Cuadernos Didácticos

Ingeniería Informática

Guía del lenguaje COOL de Multibase Cosmos

Cuaderno Nº 15

B. Cristina Pelayo García-Bustelo
Universidad de Oviedo

Juan Manuel Cueva Lovelle
Universidad de Oviedo

Oviedo, Enero 1999

Cuadernos Didácticos
Ingeniería Informática

Cuaderno Nº 15

Guía del lenguaje COOL de Multibase Cosmos

Autores:

B. Cristina Pelayo García-Bustelo
Universidad de Oviedo – España

J.M. Cueva Lovelle
Universidad de Oviedo - España

Editorial:

SERVITEC

ISBN: 84-699-0053-6

Oviedo, Enero 1999

1ª Edición

MULTIBASE

Guía del lenguaje COOL

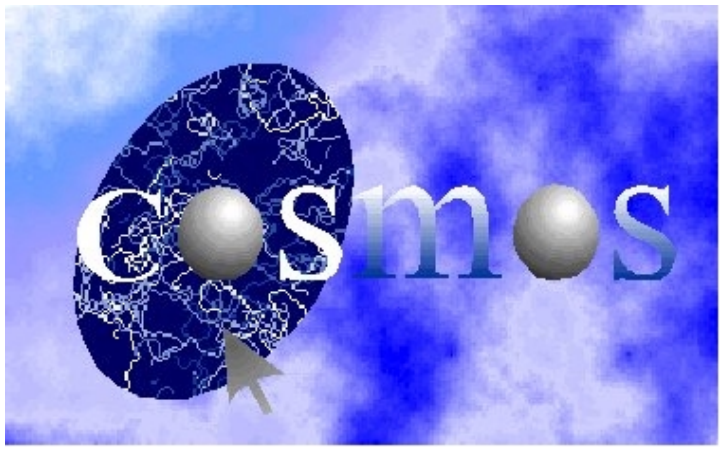


Tabla de Contenidos

Capítulo 1

Primeros pasos programando

Introducción

Se van a realizar una serie de programas sencillos con objeto de obtener el conocimiento del lenguaje y para ello se creará un proyecto en cada uno de ellos con el código que probaremos mediante el Editor de Código de COSMOS. Los pasos para crear un proyecto e insertar en el código son comunes para todos los programas que realicemos, por tanto se mostrarán detalladamente en el primero de los programas y las posibles ampliaciones se detallaran a su debido tiempo.

Construyendo un proyecto

Lo primero que debemos realizar para implementar un programa será crear el entorno donde se va a desarrollar, para ello debemos seguir los siguientes pasos:

Al abrir COSMOS aparecerá el Editor Visual de COSMOS que tiene el siguiente aspecto:

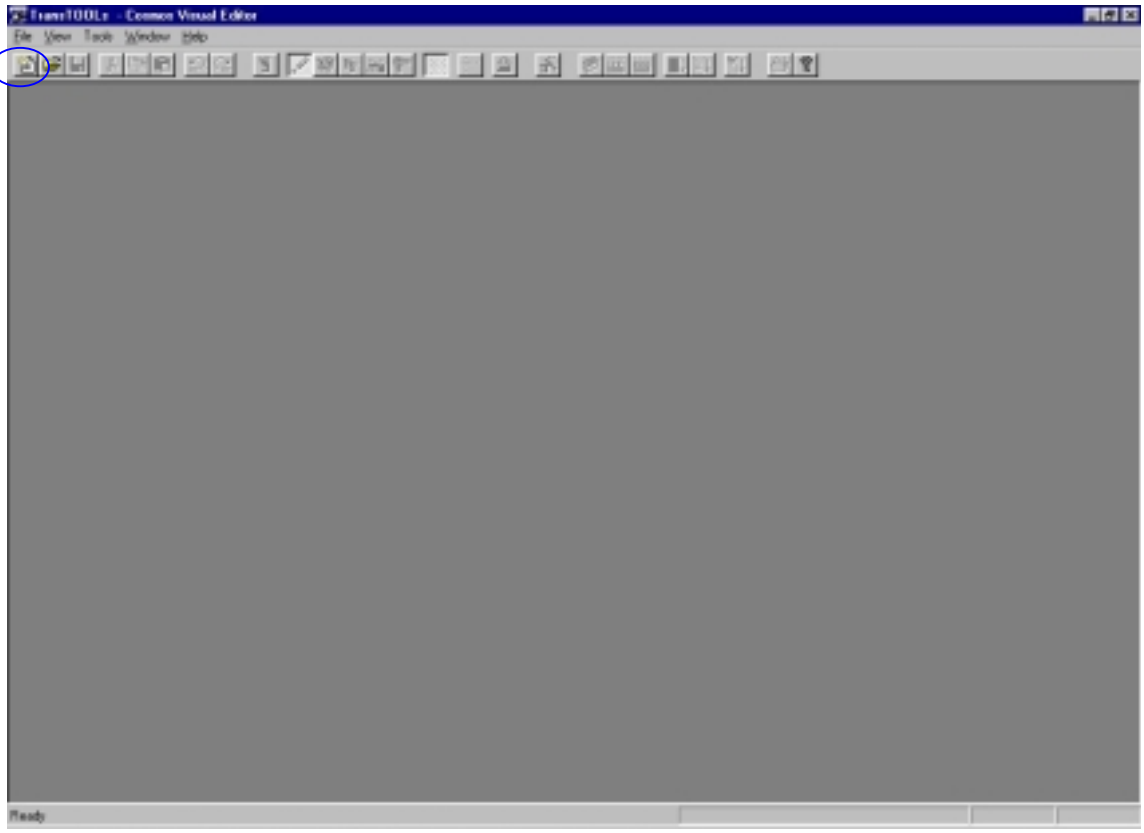


Figura 1.1. Aspecto del Editor Visual

Debemos comenzar un nuevo proyecto, y para ello podemos o hacer click en el icono marcado en la figura 1 o también mediante el menú desplegable File seleccionar la opción New.

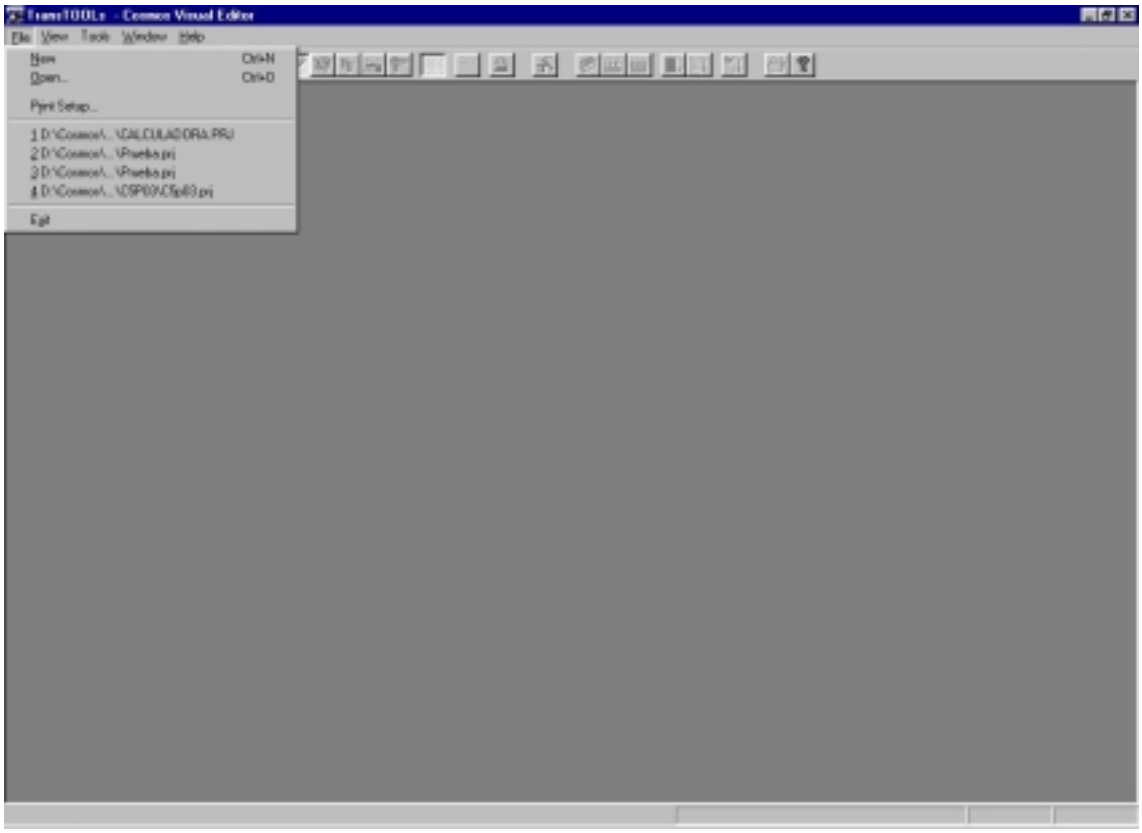


Figura 1. 2. Menú desplegable File

Aparece una nueva ventana donde debemos seleccionar el tipo nueva aplicación que queremos realizar. Nosotros seleccionaremos New Project.

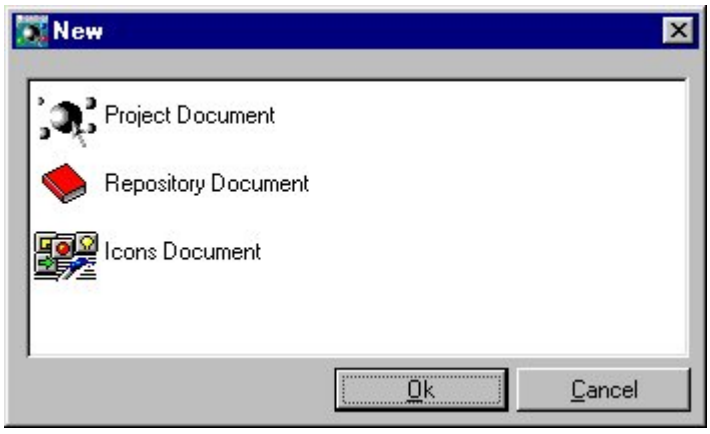


Figura 1.3. Seleccionar tipo de aplicación.

Cuando creamos un nuevo proyecto nos aparece un cuadro de diálogo preguntando los datos del nuevo proyecto:

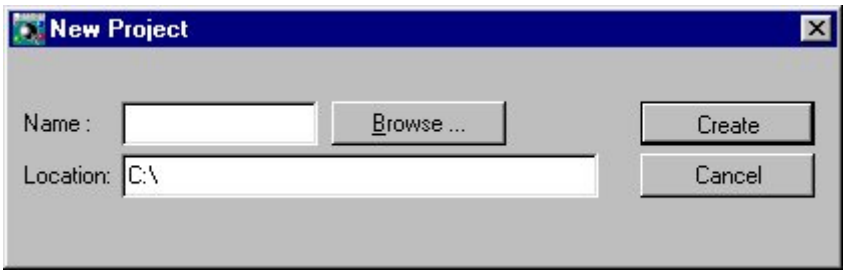


Figura 1.4. Cuadro de diálogo de Nuevo proyecto.

En este cuadro de diálogo debemos introducir el nombre del proyecto y el path donde queremos almacenar dicho proyecto. Podemos pulsar el botón Browser si queremos seleccionar el path a través del explorador de archivos.

Figura 1.5. Seleccionar la localización del nuevo proyecto.

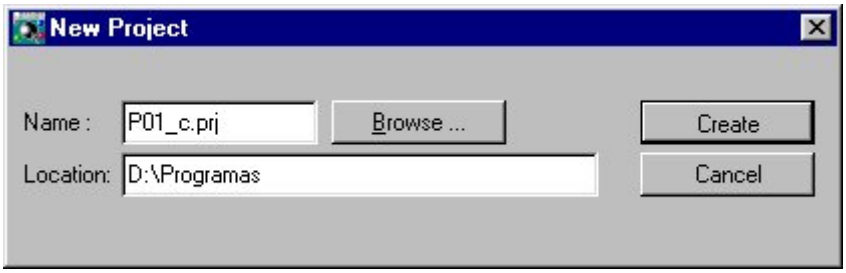


Figura 1.6. Nombre y path del nuevo proyecto.

Al crear el nuevo proyecto nos aparece en el Editor Visual la paleta del proyecto o Editor de Proyecto donde se muestran cada uno de los elementos del mismo.

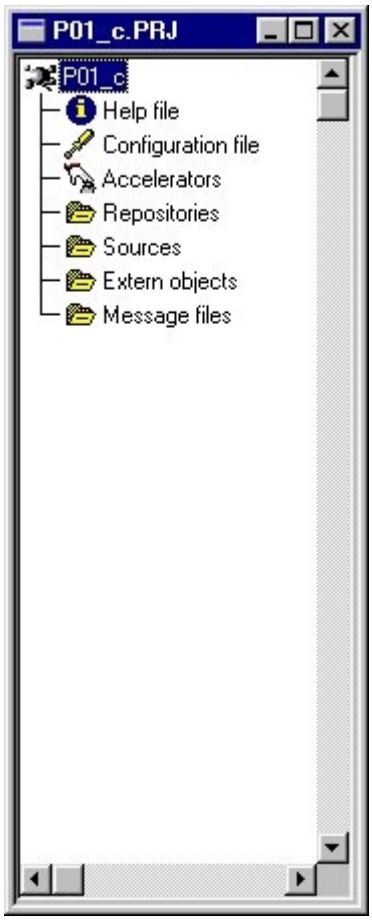


Figura 1.7. Editor del proyecto.

Ahora ya tenemos establecido el entorno apropiado para poder insertar código del COOL a un proyecto, para ello añadiremos un módulo main dentro del elemento Source del proyecto.

Un módulo main es el programa principal del proyecto, es decir en el momento que se ejecute el proyecto será donde se encuentre el código de lo que queremos que aparezca en primer lugar.

Para insertar un módulo debemos posicionar el ratón sobre el componente donde queremos insertar el módulo y pulsar el botón derecho que muestra un cuadro donde podemos elegir las acciones a realizar, nosotros seleccionaremos Add.

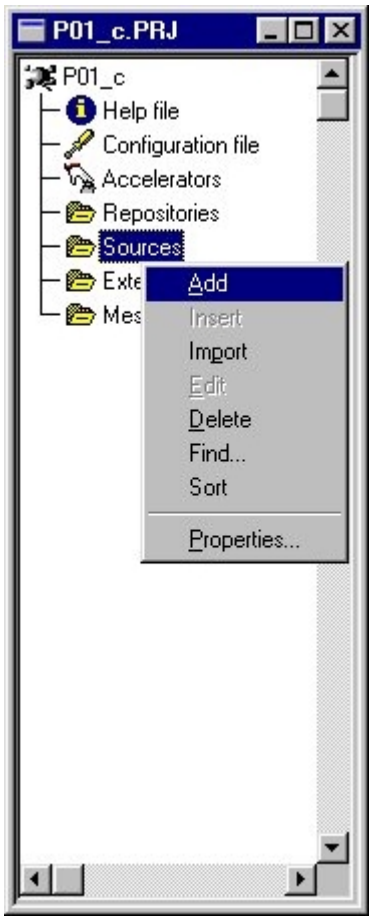


Figura 1.8. Acciones a realizar sobre Sources.

También podemos añadir un nuevo módulo al proyecto seleccionando la opcion File del menú y dentro de él la opción New. Se muestra entonces una nueva ventana que muestra el tipo de documento que podemos crear, seleccionaremos Module Document.

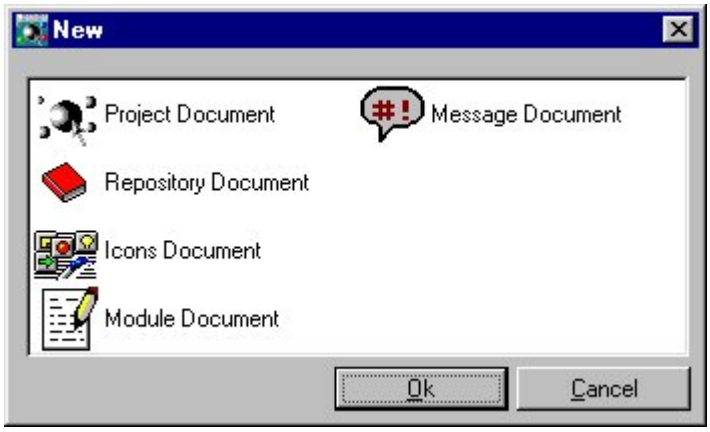


Figura 1.9. Seleccionar el tipo de documento a crear.

Despues de seleccionar esta opción de Add se muestra un cuadro de diálogo donde se nos pide el nombre del modulo, path donde se almacenará, el tipo de modulo (main o group), y si va a tratarse de un progama, un include o una librería. Nosotros seleccionaremos el nombre del programa (P01), el path y sera “main module” y “program”. La etiqueta es el nombre con el que aparece este módulo en el Editor del Proyecto, y el comentario es lo que aparecerá en la barra de estado del Editor Visual cuando nos situemos sobre este elemento.

Figura 1.10. Cuadro de diálogo de las propiedades del nodo añadido.

Al pulsar Ok en el cuadro anterior se inserta en el Editor de proyecto el nuevo modulo dentro del elemento Source:

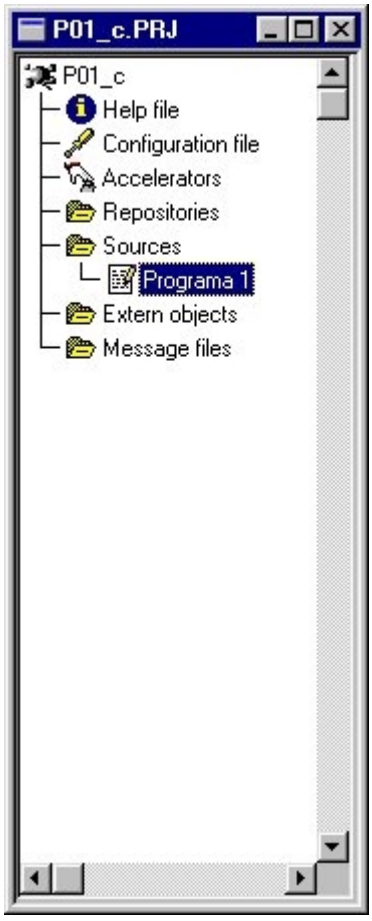


Figura 1.11. Aspecto actual del Editor de Proyecto.

Haciendo doble click sobre el modulo de programación introducido se muestran los componentes que contiene y que definen el funcionamiento de la aplicación.

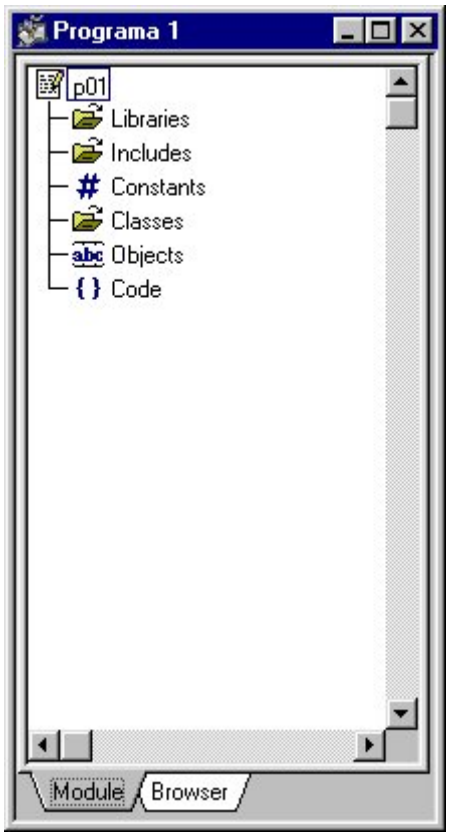


Figura 1.12. Estructura del módulo.

Estos pasos son los básicos que deberemos realizar para los programas de este tutorial.

Programa primero

Ahora vamos a introducir el código de nuestro primer programa, para ello hacemos doble click del ratón sobre la sección Code, se abre el editor de código.

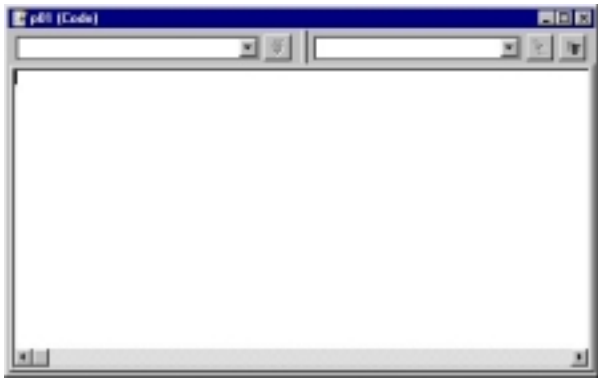


Figura 1.13. Aspecto del Editor de Código.

Este primer programa consistirá en escribir en un cuadro de diálogo el mensaje “Hola a todos/as”.

```
//Muestra por pantalla el mensaje Hola a todos/as
Main
Begin
   “Hola a todos/as”.Trace;
End
```

El método **MAIN** es un caso especial de método. Sólo se puede definir en la sección de código de un módulo. Sirve de punto de entrada para su ejecución. Al ejecutar la última instrucción de este método, se terminará la ejecución de este módulo

Begin .. end delimitan el código asociado al método Main de la sección de código. Las instrucciones que se encuentren en este bloque deben terminar en punto y coma (;).

Las comillas dobles **“ ”** indican una cadena de caracteres.

Trace es un método que en este caso pertenece a la clase Char al estar asociado a una cadena de caracteres y que muestra un cuadro de diálogo con el valor de la cadena de caracteres.

Hay que tener en cuenta en todo momento que el COOL es sensible a mayúsculas y minúsculas, y en general los métodos de las clases siempre comienzan por mayúscula y el resto de las letras en minúsculas.

Al ejecutar esta aplicación la salida que obtenemos es la siguiente:

Figura 1.14. Ejecución del Programa 1.

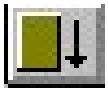
Compilación y Ejecución de Programas.

Para compilar un programa tenemos varias posibilidades:

- Ir al menú despegable Tools y seleccionar Compile
- Pulsar la tecla F8
- Pinchar en el icono:

Para ejecutar el programa tenemos, también, varias posibilidades:

- Ir al menú despegable Tools y seleccionar Execute
- Pulsar la tecla F5
- Pinchar en el icono:



Los ficheros que se generan después de realizar la compilación del programa son cuatro. Estos ficheros tienen las siguientes extensiones y significado:

Extensión	Significado
prj	Contiene la estructura y configuración del proyecto
smd	Código fuente del programa, una librería o un include
omd	Fichero objeto generado por el compilador de Cosmos. Deben encontrarse en el mismo directorio que los módulos de código fuente
pws	Estado de ejecución de un proyecto abierto, se crea cuando se guarda el proyecto por primera vez

Segundo programa

Vamos a modificar un poco el Programa 1 introduciendo un objeto de la clase char que almacenará el valor de la cadena que queremos mostrar por pantalla. La ejecución será la misma que en el programa 1:

```
//Muestra por pantalla el mensaje Hola a todos/as
Main
Objects
Begin
  s as char
End
Begin
   s = "Hola a todos/as";
   s.Trace;
End
```

Se ha introducido un nuevo bloque dentro del main denominada **Objects** donde se definirán todos los objetos que intervendrán en el código de la aplicación y se encuentra delimitados por **Begin ... end**.

