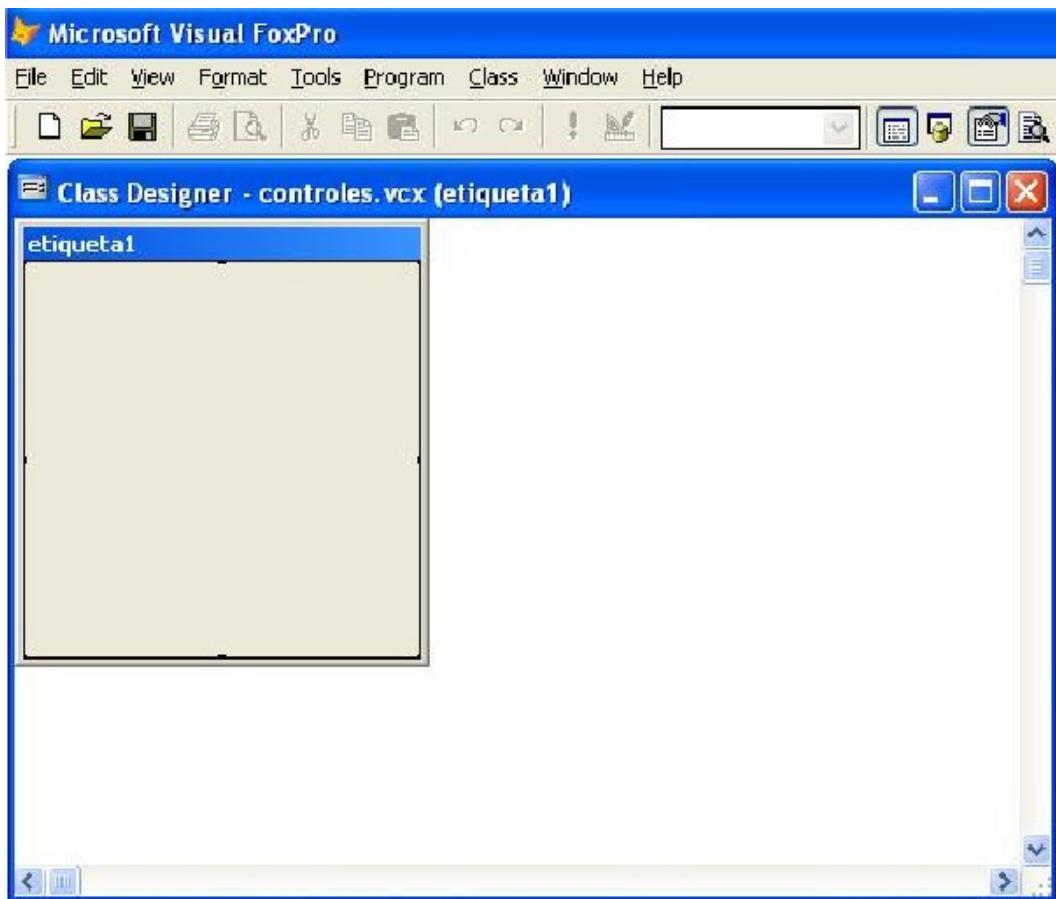
El container servirá para contener a los otros dos controles. La etiqueta mostrará el texto que deseamos, y el timer marcará la velocidad del desplazamiento.

En la ventana de comandos escribimos:

```
CREATE CLASS ETIQUETA1 OF CONTROLES AS CONTAINER
```

¿Qué le estamos diciendo aquí al Visual FoxPro? Que queremos crear una nueva **clase**, la cual se llamará ETIQUETA1, estará grabada en el archivo CONTROLES.VCX y su **clase base** será CONTAINER.

Nos mostrará la siguiente pantalla:



Pantalla N° 13

Como podemos ver en la pantalla, esta **clase** se llama ETIQUETA1 y el archivo se llama CONTROLES.VCX

Le cambiamos el tamaño y le agregamos un control *label* y un control *timer* para que quede así:



Pantalla N° 14

En la propiedad *Autosize* del control LABEL1 ponemos .F. y en la propiedad *Width* ponemos 720 (puedes poner el más adecuado a tu pantalla).

En la propiedad *Interval* del control TIMER1, ponemos: 100

Y en el método *Timer* del control TIMER1 escribimos:



Pantalla N° 15

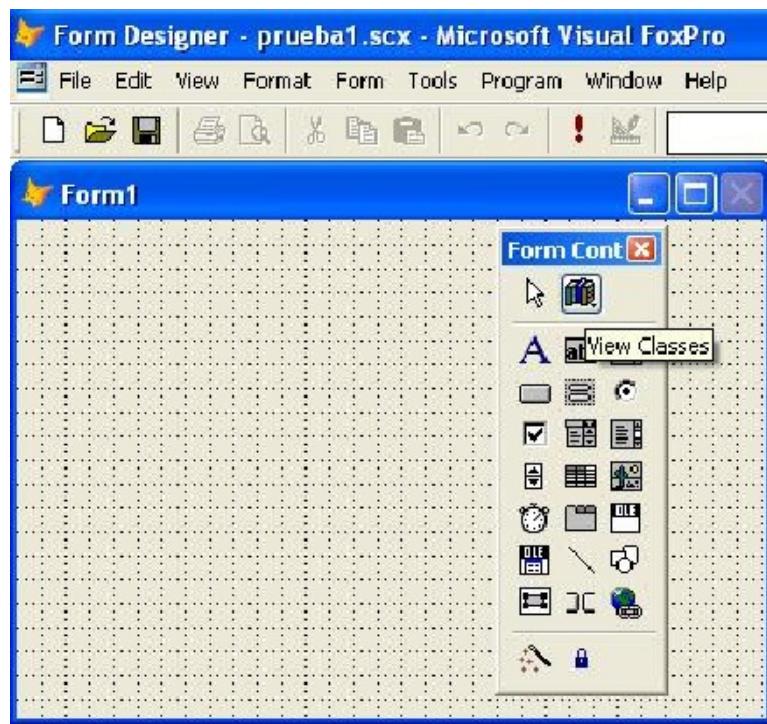
Grabamos y si nos fijamos en la carpeta veremos que ahora tenemos en ella un archivo llamado CONTROLES.VCX, es en ese archivo donde se encuentra la **clase** que acabamos de crear. Esta **clase** ahora puede ser utilizada en cualquier formulario.

Vamos a verificarlo.

En la ventana de comandos del Visual FoxPro escribimos:

Modify form PRUEBA1

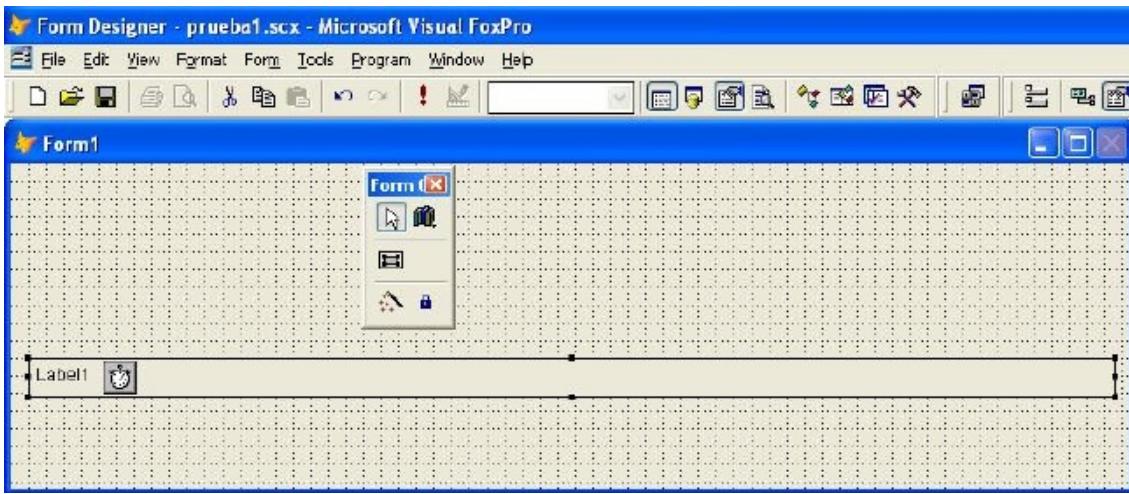
Nos mostrará un formulario vacío, hacemos click en el icono “View Classes”



Pantalla N° 16

Elegimos “Add...” y luego CONTROLES.VCX. Nos mostrará todas las **clases** que están dentro de CONTROLES.VCX, en este momento solamente hay una pero a medida que vayamos creando nuevas **clases** éstas irán agregándose.

Insertamos el control ETIQUETA1 en nuestro formulario y veremos una pantalla similar a la siguiente:



Pantalla N° 17

En el método INIT del formulario escribimos:



Pantalla N° 18

Ejecutamos el formulario y ... veremos el texto desplazándose de derecha a izquierda.

Si queremos que se mueva más rápido o más lento, es cuestión de cambiar el valor de la propiedad *Interval* del control TIMER1.

Por supuesto que también podemos cambiar el texto y escribir el que deseemos.

Como vemos, en muy poco tiempo hemos conseguido tener un control label desplazándose por la pantalla. Este es un ejemplo muy simple, iremos viendo algunos más complicados en las siguientes páginas.

El archivo CONTROLES.VCX, el formulario PRUEBA1.SCT y todos los demás ejemplos se encuentran en un archivo comprimido junto con este documento.

Es importante que los descomprimas también y los vayas observando para entender mejor. Y por supuesto, si tienes algunas dudas, puedes escribirme al e-mail que aparece en la última página.

UNA CLASE LABEL TITILANTE

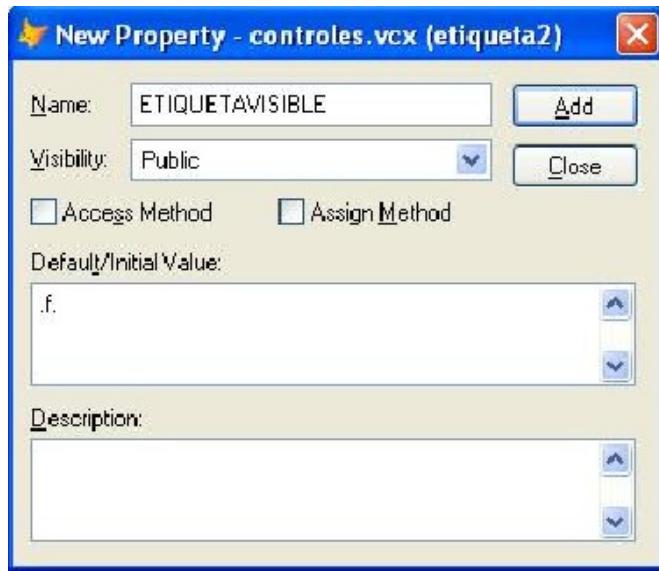
Ahora crearemos una **clase label** que titilará a intervalos regulares. La forma de hacerla es similar a la del ejemplo anterior. Utilizaremos tres controles:

- Un container
- Una etiqueta
- Un timer

En la ventana de comandos del Visual FoxPro escribimos:

```
CREATE CLASS ETIQUETA2 OF CONTROLES AS CONTAINER
```

Después de agregar el control *label* y el control *timer* creamos una nueva propiedad y la llamamos *EtiquetaVisible*, como se ve en la siguiente pantalla:



Pantalla N° 19

En la propiedad *Autosize* del control label ponemos *.F.*, en la propiedad *Width* ponemos 720 (o el que te parezca más adecuado).

En la propiedad *Interval* del control Timer1 ponemos: 500

En el método *Timer* del control Timer1 escribimos:

```
This.Parent.Label1.Visible = !This.Parent.Label1.Visible
```

Grabamos la clase y verificamos que funcione. Para eso creamos un nuevo formulario llamado PRUEBA2, le insertamos el control *Etiqueta2* y en el método INIT del formulario escribimos:

```
ThisForm.Etiqueta21.Label1.Caption = "Este texto está titilando"
```

Ejecutamos el formulario y ... ¡¡¡sí, el texto está titilando!!!

UNA CLASE LABEL SOMBREADA

Ahora crearemos una **clase label** sombreada, de esa manera parecerá que el texto está en tres dimensiones, suele ser un buen efecto, mejora mucho la estética de nuestras aplicaciones.

Utilizaremos tres controles:

- Un control container
- Dos controles label

En la ventana de comandos del Visual FoxPro escribimos:

```
CREATE CLASS ETIQUETA3 OF CONTROLES AS CONTAINER
```

En el container Etiqueta3:

- BackStyle = 0. Transparent
- BorderWidth = 0

En label1:

- BackStyle = 0. Transparent
- ForeColor = 0, 0, 128
- Left = 0
- Top = 0
- Width = 720 (o el que te parezca más adecuado)

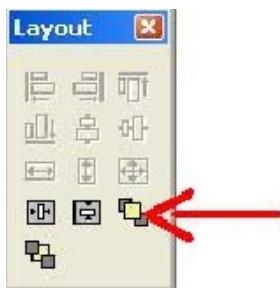
En label2:

- BackStyle = 0.Transparent
- ForeColor = 192, 192, 192
- Left = 2
- Top = 2
- Width = 720 (o el que te parezca más adecuado)

En el container Etiqueta3:

- Height = 17
- Width = 720

Para que se vea el efecto de sombra deseado, *label1* debe estar arriba de *label2*, para decirle cual de esas etiquetas queremos que esté arriba y cual queremos que esté abajo usamos la *Layout Toolbar*. Seleccionamos a *label1* y luego en la *Layout Toolbar* seleccionamos el icono que la coloca en la posición superior.