Los objetos definidos en este bloque son los atributos locales a este método.

Dentro de la sección Objects la definición de los objetos no debe terminarse con punto y coma (;) porque genera un error.

Para definir un objeto de las clases predefinidas o creadas por el usuario después del nombre que pongamos al objeto debemos añadir la partícula **"as"** seguida del nombre de la clase, en nuestro caso char por tratarse de una cadena de caracteres.

La asignación de valor a un objeto se realiza mediante el operador **"="** y luego el método Trace se aplica sobre el objeto al que se ha dado valor.

Para ilustrar la importancia de los puntos y comas (;) y su aplicación diferente en cada una de las secciones, haremos uso de una utilidad que posee el Editor de Código de Cosmos.

En un principio introducimos el código del Programa segundo, cometiendo un error: poner un punto y coma (;) despues de la definición del objeto **s** de la clase **char**:

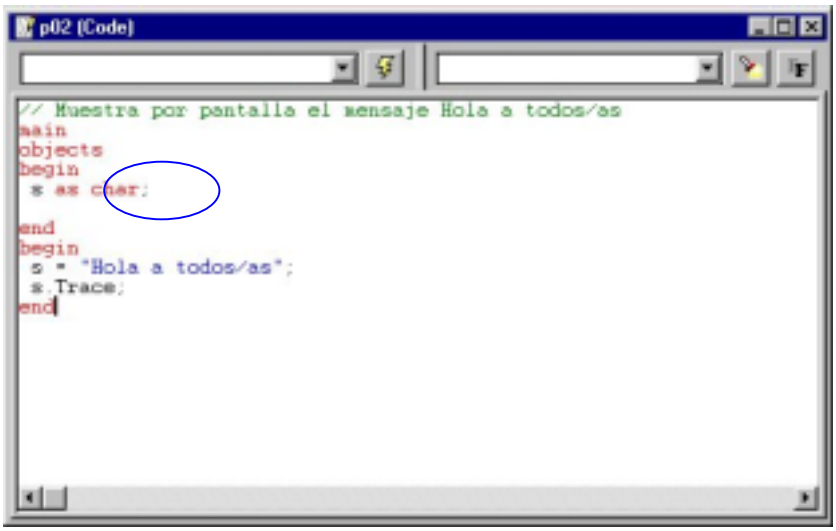


Figura 1.15. Código del programa segundo con un error.

Podemos analizar el código que hemos escrito haciendo click sobre el icono  . Cuando existe un error se muestra el tipo de este en el campo de edición despegable que se encuentra en la parte superior izquierda del editor.

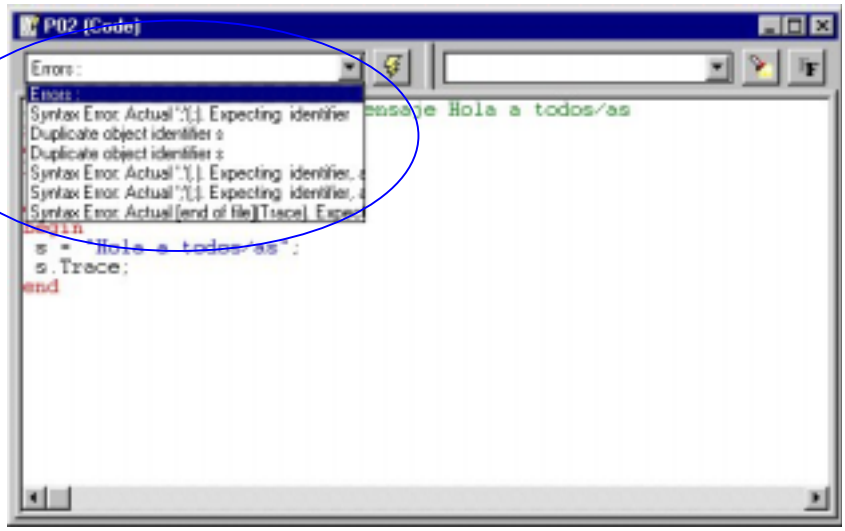


Figura 1.16. Mensaje con el error encontrado en el código analizado.

Después de corregir el error y pulsar el icono para analizar el código nuevamente, obtenemos los métodos definidos dentro del programa 2, en este caso únicamente el main.

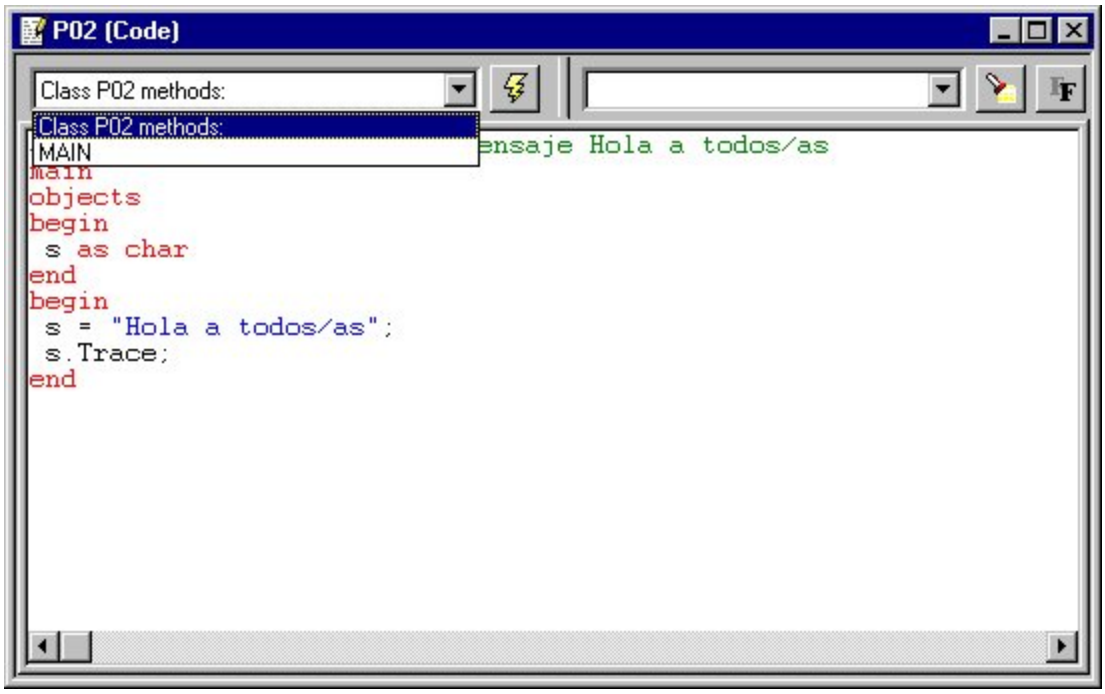


Figura 1.17. Métodos del programa segundo.

Tercer programa

En este programa vamos a realizar el cálculo del área de un triángulo, cuya base y altura hemos introducido en el propio programa.

```
//Calcula el área de un triángulo
Main
Objects
Begin
  base as integer
  altura as integer
  area as decimal
  salida as char
End
Begin
   base = 7:
   altura =3;
   area = (base*altura)/2;
   salida =area.Using(0);
   salida + =   "  es el area del triangulo de base 7 y altura 3";
   salida.Trace;
End
```

Si se mira la Guía de Referencia de Cosmos podemos comprobar que Integer y Decimal son clases que derivan de la clase Numeric que a su vez deriva de la clase Simple.

Los operadores que tienen estas clases estan recogidas en dicha Guía. En este programa se han usados los siguientes operadores de la clase Numeric:

* (multiplicación)	Multiplica dos objetos de tipo Numeric y sus clases derivadas.
/ (división)	Divide un valor de tipo integer entre un valor entero, si el resultado se almacena en un objeto de tipo decimal el resultado de la división se convierte a un número real; el resultado se trunca si se almacena en un objeto de tipo integer. En general divide dos objetos de tipo Numeric.

El método de la clase Numeric y por tanto de sus derivadas que hemos empleado en este ejemplo es un conversor:

Using(mascara)	Convierte un objeto de la clase Numeric y derivadas en un objeto de la clase Char. Máscara es un valor entero en el que se especifica la forma en la que queremos que aparezca la conversión al mostrar el dígito convertido. La máscara 0 devuelve el valor del objeto Char con la misma estructura que el objeto Numeric convertido.

El operador que hemos utilizado de la clase Char es el operador asignación-suma:

+= (suma)	Concatena dos objetos de tipo Char o como en este caso, un objeto de tipo char y una cadena expresada entre comillas. Por ejemplo en este caso sería lo mismo que escribir: salida = salida + " es el area del triangulo ....”;

La salida que obtenemos por pantalla al ejecutar este programa es:

Figura 1.18. Resultado de la ejecución del Programa 3.

Cuarto programa

Vamos a introducir una nueva sección dentro de nuestro programa, en la cual se definirán las constantes a utilizar en la aplicación. Para esto modificaremos el tercer programa definiendo la base y la altura como constantes en lugar de objetos de tipo integer.

Para poder definir constantes dentro de un módulo se hace doble click en el componente Constants. Aparece una nueva ventana que en un principio esta vacia y es la que contendrá las constantes del módulo. Pulsando el botón derecho del ratón sobre esta pantalla aparece un cuadro donde se elegirá la única opción posible, en un principio, que Add.

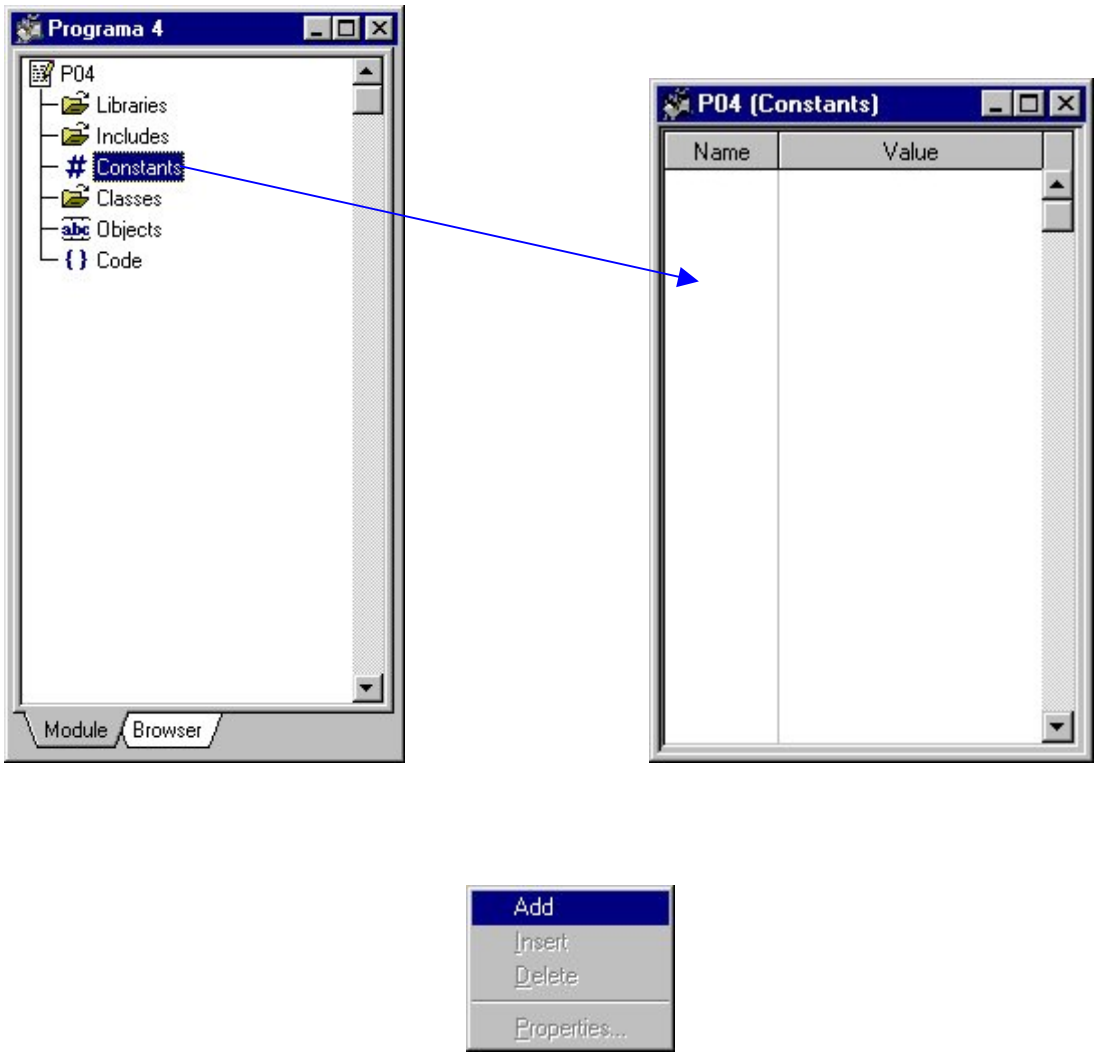


Figura 1.19. Añadir constantes a un módulo.

En ese momento aparece un cuadro de diálogo donde se debe introducir tanto el nombre de la constante como el valor que se desea asignar a la misma durante la ejecución del módulo.

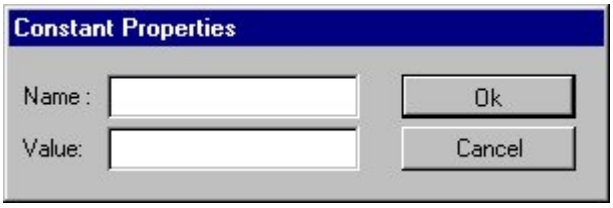


Figura 1.20. Cuadro de diálogo de las propiedades de una constante.

Después de introducir como constantes los dos objetos de tipo integer del programa tercero, la sección de Constants dentro del módulo del programa cuarto queda:

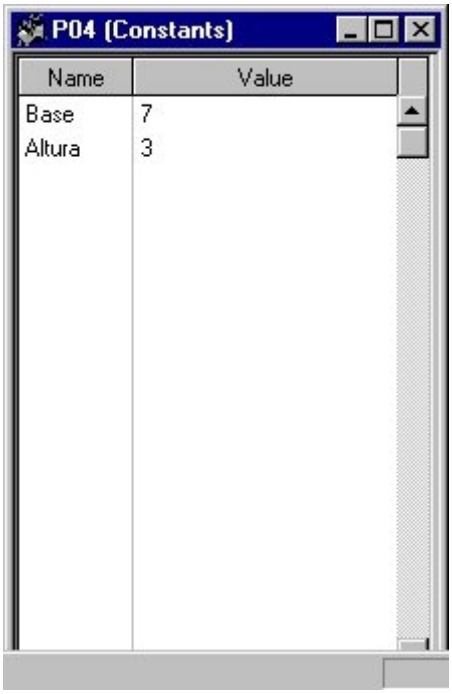


Figura 1.21. Constantes del programa cuarto.

El código de este programa después de los cambios que hemos efectuado respecto al tercer programa es:

```
//Calcula el área de un triángulo con constantes
Main
Objects
Begin
  area as decimal
  salida as char
End
Begin
   area = (Base*Altura)/2;
   salida =area.Using(0);
   salida + =   "  es el area del triangulo de base 7 y altura 3";
   salida.Trace;
End
```

La salida de este programa es similar a la del anterior. Hay que tener en cuenta las letras mayúsculas y minúsculas en el momento de utilizar las constantes, ya que deben coincidir en el mismo orden que en la declaración dentro de la sección Constants. Así este programa habría dado un error de compilación si en lugar de escribir **Base** hubiésemos escrito **base**, ya que en la declaración de constantes tiene su primera letra en mayúsculas.

Depuración de programas

En este momento veremos una utilidad de Multibase COSMOS: la depuración de programas. Para ello usaremos el programa cuarto.

En primer lugar debemos seleccionar el módulo que deseamos depurar, en nuestro caso seleccionamos P04 que representa al módulo del programa cuarto. Posteriormente existen tres formas de ejecutar el depurador de programas:

1. Mediante la opción Debug del menú desplegable Tools.
2. Con las teclas Ctrl+F5
3. Mediante el icono 

Aparece en este momento la ventana del depurador con el programa cuarto en la ventana principal del mismo.

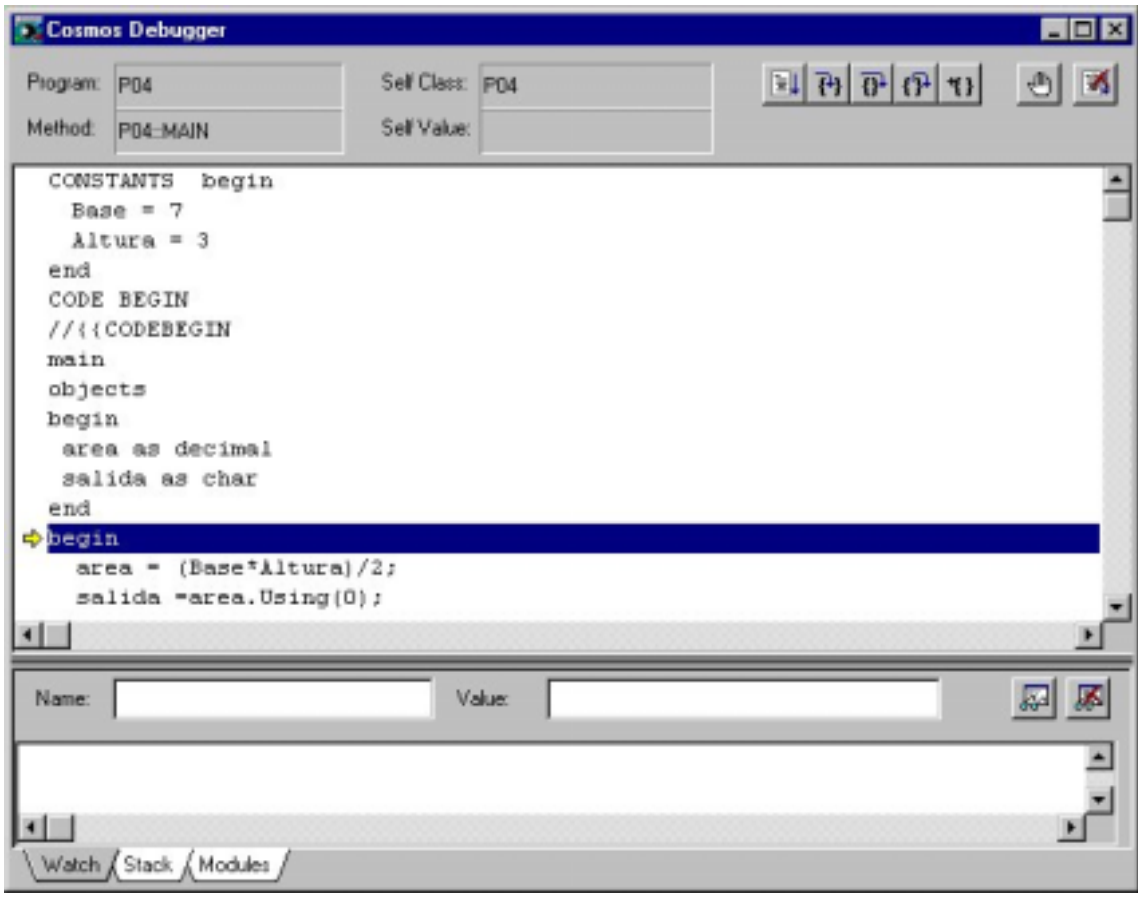


Figura 1.22. Aspecto del Depurador.

Podemos añadir en la parte inferior (Watch List) los objetos de los que queremos saber su valor a lo largo de la depuración del programa.

Para ello debemos rellenamos el campo de **Name** con el nombre del objeto que quermos observar y posteriormente hacemos click sobre el icono  . Añadiremos los objetos **base, altura, area y salida.**

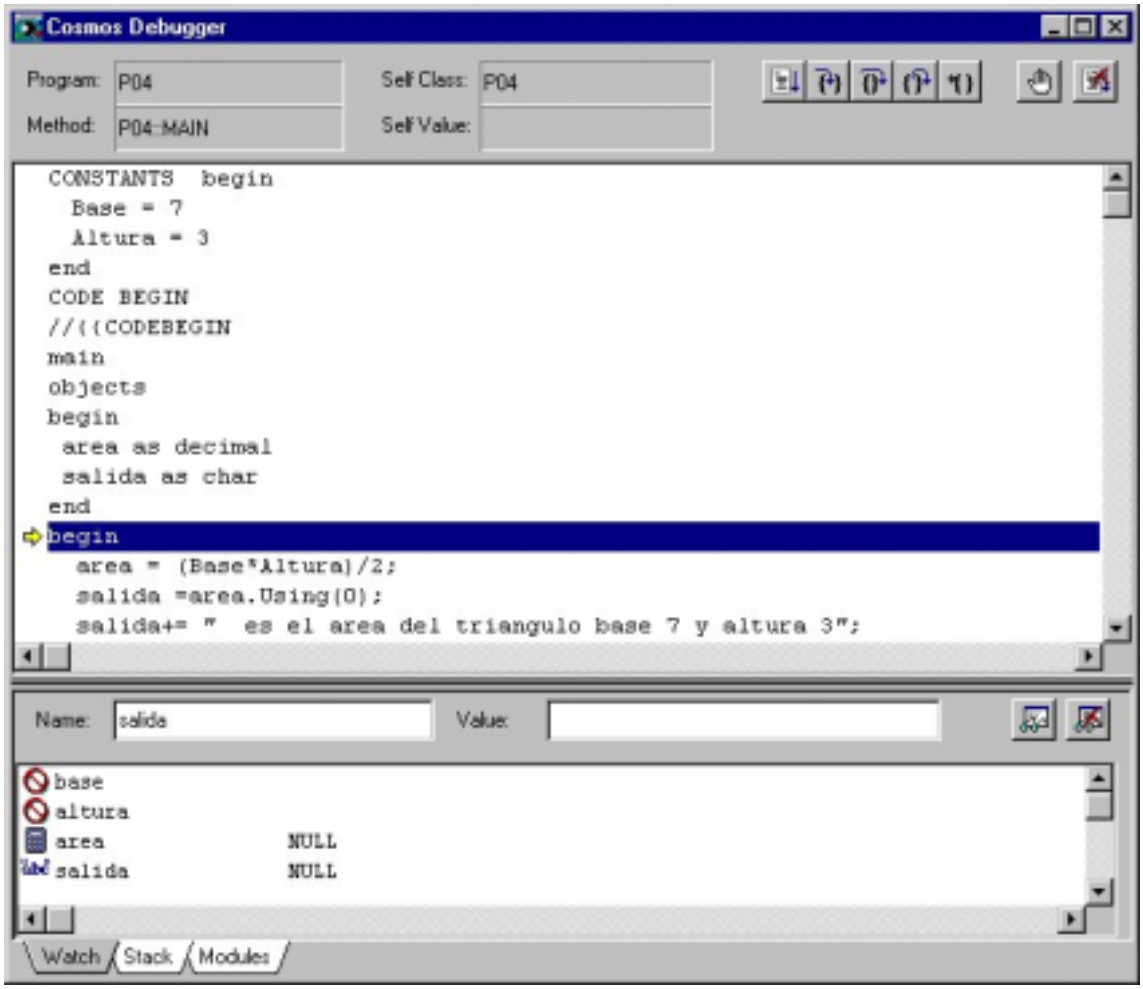


Figura 1.23. Añadir objetos a Watch List

Para ir siguiendo la ejecución del programa y comprobar como varían de valor los objetos debemos utilizar los iconos de la zona superior derecha del depurador.

La solapa Stack (pila) de la parte inferior del depurador es interesante para visualizar la pila en tiempo de ejecución. La pila se utiliza para almacenar los parametros y resultados de los métodos que se estudiarán en el capítulo 5.

La solapa Modules (módulos) de la parte inferior del depurador indica el módulo que se esta ejecutando y quien lo ejecuta. Los módulos se estudiarán en el capítulo 8.

De esta forma cuando un programa compila sin errores y en cambio cuando lo ejecutamos no obtenemos el resultado deseado podemos comprobar paso a paso su comportamiento y así encontrar el error, posiblemente de lógica. Para saber todas las posibles opciones del Depurador véase la Guía de Referencia de Multibase COSMOS.

Capítulo 2

Clase simple

Introducción

En este capítulo veremos cada una de las clases que almacenan los tipos de datos simples que posee COSMOS. Todas estas clases son derivadas de la clase virtual **Simple**.

Los objetos de tipo simple contienen los tipos elementales del lenguaje (enteros, caracteres, fechas,..).

De cada una de las clases que derivan de la clase Simple, excepto de las clases Enum y EnumSet, veremos ejemplos de sus operadores, métodos y conversores. El significado de cada uno de estos conceptos es el siguiente:

Operadores

Un método operador es un método que se referencia por un símbolo matemático o bien por el identificador de éste. Existen operadores ariméticos, relacionales y de asignación.

Métodos

En Cosmos se denomina método al conjunto de acciones y servicios que ofrece una clase. Un método se implementa en una clase, y determina cómo tiene que actuar el objeto cuando recibe un mensaje. Un método puede también enviar mensajes a otros objetos solicitando una acción o información. Los métodos en Cosmos son un grupo de instrucciones orientados a realizar una tarea específica y que se referencia mediante un nombre simbólico.

Conversor

Al aplicar un método conversor sobre un objeto de una clase, da como resultado otro objeto de una clase diferente a la primera. A este proceso se le denomina conversion. El propósito de una conversión es obtener un objeto de una clase diferente con un valor considerado como equivalente.

Clase Numeric

Esta clase es virtual, de ella derivan las clases que almacenan los tipos numéricos de datos. Una clase virtual, en COSMOS, se caracteriza por:

El usuario no puede crear una clase derivada directamente de una clase virtual. No se pueden instanciar.

Por tanto, esta clase únicamente sirve de base de otras clases. La idea es disponer de un mecanismo que soporte la noción del concepto general de número del cual se utilizaran variantes concretas. De estas variantes estudiaremos sus particulariades a continuación y son Smallint, Integer y Decimal.

La clase Numeric posee los siguientes operadores aritméticos, operadores relacionales, métodos y conversores que pueden ser utilizados por cualquiera de las clases que derivan de ella.

Operadores aritméticos

Operador	Funcionalidad
+ (suma)	Este operador suma dos objetos de tipo Numeric
- (resta)	Este operador resta dos objetos de tipo Numeric
* (multiplicación)	Este operador multiplica dos objetos de tipo Numeric
/ (división)	Este operador divide dos objetos de tipo Numeric
% (módulo)	Este operador calcula el resto de la división de dos objetos numéricos enteros
** (exponenciación)	Este operador exponenciación eleva el valor de un objeto de tipo Numeric a una potencia indicada por otro
- (signo)	Operador de inversión de signo. Los objetos numéricos positivos se convierten en negativos y viceversa

Operadores relacionales

Operador	Funcionalidad
== (igualdad)	Este operador de igualdad chequea si dos objetos numéricos tienen igual valor
(distinto)	Este operador de igualdad chequea si dos objetos numéricos no tienen igual valor
(mayor)	Este operador chequea si un objeto de tipo Numeric es mayor que otro
= (mayor o igual)	Este operador chequea si un objeto de tipo Numeric es mayor o igual que otro
(menor)	Este operador chequea si un objeto de tipo Numeric es menor que otro
= (menor o igual)	Este operador chequea si un objeto de tipo Numeric es menor o igual que otro
BETWEEN	Este operador chequea si un objeto de tipo Numeric esta comprendido o no en el rango dado por otros dos
IN	Este operador chequea si un objeto Numeric es igual o no a alguno de los valores incluidos en una lista de valores

Métodos

Método	Funcionalidad
Using	Este método realiza una conversión formateada de un objeto de tipo Numeric a Char
Trace	Este método muestra en una ventana el valor del objeto

Conversores

La clase Numeric tiene conversores a las siguientes clases:

Boolean
Smallint
Integer
Decimal
Char

Clase Smallint

Esta clase es instanciable y permite crear objetos que son números enteros. El rango de valores numéricos que admite está entre –32767 y +32767.

Esta clase dispone de operadores aritméticos de asignación, operadores aritméticos, métodos y conversores

Operadores aritméticos de asignación

Operador	Funcionalidad
= (asignación)	Este método permite asignar un valor a un objeto Smallint. Retorna el propio objeto
+= (suma)	Este método suma al valor de un objeto de tipo Smallint el valor de una expresión del mismo tipo. Retorna el primer operando, es decir, el objeto sobre el que se aplica el operador
-= (resta)	Este método resta al valor de un objeto de tipo Smallint el valor de una expresión del mismo tipo. Retorna el primer operando, es decir, el objeto sobre el que se aplica el operador
*= (multiplicación)	Este método multiplica el valor de un objeto de tipo Smallint el valor de una expresión del mismo tipo. Retorna el primer operando, es decir, el objeto sobre el que se aplica el operador
/= (división)	Este método divide el valor de un objeto de tipo Smallint entre el valor de una expresión del mismo tipo. Retorna el primer operando, es decir, el objeto sobre el que se aplica el operador
%= (módulo)	Este método calcula el resto de la división del valor del objeto entre el valor de una expresión del mismo tipo. Retorna el primer operando, es decir, el objeto sobre el que se aplica el operador

```
// Ejemplo de los operadores aritméticos de asignación
// de la clase Smallint
main
objects
begin
  s as smallint
end
begin
 //asignación
  s = 3;
 //operadores aritméticos de asignación
  s += 2; //es similar a la operación s=s+2
  s -=1;
  s *=5;
  s /= 4;
  (s %= 5).Trace;
end
```

Hemos definido un objeto de tipo Smallint sobre el que hemos aplicados todos los operadores aritméticos de asignación de esta clase.

La secuencia de las operaciones es la siguiente:

Operación	Valor de S
Asignar	3
Sumar 2	5
Restar 1	4
Multiplicar por 5	20
Dividir entre 4	5
Modulo de 5	0

Después de la ejecución del programa, por pantalla obtenemos como resultado el 0 porque únicamente hemos aplicado el método Trace de la clase Numeric sobre la operación del cálculo del modulo, en el momento que s tiene un valor de 5.

Existe una diferencia entre encerrar entre paréntesis la expresión que precede al método Trace o no hacerlo:

- Si se emplean los paréntesis Trace actúa sobre la expresión global
- Cuando no se utilizan se aplica directamente sobre el elemento anterior.

Operadores aritméticos

++ (incremento)	El operador incremento suma una unidad al operando sobre el que se aplica
-- (decremento)	El operador decremento resta una unidad al operando sobre el que se aplica

```
// Ejemplo de los operadores aritméticos
// de la clase Smallint
main
objects
begin
  s as smallint
end
begin
  s = 5;
  ++s;
  --s;
  s.Trace;
end
```

s toma un valor de 5 el cual es incrementado en una unidad y luego decrementado por lo que la salida del programa será 5.

Métodos

Método	Funcionalidad
Character	Devuelve el carácter correspondiente a la posición en la tabla ASCII que corresponde con el valor que tiene el objeto
MonthName	Este método devuelve el nombre completo del mes correspondiente al valor del objeto. El valor 1 retorna “January” y el valor 12 retorna “December”
MonthName3	Este método devuelve el nombre del mes con 3 letras, correspondiente al valor del objeto. El valor 1 retorna “Jan” y el valor 12 retorna “Dec”
WeekDayName	Este método devuelve el nombre completo del día de la semana correspondiente al valor del objeto. El valor 0 retorna “Sunday” y el valor 6 retorna “Saturday”
WeekDayName3	Este método devuelve el nombre de dia de la semana con 3 letras, correspondiente al valor del objeto. El valor 0 retorna “Sun” y el valor 6 retorna “Sat”

```
// Ejemplo de los métodos
// Clase Smallint
main
objects
begin
  s as smallint
end
begin
  s=64;
  s.Character.Trace;
  s=3;
  s.Trace;
  s.MonthName.Trace;
  s.MonthName3.Trace;
  s.WeekDayName.Trace;
  s.WeekDayName3.Trace;
end
```

Conversor

La clase Smallint posee conversor a la clase Char.

```
// Ejemplo del conversor
// Clase Smallint
main
objects
begin
  s as smallint
  c as char
end
begin
  s = 27;
  c = 'Mi edad es '+ s + 'años';
  c.Trace;
end
```

Al efectuar una asignación sobre un objeto de tipo carácter de un Smallint, este se convierte automáticamente en un objeto de tipo Char.

Clase Integer

Es una clase instanciable y permite crear objetos que son números enteros, en un rango comprendido entre -2.147.483.647 y +2.147.483.647, ambos incluidos.

Esta clase posee operadores aritméticos de asignación, operadores aritméticos y un conversor, además de los operadores heredados de la clase Numeric.

Operadores aritméticos de asignación

Operador	Funcionalidad
= (asignación)	Este método permite asignar un valor a un objeto Integer. Retorna el propio objeto
+= (suma)	Este método suma al valor de un objeto de tipo Integer el valor de una expresión del mismo tipo. Retorna el primer operando, es decir, el objeto sobre el que se aplica el operador
-= (resta)	Este método resta al valor de un objeto de tipo Integer el valor de una expresión del mismo tipo. Retorna el primer operando, es decir, el objeto sobre el que se aplica el operador
*= (multiplicación)	Este método multiplica el valor de un objeto de tipo Integer el valor de una expresión del mismo tipo. Retorna el primer operando, es decir, el objeto sobre el que se aplica el operador
/= (división)	Este método divide el valor de un objeto de tipo Integer entre el valor de una expresión del mismo tipo. Retorna el primer operando, es decir, el objeto sobre el que se aplica el operador
%= (módulo)	Este método calcula el resto de la división del valor del objeto entre el valor de una expresión del mismo tipo. Retorna el primer operando, es decir, el objeto sobre el que se aplica el operador

Operadores aritméticos

Operador	Funcionalidad
++ (incremento)	El operador incremento suma una unidad al operando sobre el que se aplica
-- (decremento)	El operador decremento resta una unidad al operando sobre el que se aplica

Conversor

La clase Integer tiene conversor a la clase Char. Su uso es similar a lo visto en la clase Smallint.

```
//Ejemplo de los métodos y operadores de la clase
// Integer
// que deriva de la clase virtual Numeric
main
objects
begin
  i1 i2 as integer
end
begin
 //asignación
  i1 = 39999;
  i2 = 1000;
//operadores aritméticos de asignación
  i1 += i2;
  i2 -=i1;
  i1 =i2; //es similar a i1 = i1*i2
  i1 /= i2;
  i1 %= i2;
//operadores aritméticos
  ++i1;
  --i1; //es similar a i1=i1-1
 end
```

Clase Decimal

Esta clase deriva de la clase Numeric, es una clase instanciable. Una clase DECIMAL [(m[,n])] permite crear objetos que son números decimales de coma flotante con un total de m dígitos significativos (precisión) y n dígitos a la derecha de la coma decimal (escala). m puede ser como máximo menor o igual a 32 (m = 32) y n menor o igual que m. Cuando se asignan valores a m y n, la variable decimal tendrá punto aritmético fijo. El segundo parámetro n es opcional, y si se omite se tratará como un decimal de coma flotante. Un elemento decimal(m) tiene una precisión m y un rango de valor absoluto entre 10^{-130} y 10^{125}. En caso de no especificar longitud a un DECIMAL, éste será tratado como decimal(16).

De esta clase deriva la clase Money. Un “Money” es un “Decimal” que tiene un número de decimales por defecto igual a 2. Esta clase es instanciable.

Operadores aritméticos de asignación

Operador	Funcionalidad
= (asignación)	Este método permite asignar un valor a un objeto Decimal. Retorna el propio objeto
+= (suma)	Este método suma al valor de un objeto de tipo Decimal el valor de una expresión del mismo tipo. Retorna el primer operando, es decir, el objeto sobre el que se aplica el operador
-= (resta)	Este método resta al valor de un objeto de tipo Decimal el valor de una expresión del mismo tipo. Retorna el primer operando, es decir, el objeto sobre el que se aplica el operador
*= (multiplicación)	Este método multiplica el valor de un objeto de tipo Decimal el valor de una expresión del mismo tipo. Retorna el primer operando, es decir, el objeto sobre el que se aplica el operador
/= (división)	Este método divide el valor de un objeto de tipo Decimal entre el valor de una expresión del mismo tipo. Retorna el primer operando, es decir, el objeto sobre el que se aplica el operador
%= (módulo)	Este método calcula el resto de la división del valor del objeto entre el valor de una expresión del mismo tipo. Retorna el primer operando, es decir, el objeto sobre el que se aplica el operador

Operadores aritméticos

Operador	Funcionalidad
++ (incremento)	El operador incremento suma una unidad al operando sobre el que se aplica
-- (decremento)	El operador decremento resta una unidad al operando sobre el que se aplica
- (signo)	Operador de inversión de signo. Los objetos Decimal positivos se convierten en negativos y viceversa

Métodos

Método	Funcionalidad
Round	Redondea el valor de un objeto Decimal con la precisión indicada
Truncate	Trunca el valor del objeto Decimal a la precisión indicada

Conversor

La clase Decimal tiene conversor a la clase Char

Clase Money

Una clase MONEY [(m[,n])] permite crear objetos que son números decimales de coma flotante con un total de m dígitos significativos (precisión) y n dígitos a la derecha de la coma decimal (escala), igual al tipo de dato DECIMAL. Los objetos de esta clase representan cantidades referentes a unidades monetarias.

Un Money es un Decimal que tiene un número de decimales por defecto igual a 2.

```
// Ejemplo de los métodos y operadores de la clase
// Decimal
// que deriva de la clase virtual Numeric
// Money deriva de Decimal, tiene un número de decimales
// por defecto igual a 2.
main
objects
begin
  d1 d2 as decimal
  c as char
end
begin
 //asignación
  d1 = 3.9999;
  d2 = 1.001;
 //operadores aritmeticos de asignación
  d1 += d2;
  d1 -= d2;
  d1 *= d2;
  d1 /= d2;
  d1 %= d2;
 //operadores aritmeticos
  ++d1;
  --d1;
  -d1;
 //métodos de la clase Decimal
  d1 = 3.9999;
  d2 = 1.0001;
  d1.Round(2).Trace; //muestra por pantalla 4 porque redondea d1
  d2.Truncate(2).Trace; //trunca d2 y muestra por pantalla 1.0
 //conversor a char
  c = 'Tengo '+ d1 + ' euros'; //muestra como un char la concatenacion
                               // de las cadenas y el decimal d1
end
```

Clase Char

Esta clase deriva de la clase Simple, es una clase instanciable. Una clase char(n) permite crear objetos que son una cadena de caracteres alfanuméricos de longitud máxima n. Siendo “n” un número entre 1 y 32.767. También se pueden definir objetos o clases char sin indicar la longitud, en este caso se entiende que es un char de longitud indefinida y se irá reservando memoria dinámicamente.

Operadores

Operador	Funcionalidad
+ (suma)	Este operador concatena dos objetos de tipo alfanumérico
= (asignación)	Este operador permite asignar un valor a un objeto Char
+= (suma)	Este operador concatena una expresión alfanumérica (Char) a un objeto de tipo Char. Modifica el objeto
[ ] (substring)	Devuelve la cadena de caracteres de un objeto Char comenzando en un carácter dado y con una longitud dada

Operadores relacionales

Operador	Funcionalidad
== (igualdad)	Este operador de igualdad chequea si dos objetos Char son iguales
<> (distinto)	Este operador chequea si dos objetos Char son distintos
> (mayor)	Este operador chequea si un objeto Char es mayor que otro
>= (mayor o igual)	Este operador chequea si un objeto Char es mayor o igual que otro
< (menor)	Este operador chequea si un objeto Char es menor que otro
<= (menor o igual)	Este operador chequea si un objeto Char es menor o igual que otro
BETWEEN	Este operador chequea si un objeto Char esta comprendido o no en el rango dado por otros dos
IN	Este operador chequea si un objeto Char es igual o no a uno de los valores incluidos en una lista de valores
LIKE	Condición de comparación de objetos Char. Devuelve TRUE o en caso de que el resultado de una expresión se ajuste o no un patrón
MATCHES	Condición de comparación de objetos Char. Devuelve TRUE o FALSE dependiendo de que el resultado de una expresión se ajuste o no a un patrón

Métodos

Método	Funcionalidad
AnsiToOem	Convierten el valor de un objeto Char, del conjunto de caracteres ANSI al conjunto de caracteres OEM
Ascii	Devuelve el código ASCII del primer carácter de un objeto Char
Count	Este método devuelve el número de veces que un patrón se encuentra en un objeto Char
DirName	Devuelve de un objeto Char que representa un path completo su valor eliminando el último elemento del path.
FileName	Devuelve la parte correspondiente al último elemento del path de un objeto Char que representa un path completo
GetWord	Devuelve la enésima palabra de un objeto Char
Length	Devuelve la longitud del valor de un objeto Char
L ocate	Este método busca un literal en otro
Lowcase	Devuelve el objeto Char (expresión alfanumérica) convertido a minúsculas
LTrim	Elimina los blancos situados a la izquierda en el objeto Char (expresión alfanumérica) y retorna dicho objeto modificado
NumWords	Este método cuenta las palabras de un objeto Char
OemToAnsi	Convierte el valor de un objeto Char, del conjunto de caracteres OEM al conjunto de caracteres ANSI

Método	Funcionalidad
Replace	Este método sustituye un literal por otro en un objeto Char
RTrim	Elimina los blancos situados a la derecha en el objeto Char (expresión alfanumérica) y retorna dicho objeto modificado
StrRepeat	Este método repite el valor de un objeto Char, tantas veces como se indique. La longitud máxima del resultado es de 512 caracteres
SubString	Devuelve la cadena de caracteres de un objeto Char comenzando en un carácter dado y con una longitud dada
Trace	Este método muestra en una ventana el valor que tiene asignado el objeto
Trim	Elimina los blancos situados a la derecha y a la izquierda del objeto Char (expresión alfanumérica) y reduce cada grupo de blancos entre palabras a uno solo. Retorna el objeto modificado
Upcase	Convierte a mayúsculas el valor de un objeto Char (expresión alfanumérica) . Retorna el objeto modificado
VarLength	Devuelve la longitud con que ha sido definido un objeto Char (espacio en Bytes). Para una clase Char(20) devolverá siempre 20. Para una clase Char de longitud indefinida devolverá el espacio en Bytes que ocupa en ese momento el objeto

Conversores

La clase Char tiene conversores a las siguientes clases:

- Time
- Date
- Boolean
- Smallint
- Integer
- Decimal
- Numeric
- Money

```
//Métodos y operadores de la Clase Char
//derivada de la clase virtual Simple
main
objects
begin
  c1 c2 total saludo as char
  fichero as char
end
begin
  //asignacion
  c1 = "Hola";
  c2 = " a todos/as";
  //operadores
  total = c1+c2;
  saludo = " ¿que tal estáis?";
  total += saludo; //total = “Hola a todos/as ¿qué tal estáis?”
  saludo[3,8].Trace; // muestra “que tal”
  //métodos
  total=c1+c2;
  c1.Ascii.Trace; //muestra el codigo Ascii de H
  total.Count("a").Trace; //la salida es 3 porque son el numero de a
  total.Length.Trace; //15 es el número de caracteres del objeto total
  fichero = "c:\cosmos\ejemplos\fichero";
  fichero.DirName.Trace; //muestra c:\cosmos\ejemplos
  fichero.FileName.Trace; //muestra fichero
end
```

Clase Date

Esta clase deriva de la clase Simple, es una clase instanciable. Esta clase permite crear objetos que son valores de fecha.

Los tipos DATE son introducidos como una secuencia día, mes y año, con caracteres numéricos. El día puede representarse como el día del mes (1 ó 01, 2 ó 02, etc.). El mes se representa como un número (Enero: 1, Febrero: 2, etc.). El año se representa como un número de cuatro dígitos (0001 a 9999). En caso de mostrar dos dígitos para el año, se asume que el año es 19yy si la fecha del sistema es anterior al año 2000 y 20yy en caso contrario.

Operadores aritméticos

Operador	Funcionalidad
+ (suma)	Suma a una fecha un número indicado de días
- (resta)	Resta a una fecha un número indicado de días
++ (incremento)	El operador incremento suma un día a la fecha sobre la que se aplica
-- (decremento)	El operador decremento resta un día a la fecha sobre la que se aplica

Operadores aritméticos de asignación

Operador	Funcionalidad
= (asignación)	Este método permite asignar un valor a un objeto Date
+= (suma)	Este método suma un número indicado de días a la fecha sobre la que se aplica
-= (resta)	Este método resta un número indicado de días a la fecha sobre la que se aplica

Operadores relacionales

Operador	Funcionalidad
== (igualdad)	Este operador de igualdad chequea si dos objetos de tipo Date son iguales
<> (distinto)	Este operador chequea si dos objetos de tipo Date son distintos
> (mayor)	Este operador chequea si un objeto de tipo Date es mayor que otro
>= (mayor o igual)	Este operador chequea si un objeto de tipo Date es mayor o igual que otro
< (menor)	Este operador chequea si un objeto de tipo Date es menor que otro
<= (menor o igual)	Este operador chequea si un objeto de tipo Date es menor o igual que otro
BETWEEN	Este operador chequea si un objeto de tipo Date esta comprendido o no en el rango dado por otros dos
IN	Este operador chequea si un objeto de tipo Date es igual o no a alguno de los valores incluidos en una lista de valores

Métodos

Métodos	Funcionalidad
Day	Este método devuelve el día del mes correspondiente a una fecha determinada
DaysTo	Este método calcula el número de días comprendidos entre dos fechas
Month	Este método devuelve el mes en el que se encuentra una fecha determinada
Using	Este método realiza una conversión a Char formateada de expresiones de tipo fecha (Date)
Trace	Este método muestra en una ventana el valor que tiene asignado el objeto
WeekDay	Este método devuelve el día de la semana de una fecha determinada
Year	Este método devuelve el año en el que se encuentra una fecha determinada

Conversor

La clase Date tiene conversor a la clase Char

```
//Ejemplo del uso de los operadores y métodos
//Clase Date
main
objects
begin
 d1 d2 as Date
end
begin
 //operadores de asignación
 d1= "05/05/1971";
 d2= "10/05/1971";
 d1.Trace; //muestra el contenido del objeto d1
 //operadores aritméticos de asignación
 d1 +=3; // suma tres días al objeto d1
 d1.Trace; //muestra el contenido actual de d1 = 08/05/1971
 //métodos
 d1.Day.Trace;
 d1.Month.Trace;
 d1.WeekDay.Trace;
 d1.Year.Trace;
 (d1.DaysTo(d2)).Trace; //muestra los días que existen entre d1 y d2
end
```

Clase Time

Esta clase deriva de la clase Simple, es una clase instanciable. Esta clase permite crear objetos que son valores horarios desde 00:00:01 a 24:00:00.