Pantalla N° 20

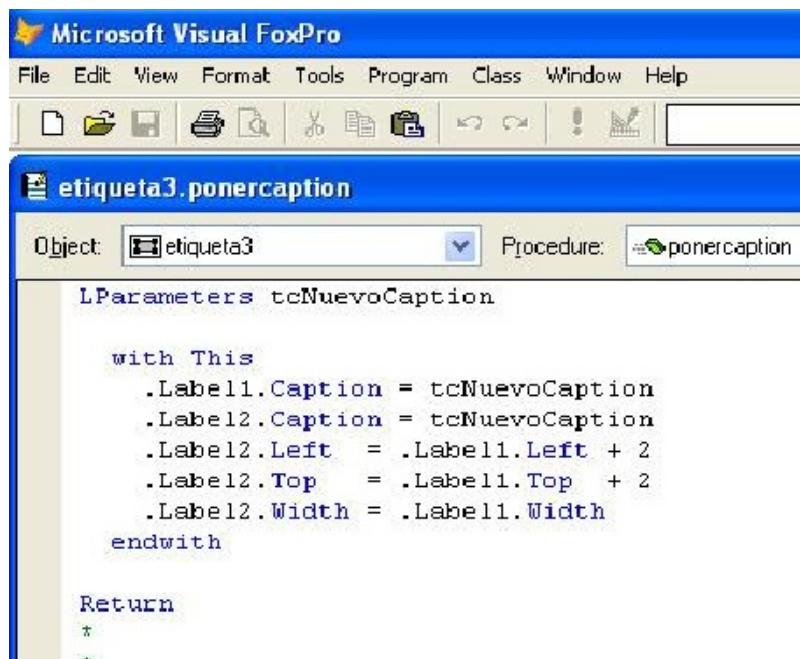
Con esto conseguiremos que *label1* esté arriba y que *label2* esté debajo

Después en el menú elegimos:
Class | New Method ...

y en Name: escribimos “PonerCaption”

Este nuevo método que estamos creando nos permitirá asignar el mismo caption a las dos etiquetas de esta **clase** y también asegurarnos que sus posiciones sean las más adecuadas para que el efecto de sombra sea visible y estéticamente agradable. Nos aseguramos que *label1* y que *label2* estén relacionadas.

En el método *PonerCaption* escribimos:



The screenshot shows the Microsoft Visual FoxPro IDE. The title bar says "Microsoft Visual FoxPro". The menu bar includes File, Edit, View, Format, Tools, Program, Class, Window, Help. The toolbar has various icons. A window titled "etiqueta3.ponercaption" is open. The Object dropdown shows "etiqueta3" and the Procedure dropdown shows "ponercaption". The code in the editor is:

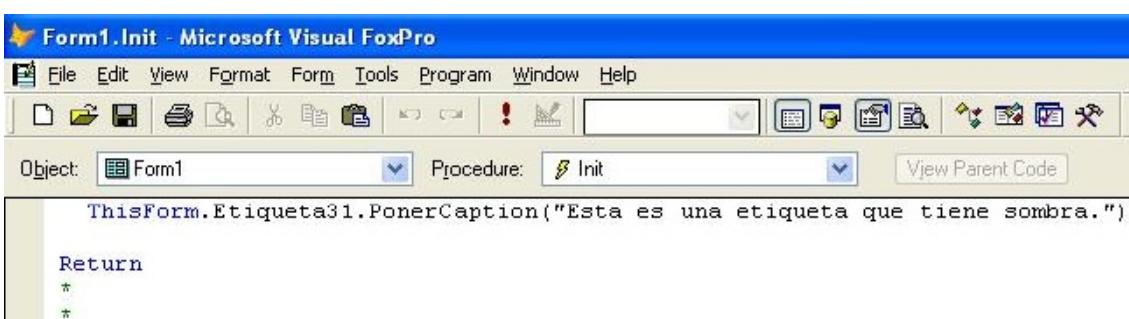
```
LParameters tcNuevoCaption  
  
with This  
    .Label1.Caption = tcNuevoCaption  
    .Label2.Caption = tcNuevoCaption  
    .Label2.Left = .Label1.Left + 2  
    .Label2.Top = .Label1.Top + 2  
    .Label2.Width = .Label1.Width  
endwith  
  
Return  
*  
*
```

Pantalla N° 21

¿Funcionará? Vamos a verificarlo. Creamos un nuevo formulario y lo llamamos FORM3.

En ese formulario insertamos el control *Etiqueta3* que acabamos de crear.

Luego, en el método INIT del formulario escribimos:



The screenshot shows the Microsoft Visual FoxPro IDE. The title bar says "Form1.Init - Microsoft Visual FoxPro". The menu bar includes File, Edit, View, Format, Form, Tools, Program, Window, Help. The toolbar has various icons. A window titled "Form1.Init" is open. The Object dropdown shows "Form1" and the Procedure dropdown shows "Init". The code in the editor is:

```
ThisForm.Etiqueta31.PonerCaption("Esta es una etiqueta que tiene sombra.")  
  
Return  
*  
*
```

Pantalla N° 22

Ejecutamos el formulario y ... ¡¡¡sí, funciona!!!

UNA CLASE LABEL PARA TÍTULOS

Es muy frecuente que en nuestros formularios necesitemos poner títulos o subtítulos para que el usuario tenga más claro el significado de los campos que se encuentran debajo de ese título o subtítulo. Ahora crearemos una **clase** que nos servirá justamente para eso. Puede usarse como una base de otras **clases** similares. Por ejemplo, una podría tener el fondo amarillo, otra podría tener el fondo azul con otro tipo o tamaño de letras, etc., ya es cuestión de tu creatividad.

En la ventana de comandos del Visual FoxPro escribimos:

```
MODIFY CLASS ETIQUETA4 OF CONTROLES AS CONTAINER
```

Le insertamos un control *shape*, como vemos en la siguiente pantalla:



Pantalla Nº 23

El control *shape* sirve para dibujar rectángulos, cuadrados, círculos, óvalos y líneas.

En el control *etiqueta4*:

- BackStyle = 0. Transparent
- BorderWidth = 0
- Height = 23
- Width = 720

En el control *shape1*:

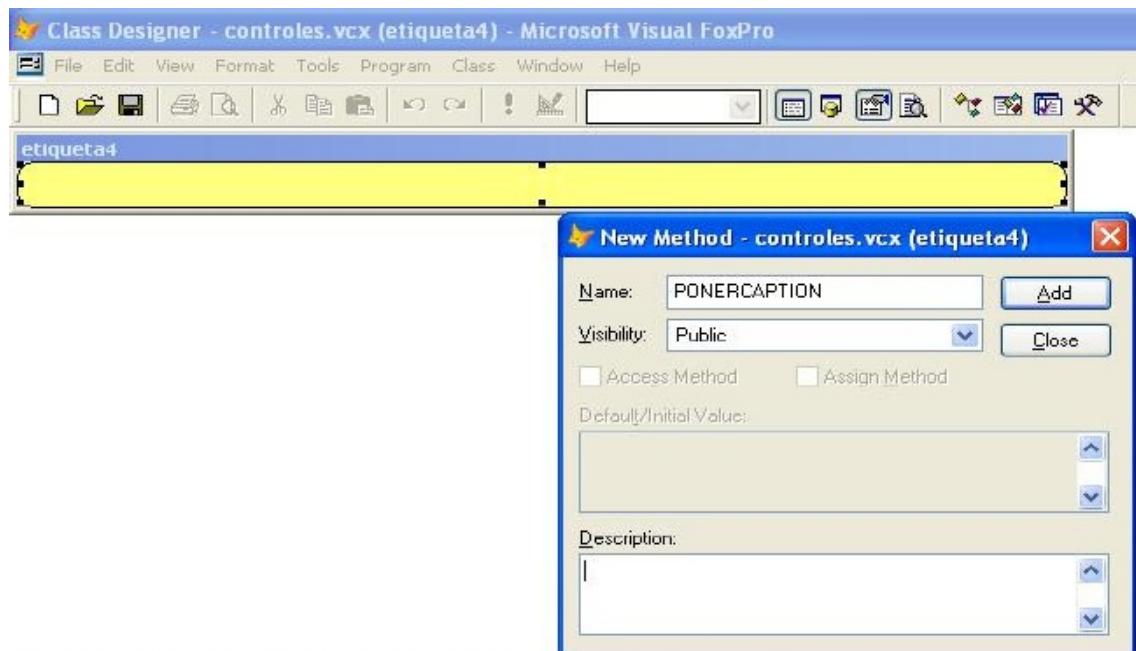
- BackColor = 255, 255, 128
- Curvature = 20
- Height = 23
- Left = 0
- Top = 0
- Width = 720

Ahora le agregamos al control *Etiqueta4* un control *Etiqueta3*, el que creamos en el ejemplo anterior. ¿Podemos agregarle a un container un control creado por nosotros mismos? Claro que sí, sin ningún problema. Los controles que nosotros creamos pueden usarse como si fueran controles nativos del Visual FoxPro, esa es justamente una de las grandes ventajas que tenemos al trabajar con **clases**, las nativas y las creadas por nosotros pueden usarse indistintamente.

En el control *Etiqueta31*:

```
- Height = 23
- Left = 0
- Top = 0
- Width = 720
- Label1.Alignment = 2 - Center
- Label1.FontBold = .T.
- Label1.FontSize = 12
- Label1.Left = 0
- Label1.Top = 3
- Label2.Alignment = 2 - Center
- Label2.FontBold = .T.
- Label2.FontSize = 12
- Label2.Left = 2
- Label2.Top = 5
```

Y ahora, para poner el texto que mostrará *Etiqueta4* creamos un método que se encargará de realizar esa tarea:



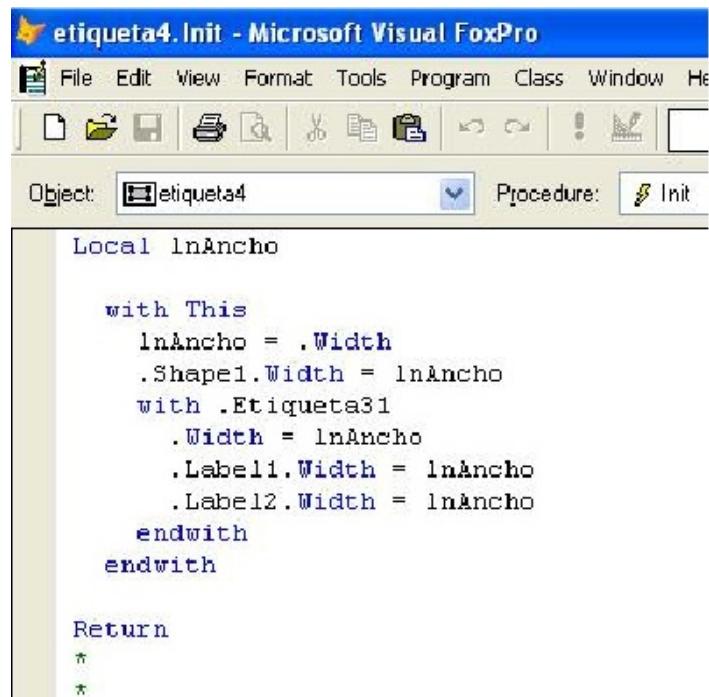
Pantalla N° 24

The screenshot shows the Microsoft Visual FoxPro code editor with the title bar "etiqueta4.ponercaption - Microsoft Visual FoxPro". The code window displays the following VFP code:

```
LParameters tcNuevoCaption
This.Etiqueta31.PonerCaption(tcNuevoCaption)
Return
*
```

Pantalla N° 25

Sin embargo aún falta algo, el tamaño de nuestra *Etiqueta* de títulos lo hemos fijado en 720. Pero ¿y si necesitamos que sea más corto o más largo? En ese caso ya no estará mostrando el texto centrado y estéticamente será feo, así que tenemos que solucionar ese inconveniente. Para ello, al método INIT de *Etiqueta4* le agregamos las siguientes líneas:



```
etiqueta4.Init - Microsoft Visual FoxPro
File Edit View Format Tools Program Class Window Help
Object: Etiqueta4 Procedure: Init
Local lnAncho

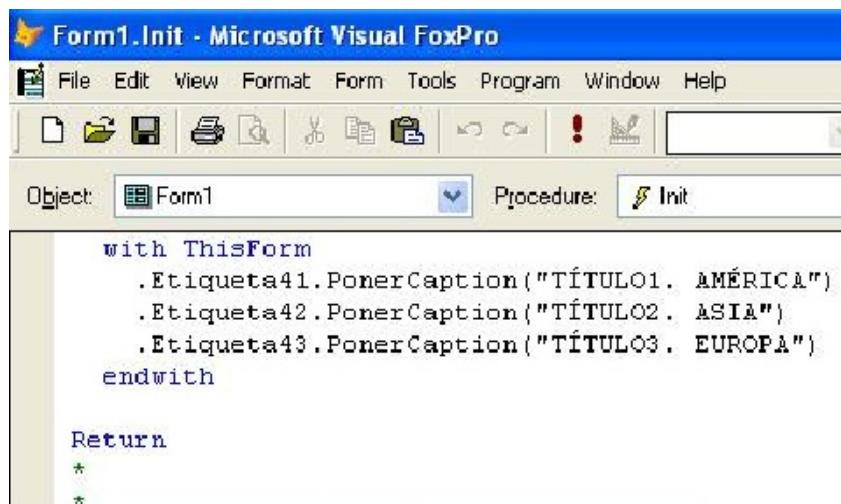
with This
    lnAncho = .Width
    .Shape1.Width = lnAncho
    with .Etiqueta31
        .Width = lnAncho
        .Label1.Width = lnAncho
        .Label2.Width = lnAncho
    endwith
endwith

Return
*
```

Pantalla N° 26

Lo cual nos asegura que *Etiqueta4* y todos sus componentes tengan siempre exactamente el mismo ancho y por lo tanto el texto escrito en *Label1* y en *Label2* estará siempre centrado.