A los objetos de esta clase se les asigna un valor como una secuencia de hora, minutos y segundos, sin separador entre ellos (tampoco espacios en blanco). Asimismo, permite ordenaciones, operaciones y comparaciones horarias entre dos objetos de la clase TIME.

Operadores aritméticos

Operador	Funcionalidad
+ (suma)	Suma a una hora un número determinado de segundos
- (resta)	Resta a una hora un número determinado de segundos
++ (incremento)	El operador incremento suma un segundo a la hora sobre la que se aplica
-- (decremento)	El operador decremento resta un segundo a la hora sobre la que se aplica

Operadores aritméticos de asignación

Operador	Funcionalidad
= (asignación)	Este método permite asignar un valor a un objeto Time
+= (suma)	Este método suma un número determinado de segundos a la hora sobre la que se aplica
-= (resta)	Este método suma un número determinado de segundos a la hora sobre la que se aplica

Operadores relacionales

Operador	Funcionalidad
== (igualdad)	Este operador de igualdad chequea si dos objetos de tipo Time son iguales
<> (distinto)	Este operador chequea si dos objetos de tipo Time son distintos
> (mayor)	Este operador chequea si un objeto de tipo Time es mayor que otro
>= (mayor o igual)	Este operador chequea si un objeto de tipo Time es mayor o igual que otro
< (menor)	Este operador chequea si un objeto de tipo Time es menor que otro
<= (menor o igual)	Este operador chequea si un objeto de tipo Time es menor o igual que otro
BETWEEN	Este operador chequea si un objeto de tipo Time esta comprendido o no en el rango dado por otros dos
IN	Este operador chequea si un objeto de tipo Time es igual o no a alguno de los valores incluidos en una lista de valores

Métodos

Método	Funcionalidad
Hour	Devuelve las horas del objeto Time
Minute	Devuelve los minutos del objeto Time
Second	Devuelve los segundos del objeto Time
SecondsTo	Este método calcula el número de segundos comprendidos entre dos objetos de la clase Time
Trace	Este método muestra en una ventana el valor que tiene asignado el objeto
Using	Este método realiza una conversión a Char formateada del objeto de tipo Hora (Time).

Conversores

La clase Time tiene conversor a la clase Char

```
//Ejemplo de uso de los operadores y métodos
//Clase Time
main
objects
begin
 t1 t2 as Time
 s as char
end
begin
 //operador de asignación
 t1 = "17:14:10";
 t2 = '00:00:02';
 t1.Trace;
 //operadores aritméticos
 t1 = t1 + 3;
 t1 = t1 - 5;
 --t1;
 t1.Trace;
 //métodos
 t1.Hour.Trace;
 t1.Minute.Trace;
 t1.Second.Trace;
 (t1.SecondsTo(t2)).Trace;
 (t1.Using(3)).Trace;
 s='La hora es ';
 s+=t1;
 s.Trace;
end
```

Para ilustrar el uso del método Using vamos a ver como quedaría una salida por pantalla con las diferentes máscaras que podemos utilizar.

Utilizaremos para ello la hora 22:15:37

Nº Máscara	Forma	Salida
0	General	22:15:37
1	Hh:mm:ss	22:15:37
2	Hh:mm:ss M	22:15:37M
3	Hh:mm	22:15
4	Hh:mm PM	10:15 PM
5	Hh	22
6	Hh PM	10 PM

Clase Boolean

Esta clase deriva de la clase Simple, es una clase instanciable. Esta clase permite crear objetos que son objetos que solamente pueden tener valores TRUE o FALSE.

Operadores lógicos

Operador	Funcionalidad
AND	Devolverá TRUE si ambas condiciones son ciertas
OR	Devolverá TRUE si al menos una de las condiciones es cierta
XOR	Es el operador OR exclusivo entre dos condiciones. Devolverá TRUE si una de las condiciones es cierta y la otra falsa, devuelve FALSE en caso contrario
NOT	El operador negación invierte el resultado que se obtiene al evaluar una condición

Operadores relacionales

Operador	Funcionalidad
== (igualdad)	Este operador de igualdad chequea si dos objetos de tipo Boolean son iguales
<>(distinto)	Este operador de igualdad chequea si dos objetos de tipo Boolean son distintos

Operador de asignación

Operador	Funcionalidad
= (asignación)	Este operador permite asignar un valor a un objeto de tipo Boolean

Métodos

Método	Funcionalidad
Trace	Este método muestra en una ventana el valor que tiene asignado el objeto

```
//Ejemplo de los operadores y métodos
//Clase Boolean
main
objects
begin
 b1 b2 b3 as boolean
end
begin
 //operador de asignación
 b1 = true;
 b2 = false;
 b1.Trace;
 //operadores lógicos
 b3 = b1 and b2;
 b3 = b1 or b2;
 b3 = b1 xor b2;
 b3 = not(b1 xor b2).Trace;
end
```

Con esto terminamos este capítulo dedicado a aprender el uso de los tipos simples de datos. Hemos dejado para un próximo capítulo las clase Enum y EnumSet que aún siendo derivadas de la clase Simple, al tratarse de clases abstractas para poder utilizarlas debemos saber crear clases derivadas y esto lo veremos en el capítulo 4 de este tutorial.

Capítulo 3

Control de Flujo

Introducción

A lo largo de este capítulo se mostrará el uso de todas las instrucciones de control de flujo de programa que tiene el COOL

Instrucciones de selección — Evalúan una condición y dependiendo de su resultado ejecutan una o u otra instrucción: IF, SWITCH.

Instrucciones iterativas — Repiten la ejecución de una instrucción en función del resultado de la evaluación de una condición: DO, FOR, FOREVER, REPEAT, WHILE

Instrucciones de ruptura de secuencia — Terminan la ejecución de un método o bien alteran la secuencia de ejecución de una instrucción iterativa: BREAK, CONTINUE, RETURN

Instrucción IF

Con esta instrucción se consigue la ejecución condicional de una instrucción.

Sintaxis:

if_statement ::= IF condition THEN statement1 [ELSE statement2]

Parámetro	Significado
Condition	Condición de bifurcación
Statement1	Instrucción a la que se cede el control si el resultado de evaluar condition es TRUE
Statement2	Instrucción a la que se cede el control si el resultado de evaluar condition es distinto de TRUE

Si condition evalúa a NULL, por tanto es distinta de TRUE, se ejecuta statement2, si se ha especificado.

```
//Instrucción IF
//Análisis del día de hoy

main
objects
begin
 d1 d2 as Date
end
begin
 d1 = "21/03/98";
 d2 = "20/06/98";
 if today between d1 and d2
  then
   "Es primavera".Trace;
  else
   "No es primavera".Trace;
end
```

Today es un literal dinámico de la clase Date cuyo valor es la fecha del sistema. Se trata de una de las palabras reservadas del lenguaje.

Between es el operador de la clase Date que chequea si un objeto de dicha clase esta comprendido o no en el rango dado por otros dos.

Instrucción SWITCH

Provoca la ejecución condicional de una instrucción. Tiene dos posibles formatos:

Formato 1

Recorre la lista de condiciones de cada rama CASE hasta que una de ellas evalúe a TRUE, en cuyo caso ejecuta la instrucción asociada.

Sintaxis:

switch_statement ::= SWITCH
BEGIN
[{CASE condition,... : statement1}...]
[DEFAULT : statement2]
END

Parámetro	Significado
Condition	Si evalúa a TRUE, provoca la ejecución de statement1
Statement1	Se ejecuta si alguna condición de las lista de condiciones de su rama CASE evalúa a TRUE
Statement2	Se ejecuta si ninguna de las condiciones anteriores evalúa a TRUE

Su funcionamiento es equivalente a una serie de instrucciones IF anidadas. El orden en que se evalúan las condiciones de cada rama CASE es indeterminado y puede variar en futuras versiones. Una vez que una de las expresiones de una rama evalúa a TRUE, se dejan de evaluar el resto de las demás condiciones de ésta y de las demás ramas CASE.

```
//Resuelve si un número definido como constante es
//positivo, negativo o cero

main
begin
 switch
  begin
   case i>0 : "El número es positivo".Trace;
   case i<0 : "El número es negativo".Trace;
   default  : "El número es cero".Trace;
  end
end
```

El número i se ha declarado en la sección de constantes como un smallint con un determinado valor.

Formato 2

Ejecuta un conjunto de instrucciones dependiendo del valor resultante de evaluar una expresión.

Sintaxis:

switch_statement ::=
SWITCH expression1
BEGIN
[{CASE switch_condition,... : statement1}...]
[DEFAULT : statement2]
END

switch_condition ::= [switch_comparison_operator] expression
| IN (expression,...)
| IS [NOT] NULL
| BETWEEN expression AND expression

switch_comparison_operator ::= < | <= | > | >= | MATCHES | LIKE

Parámetro	Significado
Expression1	Es la expresión a comparar
Expression	Es la expresión con la que se compara expresión1, aplicando el operador indicado
Statement1	Se ejecuta si alguna condición de las lista de condiciones de su rama CASE evalúa a TRUE
Statement2	Se ejecuta si ninguna de las condiciones anteriores evalúa a TRUE

expresión1 se evalúa una única vez y antes de la evaluación de las expresiones de cada rama CASE

Si no se especifica switch_comparison_operator, se realiza una comparación por igualdad mediante el operador ==.

```
//Resuelve si un número definido como constante es
//positivo, negativo o cero

main
begin
 switch i
  begin
   case >0 : "El número es positivo".Trace;
   case <0 : "El número es negativo".Trace;
   default : "El número es cero".Trace;
  end
end
```

Instrucción DO

Es una instrucción iterativa que comprueba su condición de finalización al final cada iteración.

Sintaxis:

do_statement ::= DO statement WHILE condition;

Parámetro	Significado
statement	Se ejecuta mientras condition evalúe a TRUE
condition	Si evalúa a TRUE, se produce una nueva iteración de statement. En caso contrario se cede el control a la siguiente instrucción a do_statement

Esta instrucción termina siempre con el caracter ';'.

Dado que condition se evalúa tras cada iteración, statement se ejecuta al menos una vez.

Es posible romper la secuencia de iteración mediante las instrucciones BREAK, CONTINUE o RETURN.

```
//Cálculo del factorial del número i

main
objects
begin
 i f as smallint
 s as char
end
begin
 i=3;
 s = "El factorial de "+i.Using(0)+" es ";
 f=1;
 do
  begin
   f*=i;
   --i;
  end
 while i<>0;
  s= s + f.Using(0);
  s.Trace;
end
```

La primera vez que se asigna valor al objeto de la clase char s, se hace uso de la concatenación de cadenas de caracteres y de la conversión de un objeto smallint a char mediante una máscara.

La salida que produce este programa es:

Figura 3.1. Salida del programa.

Instrucción FOR

Es una instrucción iterativa que comprueba su condición de finalización al inicio de cada iteración. Se recomienda su utilización cuando se conoce de antemano el número de iteraciones.

Formato 1

Inicializa un objeto con un valor dado y ejecuta una instrucción repetidas veces hasta que el valor de dicho objeto deje de cumplir una condición dada.

Sintaxis:

for_statement ::= FOR identifier = expression1 [DOWN] TO expression2
[STEP expression3] DO statement

Parámetro	Significado
Identifier	Referencia un objeto. Este puede ser de cualquier clase
Expression1	Valor inicial que toma identifier
Expression2	Valor que determina la continuación del bucle. Si se especifica DOWN, identifier ha de aceptar la operación >= con expresión2. En caso contrario ha de aceptar la operación <= con expresión2 Esta comparación se realiza al principio de cada iteración.Si el resultado la operación es TRUE, se ejecuta statement
Expression3	Es opcional. Si no se especifica STEP, identifier ha de aceptar la operación –, si se ha especificado DOWN o la operación ++ si no se ha especificado DOWN. Si se especifica STEP, identifier ha de aceptar la operación -= con expression3, si se ha especificado DOWN o la operación += con expression3 si no se ha especificado DOWN. Esta operación se realiza al final de cada iteración
Statement	Instrucción que se itera

Las tres expresiones (expression1, expression2 y expression3) se evalúan una única vez al principio del bucle, y no en cada iteración de éste.

Es posible romper la secuencia de iteración mediante las instrucciones BREAK, CONTINUE o RETURN.

```
//Cálculo del factorial de un número

main
objects
begin
 i f as Integer
 s as char
end
begin
 f=1;
 i=5;
 s = "El factorial de "+i.Using(0)+ " es ";
 for i = 5 down to 1 step 1 do f*=i;
 MessageBox(s+f.Using(0),"Factorial",0);
End
```

MessageBox es un método de la clase **Module** que muestra un cuadro de diálogo para pedir al usuario una respuesta o ofrecer una información. Su sintaxis es:

MessageBox(msg, title, flags) return Smallint

Donde:

Parámetro	Significado
Msg	Objeto de tipo char que contiene el texto del mensaje
Title	Objeto de tipo char que por defecto asume NULL que contiene el título del cuadro de diálogo
Flags	Especifica el contenido y la conducta del cuadro de mensajes. Para comprobar los valores que puede asumir consultar la ayuda en línea de COSMOS. En nuestros ejemplos tiene el valor 0 que indica que debe mostrar un botón de “OK”

La salida que obtenemos al ejecutar nuestro ejemplo utilizando este nuevo sistema de devolver el resultado es:

Figura 3.2. Salida utilizando MessageBox.

Formato 2

Recorre una lista de expresiones, inicializando un objeto con cada una de ellas y ejecutando una instrucción.

Sintaxis:

for_statement ::= FOR identifier AS expression,... DO statement

Parámetro	Significado
identifier	Referencia un objeto. Este puede ser de cualquier clase
expression	Expresión que se asigna identifier al inicio de cada iteración. identifier ha de aceptar la operación = con expression
statement	Instrucción que se itera. Se ejecuta tantas veces como expresiónes aparezcan

Cada expresión de la lista se evalúa en el momento de realizar la operación de asignación con identifier

Es posible romper la secuencia de iteración mediante las instrucciones BREAK, CONTINUE o RETURN.

```
//Cálculo del factorial de un número

main
objects
begin
 i f as Integer
 s as char
end
begin
 i=5;
 f=1;
 s = "El factorial de "+i.Using(0)+ " es ";
 for i as 1,2,3,4,5 do f*=i;
 MessageBox(s+f.Using(0), "Factorial",0);
end
```

i va tomando en cada iteración el valor que le asignamos mediante los números que siguen a **as**, es decir en la primera iteración i es igual a 1, en la segunda es igual a 2,... y así sucesivamente hasta que i es igual a 5 que es donde realiza la última iteración.

Instrucción FOREVER

Es una instrucción iterativa sin condición de finalización.

Sintaxis:

forever_statement ::= FOREVER statement

Parámetro	Significado
statement	Instrucción que se itera

Es posible romper la secuencia de iteración mediante las instrucciones BREAK, CONTINUE o RETURN.

```
//Pide pulsar una tecla para confirmar la terminación de la ejecución
//del programa

main
begin
 forever if Ok("¿Terminar?","Confirmacion") then break;
end
```

Ok es un método de la clase Module que muestra un cuadro de mensaje con los botones «Ok» y «Cancel».

Ok(msg, title, def) return Boolean

Donde:

Parámetro	Significado
Msg	Objeto de tipo char que contiene el texto del mensaje
Title	Objeto de tipo char que por defecto asume “” que contiene el título del cuadro de diálogo
Flags	El botón seleccionado por defecto será “Ok” si def = TRUE o el botón “Cancel” si def = FALSE:

La salida que obtenemos al ejecutar nuestro ejemplo utilizando este nuevo sistema de devolver el resultado es:

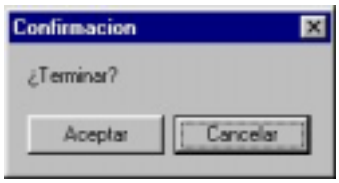


Figura 3.3. Salida del programa utilizando el método Ok.

Instrucción REPEAT

Es una instrucción iterativa que comprueba su condición de finalización al final cada iteración.

Sintaxis:

repeat_statement ::= REPEAT statement UNTIL condition;

Parámetro	Significado
Statement	Se ejecuta mientras condition no evalúe a FALSE
Condition	Si evalúa a FALSE, se produce una nueva iteración de statement. En caso contrario se cede el control a la siguiente instrucción a repeat_statement

Esta instrucción termina siempre con el caracter ‘;’.

Si condition evalúa a NULL, es decir distinto de FALSE, se cede el control a la siguiente instrucción a repeat_statement.

Dado que condition se evalúa tras cada iteración, statement se ejecuta al menos una vez.

Es posible romper la secuencia de iteración mediante las instrucciones BREAK, CONTINUE o RETURN.

```
// Cálcula el factorial de un numero

main
objects
begin
 n f as integer
 s as char
end
begin
 f=1;
 n=5;
 s="El factorial de "+n.Using(0)+" es ";
 repeat
  begin
   f*=n;
   --n;
  end
 until n==0;
 MessageBox(s+f.Using(0),"FACTORIAL",0);
end
```

Instrucción WHILE

Es una instrucción iterativa que comprueba su condición de finalización al principio de cada iteración.

Sintaxis:

while_statement ::= WHILE condition DO statement

Parámetro	Significado
Condition	Si evalúa a TRUE., se produce una nueva iteración de statement. En caso contrario se cede el control a la siguiente instrucción a while_statement
Statement	Se ejecuta mientras condition evalúe a TRUE

Si condition evalúa a NULL, es decir distinto de TRUE, se cede el control a la siguiente instrucción a while_statement.

Dado que condition se evalúa al inicio de cada iteración, statement puede no llegar a ejecutarse ninguna vez.

Es posible romper la secuencia de iteración mediante las instrucciones BREAK, CONTINUE o RETURN.

```
// Cálcula el factorial de un numero

main
objects
begin
 n f as integer
 s as char
end
begin
 f=1;
 n=5;
 s="El factorial de "+n.Using(0)+" es ";
 while n<>0 do begin
   f*=n;
   --n;
  end
 MessageBox(s+f.Using(0),"FACTORIAL",0);
end
```

Instrucción BREAK

Rompe la secuencia de iteración de una instrucción iterativa y cede el control a la siguiente instrucción a ésta. Se descartan por tanto las instrucciones del bloque posteriores a ésta.

Sintaxis:

break_statement ::= BREAK;

Esta instrucción termina siempre con el caracter ';'. Sólo puede aparecer dentro de una instrucción iterativa.

```
// Cálcula el factorial de un numero

main
objects
begin
 n f as integer
 s as char
end
begin
f=1;
n=5;
s="El factorial de "+n.Using(0)+" es ";
forever begin
   f*=n;
   --n;
   if n==0 then break;
end
MessageBox(s+f.Using(0),"FACTORIAL",0);
End
```

Instrucción CONTINUE

Rompe la secuencia de iteración de una instrucción iterativa y cede el control a la primera instrucción del bucle. Se descartan por tanto las instrucciones del bloque posteriores a ésta.

Sintaxis:

continue_statement ::= CONTINUE;

Esta instrucción termina siempre con el caracter ';'. Sólo puede aparecer dentro de una instrucción iterativa.

Provoca la evaluación de la condición de terminación asociada al bucle, y una nueva iteración de acuerdo con el resultado de la evaluación de ésta

```
// Cálcula el factorial de un numero

main
objects
begin
 n f as integer
 s as char
end
begin
n=5;
s="El factorial de "+n.Using(0)+" es ";
f=1;
while n>0 do begin
   f*=n;
   --n;
   if n==0 then begin
    MessageBox("Ultima iteracción","AVISO",0);
    continue;
   end
   MessageBox("Iteracción intermedia","AVISO",0);
end
MessageBox(s+f.Using(0),"FACTORIAL",0);
end
```

Instrucción RETURN

La instrucción RETURN finaliza la ejecución de un método y permite devolver un valor al objeto que lo invocó. Se explicará su utilización en el capítulo 4 en apartado de métodos y funciones.

Capítulo 4

Clases y Objetos

Introducción

A lo largo del presente capítulo se desarrollarán los conceptos básicos para la definición y uso de clases con sus instancias (objetos).

Definición de Clases

Una clase es la modelización de una abstracción de entidad, caracterizada por un identificador único y una Interfaz. Los objetos nacen a partir de las clases.

Durante las fases de análisis, diseño y programación, los objetos con iguales atributos y comportamiento se abstraen en clases.

Sintaxis:

class_section ::= [CLASSES BEGIN [class_definition...] END]

class_definition::=[access]identifier1IS [ABSTRACT] identifier2 [class_interface]
access ::= PUBLIC | PROTECTED | PRIVATE

Parámetro	Significado	
Access	Indica la visibilidad de la clase fuera del módulo en el que se ha declarado. Puede ser de tres tipos	
	PUBLIC	Una clase declarada como pública siempre es visible fuera del módulo
	PROTECTED	Una clase declarada como protegida siempre es visible fuera del módulo, pero no será derivable fuera del módulo
	PRIVATE	Una clase privada solamente puede ser utilizada dentro del módulo en el que se ha declarado
Identifier1	Identificador único en su ámbito. Es el nombre de la nueva clase	
Identifier2	Nombre de la clase padre de la que estamos definiendo. Si se especifica ABSTRACT, la clase que estamos definiendo se considera abstracta o incompleta. En este caso, la clase identifier2 ha de ser también abstracta y no se debe especificar class_interface	
Class_interface	Completa la interfaz de la nueva clase, si no se ha especificado ABSTRACT	

Una clase abstracta solo puede descender de otra clase abstracta.

Para explicar la creación de una clase en Cosmos utilizaremos las clases abstractas Enum y EnumSet como comentamos en el capítulo 2. Estas clases eran abstractas y como tales no podíamos crear objetos de ellas sino que teníamos que crear clases que derivasen de ellas y posteriormente crear objetos de esas nuevas clases.

Clase Enum

Esta clase deriva de la clase Simple, es una clase abstracta.. Esta clase permite almacenar objetos que pueden tener un solo valor entre una lista de valores posibles.

Un Enum no puede tener elementos repetidos y sus elementos están ordenados. Un Enum permite asignar identificadores nemónicos a valores enteros. El ejemplo clásico es la asignación de valores a los días de la semana:

```
laborales is Enum of [lunes martes miercoles jueves viernes]
```

Cada valor incluido en los corchetes en una declaración de tipo enumerado contiene un valor entero subyacente, determinado por la posición del valor en la lista.

Operador de la clase Enum

Operador	Funcionalidad
= (asignación)	Este método permite asignar un valor a un objeto Enum

Método de la clase Enum

Método	Funcionalidad
Trace	Este método muestra en una ventana el valor que tiene asignado el objeto

Clase EnumSet

Esta clase deriva de la clase Simple, es una clase abstracta. Esta clase permite almacenar valores que representan la presencia de 0, 1 o mas elementos de un conjunto. En una clase EnumSet definida por el usuario se definirán los elementos que componen el conjunto.

Un EnumSet no puede tener elementos repetidos y sus elementos no están ordenados. Un EnumSet puede tener como máximo 32 elementos.

Sirve por ejemplo para almacenar los 'flags' de apertura de un fichero. o por ejemplo para almacenar el conjunto de permisos que tiene un usuario sobre las tablas de la base de datos: añadir, borrar, modificar y consultar.

```
accesos is EnumSet of [lectura escritura modificacion]
```

Operadores de la clase EnumSet

Operador	Funcionalidad
= (asignación)	Este método permite asignar un valor a un objeto de tipo EnumSet
+ (suma)	Calcula la unión de dos objetos de tipo EnumSet

Método de la clase EnumSet

Método	Funcionalidad
Trace	Este método muestra en una ventana el valor que tiene asignado el objeto

Sabiendo ya cuales son los operadores, métodos y funcionalidad de estas clases vamos a crear un nuevo proyecto, siguiendo los pasos de siempre, con un módulo donde crearemos las nuevas clases derivadas de estas. Pulsando el botón derecho del ratón sobre el componente "Classes" del nuevo módulo, seleccionamos Add del menú que se nos muestra:

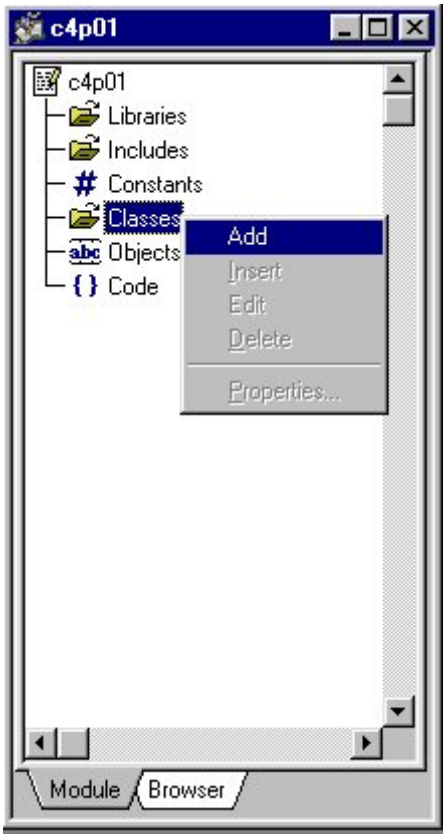


Figura 4.1. Añadir una nueva clase.

Cuando pulsamos esta opción se muestra un cuadro de diálogo en el cual se nos pide la información necesaria para crear una nueva clase. Antes de seleccionar la clase de la derivará la nueva clase el cuadro muestra las propiedades genéricas de la definición de una clase:

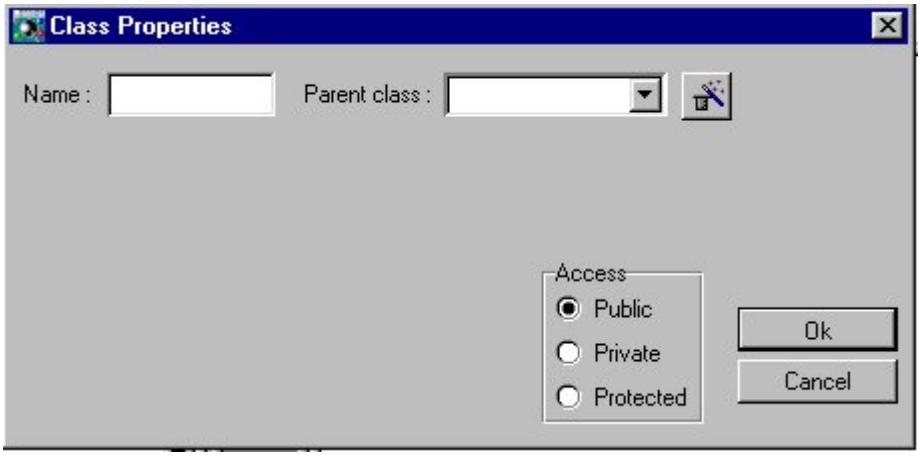


Figura 4.2. Cuadro de diálogo para crear una nueva clase.

Ahora introducimos el nombre de la nueva clase y le decimos cual será la clase padre de la que estamos creando, en nuestro caso la clase “laborales” hereda de la clase predefinida “Enum”.

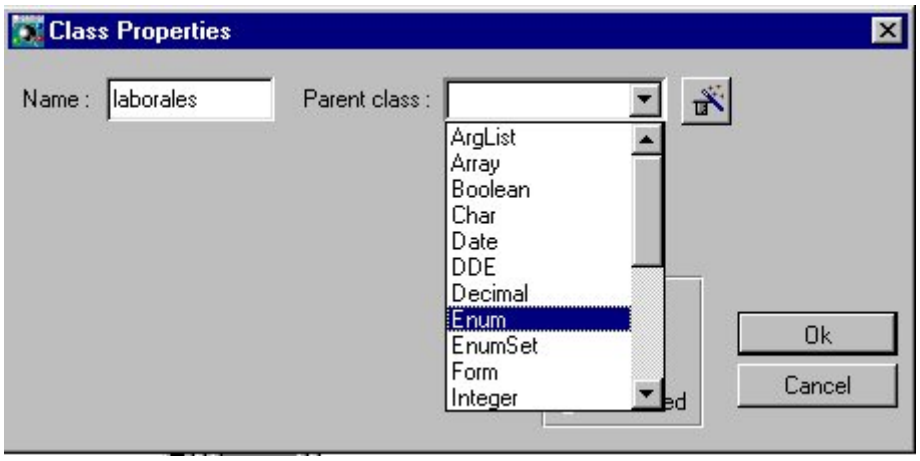


Figura 4.3. Selección de la clase padre.

Cuando se introduce la clase padre se muestra los campos de edición necesarios para introducir las propiedades que debe obligatoriamente tener la nueva clase. Estos campos de edición dependen por tanto de la clase padre que hayamos seleccionado. En nuestro caso aparece un campo de edición donde se deben introducir los valores que tendrá la clase laborales.

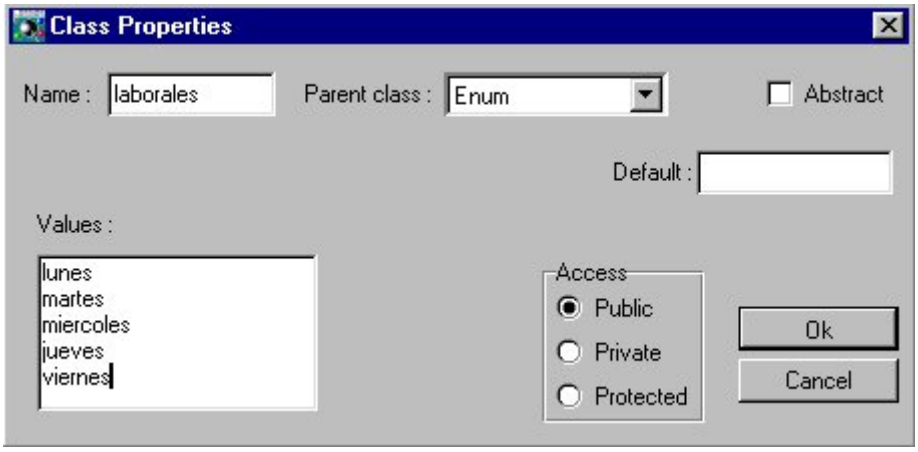


Figura 4.4. Valores de la clase “laborales”.

Pulsando el botón Ok tenemos creada la nueva clase que se situará dentro del componente “Classes” del modulo. Siguiendo estos mismos pasos hemos creado otras dos nuevas clases:

- festivos que deriva de Enum y cuyos valores son sabado y domingo
- accesos que deriva de EnumSet y cuyos valores son lectura, escritura y modificación.

El componente “Classes” despues de todas estas inserciones contendrá una clase derivada de EnumSet (accesos) y dos clases derivadas de Enum (laborales y festivos). Cada una de estas nuevas clases tiene un icono que marca cual es la clase de la que derivan.

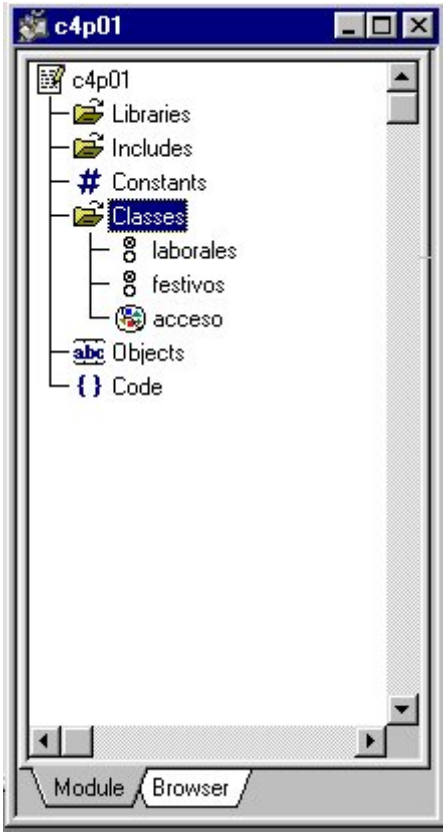


Figura 4.5. Paleta del módulo con las nuevas clases añadidas.

Podemos ver también que si seleccionamos la pestaña “Browser” que se encuentra en la parte inferior derecha de la paleta, en la jerarquía de clases de Cosmos aparecen estas nuevas clases que hemos creado derivando de sus clases padres.

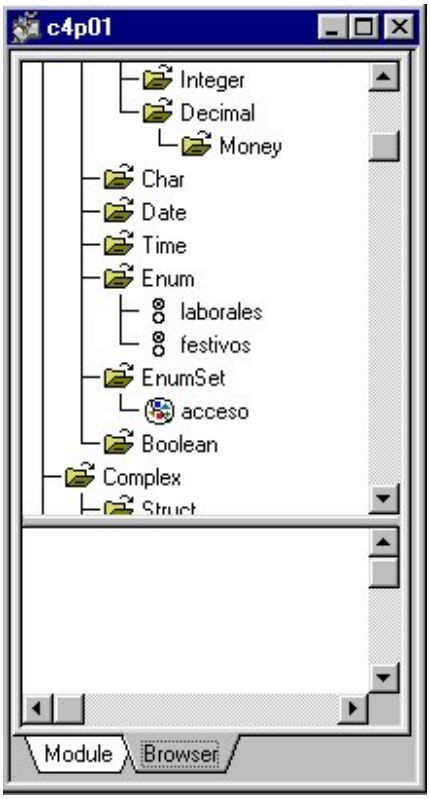


Figura 4.6. Aspecto del Browser de clases después de la inserción.

Cada clase puede tener asociado un código donde se deberán escribir sus métodos. Para poder introducir código en las clases nos situamos sobre su nombre en la paleta del módulo y haciendo doble click con el botón izquierdo del ratón, obtenemos el editor donde se debe escribir.

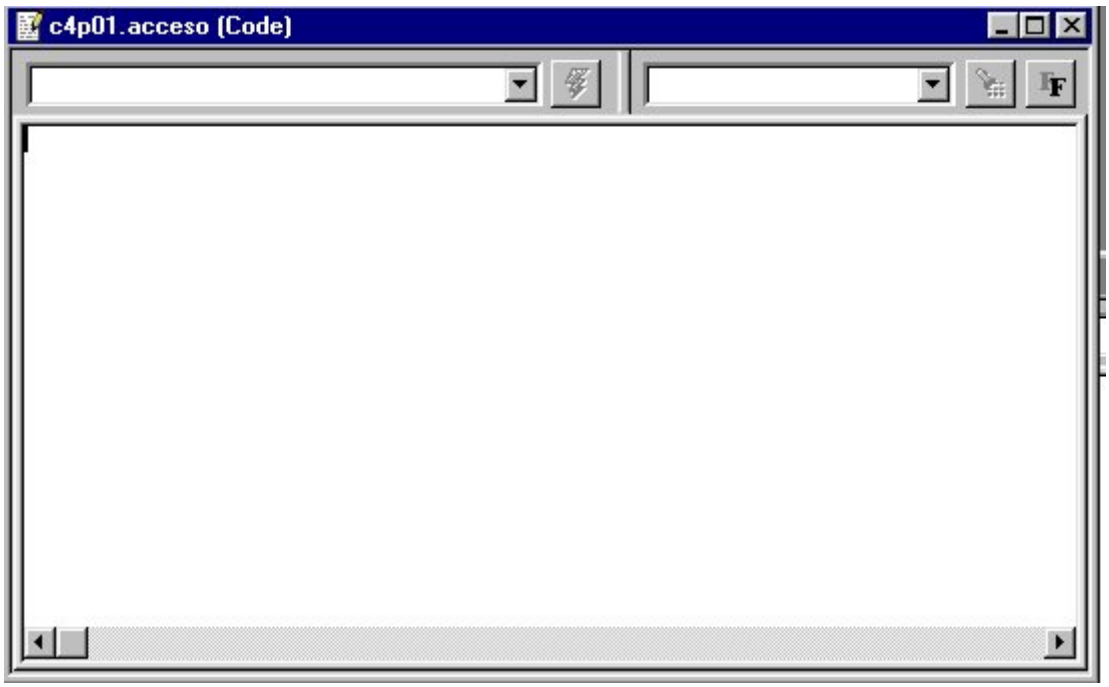


Figura 4.7. Editor de código de la clase accesos.

Hasta ahora ya hemos creado nuevas clases y hemos podido ver donde se colocan dentro de nuestro módulo. Ahora vamos a realizar un programa muy sencillo donde utilizaremos los operadores y métodos de Enum y EnumSet para comprobar como se deben definir los objetos de las nuevas clases dentro de un ámbito local.

```
// Definición de objetos de las clases creadas y utilización de los
// métodos y operadores de las clases Enum y EnumSet

main
objects
begin
 dia as laborales
 fiesta as festivos
 permiso as accesos
end
begin
 dia=lunes;
 dia.Trace;
 fiesta=domingo;
 fiesta.Trace;
 permiso=lectura+escritura;
 permiso.Trace;
end
```

Como se puede observar para crear objetos de estas nuevas clases creadas por nosotros debemos definir objetos de una forma similar a como lo hacíamos cuando se trataba de clases predefinidas.

Para comprobar el código completo del módulo, es decir la definición de las nuevas clases, su código y el código del modulo en sí vamos a utilizar otra de las herramientas que posee Cosmos: El Editor de Código.

Para utilizar este editor nos posicionaremos sobre el nombre del módulo en la paleta del proyecto o sobre cualquier componente del mismo si lo tenemos abierto. Seleccionamos en el menú “Tools” y del menú desplegable que aparece seleccionamos “Source Editor”.

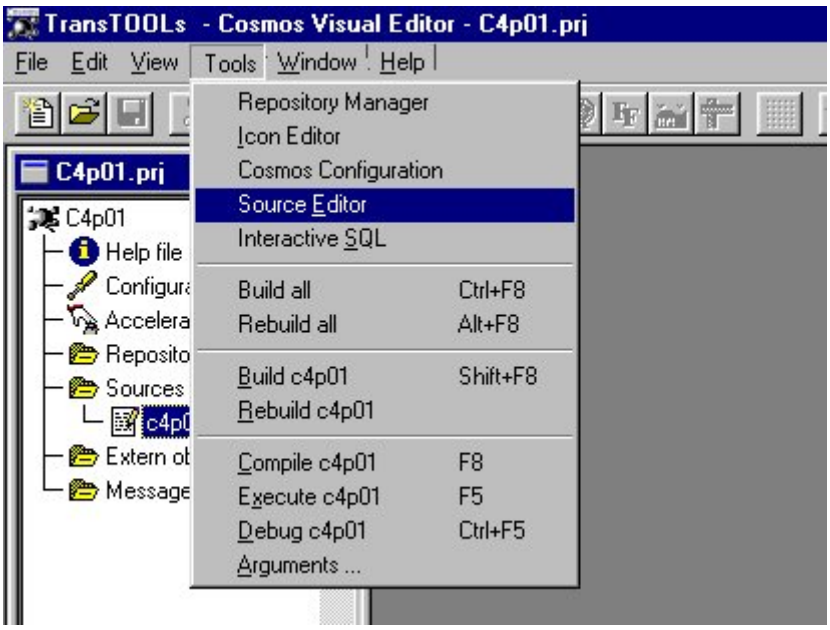


Figura 4. 8. Seleccionar el editor de código fuente.

Una vez seleccionado nos aparece una nueva ventana que contiene todo el código que se ha generado para el módulo seleccionado hasta ese momento, este código se encuentra almacenado en el fichero con extensión .smd y su nombre será el mismo que el del módulo.

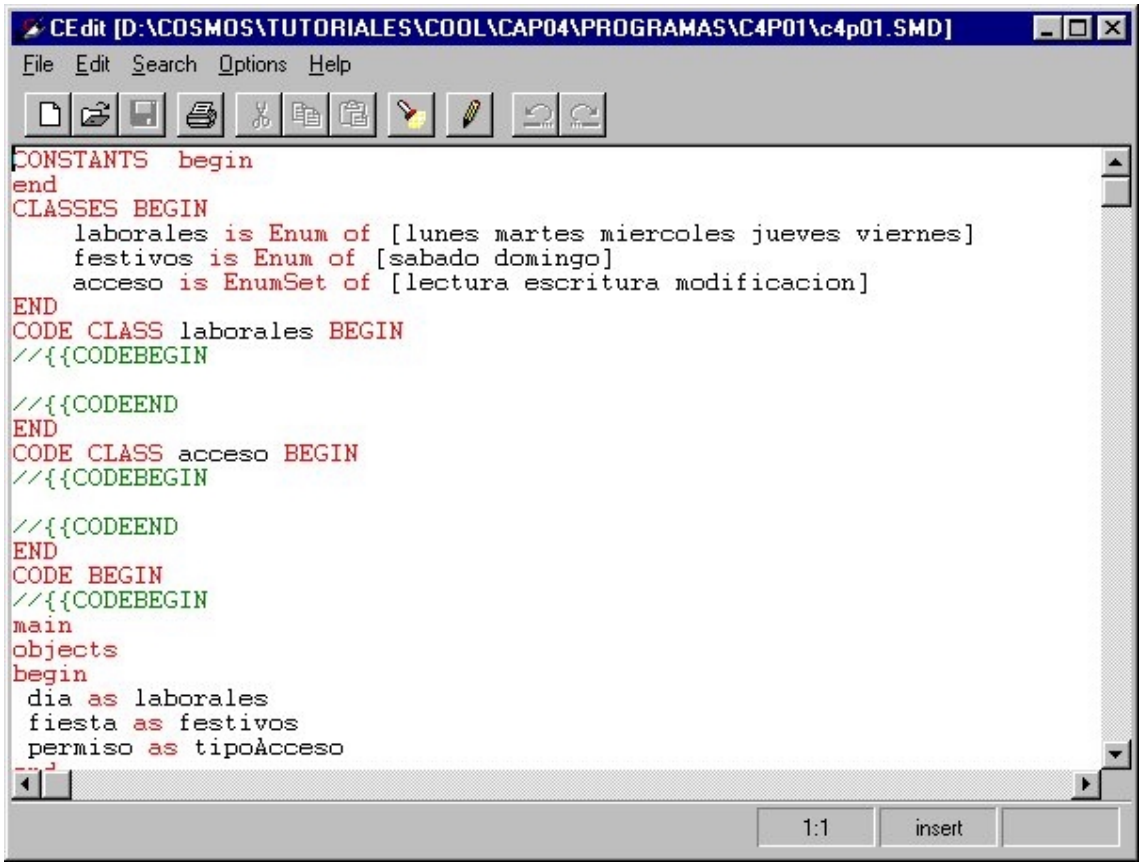


Figura 4. 9. Código generado para el módulo c4p01.

Vamos a analizar con mas detalle el código generado:

Sección de constantes vacia porque no tenemos definida en nuestro módulo ninguna.

```
CONSTANTS  begin
end
```

Sección de clases que contiene las nuevas clases creadas indicando para cada una la clase de la que derivan así como los valores que tiene cada una.

```
CLASSES BEGIN
    laborales is Enum of [lunes martes miercoles jueves viernes]
    festivos is Enum of [sabado domingo]
    accesos is EnumSet of [lectura escritura modificacion]
END
```

Código de cada una de las clases, se encuentra vacío porque no hemos definido ninguna funcionalidad especifica para las clases.

```
CODE CLASS laborales BEGIN
//{{CODEBEGIN

//{{CODEEND
END
CODE CLASS festivos BEGIN
//{{CODEBEGIN

//{{CODEEND
END
CODE CLASS accesos BEGIN
//{{CODEBEGIN

//{{CODEEND
END
```

Código del módulo, que es lo que nosotros hemos escrito en el editor de código de módulo.

```
CODE BEGIN
//{{CODEBEGIN
main
objects
begin
 dia as laborales
 fiesta as festivos
 permiso as tipoAcceso
end
begin
 dia=lunes;
 dia.Trace;
 fiesta=domingo;
 fiesta.Trace;
 permiso=lectura+escritura;
 permiso.Trace;
end
//{{CODEEND
END
```

Con esto hemos creado una nueva clase a partir de las clases predefinidas de Cosmos y a partir de aquí lo que haremos será profundizar mas en el manejo de las clases, objetos y los diferentes tipos de métodos que podemos crear.

Definición de Objetos

Los objetos son las instancias de las clases.

Sintaxis:

object_section ::= [OBJECTS BEGIN [object_definition...] END]

object_definition ::= [access] identifier... AS class [DEFAULT literal]

access ::= PUBLIC | PROTECTED | PRIVATE

Parámetro	Significado	
access	Nos limita el acceso a la interfaz del objeto. Puede ser de tres tipos	
	PUBLIC	La interfaz del objeto no está restringida. Si el módulo es de tipo include, el objeto es visible en cualquier otro módulo que dependa de éste
	PROTECTED	La interfaz del objeto está restringida al propio módulo y a todos los métodos de las clases que se definan en el módulo. El objeto no es visible en ningún otro módulo
	PRIVATE	La interfaz del objeto está restringida al propio módulo. El objeto no es visible en ningún otro módulo
identifier	Ha de ser único en la sección de objetos	
Class	Puede ser cualquier clase no abstracta. Si es una de las clases simples Smallint, Integer, Decimal, Char, Time, Date o Boolean, se puede especificar literal como valor por defecto mediante la cláusula DEFAULT. Si la clase es Char, se puede especificar una longitud. Si la clase es Decimal o Money, se puede especificar una longitud y una precisión	

Hasta ahora hemos creado objetos dentro de la sección de código de un módulo, pero de esta forma creamos objetos de ámbito local o de método. Pero existen otras formas de definir objetos y dependiendo de esta definición se modifica el ámbito de dicho objeto.

Se puede definir un objeto de ámbito de clase o atributo dentro de la sección de interfaz de una clase, aunque únicamente las clases Struct, Form, Page y Menu pueden definir este tipo de objetos. Este tipo de definición lo veremos en el capítulo 7 cuando utilicemos las clases derivadas de la clase abstracta y virtual Complex.

Por último podemos definir objetos que serán los atributos de un módulo de programación. Para ello debemos seleccionar, dentro de los componentes que contiene el módulo, la sección de **Objects.** Hacemos doble click del ratón sobre él o pulsado el botón derecho del ratón seleccionamos la opción Edit. Se muestra el contenido de la sección de objetos globales del módulo, que en este momento se encontrará vacio.