Bueno, ahora solamente nos falta hacer una prueba para verificar que *Etiqueta4* funcione como lo deseamos. Para ello creamos un nuevo formulario al cual lo llamamos FORM4, le agregamos tres controles *Etiqueta4*, y en su método INIT escribimos:



```
Form1.Init - Microsoft Visual FoxPro
File Edit View Format Form Tools Program Window Help
Object: Form1 Procedure: Init
with ThisForm
    .Etiqueta41.PonerCaption("TÍTULO1. AMÉRICA")
    .Etiqueta42.PonerCaption("TÍTULO2. ASIA")
    .Etiqueta43.PonerCaption("TÍTULO3. EUROPA")
endwith

Return
*
```

Pantalla N° 27

Así que solamente nos resta ejecutar el formulario y ver si cumple con nuestras expectativas, lo hacemos y esto es lo que obtenemos:



Pantalla N° 28

Y sí, los títulos aparecen perfectamente centrados, que es justamente lo que queríamos conseguir.

Como ya dijimos anteriormente, podríamos tener varios controles encargados de mostrar los títulos y los subtítulos de nuestras aplicaciones. Algunos de esos controles podrían tener el control de fondo amarillo, otros podrían tenerlo rojo, otros azul, otros blanco, otros negro, etc. También los tipos de letra y los tamaños de las letras podrían ser distintos, así como el color del texto. Ya todo dependerá de nuestra creatividad porque el Visual FoxPro con sus clases visuales realmente nos permite hacer maravillas.

UNA CLASE DE BOTONES

Hay muchísimos botones que podríamos hacer, ya que los botones se usan mucho y en muchos contextos. Ahora haremos uno bien sencillo cuya única función será limpiar los eventos. Lo hacemos así para poder incluirlo en algunas de las clases que estaremos haciendo más adelante.

En la ventana de comandos del Visual FoxPro escribimos:

```
CREATE CLASS BOTONCANCELAR OF CONTROLES AS COMMANDBUTTON
```

En la propiedad *Caption* escribimos:

Cancelar

Y en el método *Click* escribimos:

```
CLEAR EVENTS
```

Y eso es todo, a partir de ahora ya contamos con un botón que nos permitirá cancelar los eventos y el cual podremos usar en muchos otros controles.

UNA CLASE PARA MOSTRAR FOTOGRAFÍAS

En muchas aplicaciones podemos necesitar mostrar fotografías: de un artículo, de un empleado, de un alumno, de un profesor, de un automóvil, de una casa, etc., según sea la aplicación que estemos realizando mostrar fotos puede ser una necesidad o quizás una conveniencia.

Con la **clase** que desarrollaremos a continuación les proveeremos a nuestras aplicaciones de esa característica. Desde luego que podemos mostrar fotografías sin necesidad de usar una **clase** para ello, pero haciéndolo de esta manera lo tendremos más fácil.

En la ventana de comandos del Visual FoxPro escribimos:

```
CREATE CLASS FOTOS OF CONTROLES AS CONTAINER
```

Luego le insertamos:

- Dos controles *image*
- Dos controles *shape*
- Un control *label*

En el control *Foto* escribimos:

```
- BackStyle = 0 - Transparent  
- BorderWidth = 0  
    - Height = 299  
    - Width = 331
```

En el control *Image1* escribimos:

```
- Height = 32  
- Left = 149  
- Top = 0  
- Width = 32
```

En el control *Image2* escribimos:

```
- Height = 237  
- Left = 7  
- Stretch = 2 - Stretch  
- Top = 36  
- Width = 316
```

En el control *Label1* escribimos:

```
- Alignment = 2 - Center  
- BackStyle = 0 - Transparent  
- Caption = (nada, lo dejamos en blanco)  
- FontSize = 8  
- Height = 15  
- Left = 3  
- Top = 282  
- Width = 324
```

En el control *Shape1* escribimos:

- BackColor = 0, 128, 128
- Curvature = 20
- Height = 243
- Left = 3
- Top = 33
- Width = 324

En el control *Shape2* escribimos:

- BackColor = 192, 192, 192
- Curvature = 20
- Height = 243
- Left = 7
- Top = 37
- Width = 324

Con esto ya hemos completado la parte visual de nuestro control, o sea que ya hemos puesto valores a todas las propiedades que nos interesan, ahora falta escribir los métodos, es lo que hacemos a continuación:

En el método *Init* de *Fotos*, escribimos:

```
This.Image1.Picture = LocFile("Cámera.gif")  
Return
```

Pantalla N° 29

Le agregamos un método al cual llamamos *PonerNombreFotografía* y en él escribimos:

```
LParameters tcNombre  
  
with This  
    .Image2.Picture = tcNombre  
    .Label1.Caption = tcNombre  
endwith  
  
Return
```

Pantalla N° 30

El método anterior nos servirá cuando deseemos mostrar una fotografía. Le enviamos como parámetro el nombre de la fotografía que queremos visualizar y el método se encargará de eso.

El siguiente método, por el contrario, nos devolverá el nombre de la fotografía que estamos visualizando. Su nombre es *ObtenerNombreFotografia*.

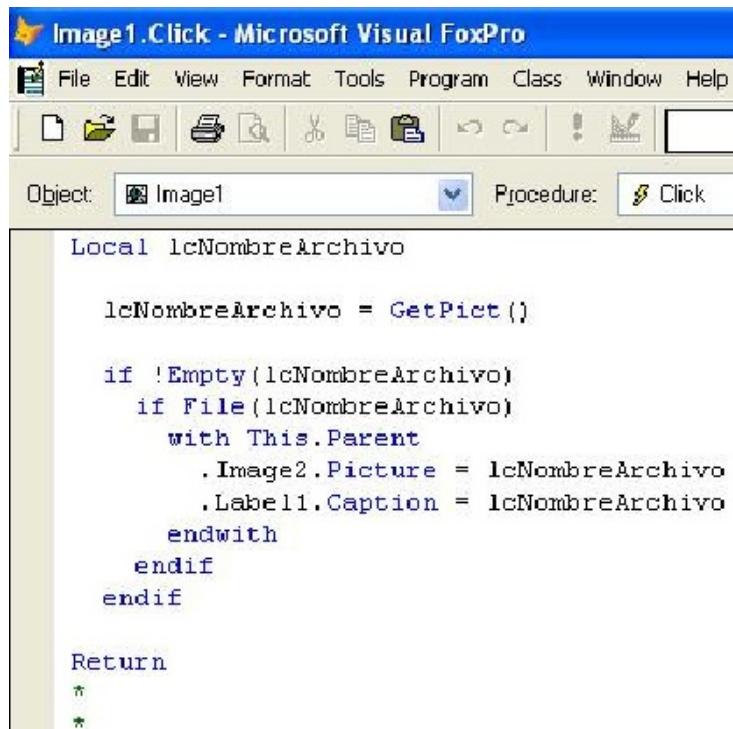


Pantalla N° 31

Así que cuando queremos obtener el nombre de la fotografía que estamos viendo podemos escribir algo como:

```
ThisForm.Fotos1.NombreFotografia()
```

Y para cuando queremos que sea el usuario quien elija la fotografía que se mostrará, ponemos el siguiente código en el método *Click* del control *Image1*.

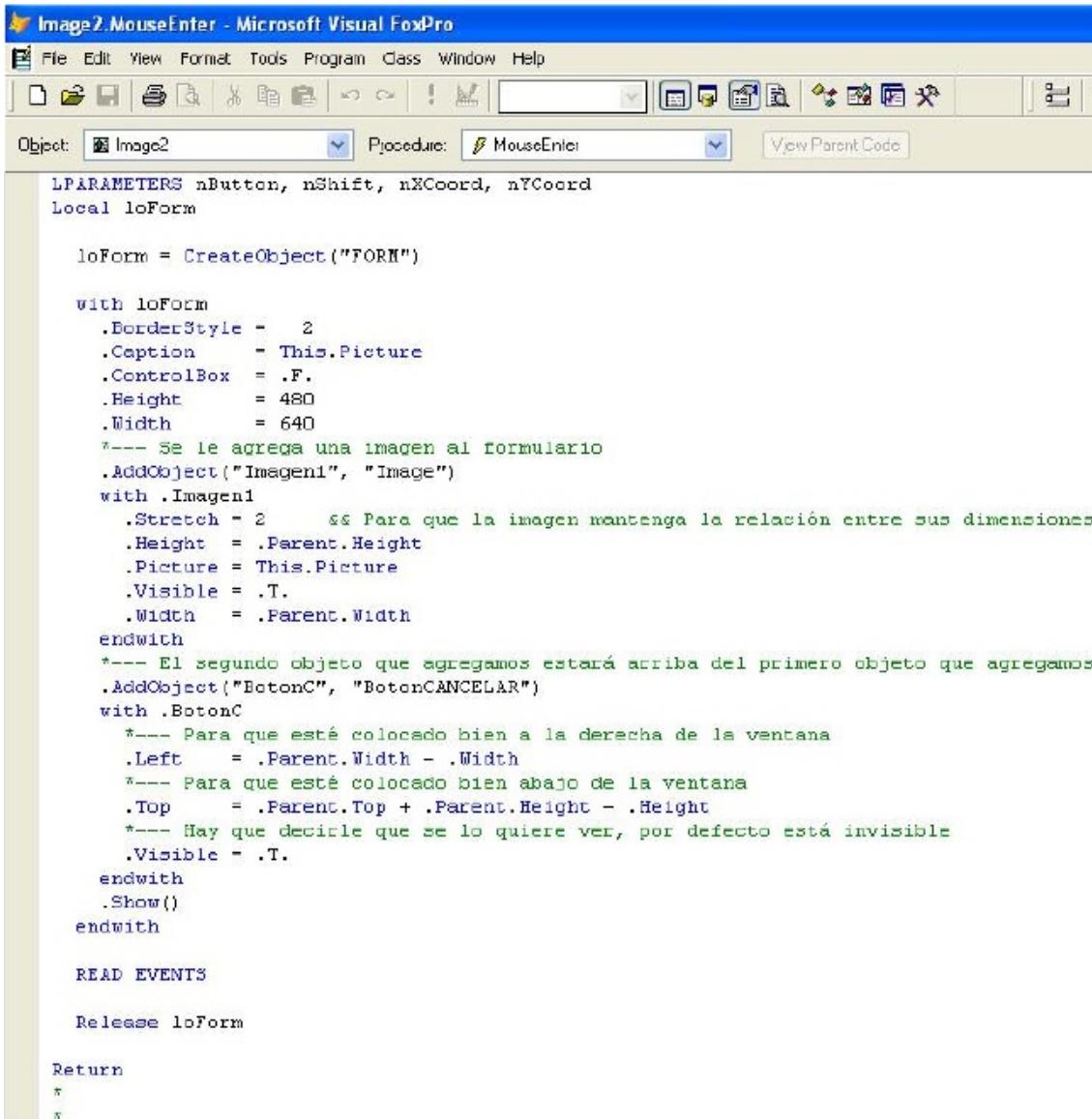


Pantalla N° 32

Cuando él haga click sobre la imagen de la camarita se abrirá un cuadro de diálogo que le pedirá que elija un archivo. Si eligió un archivo entonces será mostrado en el control *Image2*.

Otra funcionalidad interesante que podemos agregarle a nuestro control FOTOS es la posibilidad de mostrar en un nuevo formulario la misma imagen que tenemos en el control *Image2*, pero en un tamaño mucho mayor para que sea visualizado mejor.