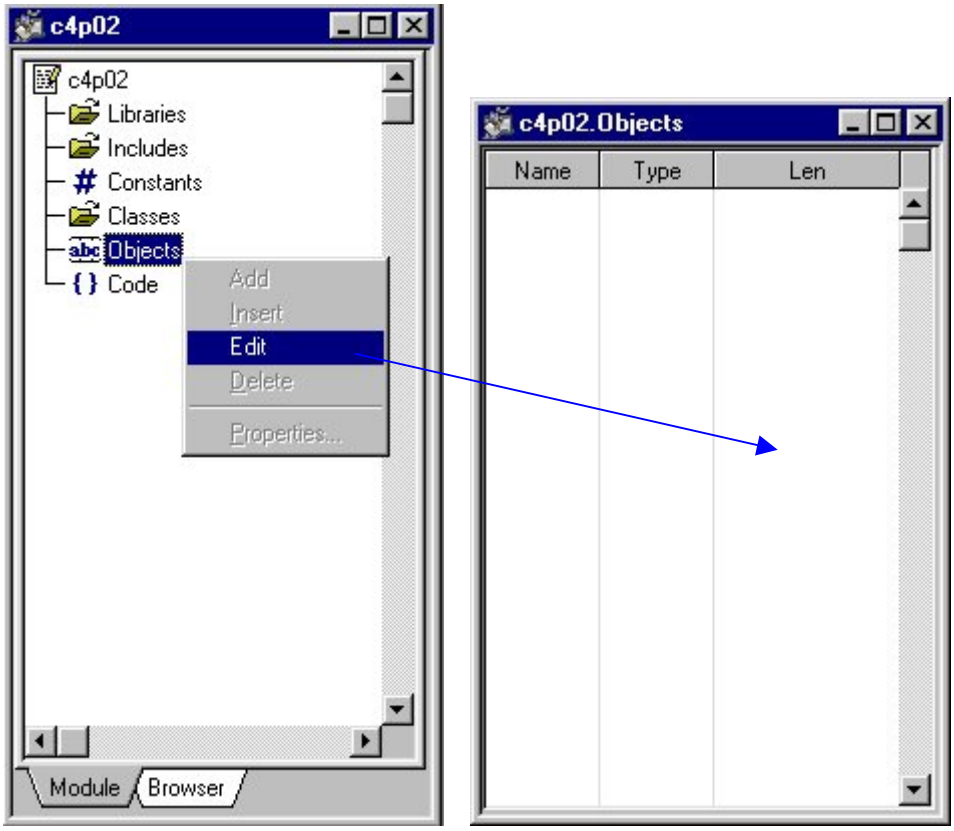


Figura 4.10. Seleccionar la sección Objects.

Para añadir objetos a esta sección debemos pulsar sobre esta ventana el botón derecho del ratón. Se mostrará una pantalla donde se puede seleccionar la acción a realizar (en un principio, únicamente se encuentra activa la opción de Add)

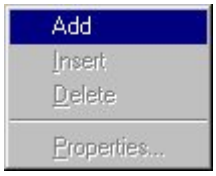


Figura 4.11. Seleccionar la opción.

Se muestra entonces un cuadro de diálogo donde debemos introducir las propiedades del objeto que queremos añadir.

Figura 4.12, Cuadro de diálogo de las propiedades de los objetos.

En nuestro programa crearemos un objeto de nombre **miObjeto** que deriva de la clase predefinida Integer. Para ello rellenamos el campo de edición de Name y seleccionamos la clase Integer de la lista desplegable, nos aparece entonces un campo de edición para introducir el valor por defecto de este objeto. Estos campos de edición sirven para introducir los atributos específicos de cada tipo de objeto, por ejemplo cuando creamos un objeto de la clase Char aparece la longitud, si es de tipo Money la longitud y la precisión y asi con todas las clases derivadas de las clases predefinidas.

Figura 4.13. Introducimos los datos del objeto.

Después de pulsar el botón Ok obtenemos la información del nuevo objeto creado en la sección de Objects del módulo actual.

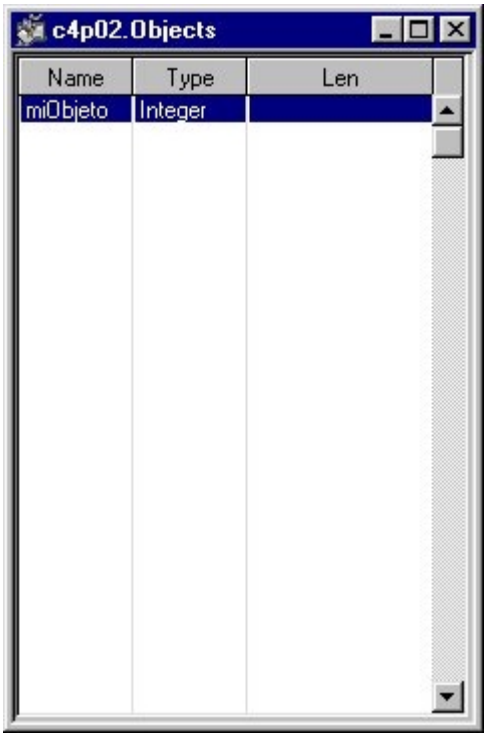


Figura 4.14. Nuevo objeto.

Ahora utilizaremos este objeto en tres métodos sencillos del módulo para comprobar que definiéndolo de esta forma el ámbito de actuación del objeto es todo el módulo y por tanto no hay que definir objetos específicos en cada método si no es necesario.

Añadimos en la sección de código del módulo un método Main desde el que llamamos consecutivamente a tres métodos que utilizan miObjeto. El código completo que se genera con las operaciones que hemos realizado hasta el momento es:

```
CONSTANTS  begin
end
OBJECTS BEGIN
    miObjeto AS Integer default 0
END
CODE BEGIN
//{{CODEBEGIN
// Manejo de un objeto de ámbito global

// Método Main que lanza la ejecución de los métodos sencillos
main
objects
begin
end
begin
 darValor(3);
 suma(5);
 verValor;
end


// Recibe como parámetro un valor numérico que asigna al objeto
// global
public darValor(m as integer) begin
 miObjeto=m;
end

// Muestra el contenido del objeto global
public verValor begin
 MessageBox(miObjeto.Using(0),"Visualizar Datos",0);
end

// Suma al objeto global una cantidad que se pasa como parametro
public suma(m as integer) begin
 miObjeto+=m;
end
//{{CODEEND
END
```

La salida de este programa es:

Figura 4.15. Salida del programa.

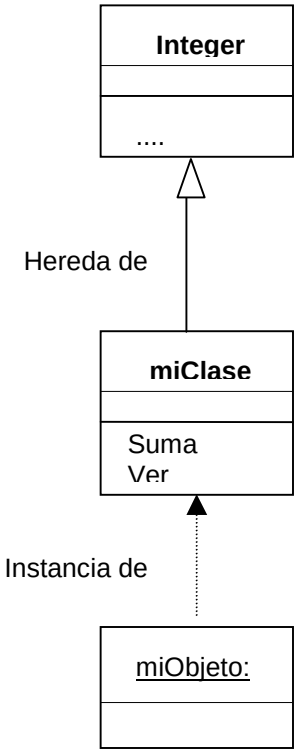
Por tanto se ha realizado de forma secuencial las acciones de dar valor al objeto global, sumar una cantidad al valor dado y mostrar el contenido de dicho objeto. Como se puede observar no hemos necesitado definir objetos locales a los métodos y todas las acciones son realizadas sobre el propio objeto global.

El objeto Self

Todos los objetos de una clase comparten el mismo código, cuando un método se aplica sobre un objeto y debe modificarse a sí mismo nos debemos referir al objeto mediante la palabra reservada **Self**.

Para clarificar este concepto creamos un nuevo proyecto con un módulo que tiene una clase llamada miClase derivada de la clase predefinida Integer. Esta clase tiene un método que suma 5 al contenido del objeto que lo invoca y un método que muestra el valor del objeto.

Además en la sección de Objects del módulo creamos un objeto derivado de esta clase llamado miObjeto. Esto queda reflejado en el diagrama UML de instancias de objetos a clases.



Con esta estructura y desde la sección de código del módulo invocamos la ejecución de los métodos de miClase. El código de los métodos de la clase miClase y el código del módulo son los siguientes:

```
CONSTANTS  begin
end
CLASSES BEGIN
    miClase is Integer
END
OBJECTS BEGIN
    miObjeto AS miClase
END
CODE CLASS miClase BEGIN

//{{CODEBEGIN

// Código del método suma que utiliza el objeto self
public suma begin
 Self=Self+5;
end
```

```
// Código del método ver que utiliza el objeto self
public ver begin
 MessageBox(Self.Using(0),"Visualizar Datos",0);
end

//{{CODEEND
END
CODE BEGIN
//{{CODEBEGIN
main
objects
begin
end
begin
 // se le asigna valor al objeto global
 miObjeto=5;
 // se invoca al método sumar que suma 5 al contenido del objeto
 // global
 miObjeto.suma;
 // se muestra el contenido del objeto global
 miObjeto.ver;
end
//{{CODEEND
END
```

La salida de este módulo después de la ejecución es:

Figura 4.16. Salida de la ejecución.

En el próximo capítulo profundizaremos mas en los conceptos desarrollados al explicar los definición de métodos.

Capítulo 5

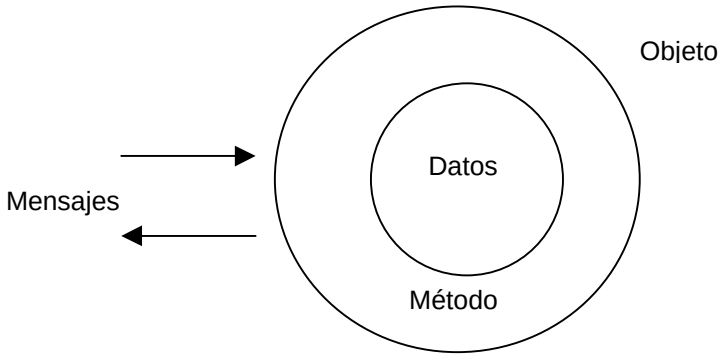
Métodos

Introducción

En el presente capítulo se desarrollan los conceptos relativos a los métodos. Se divide el capítulo en dos partes diferenciadas, por una parte los conceptos generales de los métodos y por otro lado se desarrollan ejemplos de cada tipo de método existente.

Definición de métodos

En Cosmos se denomina método al conjunto de acciones y servicios que ofrece una clase. Un método se implementa en una clase, y determina cómo tiene que actuar el objeto cuando recibe un mensaje. Un método puede también enviar mensajes a otros objetos solicitando una acción o información.



La estructura más interna de un objeto está oculta para otros usuarios y la única conexión que tiene con el exterior son los mensajes. Los datos o atributos del objeto, solamente pueden ser manipulados por los métodos asociados al propio objeto.

Los métodos en Cosmos son un grupo de instrucciones orientados a realizar una tarea específica y que se referencia mediante un nombre simbólico.

Cuando se define un método es necesario especificar el tipo de acceso:

Acceso	Significado
private	Los métodos de una clase solo pueden ser accedidas por funciones miembro de la clase
public	Los métodos de una clase pueden ser accedidos sin restricción desde cualquier punto del programa
protected	Los métodos puede ser utilizado por los miembros de la propia clase o por los métodos de sus clases derivadas

El comportamiento de un objeto está definido por la lista de métodos que soporta. En definitiva, un método permite actuar sobre los datos almacenados en los atributos de un objeto.

Los métodos están definidos por los siguientes parámetros:

Parámetro	Significado
Nombre	Es el encargado de identificar a la operación
Cuerpo	Algoritmo o conjunto de instrucciones asociado al método
Características	Pueden ser los: argumentos de entrada, valor de retorno, rango etc.. El rango es el conjunto de clases cuyos objetos son creados o modificados por la operación

Mensajes

Cuando se ejecuta un programa orientado a objetos, los objetos están recibiendo, interpretando y respondiendo a mensajes de otros objetos. Es decir, entre las distintas clases definidas dentro de un programa orientado a objetos se definen una serie de mensajes, que representan la forma en que los diferentes objetos pueden interactuar entre sí. En Cosmos un mensaje está asociado con un método, de tal forma que cuando se produce ese mensaje se ejecuta el correspondiente método de la clase a la que pertenece el objeto. Esto es, el envío de un mensaje equivale a llamar a un método.

Cuando un objeto necesita utilizar una operación especificada en la interfaz de otro objeto, se lo hace saber mediante un mensaje que consta de las siguientes partes:

Concepto	**Significado**
Emisor	Es el objeto que envía el mensaje
Receptor	Es el objeto que recibe el mensaje
Selector	Es el identificador de la operación que el objeto emisor solicita al objeto receptor
Lista de Argumentos	Es la información adicional que necesita el objeto receptor para completar la operación solicitada
Valor de Retorno	Es un valor que devuelve el objeto receptor al objeto emisor. Es el resultado de la operación solicitada

Un paso de mensaje, que implica la invocación de un método, se implementa mediante una expresión. Esta expresión es de la forma:

message ::= [expression .] operation [([argument,...])]
argument ::= expression

Parámetro	Significado
expression	Es el receptor del mensaje. La evaluación de esta expresión determina el objeto receptor del mensaje. Su especificación es opcional, pues dicho objeto puede estar especificado implícitamente. Consulte el apartado ámbito y visibilidad de un objeto
operation	Es el selector del mensaje. Es la única parte de éste cuya especificación es obligatoria
argument,...	Es la lista de argumentos del mensaje. Cada argumento se valida con el parámetro que ocupa la misma posición en el prototipo. Para que esta validación sea correcta, se ha de cumplir una y solo una de estas condiciones: 1. El argumento ha de ser el literal NULL 2. La clase del argumento ha de ser la misma que la clase del parámetro, o descender de ésta 3. La interfaz de la clase del argumento ha de especificar un método conversor hacia la clase del parámetro 4. Si la clase del parámetro es igual o descendiente de la clase ArgList, el argumento y todos los que le siguen, han de cumplir una de las tres condiciones anteriores con respecto a la clase asociada a la clase ArgList

Dado que operation puede ser polimórfica, pues puede haber más de una clase cuya interfaz especifique esta operación, se determina sobre la clase asociada a expression.

Un ejemplo muy sencillo para clarificar este concepto es:

```
MAIN
BEGIN
 "Hola".Trace;
END
```

El mensaje "Hola".Trace consta de las siguientes partes desglosadas:

Emisor: el objeto de clase Module al que pertenece el método main (SELF).
Receptor: el literal "hola" de clase Char.
Selector: la operación Trace de la clase Char.
Lista de argumentos: vacía.
Valor de retorno: objeto de la clase Char, que no se utiliza.

Otro ejemplo un poco mas complicado es el siguiente en el cual se muestra el número de letras "o" que existen en una cadena:

```
MAIN
BEGIN
 "Hola a todos/as".Count('o').Trace;
END
```

Tenemos dos mensajes anidados, pues aparecen dos operaciones, Count y Trace.

Primer mensaje:

Emisor: el objeto de la clase Module al que pertenece el método main (SELF).
Receptor: el literal "Hola a todos/as" de la clase Char.
Selector : la operación Count de la clase Char.
Lista de argumentos: Hay un único argumento que es un literal de la clase Char.
Valor de retorno: objeto de la clase Smallint que actúa como receptor del segundo mensaje.

Segundo mensaje:

Emisor: el objeto de la clase Module al que pertenece el método main (SELF).
Receptor: el objeto de la clase Smallint devuelto por el primer mensaje.
Selector: la operación Trace de la clase Smallint.
Lista de argumentos: vacía.
Valor de retorno: objeto de la clase Smallint, que no se utiliza.

Sintaxis

method_definition ::= prototype object_section method_body

prototype ::= [access] method_identifier [([parameter,...])] [RETURN class1]
access ::= PUBLIC | PROTECTED | PRIVATE
parameter ::= [VAR] identifier... AS class2 [DEFAULT literal]

object_section ::= [OBJECTS BEGIN [object_definition...] END]
object_definition ::= identifier... AS class [DEFAULT literal]

method_body ::= [compound_statement]

Vamos a explicar cada uno de los componentes de la definición de métodos por separado, es decir: prototype, object_section y method_body

Prototype (Prototipo del método)

Es la interfaz del método. Describe las características del mensaje asociado a la operación que implementa el método. Contiene toda la información necesaria para invocarlo correctamente

Parámetro	Significado	
Access	Nos limita "quien" puede invocar el método, es decir, restringe la clase del objeto emisor del mensaje asociado a la operación que implementa el método. Hay tres tipos de acceso:	
	PUBLIC	la clase del objeto emisor no está restringida
	PROTECTED	la clase del objeto emisor ha de ser la misma que la clase del objeto receptor, o descender de ésta
	PRIVATE	la clase del objeto emisor ha de ser la misma que la clase del objeto receptor
method_identifier	Tipo de método junto con un identificar adicional si es necesario. Especifica la operación asociada al método. Ha de ser único en el interfaz de la clase en la que se define el método. Los métodos se clasifican en: Función, Dll, Operador, Conversor, Notificación, Comando, Main.	
Class1	Es la clase del valor de retorno. Si se especifica, ha de ser de clase Simple y además el valor de retorno será un objeto temporal. En caso contrario el valor de retorno es el mismo objeto sobre el cual se ha invocado al método (SELF)	
Parameter	Un parámetro es un objeto que describe los requisitos que ha de cumplir un argumento en la invocación de un método. Su identificador identifier ha de ser único en la lista de parámetros.	

Object Section (Sección de Objetos Locales)

Define objetos auxiliares. Estos objetos se crean al inicio de la invocacíon del método, y se destruyen en la finalización de éste.

Parámetro	Significado
Identifier	Ha de ser único tanto en la sección de objetos locales, como en la lista de parámetros
Class	Puede ser cualquier clase no abstracta. Si es una de las clases simples Smallint, Integer, Decimal, Char, Time, Date o Boolean, se puede especificar literal como valor por defecto mediante la cláusula DEFAULT. Si la clase es Char, se puede especificar una longitud. Si la clase es Decimal o Money, se puede especificar una longitud y una precisión

Method Body (Cuerpo del método)

Consta de una secuencia de instrucciones orientadas a realizar una tarea específica. La invocación del método finaliza cuando se ejecuta la última instrucción de la secuencia o cuando se ejecuta la instrucción RETURN. En este caso, es obligatorio especificar un valor de retorno, si el prototipo del método lo exige.

Si una clase redefine un método, el prototipo de éste ha de ser idéntico al del método que se está redefiniendo, salvo la clase de retorno, que puede ser descendiente de la clase de retorno del método original.

Paso de parámetros por referencia y valor

Un argumento se pasa por referencia si en la definición del parámetro del método se especifica la cláusula VAR. También se pasan por referencia todos los objetos que su clase no desciende de la clase Simple, aunque no se especifique la cláusula VAR.

Si un objeto se pasa por valor, el runtime realiza una copia del argumento antes de la invocación del método y la destruye después. En caso contrario (paso por referencia), el estado resultante de las operaciones realizadas sobre el argumento durante la invocación del método es permanente, es decir las operaciones realizadas sobre el parámetro dentro del método se realizarán sobre el objeto original pasado como argumento.

Hay que tener en cuenta los siguientes puntos a la hora de decidir si se pasan los parámetros por referencia o por valor:

- Se debe definir un parámetro por valor (sin cláusula VAR) cuando quiera que el estado del objeto pasado como argumento permanezca inalterado durante la invocación del método.

- Se debe definir un parámetro por referencia (cláusula VAR) cuando quiera que el método pueda modificar el estado del objeto pasado como argumento.

- Se debe definir los parámetros de tipo Char por referencia (aunque no se desee modificar el valor del mismo), si desea optimizar el tiempo y gasto de memoria consumidos en la llamada, siempre que en el código del método no se altere el valor del mismo.

Definición de parámetros por valor por defecto

Un parámetro de un método puede definirse con un valor por defecto. El compilador permite la invocación a un método sin pasar valores en los parámetros definidos con valor por defecto. En este caso el compilador inserta en la llamada todos los argumentos no especificados usando para ello el valor por defecto definido para cada uno.

Como ejemplo usamos el método MessageBox de la clase Module que hemos utilizado para obtener las salidas de los programas y cuyo prototipo es:

MessageBox(msg as char, title as char default NULL, flags as smallint default 1) return smallint

Podemos utilizar la invocación siguiente:

```
MessageBox("Hola a todos/as");
```

que es equivalente a:

```
MessageBox("Hola a todos/as", NULL, 1);
```

Si se declaran parámetros por defecto en la definición de un método, estos deben indicarse consecutivamente sin interrupciones hasta el final de la lista de argumentos.

Redefinición de métodos

Si una clase define un método que ya estaba definido en la interfaz de su clase padre, decimos que esta clase redefine este método.

El prototipo ha de ser igual al del método que se redefine, exceptuando la clase del valor de retorno que puede ser una clase descendiente de la clase de retorno del método redefinido.

Si en el código de esta clase se desea invocar al método de la clase padre, es necesario especificar el cualificador Parent.

En una clase se permite redefinir los operadores aritméticos y el operador de asignación de su clase base.

Como ejemplo vamos a redefinir
Ejemplo:

```
CLASSES BEGIN
    miClase is Char (20) //definición de miClase
END
OBJECTS BEGIN
    miObjeto AS miClase //definición de un objeto de la clase miClase
END

//Código de miClase
CODE CLASS miClase BEGIN
// Redefinición del método Trace de la clase char
public Trace()
begin
("Llamamos al metodo Trace de miClase" + self).Trace;
"Ahora llamamos al metodo Trace de la clase char".Trace;
parent.Trace;
end
END
```

```
//Código del módulo
CODE BEGIN
// Se da valor al objeto y se llama al método redefinido
main
begin
 miObjeto="Hola a todos/as";
 miObjeto.Trace;
end

END
```

La salida general de este programa es:

Figura 5.1. Salida del programa de redefinición del método Trace de la clase Char.

Clasificación de los métodos

Un método puede ser de uno de los siguientes tipos y que veremos a continuación con un mayor detalle:

Tipo	Significado
Función	Un método función es un método que se referencia por un identificador
Dll	Un método dll es un método que se implementa mediante una llamada un procedimiento de una librería de enlace dinámico (DLL)
Operador	Un método operador es un método que se referencia por un símbolo matemático o bien por el identificador de éste
Conversor	Un método conversor es un método que se referencia por el identificador de una clase. Su finalidad consiste en devolver un objeto de dicha clase, equivalente a la clase sobre la que se invoca el método
Notificación	Un método notificación es la implementación de una respuesta a un mensaje provocado por un suceso o evento debido a una interacción con la interfaz gráfica o con la interfaz SQL. Los sucesos o eventos que pueden enviar una notificación están predeterminados
Comando	Un comando se define como una acción genérica de una aplicación
Main	El método MAIN es un caso especial de método. Sólo se puede definir en la sección de código de un módulo. Sirve de punto de entrada para su ejecución

Función

Como hemos comentado una función es un método que se referencia por un literal. Todos los métodos que hemos realizado en nuestros módulos, eran funciones. La sintaxis para la definición de funciones es la siguiente:

function ::= function_prototype object_section function _body

function_prototype ::= access function_identifier[([parameter,...])]
[RETURN class1]
function_identifier ::= [FUNCTION] identifier
function _body ::= compound_statement

Parámetros	Significado
Identifier	Es el nombre de la operación que implementa el método. Este nombre ha de ser único, es decir no debe coincidir con el nombre de otra operación definida en la misma clase ni con el identificador de algún atributo de ésta

Sintaxis de invocación:

function_message ::= [expression .] identifier [([argument,...])]
argument ::= expression

Parámetros	Significado
Identifier	Identificador de la función que queremos invocar

Como ejemplo hemos construido un módulo en el cual hemos definido una clase que deriva de la clase predefinida Smallint y en el código de la clase hemos definido una función que devuelve true si el valor pasado como parámetro es menor que el objeto que se utiliza para invocar el método.

```
CLASSES BEGIN
    miClase is Smallint // creamos una clase derivada de Smallint
END
OBJECTS BEGIN
    s1 AS miClase //Definimos dos objetos globales derivados de la
    s2 AS miClase //clase creada
END

// Código de la clase
CODE CLASS miClase BEGIN
//{{CODEBEGIN
//Función que calcula el mayor de dos objetos
public function mayor(n as smallint) return boolean
begin
 if self>n //se debe utilizar el objeto self para indicar sobre quien
           //se debe hacer la comparación
  then return true;
  else return false;
end
//{{CODEEND
END
```

```
//Código del módulo
CODE BEGIN
//{{CODEBEGIN
main
begin
 s1=8;
 s2=5;
 if s1.mayor(s2)
  then "Es mayor".Trace;
  else "Es menor".Trace;
end
//{{CODEEND
END
```

La salida de este programa es:

Figura 5.2. Salida de la ejecución del módulo.