Para conseguir el efecto, en el método *MouseEnter* escribimos:



The screenshot shows the Microsoft Visual FoxPro IDE interface. The title bar says "Image2.MouseEnter - Microsoft Visual FoxPro". The menu bar includes File, Edit, View, Format, Tools, Program, Class, Window, Help. The toolbar has various icons for file operations. The status bar shows "Object: Image2" and "Procedure: MouseEnter". The main code editor contains the following VFP code:

```
LPARAMETERS nButton, nShift, nXCoord, nYCoord
Local loForm

loForm = CreateObject("FORM")

With loForm
    .BorderStyle = 2
    .Caption = This.Picture
    .ControlBox = .F.
    .Height = 480
    .Width = 640
    --- Se le agrega una imagen al formulario
    .AddObject("Imagen1", "Image")
    With .Imagen1
        .Stretch = 2      --- Para que la imagen mantenga la relación entre sus dimensiones
        .Height = .Parent.Height
        .Picture = This.Picture
        .Visible = .T.
        .Width = .Parent.Width
    EndWith
    --- El segundo objeto que agregamos estará arriba del primero objeto que agregamos
    .AddObject("BotonC", "BotonCANCELAR")
    With .BotonC
        --- Para que esté colocado bien a la derecha de la ventana
        .Left = .Parent.Width - .Width
        --- Para que esté colocado bien abajo de la ventana
        .Top = .Parent.Top + .Parent.Height - .Height
        --- Hay que decirle que se lo quiere ver, por defecto está invisible
        .Visible = .T.
    EndWith
    .Show()
EndWith

READ EVENTS

Release loForm

Return
```

Pantalla N° 33

¿Qué hicimos aquí? Creamos un nuevo formulario, al cual le agregamos dos controles: una imagen y un botón. El control imagen servirá para mostrar la misma imagen que estábamos viendo en el control *Image2* del control *Fotos*, obviamente en un tamaño mucho mayor. Y el botón servirá para cerrar el formulario.

Este formulario existe solamente en la memoria de la computadora, no está grabado en el disco duro. Es un formulario temporal, el cual utilizamos para una tarea específica (en este caso, mostrar una fotografía) y luego lo eliminamos.

Recordemos que *Parent* se refiere a la **clase padre**. Por ejemplo cuando en *Imagen1* escribimos: *.Height = .Parent.Height* lo que estamos haciendo es diciéndole al

Visual FoxPro que deseamos que la altura de la imagen *Imagen1* sea la misma que la altura del formulario (porque el formulario es la **clase padre** del control *Imagen1*).

Similarmente, cuando escribimos `.Width = .Parent.Width` lo que estamos haciendo es diciéndole al Visual FoxPro que el ancho de la imagen debe ser igual al ancho del formulario.

Bueno, ahora solamente nos queda crear un formulario para probar si nuestro control *Fotos* funciona correctamente o no.

Así que en la ventana de comandos del Visual FoxPro escribimos:

```
modify form FORM5
```

Le agregamos un control *Fotos* y en el método *Init* del formulario escribimos:



Pantalla N° 34

¿Para qué esto? Para que desde dentro del formulario podamos referirnos a cualquiera de las **clases** que están dentro del archivo CONTROLES.VCX.

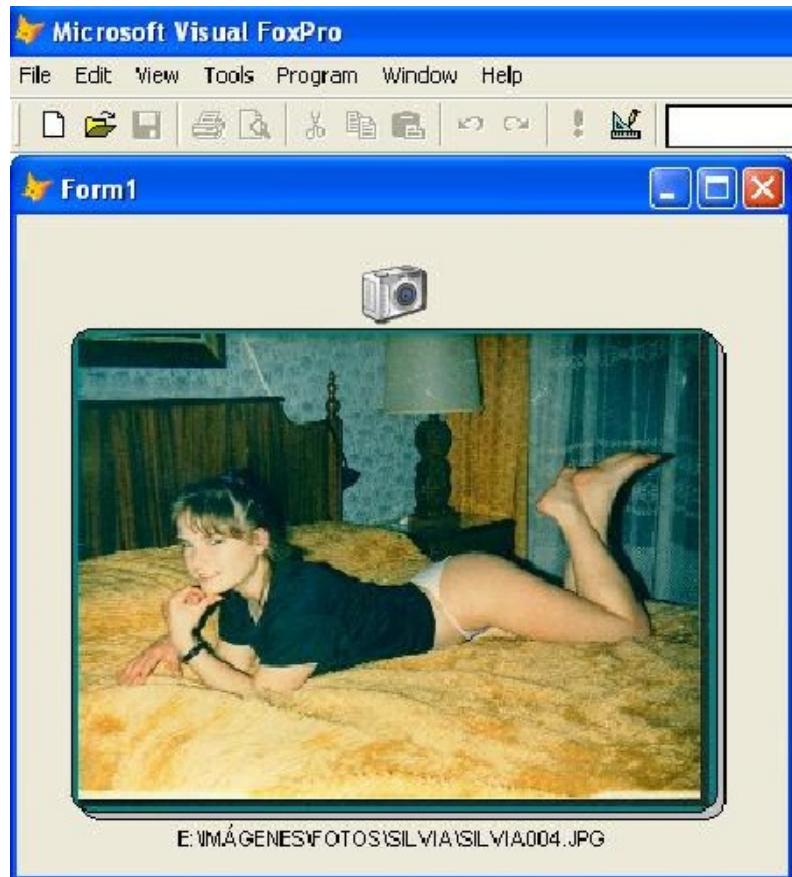
¿Y necesitamos hacer eso? Sí, porque en el método *MouseEnter* del control *Image2* habíamos escrito:

```
.AddObject("BotonC", "BotonCANCELAR")
```

Y como “BotonCANCELAR” no es uno de los controles estándar del Visual FoxPro sino que es un control que creamos nosotros entonces tenemos que indicarle donde se encuentra. Y eso es justamente lo que hace la instrucción SET CLASSLIB, le dice donde debe buscar los controles.

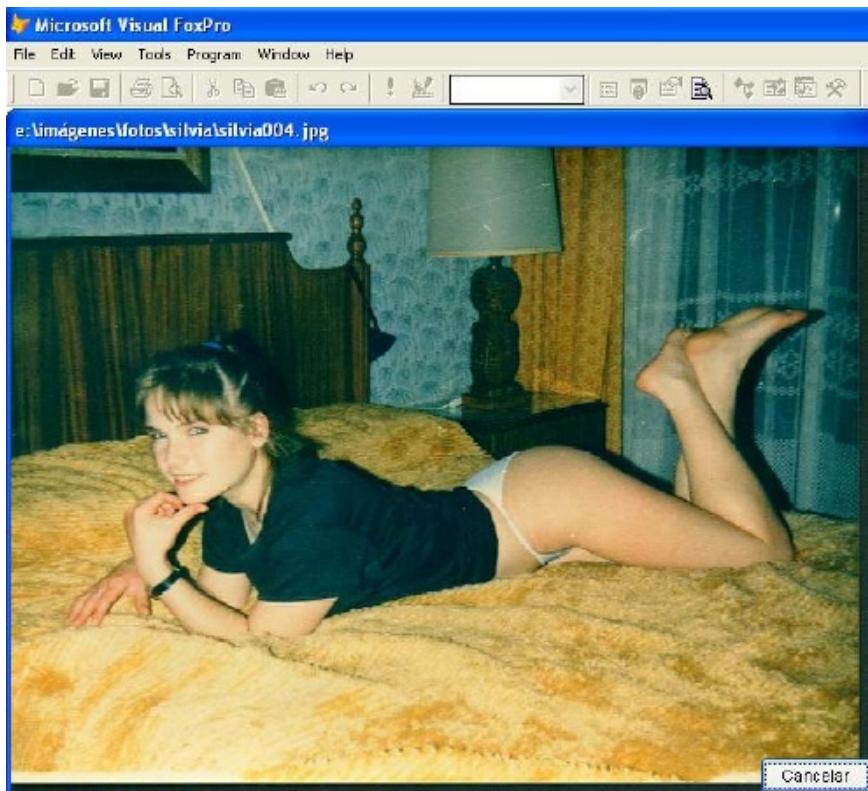
Entonces el Visual FoxPro busca al control *BotonCANCELAR* dentro del archivo CONTROLES.VCX, lo encuentra allí y ... todos felices.

Bueno, ahora solamente nos resta ejecutar el formulario y ver si lo que hemos hecho funciona bien. Así que lo ejecutamos, hacemos click sobre la imagen de la camarita, elegimos una fotografía que tenemos en el disco duro y esto es lo que obtenemos:



Pantalla N° 35

Ahora movemos el Mouse sobre la fotografía y esto es lo que obtenemos:



Pantalla N° 36

VARIAS CLASES PARA INGRESAR DATOS

Los usuarios normalmente ingresan muchos datos en nuestras aplicaciones, tener unos controles preparados para la entrada de sus datos nos facilitará la creación de las pantallas.

Ahora crearemos un control derivado de *Textbox* que utilizaremos para ingresar cualquier tipo de datos. Para diferenciarlo de su padre a éste lo llamaremos *IngresoDatos*

En la ventana de comandos del Visual FoxPro escribimos:

```
MODIFY CLASS INGRESODATO OF CONTROLES AS TEXTBOX
```

Y les asignamos estos valores a sus propiedades:

- BackColor = 250, 250, 250
- BackStyle = 1 - Opaque
- DisabledBackColor = 232, 232, 214
- DisabledForeColor = 0, 0, 0
- FontName = Courier New
- ForeColor = 0, 0, 255
- Height = 23
- Width = 100

Este control podríamos utilizarlo para ingresar cualquier tipo de datos, pero lo iremos afinando para ingresar datos específicos. Para ingresar fechas crearemos una nueva **clase** a la cual llamaremos *IngresoFecha*. En la ventana de comandos del Visual FoxPro escribimos:

```
MODIFY CLASS INGRESOFECHA OF CONTROLES AS INGRESODATO FROM CONTROLES
```

En este caso hemos escrito una nueva cláusula, llamada FROM ¿por qué hicimos eso? En los casos anteriores no era necesario escribirla porque los controles eran derivados de las **clases** estándar del Visual FoxPro, pero el control *IngresoFecha* no es derivado de ninguna de las **clases** estándar sino de una **clase** nuestra, por lo tanto debemos indicarle donde se encuentra esa **clase**.

Les asignamos estos valores a sus propiedades:

- DateFormat = 3 - British
- InputMask = 99/99/9999
- Value = {}
- Width = 81

Similarmente, ahora crearemos un nuevo control, en este caso para ingresar cantidades, al cual llamaremos *IngresoCantidad*. En la ventana de comandos del Visual FoxPro escribimos:

```
MODIFY CLASS INGRESOCANTIDAD OF CONTROLES AS INGRESODATO FROM CONTROLES
```

Como vemos, también está derivado del control *IngresoDatos*.

Les asignamos estos valores a sus propiedades:

- InputMask = 9,999.99
- Value = 0
- Width = 67

Y en el método *Init* escribimos.

The screenshot shows the Microsoft Visual FoxPro interface. The title bar says "Microsoft Visual FoxPro". The menu bar includes File, Edit, View, Format, Tools, Program, Class, Window, Help. Below the menu is a toolbar with various icons. The main window has a title bar "ingresocantidad.Init". The status bar shows "Object: ab!ingresocantidad Procedure: Init". The code area contains the following VFP code:

```
with This
    .Width = Len(.InputMask) * 7 + 11
endwith

Return
*
```

Pantalla N° 37

¿Qué estamos haciendo aquí? Diciéndole que ajuste el ancho del control a la cantidad de caracteres que hay en la propiedad *InputMask*. ¿Y para qué hacemos eso? Para que nuestro control siempre tenga el ancho exacto, que no sea mayor (en cuyo caso estaríamos desperdiando espacio) ni menor (en cuyo caso algunos dígitos podrían no verse) del adecuado.

Y ahora crearemos un control para los importes. Es muy similar al de cantidades, sólo cambian el *InputMask* y el *Width*. En la ventana de comandos del Visual FoxPro escribimos:

```
MODIFY CLASS INGRESOIMPORTE OF CONTROLES AS INGRESODATO FROM CONTROLES
```

Y les asignamos estos valores a sus propiedades:

- InputMask = 999,999,999.99
- Value = 0
- Width = 109

En el método *Init* escribimos lo mismo que habíamos escrito en el método *Init* del control *IngresoCantidad*.

Así que hemos creado 4 controles para el ingreso de los datos del usuario:

- **IngresoDatos** que sirve para cualquier tipo de datos
- **IngresoFecha** que está especializado en fechas
- **IngresoCantidad** que está especializado en cantidades
- **IngresoImporte** que está especializado en importes

Si fuera necesario podríamos agregar algunos más, por ejemplo **MostrarTotales** el cual podría especializarse en totales, sería muy similar a **IngresoImporte** pero su

propiedad *InputMask* tendría mayor cantidad de números 9, para aceptar importes aún mayores y estaría deshabilitado para que el usuario no pueda cambiar su contenido.

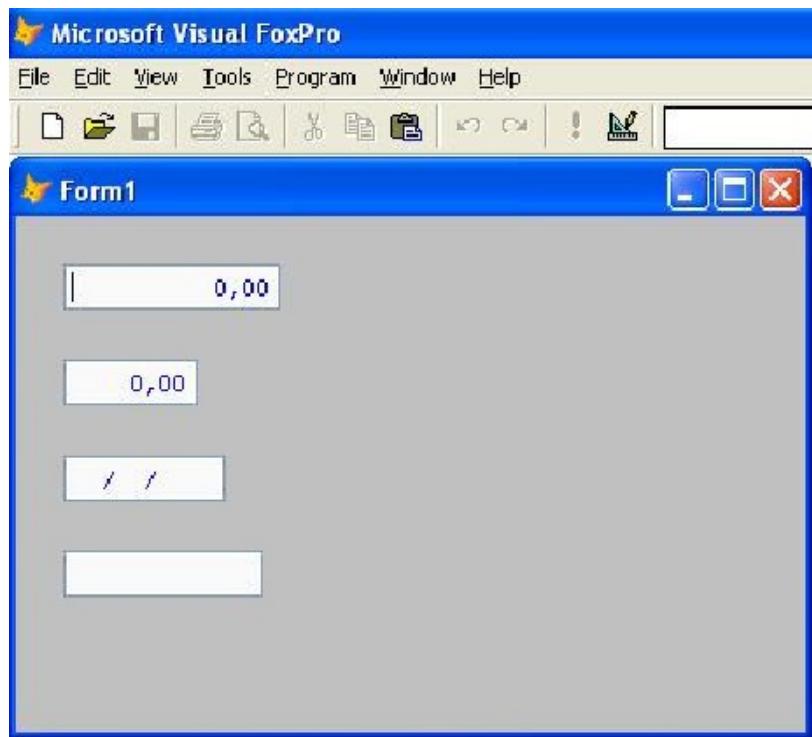
O quizás **IngresoMayusculas** para las situaciones en las cuales todos los caracteres deben estar en mayúsculas. Y así sucesivamente.

Bueno, como de costumbre nos falta probar lo que hemos hecho, debemos verificar que funciona todo ok antes de utilizar estos controles en nuestras aplicaciones.

Por lo tanto creamos un nuevo formulario, al cual llamaremos FORM6, como se ve en la siguiente línea:

```
modify form FORM6
```

Le ponemos un BackColor de 192, 192, 192, insertamos un control de cada tipo y ejecutamos el formulario, como vemos en la siguiente pantalla:



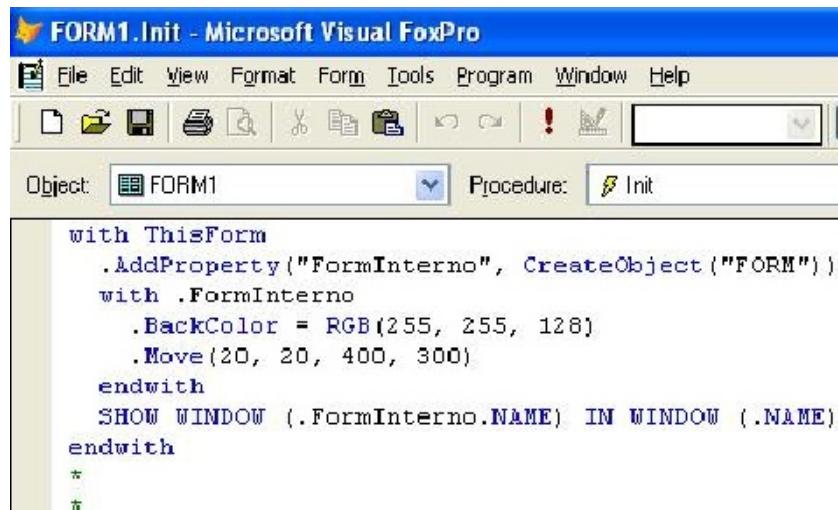
Pantalla N° 38

Y sí, cada control tiene el ancho adecuado y acepta los valores adecuados. Por ejemplo en los controles *IngresoCantidad*, *IngresoImporte* e *IngresoFecha* no podemos ingresar letras. Y en este último debemos ingresar una fecha válida porque si intentamos escribir una fecha como: 21/21/2011 el Visual FoxPro la rechazará porque no existe el mes número 21.

AGREGAR UN FORMULARIO A OTRO FORMULARIO

En el ejemplo Nº 6 (Creando formularios por programa) y en la clase para mostrar fotografías ya habíamos visto algunas formas de agregarle un formulario a otro formulario. Aquí presentamos otra técnica:

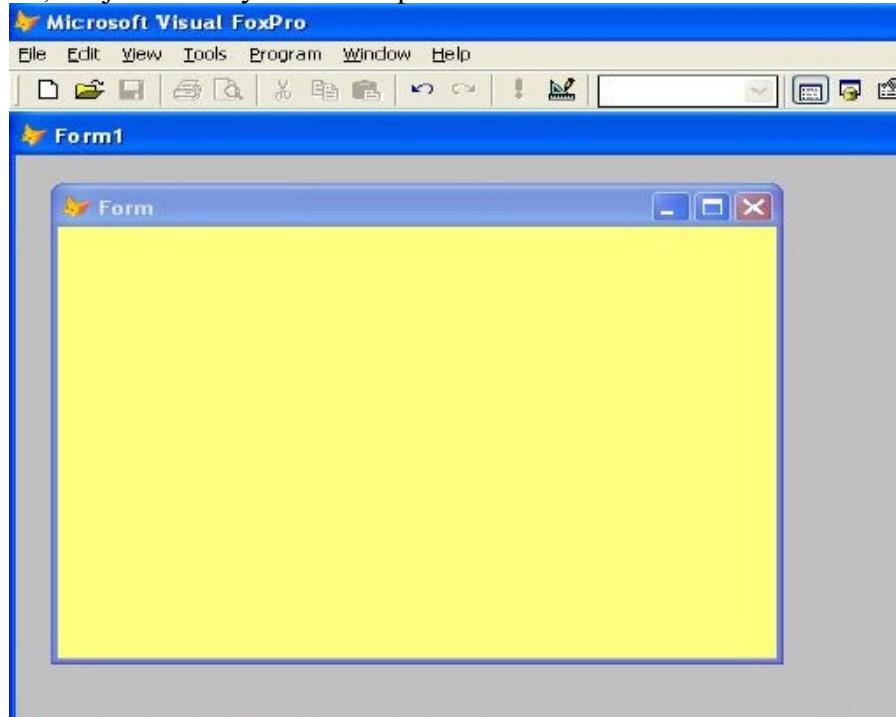
En el método *Init* de nuestro formulario principal o en cualquier otro método (por ejemplo, cuando se hace click sobre un botón de comandos) escribimos el siguiente código:



```
FORM1.Init - Microsoft Visual FoxPro
File Edit View Format Form Tools Program Window Help
Object: FORM1 Procedure: Init
with ThisForm
    .AddProperty("FormInterno", CreateObject("FORM"))
    with .FormInterno
        .BackColor = RGB(255, 255, 128)
        .Move(20, 20, 400, 300)
    endwith
    SHOW WINDOW (.FormInterno.NAME) IN WINDOW (.NAME)
endwith
*
```

Pantalla N° 39

Creamos un nuevo formulario, le agregamos el código de la Pantalla N° 39 en su método *Init*, lo ejecutamos y esto es lo que obtenemos:



Pantalla N° 40

O sea que ahora tenemos un formulario incrustado dentro de otro formulario. El nuevo formulario existe solamente en la memoria, no está grabado en el disco duro, eso significa que cuando se lo cierra desaparece totalmente.

Puede ser muy útil para mostrar mensajes de error, de advertencia, grillas, etc. En general, si no es un formulario de ingreso de datos o un formulario que se usa mucho entonces podemos mostrarlo con alguna de las técnicas de formularios temporales que hemos visto.

El formulario temporal *hereda* todas las propiedades y todos los métodos de su **clase padre**, FORM. Eso significa que podemos cambiarle el color de fondo, el caption, el tamaño, la posición, el tipo de letras, etc., lo mismo que podemos hacer en los formularios normales también podemos hacerlo con los formularios temporales.

Si ya tenemos definidas las características estándares de nuestros formularios temporales, entonces crearlos será más rápido que crear formularios normales. Además, podemos estar seguros de que siempre estarán disponibles, no correremos el riesgo de olvidarnos de copiar algún archivo .SCT o .SCX cuando nos cambiamos de computadora y que por eso nuestra aplicación deje de funcionar correctamente. Con los formularios en memoria, nuestra aplicación siempre funciona.

MÉTODOS ACCESS Y ASSIGN

El Visual FoxPro nos proporciona dos métodos muy útiles: *Access* y *Assign*. El primero se ejecuta automáticamente cuando se obtiene (lee, recupera) el valor de una propiedad. Y el segundo se ejecuta automáticamente cuando se le asigna un nuevo valor a una propiedad.

Estos métodos tienen el mismo nombre que la propiedad a la cual referencian pero finalizan con *_Access* o con *_Assign*, según corresponda.

¿Cuáles son las ventajas que obtenemos al utilizarlos?

Que podemos validar los valores de las propiedades antes de ser usados, asegurándonos así que solamente tenemos valores válidos.

Estos métodos están marcados como **protegidos** en tiempo de ejecución, lo cual implica que no pueden ser accedidos desde afuera de una clase y de sus subclases.

Y están disponibles tanto para nuestros propios objetos como para los objetos nativos del VFP.

EJEMPLO10.PRG

```
Local loFecha

SET CENTURY ON
SET DATE BRITISH

CLEAR

loFecha = CreateObject ("Fecha")

with loFecha
    .dFecha = CtoD ("01/01/1940")
    .ImprimirFecha()
    .PonerFecha (CtoD ("31/12/1995"))
    .ImprimirFecha()
    .PonerFecha (CtoD ("21/04/2011"))
    .ImprimirFecha()
endwith

Release loFecha

Return
*
```

```
DEFINE CLASS Fecha AS CUSTOM
dFecha = {}

PROCEDURE dFecha_Assign
LParameters tdFecha
if DtoS(tdFecha) < "20000101"
=MessageBox("ERROR...la fecha " + DtoC(tdFecha) + " es
inválida")
else
=MessageBox("La fecha " + DtoC(tdFecha) + " es correcta")
This.dFecha = tdFecha
endif
ENDPROC

PROCEDURE ImprimirFecha
? This.dFecha
ENDPROC

PROCEDURE PonerFecha
LParameters tdNuevaFecha
This.dFecha = tdNuevaFecha
ENDPROC

ENDDEFINE
*
```

Como podemos ver, el método `dFecha_Assign` no acepta fechas que sean anteriores al 1 de Enero de 2011. Cualquier intento de asignarle a la propiedad `dFecha` una fecha anterior será rechazada y además se mostrará un mensaje de error.

Tanto si la asignación se desea hacerla directamente con:

`loFecha.dFecha = CtoD("01/01/1940")`

o indirectamente a través de:

`loFecha.PonerFecha(Ctod("31/12/1995"))`

el método `ASSIGN` se encargará de rechazarlas.

Cuando la fecha ingresada es válida y queremos utilizarla debemos asignarla expresamente, como se ve en la línea:

`This.dFecha = tdFecha`

Los métodos ACCESS y ASSIGN también están disponibles para los objetos nativos del Visual FoxPro, por ejemplo podemos escribir:

EJEMPLO11.PRG

```
Local loForm1

loForm1 = CreateObject ("MiForm")

with loForm1
    .Caption = "*** CAPTION DEL FORMULARIO PRINCIPAL ***"
    .Left     = 100
    .Width    = 500
    =MessageBox ("Este es el caption: " + .Caption)
endwith

Release loForm1

Return
*
*

DEFINE CLASS MiForm AS FORM

PROCEDURE THIS_ACCESS
LParameter tcPropiedad
    =MessageBox ("Se accedió a la propiedad: " + tcPropiedad)
    Return(This)
ENDPROC

PROCEDURE LEFT_ASSIGN
LParameters tnLeft
    =MessageBox ("Se le asignó a la propiedad Left el valor: " +
Transform(tnLeft))
    This.Left = tnLeft
ENDPROC

ENDDEFINE
```

EL OBJETO _SCREEN

El Visual FoxPro 9 tiene un objeto público cuyo nombre es _Screen el cual nos permite especificar propiedades y métodos para la pantalla principal.

Lo bueno de hacer eso es que podemos tener propiedades o métodos visibles a toda la aplicación sin necesidad de declarar a las variables como públicas (con la instrucción PUBLIC). Y lo malo ... es que nos permite tener propiedades y métodos que son públicos. En síntesis, puede ser muy útil pero hay que manejarlo con cuidado.