Vamos a analizar detenidamente el prototipo de la función que hemos realizado en el ejemplo anterior:

```
public function mayor(n as smallint) return boolean
```

- Public: indica que el acceso a esta función es público, es decir, la clase del objeto emisor es público.
- Function: indica que el método es una función, se puede suprimir.
- Mayor: es el identificador de la función. Este nombre ha de ser único.
- (n as smallint): parámetros de la función. Al no poner VAR se trata de parámetros por valor, es decir se realiza una copia del objeto que únicamente pervive durante la ejecución de la función, y luego es destruido.
- Return boolean: indica el tipo de retorno que provoca la ejecución del programa. Es necesario añadir la clausula return cuando la función devuelve algún valor.

Las funciones en Cosmos pueden ser **recursivas**. Una función es recursiva cuando en el cuerpo de la función existe una llamada a si misma. Las funciones recursivas son muy útiles sobretodo en dos casos:

- cuando un problema se puede definir en función de sí mismo
- cuando un problema se puede resolver en función de uno o varios casos triviales

Un ejemplo típico de una función recursiva es el cálculo del factorial de un número que se define matemáticamente como:

```
n!=n*(n-1)*(n-2)*(n-3)*...*2*1
```

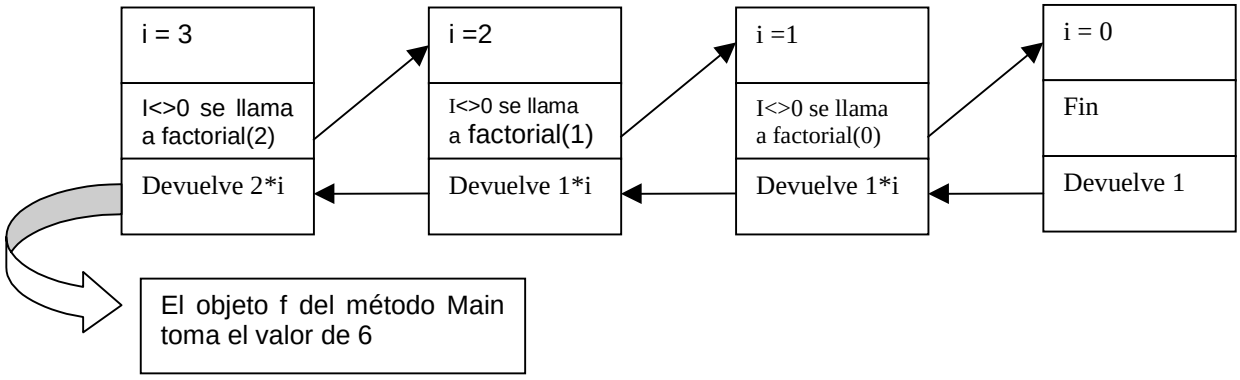
y se puede definir a partir de una función recursiva de la siguiente manera:

```
factorial(1)=1
factorial(2)=2*1=2*factorial(1)
factorial(3)=3*2*1=3*factorial(2)
   .
   .
   .
factorial(n)=n!=n*factorial(n-1)
```

Realizamos este ejemplo del calculo del factorial de un número para ilustrar el funcionamiento de la recursividad en Cosmos.

```
CONSTANTS  begin
end
CODE BEGIN
//{{CODEBEGIN
// Calcula recursivamente el factorial de un número
main
objects
begin
 i f as smallint
 s as char
end
begin
  i=3;
  f=factorial(i);
  s= "El factorial de " +i.Using(0)+ " es " + f.Using(0);
  MessageBox(s,"Factorial",0);
end

// Funcion recursiva que calcula el factorial
public function factorial(i as smallint) return smallint
begin
 if i==0
  then return 1;
  else return factorial(--i)*i;
end
//{{CODEEND
END
```

La secuencia de ejecución de este método main se muestra en la figura a continuación, en la primera casilla se indica el valor actual del parámetro i, en la central se marca la acción que se realiza y en la última lo que devuelve en cada momento el programa:



La salida de este método Main por pantalla es la siguiente:

Figura 5.3. Salida del cálculo del factorial.

Operador

Un método operador es un método que es referenciado por un símbolo matemático o bien por el identificador de éste. Su sintaxis de definición es:

operator ::= operator_prototype object_section operator_body

operator_prototype::=access operator_identifier[([parameter])][RETURN class1]
operator_identifier ::= OPERATOR op

operator_body ::= compound_statement

El nombre del operador es el nombre de la operación que implementa el método. Este nombre ha de ser único, es decir no debe coincidir con el nombre de otra operación definida en la misma clase ni con el identificador de algún atributo de ésta

Hay tres tipos de sintaxis de invocación dependiendo del número de si el operador es unario, binario o ternario:

Tipo	Invocacion
Unario	operator_message::=op expression | expression op | [expression.] OPERATOR op [()]
Binario	operator_message::=[expression] op argument | [expression .] OPERATOR op (argument)
Ternario	operator_message::=[expression] op argument1 op1 argument2 | [expression .] OPERATOR op (argument1, argument2)

Cada clase predefinida en Cosmos posee operadores o no. Son de tipo aritmético, de asignación o relacional. Para saber cuales son estos operadores consultar La guía de referencia de MultiBase Cosmos.

Sólo es posible redefinir los operadores aritméticos (*, *=, /, /=, %, %=, +, +=, ++, -, -=, --, =).

Para ilustrar el uso de operadores creamos un nuevo módulo que dada una fecha nos diga si el año es bisiesto o no. Un año es bisiesto si es divisible entre 4 y no divisible entre 100, excepto los que son divisibles entre 400 que si son bisiestos.

Hemos creado una clase, miDate, derivada de la clase predefinida Date, sobre la que hemos creado tres métodos:

- Constructor: inicializa el valor del objeto que lo invoca
- OPERATOR ++: redefine el operador incremento de la clase padre Date. Este operador inicialmente incremente en un día el objeto Date. En nuestro caso hemos incrementado el objeto en un año (365 días)
- Bisiesto: devuelve un char que contiene el resultado de analizar si el objeto es o no bisiesto.

En el código del módulo hemos realizado una serie de invocaciones a los métodos de miDate.

```
CLASSES BEGIN
    miDate is Date
END
OBJECTS BEGIN
    fecha AS miDate
    resultado AS Char (40)
END
```

```
CODE CLASS miDate BEGIN
// Inicio del código de la clase miDate

// Constructor de la clase miDate, inicializa el valor del
// objeto que lo invoca mediante el objeto f pasado como
// parametro
public Constructor(f as Date) begin
 self=f;
end

//Redefinición del operador aritmético ++ de la clase Date
//para que incremente la fecha del objeto que lo invoca en un año
//en lugar de un día
public OPERATOR ++ return miDate begin
 self+=365;
end

// Otorga valor al objeto global resultado
// con el resultado del análisis de si es o no bisiesto
// el objeto que invoca a este método
public function Bisiesto return char begin
if ((fecha.Year % 4 == 0) and
    (fecha.Year % 100<>0)) or
   (fecha.Year%400==0)
  then return "Este año " +fecha.Using(0)+ "es bisiesto";
  else return "Este año " +fecha.Using(0)+ "NO es bisiesto";
end
// Fin del código de la clase miDate
END

//Se redefine el operador between
public function miBetween(f1 f2 as Date) return char begin
 if self between f1 and f2
  then return"Estamos en los 90";
  else return "No estamos en los 90";
end


CODE BEGIN
//Inicio del código del módulo
main
begin
fecha.Constructor("05/05/1992");
resultado=fecha.Bisiesto;
resultado.Trace;

++fecha;
fecha.Trace;

resultado=fecha.miBetween(“01/01/90”, “31/12/99”);
resultado.Trace;
end
//Fin del código del módulo
END
```

Un ejemplo de invocación a un operador unario es:

```
++fecha
```

que cumple el prototipo de invocación op expression, que es equivalente, mediante el otro formato de invocación a:

```
fecha.OPERATOR++()
```

Un ejemplo de invocación a un operador binario es:

```
Self+=365
```

que es equivalente a Self=Self+365, que cumple el prototipo de invocación [expresion] op arguments, que es equivalente, mediante el otro formato de invocacion a:

```
Self.OPERATOR=(Self.OPERATOR+(365))
```

Un ejemplo de invocación a un operador ternario es:

```
Self Between f1 and f2
```

que cumple el prototipo de invocación [expresion] op argument1 op argument2, que es equivalente, mediante el otro formato de invocación a:

```
Self.OPERATOR Between(f1,f2)
```

Conversor

Un método conversor es un método que se referencia por el identificador de una clase. Su finalidad consiste en devolver un objeto de dicha clase, equivalente a la clase sobre la que se invoca el método. Su sintaxis de definición es:

conversor ::= conversor_prototype object_section conversor_body

conversor_prototype ::= access conversor_identifier
conversor_identifier ::= CONVERSOR identifier

conversor_body ::= compound_statement

El identificador es la clase del valor de retorno. Siempre debe ser un identificador de una clase Simple. Dicha clase no puede ser una clase padre de la misma que lo implementa, aunque si una clase hija.

Los métodos de este tipo no aceptan parámetros pues la conversión ha de ser única.

Este método devuelve un valor de la clase indicada, que se supone equivalente al objeto sobre el que se ha invocado el método conversor. Nunca modifica el objeto sobre el que se invoca.

Podemos ver un método conversor como un método función sin parámetros y que devuelve un valor de la clase indicada. También podemos ver un método conversor como un método operador sin parámetros y cuyo operador es la clase.

En el proceso de validación de un argumento respecto a un parámetro, el compilador puede insertar una invocación a un conversor sobre el argumento con el fin de validar el parámetro.

La sintaxis de invocación para los conversores es:

conversor_message ::= [expression.] [CONVERSOR] identifier [()]

El identificador es el identificador de una clase Simple. Dicha clase no puede ser una clase padre, aunque si una clase hija

Para ilustrar el uso de los conversores, hemos creado un módulo que tiene dos clases una llamada miSmallint que deriva de Smallint y otra llamada miChar que deriva de Char. También hemos creado dos objetos uno de cada clase creada, llamados s y c respectivamente. Cada una de estas clases tienen un método constructor que da valor al objeto que lo invoca y otro método que es un conversor a la clase contraria. El código del módulo contiene simplemente unas instrucciones que invocan al método constructor de cada clase sobre cada objeto y luego compara el contenido de los objetos a través del conversor, mostrando un mensaje.

```
// Definición de las clases
CLASSES BEGIN
    miSmallint is Smallint
    miChar is Char (30)
END

// Definición de los objetos
OBJECTS BEGIN
    s AS miSmallint
    c AS miChar
END

CODE CLASS miSmallint BEGIN
// Inicio del código de la clase miSmallint

// Constructor de la clase miSmallint, inicializa el valor del
// objeto que lo invoca mediante el objeto num pasado como
// parametro
public Constructor(num as Smallint) begin
 self=num;
end

//Conversor de miSmallint a miChar
public function IntToChar return Char begin
 return self.conversor char;
end

// Fin del código de la clase miSmallint
END

CODE CLASS miChar BEGIN
// Inicio del código de la clase miChar

// Constructor de la clase miChar, inicializa el valor del
// objeto que lo invoca mediante el objeto car pasado como
// parametro
public Constructor(car as Char) begin
 self=car;
end

//Conversor de miChar a miSmallint
public function CharToInt return Smallint begin
 return self.conversor smallint;
end

// Fin del código de la clase miChar
END

CODE BEGIN
// Inicio del código del módulo
main
begin
 s=8;
 c="5";
 if c.CharToInt < s
  then "Es menor".Trace;
  else "Es mayor".Trace;
end
// Fin del código del módulo
END
```

Un ejemplo de invocación a un conversor es:

```
Self.conversor Smallint
```

que cumple el prototipo de invocación [exoression.] [Conversor] identifier[()], que es equivalente, mediante el otro formato de invocación a:

```
self.Smallint
```

Main

El método MAIN es un caso especial de método. Sólo se puede definir en la sección de código de un módulo. Sirve de punto de entrada para su ejecución. Al ejecutar la última instrucción de este método, se terminará la ejecución de este módulo. Su sintaxis de definición es:

main ::= main_prototype object_section main_body

main_prototype ::= main_identifier [([parameter,...])]
main_identifier ::= MAIN

main_body ::= compound_statement

No es posible invocar el método main explícitamente. Ha de invocarse indirectamente, bien a través del comando cosrun o bien a través del método Run de la clase Module.

El método Main siempre recibe sus argumentos por referencia, aunque no se especifique la palabra reservada Var en su definición de parámetros.

La finalización de la ejecución de un método Main implica una descarga del módulo en el que se ha definido si este no es referenciado por ninguno de los módulos de la aplicación que están en memoria en ese momento.

Si un módulo va a ser invocado mediante el comando cosrun, ha de tener un prototipo tal que no necesite argumentos para su invocación, o bien que solo necesite un argumento de tipo Char.

Todos los módulos que hemos realizado hasta el momento utilizan el método main sin argumentos por tanto cualquiera nos sirve de ejemplo de este tipo de método.

Un ejemplo sencillo del uso de argumentos en el método Main es el que recibe como parametro una cadena y la muestra por pantalla, por defecto asume la cadena “Hola a todos/as”.

```
CODE BEGIN
//Inicio del código del módulo
main (c as Char default "Hola a todos/as")
begin
 c.Trace;
end
//Fin del código del módulo
END
```

Si ejecutamos este código directamente sin indicar argumentos asume el valor por defecto y la salida de la ejecución del módulo es:

Figura 5.4. Salida sin especificar argumentos.

Para indicar argumentos desde el Editor Visual de Cosmos podemos acudir al menú Tools y seleccionar la entrada Arguments... que muestra la siguiente ventana donde se introduce el argumento y además nos permite seleccionar si deseamos que siempre que se ejecute el módulo se nos pregunte por los argumentos.

Figura 5.5. Introducir argumentos.

Con este nuevo argumento la salida que obtenemos al ejecutar el módulo es:

Figura 5.6. Salida con el nuevo argumento.

DLL

Un método dll es un método que se implementa mediante una llamada un procedimiento de una librería de enlace dinámico (DLL). Podemos ver un método dll como un método función con implementación externa.

Las librerías de enlace dinámco (DLL) son una característica clave del sistema operativo Windows. Como su nombre indica, las DLL's son bibliotecas de funciones que las aplicaciones pueden vincular y usar en tiempo de ejecución en lugar de hacerlo estáticamente en tiempo de compilación. Esto significa que las bibliotecas pueden actualizarse independientemente de la aplicación y que muchas aplicaciones pueden compartir una misma DLL.

Para poder llamar en nuestro programa a una función de una DLL debemos indicar de alguna forma al compilador en que DLL se encuentra el método y que parámetros recibe. Es decir debemos declarar el método para que el compilador sea capaz de encontrarla.

Sintaxis de definición:

dll ::= function_prototype object_section dll_body

dll_prototype ::= access dll_identifier [([parameter,...])] [RETURN class1]
dll_identifier ::= [SELF] DLL dll_path identifier

dll_body ::= compound_statement

Identifier es el nombre de la operación que implementa el método. Este nombre ha de ser único, es decir no debe coincidir con el nombre de otra operación definida en la misma clase ni con el identificador de algún atributo de ésta. Este identificador ha de coincidir además con el nombre del procedimiento exportado por la librería dinámica.

dll_path es un literal de clase Char. Es el path en el que se busca la dll, en la primera invocación al método. No es necesario indicar el path completo de la Dll, basta con indicar su nombre siempre que esta sea accesible para Windows. Sólo se comprueba en tiempo de ejecución y no en tiempo de compilación.

La clase de los parámetros ha de ser Simple. No obstante, no se da ningún error en caso contrario.

La sintaxis de invocación de un método dll es idéntica a la de un método función:

dll_message ::= function_message

Si se especifica SELF en la definición del método, el objeto sobre el que se invoca se pasará como primer parámetro.

El paso de argumentos al procedimiento de la Dll se realiza mediante el convenio Pascal. El paso de argumentos se realiza de la siguiente manera:

Clase	Por Valor	Por referencia
Smallint	entero de 16 bits	Puntero de 32 bits
Integer	entero de 32 bits	Puntero de 32 bits
Decimal	no implementado	no implementado
Char	puntero de 32 bits	Puntero de 32 bits
Date	entero de 32 bits	Puntero de 32 bits
Time	entero de 32 bits	Puntero de 32 bits
Enum	entero de 32 bits	puntero de 32 bits
EnumSet	entero de 32 bits	puntero de 32 bits
Complex	no implementado	no implementado
Container	no implementado	no implementado

Si un argumento se pasa por valor y su estado es desconocido (IS NULL), el valor que se pasa es indeterminado.

Si un argumento con valor NULL se pasa por referencia, tras la invocación del método, se asume que su valor ha sido modificado, por lo que a partir de este momento, sea cual sea su valor se tomará como valor real (NOT NULL). Si el argumento es el literal NULL, se pasa 0.

El valor de retorno se realiza de manera análoga al paso de argumentos por valor. Si la clase del valor de retorno es la clase Char y el valor de retorno es 0, se considera el valor de retorno como desconocido (IS NULL). En caso contrario el estado del valor de retorno no es desconocido (IS NOT NULL).

La librería dinámica se carga la primera vez que se invoca un procedimiento de ella, y se descarga cuando finaliza la ejecución de la aplicación.

Para ilustrar el uso de las Dll's hemos escrito un módulo que invoca el procedimiento ver_reloj de una Dll construida en Borland Delphi 3.0, que muestra un reloj digital con alarma. El código de esta Dll lo hemos incluido en el Anexo B de este tutorial. El código del módulo miDll es el siguiente:

```
CODE BEGIN
//Inicio del código del módulo
public dll "d:\programas\delphi\Uti.dll" ver_reloj
//Invoca una función que pertenece a una Dll
main
begin
  ver_reloj;
end
//Fin del código del módulo
END
```

En la definición de la Dll hemos incluido la ruta completa donde se encuentra almacenado el código de la Dll y el identificador de la función que vamos a invocar. La ejecución de este módulo produce la siguiente salida:

Figura 5.7. Salida de la ejecución de la Dll.

Notificaciones

Un método notificación es la implementación de una respuesta a un mensaje provocado por un suceso o evento debido a una interacción con la interfaz gráfica o con la interfaz SQL. Los sucesos o eventos que pueden enviar una notificación están predeterminados.

Sintaxis:

notification ::= form_notification | control_notification | table_notification

Se puede observar en la sintaxis que las notificaciones pueden ser de tres tipos:

Notificaciones de Control	Eventos mandados por un control de la Screen que son procesados por su Form.
Notificaciones de Tabla	Eventos mandados por una tabla, que son procesados por su Form.
Notificaciones de Form	Eventos mandados y procesados por el Form.

Una notificación solo se puede definir dentro del código de una clase Form. Las notificaciones están ya predefinidas, el usuario no puede crear notificaciones. Es decir, solo puede definir las instrucciones a ejecutar cuando se manda una notificación de las que hay definidas en Cosmos.

Debido a la funcionalidad de las notificaciones en el presente capítulo únicamente reflejaremos la sintaxis de definición de cada uno de los tipos de notificación y dejamos para el capítulo dedicado a Clases Complejas dentro de Form los ejemplos donde se hace el tratamiento de las notificaciones.

1. Notificaciones de Control.

Un método notificación de control es la respuesta a un mensaje que envía un control gráfico a la clase Form en la que se ha definido para informarle de un suceso o evento.

Sintaxis de definición:
control_notification::=control_notification_prototype object_section notification_body

control_notification_prototype ::= ON event control [(idm AS Integer)]
| ON event (control AS Char[, idm AS Integer])
notification_body ::= compound_statement

Parámetros	Significado
Event	Identificador de una de las notificaciones que puede mandar un objeto de clase Control a su Form, tales como Click, Rclick, Dblclick, SelChange, etc..
Control	Identificador del objeto de clase Control que envía la notificación al Form
Idm	Indice del elemento seleccionado en un control de tipo: Lista, grupo de cajas, grupo de botones, grupo de botones radio

Estos métodos no pueden ser invocados explícitamente. Solo pueden ser invocados como consecuencia de una interacción con un control de la interfaz gráfica.

Cuando se recibe una notificación de un control, se busca primero si se ha definido un método de prototipo “ON event control [(idm AS Integer)] “ específico para este caso. En caso contrario se buscará un método generico “ON event (control AS Char[, idm AS Integer]) “ que captura las notificaciones de tipo “event” para cualquer control, pasando el nombre del control como parámetro.

Si el control gráfico tiene asociado un comando, esta notificación se envía como comando.

2. Notificaciones de Form.

Un método notificación de form es la respuesta a un mensaje que envía a sí mismo un objeto de clase Form informar de un suceso o evento.

Sintaxis de definición:

form_notification::= form_notification_prototype object_section notification_body

form_notification_prototype ::= notification_identifier

notification_body ::= compound_statement

notification_identifier ::= ON event

Parámetros	Significado
event	Identificador de una de las notificaciones que puede mandar un objeto de clase Form, tales como Open y Close

Estos métodos no pueden ser invocados explícitamente. Solo pueden ser invocados como consecuencia de la ejecución de determinados métodos (Open, Close, Run y Exit).

3. Notificaciones de Tabla.

Una notificación de tabla es un mensaje que envía un objeto de la clase FormTable al objeto de la clase Form en la que se ha definido, para informarle de un suceso o evento.

Sintaxis de definición:

table_notification::=table_notification_prototype object_section notification_body

table_notification_prototype ::= ON event TABLE table
| ON event TABLE (table AS Char)

notification_body ::= compound_statement

Parámetros	Significado
event	Identificador de una de las notificaciones que puede mandar un objeto de clase FormTable a su Form definidas en Cosmos, puede ser una de las siguientes: Add, New, Update, etc..
tabla	Objeto de la clase FormTable afectado por el evento
identifier	Identificador del objeto de la clase FormTable afectado por el evento

Comandos

Un comando se define como una acción genérica de una aplicación. Esto permite asociar la misma respuesta por ejemplo al pulsar un botón, opción de menú o una secuencia de teclas (acelerador).

Si el control gráfico que envía el comando esta definido en el área gráfica asociada a una tabla del Form, este comando se enviará en primer lugar a la tabla para ver si se procesa, y en caso contrario se enviará al Form.

El hecho de que tanto la clase Form y la Clase FormTable tenga preimplementados métodos de respuesta a comandos predefinidos, permite un comportamiento automático de los forms y tablas sin más que definir controles gráficos que envíen estos comandos

Además de los comandos predefinidos Cosmos, el programador puede definir todos aquellos que necesite en cada aplicación.