¿Qué podríamos tener en el objeto _Screen? Cualquier dato que nos interese tener disponible en cualquier parte de nuestra aplicación.

```
*--- Se definen las propiedades y métodos del objeto _Screen
*--- Es decir, los que se usarán en todo el Sistema
with _Screen
    .AddProperty("cNombreAplicacion", "Mi Gran Sistema Informático")
    .AddProperty("cCarpetaUbicacionEjecutable", JustPath(SYS(16, 0)))
    .AddProperty("cUsuario")
    .AddProperty("cContrasena")
    .AddObject("MiProcedimiento", "MiClase")
endwith

*--- AQUI VIENE TU PROGRAMA PRINCIPAL

with _Screen
    .cUsuario      = "nombre del usuario"
    .cContrasena = "contraseña del usuario"
endwith

*--- AQUI FINALIZA TU PROGRAMA

*--- Se eliminan las propiedades y métodos del objeto _Screen
with _Screen
    .RemoveProperty("cNombreAplicacion")
    .RemoveProperty("cCarpetaUbicacionEjecutable")
    .RemoveProperty("cUsuario")
    .RemoveProperty("cContrasena")
    .RemoveObject("MiProcedimiento")
endwith

Return
*
*

DEFINE CLASS MiClase AS CUSTOM
    PROCEDURE MiProcedimiento
    ENDPROC
ENDDEFINE
```

Fíjate que la instrucción with _Screen está repetida 3 veces:

La 1^a sirve para agregarle propiedades y métodos al objeto _Screen

La 2^a le asigna valores a algunas de sus propiedades

La 3^a elimina las propiedades y métodos que se habían agregado en la 1^a

El método AddProperty() permite agregarle una nueva propiedad a un objeto. Si lo deseas también puedes especificar el nuevo valor de la propiedad.

Por ejemplo, en la línea:
`.AddProperty("cUsuario")`

se le agrega una nueva propiedad al objeto `_Screen`, llamada `cUsuario`, pero no se le dice el valor de esa propiedad, en cambio en la línea:

`.AddProperty("cNombreAplicacion", "Mi Gran Sistema Informático")`

se le agrega una nueva propiedad al objeto `_Screen`, llamada `cNombreAplicacion` y también se le asigna una valor, en ese caso: “Mi Gran Sistema Informático”.

El segundo `with _Screen` le asigna valores a las propiedades anteriormente agregadas a `_Screen` pero a las cuales aún no se le habían asignando valores.

El tercer `with _Screen` remueve (borra, elimina, desecha) las propiedades y objetos que el primer `with _Screen` había agregado ¿por qué hacer eso? Porque una buena regla de la programación dice: “cuando dejes de usar algo, déjalo como estaba antes de usarlo”.

En este caso, si no removemos las propiedades y los métodos, cuando termine nuestra aplicación ellos aún continuarán dentro del objeto `_Screen`. Si trabajamos en la ventana de comandos del Visual FoxPro, podremos comprobarlo.

¿CUÁNDΟ USAR EL OBJETO _SCREEN?

Cuando queremos que un valor o un método esté disponible en toda la aplicación. Pero siempre ateniéndonos a la siguiente regla:

“La asignación se debe hacer en un solo lugar y solamente en uno”.

Asignación → en un solo lugar
Utilización → en muchos lugares

Eso significa que nunca jamás debemos asignarle a una propiedad de `_Screen` valores en 2 ó más lugares diferentes porque en ese caso podríamos tener problemas, típicamente cuando creemos que la propiedad tiene un valor pero en realidad tiene otro.

Podemos usar el valor de la propiedad todas las veces que lo deseemos y en todos los lugares que lo deseemos...pero le asignamos un valor en un solo lugar.

Si nos atenemos a esa regla, las cosas nos irán bien. Y si no nos atenemos, entonces, pues

MANEJO DE ERRORES EN LA P.O.O.

En el FoxPro tradicionalmente el manejo de los errores del lenguaje se lo hacía con la instrucción ON ERROR. Funcionaba y sigue funcionando, pero si vamos a usar clases hay otra construcción que es mucho mejor.

Se trata de TRY...CATCH...FINALLY

¿Cómo se la utiliza?

Debajo de TRY ponemos las instrucciones que queremos controlar.

Debajo de CATCH las instrucciones que manejan los errores encontrados.

Debajo de FINALLY las instrucciones que queremos ejecutar siempre.

EJEMPLO12.PRG

Local lnMiNumero

CLEAR

*--- Quita uno solo de los asteriscos de abajo y mira lo que sucede.
Luego ponlo a ese y quita otro. Todos son errores puestos a propósito.

```
TRY
*      abc
*      lnMiNumero = lnMiGranNumero
*      do MiProcedure
*      set printer ot
CATCH TO oExcepcion
lnErrorNro = oExcepcion.ErrorNo
lcMensaje = ""
do case
    case lnErrorNro = 1
        lcMensaje = "El archivo no existe"
    case lnErrorNro = 10
        lcMensaje = "Error de sintaxis"
    case lnErrorNro = 12
        lcMensaje = "La variable no existe"
    case lnErrorNro = 16
        lcMensaje = "Comando no reconocido"
    case lnErrorNro = 36
        lcMensaje = "El comando contiene una palabra no reconocida"
endcase
=StrToFile(DtoC(Date()) + " " + Time() + " Ocurrió el error N° "
+ Transform(lnErrorNro) + " " + lcMensaje + " " + Chr(13),
"SISTEMA.ERR", .T.)
FINALLY
ENDTRY

Return
*
```

En el programa de arriba todos los mensajes de error capturados se enviaron a un archivo de texto llamado SISTEMA.ERR, revisando ese archivo se podrá conocer la fecha y la hora en la cual ocurrió el error, así como también su número y un mensaje

descriptivo. Es muy importante para los Analistas y Programadores tener esos datos. Y naturalmente que se podrían agregar otros más, como nombre de la aplicación, nombre del programa, nombre del usuario, nombre de la computadora, etc.

No es necesario hacerlo de esta manera. También podrían haberse capturado los errores dentro del CATCH y manejados allí. Por ejemplo, se pudo haber escrito:

```
CATCH TO oExcepcion
lnErrorNro = oExcepcion.ErrorNo
lcMensaje = ""
do case
  case lnErrorNro = 1
    lcMensaje = "El archivo no existe"
  case lnErrorNro = 10
    lcMensaje = "Error de sintaxis"
  case lnErrorNro = 12
    lcMensaje = "La variable no existe"
  case lnErrorNro = 16
    lcMensaje = "Comando no reconocido"
  case lnErrorNro = 36
    lcMensaje = "El comando contiene una palabra no reconocida"
endcase
=MessageBox(lcMensaje)
```

Y por supuesto que hay varias otras alternativas, ya todo es cuestión de gusto del programador. Lo importante aquí es que con la construcción TRY ... ENDTRY se pueden atrapar todos los errores y manejarlos centralizadamente.

Eso hará que nuestra aplicación sea más estable y que nos resulte sencillo encontrar y corregir los errores que tuviera.

USANDO EL MÉTODO ERROR()

Además de TRY ... ENDTRY otra posibilidad que tenemos para manejar los errores de nuestra aplicación es escribir código en el método ERROR() que tienen todos los objetos.

En este método podemos capturar y procesar todos los errores que ocurran.

AUTOMATIZACIÓN DE WORD

Una de las tantas ventajas que tenemos al utilizar clases es que podemos automatizar algunos programas, por ejemplo: Word y Excel. Ahora veremos como automatizar Word y en el siguiente capítulo veremos como automatizar Excel.

Para decirle al Visual FoxPro que queremos automatizar Word empezamos creando un objeto de clase ... MSWord.

```
Local loWord

loWord = CreateObject("MSWORD")

with loWord

endwith

Release loWord

Return
*
```

Y ahora, dentro del with ... endwith escribiremos todas las instrucciones para hacer lo que deseamos. Por ejemplo:

- Crear un nuevo documento
- Abrir un documento
- Agregar texto
- Mostrar el texto en negritas, subrayado, etc.
- Mostrar la vista previa del documento
- Imprimir el documento
- Guardar el documento (como .DOC, .PDF, etc.)
- Cerrar el documento

Para ello, dentro de nuestra clase MSWORD pondremos las propiedades:

```
DEFINE CLASS MSWORD AS CUSTOM

cNombreDocumento = ""
cNombreImpresora = ""
oWord           = NULL
oDocumento       = NULL
oRango          = NULL

ENDDDEFINE
```

La propiedad *cNombreDocumento* mantendrá el nombre del documento que estamos editando.

La propiedad *cNombreImpresora* mantendrá el nombre de la impresora que usaremos cuando vayamos a imprimir. Debe ser el nombre de una impresora instalada en nuestra computadora.

El valor inicial de la propiedad *oWord* es NULL ¿por qué eso? porque en ella tendremos un objeto y el valor inicial de los objetos podemos definirlo como NULL. En el objeto *oWord* tendremos propiedades y métodos globales.

En el objeto *oDocumento* tendremos las propiedades y los métodos que se refieren a un solo documento.

Y en el objeto *oRango* las propiedades y métodos que se refieren a un rango del documento.

¿Cómo haríamos un método para crear un nuevo documento?

```
PROCEDURE AgregarDocumento
  with This
    .oWord          = CreateObject ("WORD.APPLICATION")
    .oWord.Visible = .F.      && No queremos que Word sea visible
    .cNombreDocumento = ""
    .oDocumento     = .oWord.Documents.Add()
    .oRango         = .oDocumento.Range()
  endwith
ENDPROC
```

¿Y para abrir un documento ya existente?

```
PROCEDURE AbrirDocumento
  LParameters tcNombreDocumento
  with This
    .oWord          = CreateObject ("WORD.APPLICATION")
    .oWord.Visible = .F.      && No queremos que Word sea visible
    .cNombreDocumento = tcNombreDocumento
    .oDocumento     = .oWord.Documents.Open(.cNombreDocumento)
    .oRango         = .oDocumento.Range()
  endwith
ENDPROC
```

Si comparamos ambos métodos veremos que son bastante parecidos. La gran diferencia es que en el método *AbrirDocumento* especificamos el nombre que tiene el documento que queremos abrir y usamos el método *Open()* del objeto *oWord*.

Similarmente, crearemos los métodos para realizar las demás tareas. ¿Queremos mostrar el texto en negritas?

```
PROCEDURE FontNegrita
  LParameters tlNegrita      && debe ser .T. o .F.
  This.oRango.Font.Bold = tlNegrita
ENDPROC
```

¿Queremos imprimir el documento?

```
PROCEDURE ImprimirDocumento
  with This.oWord
    .ActivePrinter = This.cNombreImpresora
    .PrintOut()
  endwith
ENDPROC
```

¿Queremos mostrar el documento?

```
PROCEDURE MostrarDocumento  
    This.oWord.Application.Visible = .T.  
ENDPROC
```

En el programa EJEMPLO13.PRG que acompaña a este documento podrás encontrar un programa que escribe texto en Word. No he incluido ese programa aquí porque es muy extenso, pero allí lo encontrarás, bien explicado.

Automatizar WORD parece complicado pero no lo es tanto, lo más importante a recordar es que podemos referirnos al documento completo, a un rango del documento, a una selección del documento o a una palabra.

Y de acuerdo a eso, podemos utilizar las propiedades y métodos que les afectan, por ejemplo el tipo de letra, el tamaño de la letra, etc.

También debemos recordar que debemos colapsar los rangos para que funcionen los valores que pusimos.

Y que para pasar a la siguiente línea debemos enviar un retorno de carro (o sea, el código CHR(13)). Si no lo hacemos así, el texto continuará en la misma línea.

AUTOMATIZACIÓN DE EXCEL

Automatizar Excel, sea para grabar datos en una planilla o para obtener los datos de una planilla es una tarea bastante frecuente, así que vale la pena saber como hacerla.

Cuando vamos a grabar en una planilla Excel tenemos dos opciones:

- Grabar en una planilla nueva
- Grabar en una planilla existente

En general, para ahorrar tiempo y hacerlo bien desde la primera vez, la segunda opción es la preferible.

Si creamos una planilla nueva, entonces nosotros desde el Visual FoxPro tenemos que dedicarnos a escribir su título, sus recuadros, los nombres de las columnas, los colores, los tamaños de las letras, los sombreados, etc. Puede hacerse, claro que sí, pero lleva tiempo eso de estar revisando una y otra vez si el resultado obtenido es el resultado deseado.

En cambio, si modificamos una planilla ya existente lo tenemos mucho más fácil. ¿Por qué? porque todo lo anterior (título de la planilla, recuadros, nombres de las columnas, tipos de letras, colores, etc.) ya está dentro de la planilla y lo único que hacemos es colocar datos en las celdas.

Con la clase que veremos a continuación podremos hacerlo de cualquiera de las dos maneras, aunque lo recomendable es usar la segunda, para ahorrar mucho tiempo.

Nuestra clase inicial es la siguiente:

```
DEFINE CLASS EXCEL AS CUSTOM

cNombrePlanilla = ""
oExcel          = NULL
oAplicacion     = NULL

ENDDEFINE
```

A la cual le iremos agregando funciones y procedimientos para hacerla más útil y poderosa.

Para crear una nueva planilla escribimos el siguiente procedimiento:

```
PROCEDURE NuevaPlanilla
with This
    .oExcel      = CreateObject ("EXCEL.APPLICATION")
    .oAplicacion = .oExcel.Application
    with .oAplicacion
        .Visible = .F.
        .WorkBooks.Add()
    endwith
endwith
ENDPROC
```

Y para abrir una planilla que ya teníamos grabada en el disco, escribimos lo siguiente:

```
PROCEDURE AbrirPlanilla
LParameters tcNombrePlanilla
with This
    .oExcel      = CreateObject ("EXCEL.APPLICATION")
    .oAplicacion = .oExcel.Application
    with .oAplicacion
        .Visible = .F.
        .WorkBooks.Open(tcNombrePlanilla)
    endwith
endwith
ENDPROC
```

Para escribir un valor (texto, número, fecha) en una celda, tenemos el siguiente procedimiento, muy sencillo realmente:

```
PROCEDURE EscribirCelda
LParameters tnFila, tnColumna, tuValor
    This.oAplicacion.Cells(tnFila, tnColumna).Value = tuValor
ENDPROC
```

Y para obtener el valor (texto, número, fecha) que se encuentra en una celda, tenemos la siguiente función, también muy sencilla:

```
FUNCTION LeerCelda
LParameters tnFila, tnColumna
Local lnValor
    lnValor = This.oAplicacion.Cells(tnFila, tnColumna).Value
RETURN (lnValor)
```

Y para mostrar la planilla (o sea, para que sea visible al usuario) tenemos otro procedimiento, sencillísimo:

```
PROCEDURE MostrarPlanilla
    This.oAplicacion.Visible = .T.
ENDPROC
```

Como ves, automatizar Excel es muy fácil, podríamos agregarle aún muchos procedimientos y funciones: para dibujar recuadros, poner texto en negrita, cambiar el color del texto, etc., pero eso ya queda a tu cargo, como ejercicio. Es facilísimo, si te hace falta información, utiliza al gran dios Google, quien con toda seguridad te proveerá de las respuestas que necesitas.

UNA CLASE GRILLA MEJORADA

Una **clase** que visualmente llama mucho la atención es la grilla. Y aunque la grilla que provee el Visual FoxPro en forma estándar es muy buena, hay varias cosas que le faltan y que podemos agregarle para hacerla mejor y más adecuada a nuestras necesidades. Por ejemplo, podríamos agregarle las siguientes características:

- Que ordene el contenido según una columna cuando se hace click sobre el *Header* de esa columna
- Que muestre una celda con un color distinto al color de las demás celdas, siempre o solamente cuando se cumple una condición
- Que muestre las filas en distintos colores, según se cumplan algunas condiciones
- Que muestre los totales de las columnas numéricas
- Que muestre gráficos en algunas celdas
- Que permita realizar búsquedas

En la ventana de comandos del Visual FoxPro escribimos:

```
MODIFY CLASS GRILLA OF CONTROLES AS CONTAINER
```

Con lo cual le estamos ordenando crear una nueva **clase**, llamada GRILLA, que se grabará en el archivo CONTROLES.VCX y que será descendiente de la **clase** CONTAINER. ¿Container, por qué un container? Porque nuestra **clase** no tendrá sólo una grilla sino que también tendrá textboxs que mostrarán los totales de las columnas, por eso no podemos hacer que descienda solamente de GRID. Cuando una **clase** debe estar compuesta por dos o más **objetos** entonces debemos crearla con un objeto contenedor, por ejemplo, con un CONTAINER.

COLORES DE LAS FILAS

Usualmente cuando se quiere mostrar las filas de una grilla en dos colores se escribe algo como:

```
Thisform.Grid1.SetAll("DynamicBackColor", ;  
"IIF(MOD(RECNO(), 2)=0, RGB(255, 255, 255), RGB(0, 255, 0))", ;  
"Column")
```

Lo cual tiene el efecto de mostrar las filas pares (2, 4, 6, 8, etc.) en color blanco y las filas impares (1, 3, 5, 7, 9, etc.) en color verde. Es una solución aceptable si sólo queremos mostrar dos colores, pero ... ¿y si queremos mostrar 3, 5, 8, 10 colores? Allí tenemos dos soluciones: o utilizamos un montón de funciones IIF() o utilizamos una función ICASE(). Sin embargo hay una alternativa mucho mejor: utilizar un método o una función propia, especializada en devolver un color.

Supongamos que queremos mostrar las filas en tres colores:

- Rojo, si el importe es negativo
- Blanco, si el importe está entre 0 y 1.000.000
- Azul, si el importe es mayor que 1.000.000

En este caso escribiríamos:

```
ThisForm.Grid1.SetAll("DynamicBackColor", ;  
"Thisform.ObtenerColor(MiCondicion)", "Column")
```

Y el método ThisForm.ObtenerColor() podría ser algo similar a esto:

MÉTODO ObtenerColor

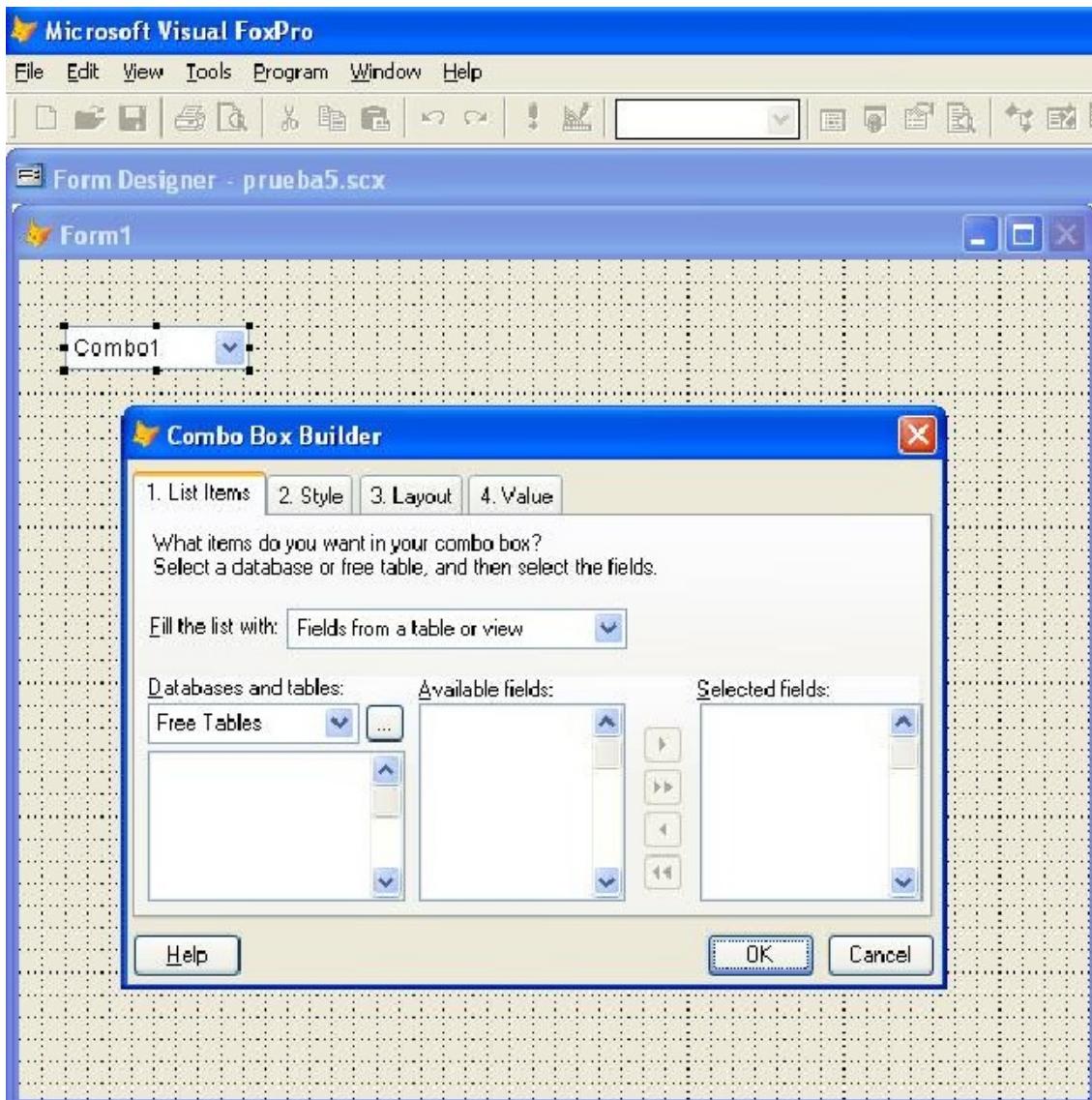
LParameters tnImporte

Local lnColor

```
do case  
    case tnImporte < 0  
        lnColor = RGB(255, 0, 0)      && Color rojo  
    case tnImporte >= 0 .AND. tnImporte <= 1000000  
        lnColor = RGB(255, 255, 255)  && Color blanco  
    case tnImporte > 1000000  
        lnColor = RGB(0, 0, 255)      && Color azul  
    endcase  
  
Return(lnColor)  
*  
*
```

CREANDO UN CONSTRUCTOR

Cuando le agregamos un combobox a un formulario, si hacemos click con el botón derecho sobre él y luego click sobre la opción “Builder” vemos algo como esto:



Pantalla N° 41

Esta ventana que aparece y que tiene como título: “Combo Box Builder” es el **constructor** de la clase Combobox.

¿Y para qué sirve? Para facilitarnos la tarea de usar el combobox: agregándole opciones, eligiendo el estilo, el origen de los datos, etc.

Es mucho más fácil definir al combobox usando a su constructor que definirlo sin usarlo, justamente para eso existe.

Y como el Visual FoxPro es tan bueno y tan poderoso ... nos permite crear constructores para nuestras propias clases. O sea que no solamente podemos utilizar los constructores predefinidos, también podemos crear constructores propios.

PASOS A SEGUIR PARA CREAR UN CONSTRUCTOR

1. A nuestra clase visual debemos agregarle una propiedad llamada BUILDER
2. El valor de esa propiedad será el nombre que tiene (o tendrá) un archivo .APP
3. Creamos un archivo .PRG que recibirá dos parámetros y que llamará a un formulario, por ejemplo:

MAIN_GRILLA.PRG

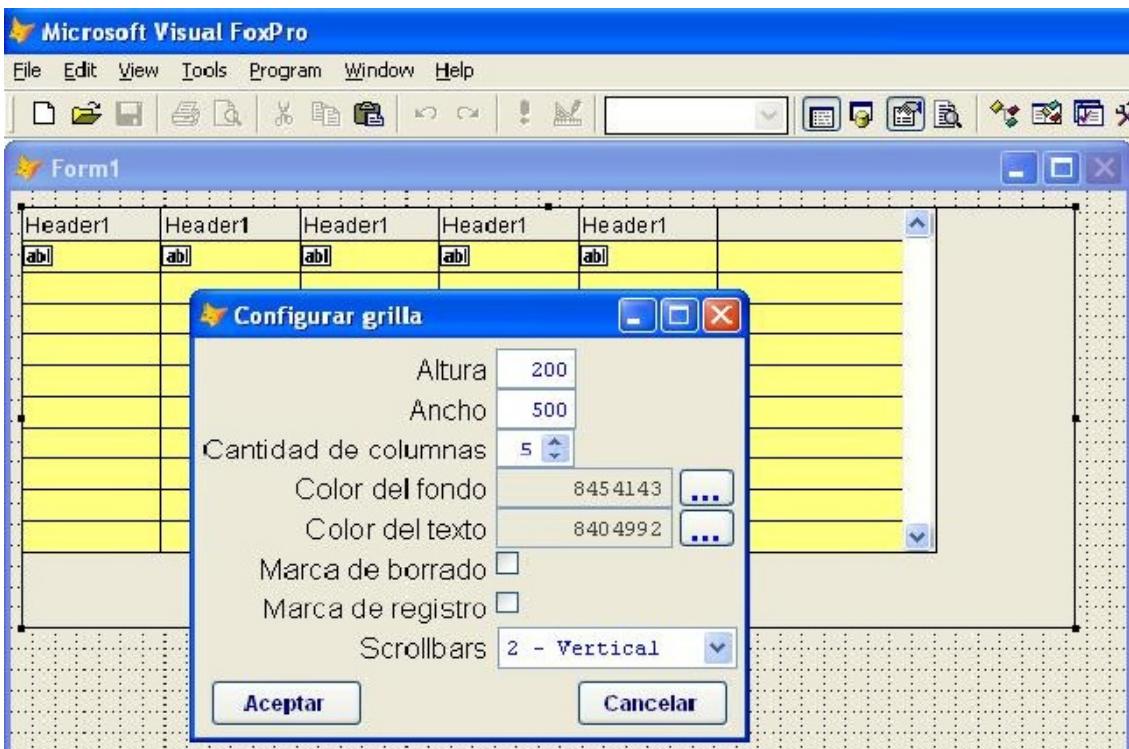
```
LParameters toMiObjeto, toParametro2
```

```
DO FORM CONFIGURAR_GRILLA with toMiObjeto, toParametro2
```

4. Creamos un nuevo formulario, con todas las opciones que deseamos poner a disposición de los usuarios (en este ejemplo: CONFIGURAR_GRILLA)
5. Creamos un nuevo proyecto y ponemos como archivo principal al programa creado en el paso 3 (en este caso: MAIN_GRILLA.PRG)
6. Hacemos click sobre el botón “Build...” y elegimos “Application (app)”
7. Elegimos el nombre para nuestra aplicación (que debe ser igual al nombre que pusimos en la propiedad, en el paso 2) y la generamos
8. Ya está

Ahora, cuando deseamos utilizar el constructor que acabamos de crear, simplemente le agregamos esa clase visual a un formulario (en nuestro ejemplo, una grilla), hacemos click con el botón derecho sobre nuestra clase (en este ejemplo, la grilla), luego click en “Builder...” y ... veremos el formulario que habíamos creado en el paso Nº 4.

Veamos un ejemplo, hemos creado un nuevo formulario, le hemos agregado una grilla, pusimos el mouse sobre ella, presionamos el botón derecho, elegimos la opción “Builder...” y esto es lo que se nos muestra.



Pantalla N° 42

Por supuesto que nuestro **constructor** puede ser muchísimo más completo. Por ejemplo podría permitirnos elegir el nombre de cada columna, el tipo de letra, el tamaño de las letras, si queremos que debajo de la columna nos muestre el total de ella, etc., opciones tenemos muchísimas y queda a tu criterio, amable lector, elegir cuales de ellas quieras agregar a tu clase grilla.

Entonces, ahora no solamente sabes como crear nuevas clases, también sabes como agregarles constructores a esas clases, los cuales las harán mucho más valiosas.

Si tus clases las harás para tu propio uso, entonces quizás no necesites crear constructores para ellas, pero si tienes la intención de que otras personas las utilicen entonces sí sería muy bueno que tengan constructores, para facilitar el uso.

LA FUNCIÓN BINDEVENTO

Esta función puede realmente ser muy, pero muy útil. Nos permite enlazar un método de una clase con otro método de otra clase.

Hay dos ejemplos que la utilizan y en ellos puede verse como funciona y la utilidad que puede obtenerse de esta función.

Ejecuta el formulario DEMO12.SCX para verla en acción.



Pantalla N° 43

Cambia el tamaño de la ventana y la cantidad de botones, luego haz click sobre el botón “Crear botones” y fíjate lo que sucede. Prueba con varios tamaños de ventana y con distinta cantidad de botones.

Te recomiendo que estudies el funcionamiento de este formulario, lo que aprendas te podrá resultar muy útil.

¿SE PUEDE CONVERTIR A VISUAL FOXPRO EN UN LENGUAJE FUERTEMENTE TIPEADO?

Un lenguaje **fuertemente tipeado** es aquel que:

1. Requiere que todas las variables sean declaradas antes de ser usadas
2. En la declaración de la variable se debe especificar el tipo de datos que acepta
3. No acepta un tipo de datos distinto al declarado

Un ejemplo de lenguaje fuertemente tipeado es C. Por ejemplo, podríamos escribir:

```
main() {  
    int MiNumeroEntero;  
    char MiNombre[40];  
}
```

Con lo cual le decimos que MiNumeroEntero es una variable que solamente aceptará números enteros y que MiNombre es una variable que aceptará como máximo 40 caracteres.

La variable MiNumeroEntero no aceptará caracteres ni la variable MiNombre aceptará números. Por ejemplo, si intentamos realizar una asignación como la siguiente, obtendremos un error de compilación:

```
MiNombre = 123456;
```

En cambio, Visual FoxPro es un lenguaje **débilmente tipeado**. Eso implica que podemos escribir:

```
MiNombre = "Walter"  
MiNombre = 123456  
MiNombre = 2.79
```

y él acepta sin chistar todos esos valores para la variable MiNombre.

Un lenguaje **fuertemente tipeado** tiene sus ventajas y sus desventajas. La ventaja es que rechazará asignaciones que no corresponden a su tipo declarado de variable (como las asignaciones numéricas a MiNombre de las líneas anteriores) y la desventaja es que ... rechazará esas asignaciones.

En la ayuda del Visual FoxPro hay una sección titulada:

[How to: Implement Strong Typing for Class, Object and Variable Code](#)

Leyendo ese título (como implementar el tipeado fuerte para clases, objetos y variables de código) se podría pensar que el Visual FoxPro nos permite de alguna manera convertirlo en fuertemente tipeado.

Más abajo encontraremos un subtítulo que dice:

To implement strong typing

En el cual dice que hay que usar la cláusula AS para conseguir ese objetivo. ¿Funcionará? Hmmmm....probemos. Escribimos un nuevo programa:

```
LOCAL lnMiNumero AS INTEGER, lcMiNombre AS STRING
```

```
lnMiNumero = "Esto es un texto"  
lcMiNombre = 123321
```

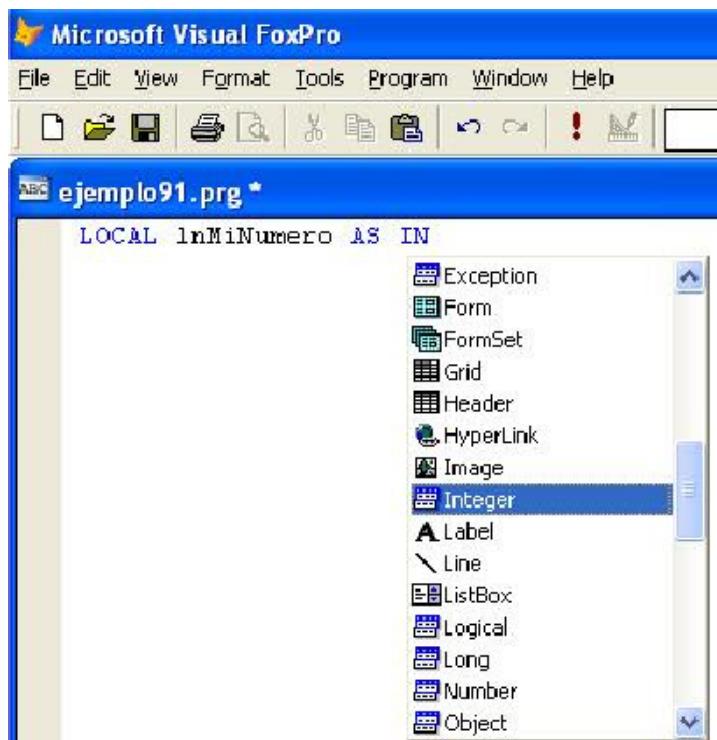
```
? lnMiNumero, lcMiNombre
```

Ejecutamos este programa para ver que sucede y ¡¡¡sorpresa!!! Vemos que en la pantalla aparece:

Esto es un texto 123321

Lo cual **no debería ocurrir** si con la cláusula AS se pudiera convertir al VFP en un lenguaje **fuertemente tipado**.

Con la cláusula AS sí podemos obtener del IntelliSense la funcionalidad de que nos muestre todos los tipos de variables disponibles, por ejemplo, cuando escribimos la siguiente línea en un programa:



Pantalla N° 44

Nos indica cuales son los tipos de datos disponibles y cuando escribimos IN, vemos que se posiciona sobre la palabra "Integer".

Esto es sin dudas una ayuda pero esa sola característica no lo convierte al Visual FoxPro en un lenguaje fuertemente tipeado. En otras palabras, la ayuda del VFP está equivocada, **mintió**.

Lo cual tiene una moraleja: no creas todo lo que leas, pruébalo, verifícalo. Siempre es preferible ser escéptico y conocer la verdad a ser crédulo y creerlo todo sin haberlo comprobado.

TRUCOS Y CONSEJOS

PROPIEDAD DE SOLO LECTURA

A veces podemos desear o necesitar que una propiedad sea de solo lectura, eso lo podemos hacer muy fácilmente con el método ASSIGN, como vemos en el siguiente ejemplo:

```
Local loSoloLectura

loSoloLectura = CreateObject("PROPIEDAD_SOLO_LECTURA")

with loSoloLectura
    .cMiVariable = "Este es el nuevo texto"
    ? "Mi texto = " + .cMiVariable
endwith

Release loSoloLectura

Return
*
*
*
DEFINE CLASS PROPIEDAD_SOLO_LECTURA AS CUSTOM
    cMiVariable = "Este es el valor original"
    PROCEDURE cMiVariable_ASSIGN
        LPARAMETERS tcValor
        =MessageBox("Error, la propiedad cMiVariable es de solo
lectura")
        *---- Quita el asterisco de la siguiente linea para asignar
        *      This.cMiVariable = tcValor
    ENDPROC
ENDDEFINE
*
*
```

En este caso la propiedad *cMiVariable* es de solo lectura, si le asignamos un valor éste no tendrá efecto, la propiedad continuará con el valor anterior.

El método *cMiVariable_ASSIGN* se ejecutará cada vez que se le asigne algún valor a la propiedad *cMiVariable*.

Si queremos que el valor asignado pase a la propiedad entonces debemos escribir:

```
This.cMiVariable = tcValor
```

¿CÓMO SABER SI UNA PALABRA ESTÁ CORRECTAMENTE ESCRITA?

A veces queremos saber si las palabras que escriben los usuarios son correctas, si existen en el idioma castellano o no. La forma más sencilla de saberlo es a través de la automatización con Word, como vemos en el siguiente ejemplo:

```
Local lcMiPalabra, loWord, llResultado

lcMiPalabra = "Visuel"      && quiso escribir 'Visual'

loWord = CreateObject ("WORD.APPLICATION")

llResultado = loWord.CheckSpelling(lcMiPalabra)

? "¿Existe la palabra " + lcMiPalabra + "? ", llResultado

Release loWord

Return
*
```

VARIABLES Y FUNCIONES EN LOS CAPTION

Todos sabemos que podemos cambiar el valor del caption de un formulario, botón o etiqueta escribiendo en un método algo como:

```
ThisForm.Caption = "Este es el caption de mi formulario"
```

```
ThisForm.Label1.Caption = "Hoy es el día " + DTOC(DATE())
```

Y también sabemos que podemos cambiar directamente la propiedad, para eso hacemos click sobre el control y luego en la ventana de propiedades buscamos *Caption* y ponemos allí el valor que nos interesa.

Sin embargo el Visual FoxPro nos da otra posibilidad, una que muy poca gente conoce. En la ventana de propiedades, buscamos el *Caption* y escribimos:

```
= "Hoy es el día " + DTOC(DATE())
```

Lo importante aquí es colocar el símbolo de igualdad (=) en el *Caption* ya que si no lo ponemos, el truco no funciona.

Podemos ver un ejemplo en el formulario DEMO08.SCX

¿USAR EL MÉTODO SHOW() O LA PROPIEDAD VISIBLE PARA MOSTRAR UN FORMULARIO?

Hay dos formas posibles de mostrar un formulario, podemos escribir:

```
loMiFormulario.SHOW()
```

o podemos escribir:

```
loMiFormulario.Visible = .T.
```

¿Cuál es la diferencia, cuál es mejor?

El método *SHOW()* muestra el formulario y le asigna a la propiedad *Visible* el valor .T.

La diferencia entre *SHOW()* y *Visible* es que *SHOW()* es un método o sea que puedes escribir código dentro de él. En cambio a la propiedad solamente le pones verdadero o falso.

Eso implica que si cuando se muestra el formulario también quieras hacer otra cosa entonces usarías el método *SHOW()* pero si lo único que te interesa es que el formulario esté visible entonces puedes utilizar a cualquiera de los dos.

¿CÓMO SABER CUALES SON LAS VARIABLES DECLARADAS EN UNA FUNCIÓN O PROCEDIMIENTO?

Si tienes habilitado el IntelliSense, entonces escribe **zloc** y luego un espacio en blanco y así verás todas las variables que has declarado.

COMO CAMBIAR LOS VALORES DE LAS PROPIEDADES DE UNA GRILLA

En modo ejecución, con la función *SetAll()*:

```
ThisForm.Grid1.SetAll('WIDTH', 40, 'COLUMN')
```

En modo diseño, selecciona la grilla, y escribe en la ventana de comandos:

```
ASelObj(laGrilla, 1)
laGrilla[1].SetAll('Width', 120, 'column')
Release laGrilla
```

Vuelve a tu formulario, mira la grilla y verás que ... todas las columnas tienen el mismo ancho y todas tienen un ancho de 120 pixeles.

Por supuesto que puedes utilizar otras propiedades también si lo deseas (*BackColor*, *ForeColor*, *FontName*, *FontSize*, etc.).

CONCLUSIÓN

Como te habrás dado cuenta, aprender como usar las clases del Visual FoxPro no es algo ni difícil, ni complicado, ni estratosférico. Tiene sus vueltas, sí, pero todo es muy lógico cuando empiezas a pensar en ellas.

Aún hay mucho más que puedes aprender sobre las clases, aquí no se ha dicho todo sobre ellas, pero con el material que ahora tienes en tus manos ya puedes empezar a dar tus primeros pasos.

Es fuertemente aconsejable que trates de dominarlas, porque el futuro de los lenguajes de programación sin ninguna duda está allí, las ventajas son muchas, como habrás podido comprobar después de leer este documento.

Como siempre en programación, lo importante es que practiques mucho, así le tomarás el pulso y te sentirás muy a gusto con ellas.

Recuerda también que hay muchos ejemplos en el disco duro, en la carpeta donde descargaste este documento, y cuyas explicaciones no han sido incluidas aquí por falta de tiempo del autor ya que de haberlo intentado hubiera tardado varios días más en publicar lo que has leído.

Pero esos ejemplos están allí, esperando que los revises.

Cualquier duda, sugerencia, consulta, o corrección que deseas realizar, puedes comunicarte conmigo a mi e-mail, que siempre serás bienvenido si escribes con buena onda.

Walter R. Ojeda Valiente
wrov@hotmail.com