Sintaxis:

command ::= form_command | table_command

Como se puede observar en la sintaxis los comandos pueden ser de dos tipos:

Comandos de Tabla	Código de respuesta al realizar una determinada acción sobre una tabla de un Form (añadir, modificar, borrar,etc.). Estos comandos son procesados por la tabla activa del Form.
Comandos de Form	Código de respuesta al realizar una determinada acción sobre el Form y/o su tabla maestra.

En este capítulo únicamente mostraremos la sintaxis de los dos tipos de comandos y como ejemplo del manejo de comandos os remitimos al capítulo sobre Clases Complejas dentro de la clase Form.

1. Comandos de Form

Sintaxis:

form_command ::= form_command_prototype object_section command_body

form_command_prototype ::= ON COMMAND cmd
| ON COMMAND (cmd AS Char)

command_body ::= compound_statement

Parámetros	Significado
Cmd	Es un identificador de comando. Puede aparecer explícito, o como el valor de un objeto de clase Char

2. Comandos de Tabla

Sintaxis:

table_command ::= table_command_prototype object_section command_body

table_command_prototype ::= ON COMMAND cmd TABLE table
| ON COMMAND cmd TABLE (table AS Char)
| ON COMMAND table TABLE (cmd AS Char)
| ON COMMAND TABLE(tableASChar,cmd AS Char)

command_body ::= compound_statement

Parámetros	Significado
cmd	Es un identificador de comando. Puede aparecer explícito, o como el valor de un objeto de clase Char
table	Es un identificador de tabla receptora del comando. Puede aparecer explícito, o como el valor de un objeto de clase Char

Estos métodos no pueden ser invocados explícitamente.

Capítulo 6

Clase Container

Introducción

La clase Container es una clase virtual que deriva de la clase Object de Cosmos. Esta clase al ser virtual no pueden realizar clases derivada de ella ni tampoco se puede instanciar. De esta clase derivan las clases Array y ArgList, que son contenedores de objetos.

La clase Container tiene un método y un operador que son:

Operador [ ]

Este operador permite acceder a un elemento de un objeto de la clase Container (Array, ArgList). Retorna el elemento del Container al que se ha accedido por medio de índice idx. Su sintaxis es:

object_container[idx as Smallint]

Parámetros	Significado
Idx	Es una expresión que evalúa a un valor entero, indicando a que elemento del Objeto Container se va a acceder. Es importante recordar que siempre se debe utilizar un índice de Container con origen en 1. Esto significa que el primer elemento del Container es el elemento 1.

Método Size

Este método devuelve el número de elementos que tiene un objeto de la clase Container (Array, ArgList). Retorna el número de elementos del objeto container. Su sintaxis es:

Size() return Smallint

Clase Array

Es una clase abstracta, por tanto no podemos crear instancias de ella, únicamente podemos crear clases derivadas.

Un Array es un conjunto de objetos de la misma clase a los cuales se accede por un índice. El índice de un Array es una expresión que evalúa a un valor entero, indicando a que elemento del Array se va a acceder. Siempre se debe utilizar un índice de Array con origen en uno (1). Esto significa que el primer elemento del Array es el elemento 1 y no el elemento 0. Esto es importante tenerlo en cuando se recorre un Array mediante un bucle que debemos inicializar a uno (1).

Los Arrays del lenguaje COOL de Cosmos son unidimensionales, pero se puede extender a más dimensiones definiendo un Array conjunto de Arrays.

Se hace referencia a cada elemento de un Array mediante el nombre del Array, seguido de su índice encerrado entre corchetes, es decir utilizando el operador [ ] de la clase Container.

Las propiedades de la clase Array son:

Propiedad	Significado
Name	Nombre de la clase derivada, este campo es obligatorio.
Parent Class	En este campo se indica la clase base de la nueva clase. En este caso su valor será Array u otra clase derivada de Array
Len	Número de objetos del Array
Referenced class	Lista de las clases predefinidas en cosmos y definidas del módulo y en sus includes. Los objetos del Array serán instancias de la clase que se seleccione en la lista
Default	Valor con el que se inicializarán los elementos del Array
Abstract	Permite definir la clase como abstracta o instanciable. Este campo solo estará visible cuando la clase base de la que se está creando sea abstracta
Access	Indica la visibilidad de la clase fuera del módulo en el que se ha declarado

Dependiendo de la clase referenciada que se haya seleccionado, se piden también los parámetros necesarios para el constructor de dicha clase.

Hemos creado un modulo que contendrá una clase llamada Unidimensional con 10 objetos de tipo Integer, que deriva de la clase Array. Cuando seleccionamos la opción de añadir una clase al módulo se nos muestra una ventana donde debemos introducir las propiedades de la nueva clase:

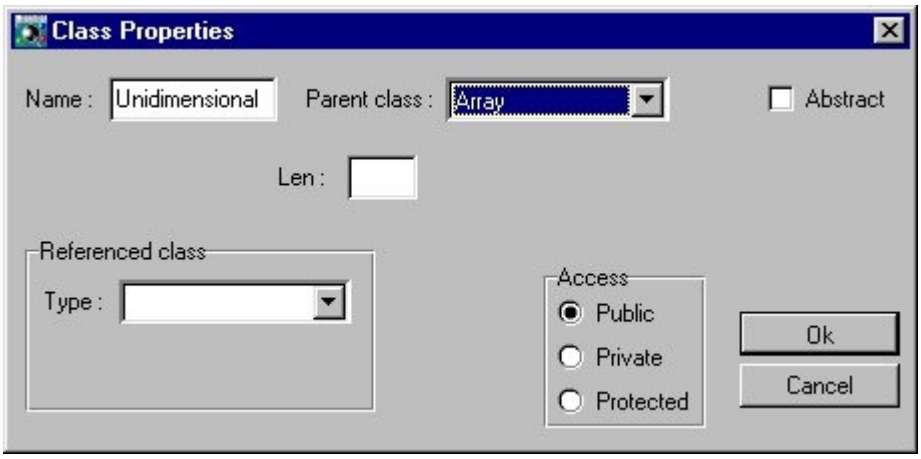


Figura 6.1. Propiedades de la nueva clase.

Estas propiedades introducidas quedan reflejadas en el código del módulo, de la siguiente manera:

```
CLASSES BEGIN
    Unidimensional is Array [10] of Integer default 0
END
```

Esta nueva clase tendrá cuatro métodos:

- Constructor: método que inicializa de forma ascendente los elementos del objeto de la clase Unidimensional que lo invoca.
- VerArray: muestra el contenido del objeto de la clase Unidimensional que lo invoca.
- Ordenar: ordena los elementos del objeto de la clase Unidimensional que lo invoca de forma descendente.
- VerTamaño: muestra el número de elementos que tiene el objeto de la clase Unidimensional que lo invoca.

El código completo de la clase Unidimensional es el siguiente:

```
CODE CLASS Unidimensional BEGIN
// Inicio del código de la clase Unidimensional

// Método que inicializa el objeto de la clase
// Unidimensional que lo invoca desde el número 1 al 10
public Constructor
objects
begin
 i as smallint
end
begin
 for i = 1 to 10 do begin
  Self[i]=i;
 end
end

//Muestra el contenido del objeto de la clase
//Unidimensional que lo invoca
public VerArray
objects
begin
 i  as smallint
end
begin
 for i=1 to 10 do begin
  Self[i].Trace;
 end
end

//Muestra el tamaño del objeto de la clase
//Unidimensional que lo invoca
public VerTamaño
begin
 Self.Size.Trace;
end

//Ordena de manera descendente el contenido
//del objeto de la clase Unidimensional
//mediante el método de la burbuja
public Ordenar
objects
begin
 i j as smallint
 aux as integer
end
begin
 for j = 2 to 10 do begin
  for i = 10 down to j do begin
    if Self[i-1] < Self[i]
     then begin
       aux= Self[i-1];
       Self[i-1]= Self[i];
       Self[i]=aux;
     end
  end
 end
end
// Fin del código de la clase Unidimensional
END
```

Hemos definido un objeto de la clase Unidimensional dentro de la sección de objetos locales del método Main que es donde llamamos a todos los métodos de la clase. El código del módulo donde se maneja la clase Unidimensional es:

```
CODE BEGIN
// Inicio del código del módulo

// Programa principal del módulo que maneja un array unidimensional
main
objects begin
    uni as Unidimensional
end
begin
 uni.Constructor;
 uni.VerTamaño;
 uni.VerArray;
 "Ahora ordenamos descendentemente 10-9- ...-2-1".Trace;
 uni.Ordenar;
 uni.VerArray;
end

// Fin del código del módulo
END
```

Un ejemplo del uso del operador [] es la instrucción que se encuentra dentro del bucle For del método Constructor de la clase Unidimensional:

```
uni[i]=i
```

El efecto que produce es que se asigna el valor i de cada iteración al objeto que ocupa la posición i dentro de la instancia de la clase Unidimensional que lo invoca. De esta forma se puede acceder a cada uno de los elementos de la estructura.

Un ejemplo del uso del método Size heredado por la clase Array de la clase Container es el método VerTamaño donde se invoca sobre el propio Objeto al método Size cuyo resultado se muestra por pantalla:

```
Self.Size.Trace;
```

Clase ArgList

Esta clase abstracta permite almacenar listas variables de argumentos. Una lista de argumentos es un conjunto de objetos del mismo tipo, es decir, todos los objetos de la lista tiene la misma clase base. A los elementos de una lista de argumentos se accede por su índice. Esta clase se utiliza para definir métodos que tiene un número variable de parámetros.

Una lista variable de argumentos siempre se pasa por referencia. Cuando en un método se define como parámetro una lista variable de argumentos, este debe indicarse en último lugar.

Las propiedades de la clase ArgList son:

Propiedad	Significado
Name	Nombre de la clase derivada, este campo es obligatorio.
Parent Class	En este campo se indica la clase base de la nueva clase. En este caso su valor será ArgList u otra clase derivada de ArgList
Referenced class	Lista de las clases predefinidas en cosmos y definidas del módulo y en sus includes. Los objetos del ArgList serán instancias de la clase que se seleccione en la lista
Abstract	Permite definir la clase como abstracta o instanciable. Este campo solo estará visible cuando la clase base de la que se está creando sea abstracta
Access	Indica la visibilidad de la clase fuera del módulo en el que se ha declarado

Para ilustrar el manejo de una lista variable de argumentos hemos realizado un módulo donde hemos definido una clase llamada unEntero derivada de la clase ArgList, referenciando a la clase integer:

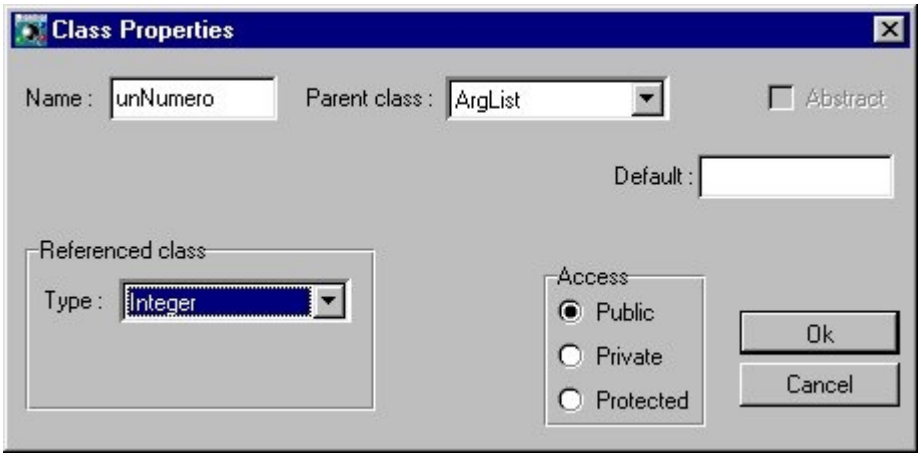


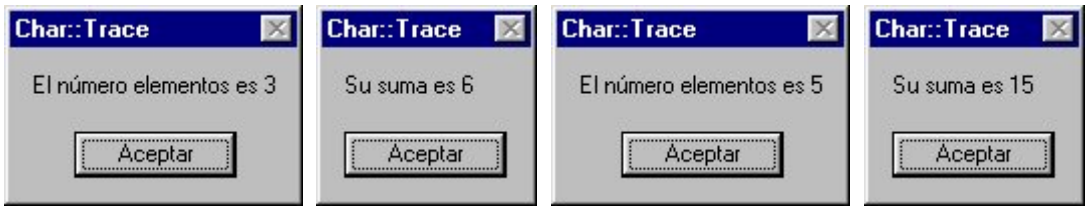
Figura 6.2. Creación de la clase unNumero

Dentro del código del módulo hemos definido una función Suma que admite como parámetro un objeto de la clase unNumero, es decir, admite un número variable de argumentos. Esta función muestra por pantalla el número de parametros que se le han pasado y retorna la suma de los objetos enteros pasados como parámetro. Desde el código del método Main llama a la función Suma dos veces, una con tres parametros (1, 2 y 3) y otra vez con cinco parámetros (1, 2, 3, 4, 5) y muestra el valor retornado por Suma. El código completo del código es:

```
CLASSES BEGIN
    unNumero is ArgList of Integer
END
CODE CLASS unNumero BEGIN
//Inicio del código de la clase unNumero
//Fin del código de la clase unNumero
END

CODE BEGIN
//Inicio del código del módulo

//Llama a la función suma pasando como parámetro
//n objetos de tipo integer
```

```
main
objects
begin
 total as integer
end
begin
 total=suma(1,2,3);
 ("Su suma es "+total.Using(0)).Trace;
 total=suma(1,2,3,4,5);
 ("Su suma es "+total.Using(0)).Trace;
end

//Función que suma los objetos integer que se le
//pasa como parámetro
public function suma(VAR unNumeroVble as unNumero) return integer
objects
begin
 i total as integer
end
begin
 total=0;
 for i = 1 to unNumeroVble.Size do begin
   total+=unNumeroVble[i];
 end
 ("El número elementos es "+unNumeroVble.Size.Using(0)).Trace;
 return total;
end
//Inicio del código del módulo
END
```

La salida de la ejecución es, de izquierda a derecha la siguiente:



Así terminamos la explicación de las clases derivadas de la clase predefinada Container.

Capítulo 7

La clase Complex

Introducción

Esta clase es abstracta y virtual, de ella derivan las clases que almacenan los objetos que tiene una estructura compleja.

De la clase Complex derivan: Struct, Window, Menu, Page, Module, Stream, SqlCursor, SqlStatement, SqlServer, PrnDocument, DDE, Control y la clase FormTable.

Clase Struct

Esta clase deriva de la clase Complex, es abstracta. Esta clase permite derivar clases con una estructura definida por el programador.

Para referenciar un miembro de una estructura en una expresión, se emplea una construcción de la forma:

nombre_estructura.miembro

El operador miembro "." permite seleccionar un miembro de una estructura.

Las estructuras ayudan a organizar datos complicados. Permiten tratar como unidad un conjunto de objetos relacionados lógicamente unos con otros, en lugar de tratarlos como entidades independientes.

Como ejemplo vamos a crear una clase miStruct con dni(integer), nombre(char) y apellidos(char), derivada de Struct, cuando creamos una clase de este tipo nos aparece en la ventana de propiedades de clase un recuadro donde debemos introducir los elementos de la estructura:

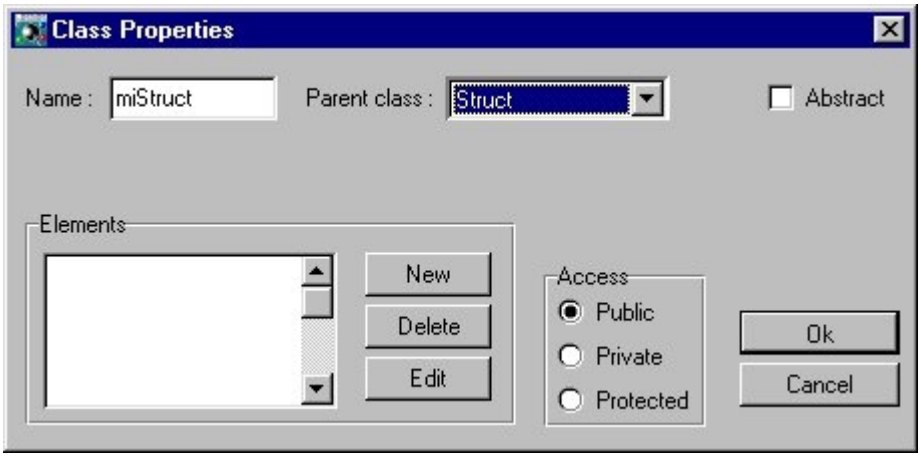


Figura 7.1. Ventana de propiedades de la nueva clase.

Haciendo click con el botón izquierdo del ratón sobre el botón New, visualizamos una nueva ventana donde debemos introducir las propiedades de un objeto de la estructura:

Figura 7.2. Propiedades del objeto mienbro de la estructura

Una vez introducidos todos los objetos que componen nuestra estructura miStruct, la ventana de propiedades de clase queda:

Figura 7.3. Propiedades de la estructura persona.

En esta clase hemos construido un método llamado Trace, que cuando es invocado, muestra la información de toda la estructura invocando a su vez a los métodos Trace de cada objeto.

Desde el código del método Main del módulo hemos definido un objeto derivado de miStruct llamado Persona, le damos valor a cada uno de sus miembros mediante el operador miembro "." y posteriormente invocamos a Trace.

El código completo de este módulo es:

```
CLASSES BEGIN
    miStruct is Struct begin
        dni as Integer
        nombre as Char (20)
        apellidos as Char (40)
    end
END
```

```
CODE CLASS miStruct BEGIN
//Inicio del código de la clase miStruct
public Trace()
begin
 self.dni.Trace;
 self.nombre.Trace;
 self.apellidos.Trace;
end
//Fin del código de la clase miStruct
END

CODE BEGIN
//Inicio del código del modulo
main
objects
begin
 persona as miStruct
end
begin
 persona.dni=9330150;
 persona.nombre="Cristina";
 persona.apellidos="Pelayo Gª-Bustelo";
 persona.Trace;
end
//Fin del código del modulo
END
```

Clase Window

Esta clase deriva de la clase Complex, es virtual. De ella derivan todas las clases que pueden usar ventanas para comunicarse con el usuario final de la aplicación.

De la clase Window deriva la clase Form

Clase Form

Esta clase define la funcionalidad de todos los objetos de tipo Form de una aplicación. Un Form contiene fundamentalmente un componente gráfico denominado Screen que permite dialogar con el usuario final mostrando información de la aplicación y recogiendo la respuesta de éste. Así mismo un Form de Cosmos tiene predefinida toda la funcionalidad necesaria para la edición directa de tablas de una Base de Datos.

La clase Form será la clase padre de las diferentes clases de formato de pantalla que defina el programador. Una vez definida una clase de tipo Form, el programador podrá definir objetos de esta clase en su programa si esta no ha sido definida como abstracta.

Esta clase tiene predefinidos una serie de métodos que permiten realizar tareas sencillas, como añadir, borrar o modificar filas de una tabla sin necesidad de programarlas. También permite realizar tareas más complejas, por ejemplo, el mantenimiento de tablas de la base de datos enlazadas mediante una relación “1 a N”.

Por lo general se hará uso de un Form siempre que se desee establecer un dialogo con el usuario.

Métodos de la clase Form

Método	Funcionalidad
AcceptEdit	Este método da validez a los datos introducidos en el formulario de la tabla activa del Form y inhabilita la edición en dicha tabla. (Sólo en modo edición)
AttachServer	Este método permite asociar un servidor al Form, cualquier operación automática que se ejecute en el Form sobre una base de datos, se hará contra el servidor asociado
CancelEdit	Este método cancela la edición, no da validez a los datos introducidos en el formulario de la tabla activa del Form y desactiva la edición en dicha tabla. (Sólo en modo edición).
Close	Este método permite cerrar un Form
Control	Este método retorna por referencia el control del Form que tiene de identificador el indicado como parámetro
CurrentFocus	Este método devuelve el control de la screen del Form que tiene el foco
DisableCommand	Este método habilita o inhabilita un comando en el Form.
DrawMenuBar	Este método repinta la barra de menú del Form. Por ejemplo, será necesario utilizarlo cuando se añade dinámicamente una opción al menú raíz del Form si este se presenta como “Pulldown”.
EditingTable	Este método retorna la tabla en edición del Form. (Sólo en modo edición).
EditQueryLike	Este método activa la edición de la tabla maestra del Form, sin mostrar los valores por defecto, y permite al usuario introducir los valores en los campos sobre los que se quiere realizar una consulta por condiciones de igualdad (QueryLike). (Sólo en modo edición)
Exit	Este método permite cerrar un Form ejecutado mediante el método “Run” (Form Modal) y devolver un valor de retorno.
FocusField	Este método devuelve una la variable asociada al control de la screen del Form que tiene el foco.
FocusTable	Este método devuelve la tabla del Form (objeto FormTable) que contiene el control que tiene el foco.
Frame	Este método devuelve el control asociado a la ventana del Form.
GetQueryByForm	Muestra un cuadro de diálogo que permitirá al usuario construir una condición de búsqueda sobre la tabla maestra del Form
GetQueryLike	Este método permite generar una condición de búsqueda sobre la tabla maestra del Form, tomando como condiciones de igualdad los valores de los controles de la screen asociados a dicha tabla.
Hwnd	Este método devuelve un número entero (INTEGER o DWORD), que se corresponde con el manejador (handle) para Windows de la ventana principal del Form.
InEditQueryLike	Este método indica si se ha esta en edición para hacer un QueryLike en la tabla maestra del Form mediante el método “EditQueryLike”. (Sólo en modo edición).
IsCommandDisabled	Este método indica si un comando esta habilitado o inhabilitado en el Form
IsOpen	Este método indica si el Form esta abierto
LosingFocus	Este método permite consultar qué control de la screen está perdiendo el foco cuando se esta procesando el evento On Enter de otro control.
MasterTable	Este método retorna la tabla maestra del Form
Maximize	Amplia la ventana del Form hasta su tamaño máximo

Método	Funcionalidad
MessageBox	Este método muestra un cuadro de mensaje para pedir una respuesta al usuario e inhabilita el Form hasta que se cierra dicho cuadro de mensaje
Minimize	Minimiza la ventana del Form
NextFocus	Este método devuelve una referencia al siguiente control del Form que admite el foco, o al anterior si se le indica en el parámetro
Open	Este método muestra en pantalla la screen de un Form de forma no modal.
Query	Este método permite la consulta de filas sobre la tabla maestra del Form
QueryByForm	Este método permite la consulta de filas sobre la tabla maestra del Form. Muestra un cuadro de diálogo que permitirá al usuario construir la condición de búsqueda.
QueryLike	Este método permite la consulta de filas sobre la tabla maestra del Form, tomando como condiciones de igualdad los valores de los controles de la screen asociados a dicha tabla
ReceivingFocus	Este método permite consultar qué control de la screen esta recibiendo el foco cuando se esta procesando el evento On Exit de otro control.
Reset	Este método permite borrar la lista en curso de todas las tablas del Form. Pone la tabla maestra en estado New.
Restore	Este método restaura la ventana del Form a su tamaño y posición anteriores a la maximización o minimización.
Run	Este método permite realizar una ejecución modal de un Form, mostrando en pantalla su screen.
SendCommand	Este método permite ejecutar un comando del Form
SetEditMode	Este método permite activar o desactivar el modo edición
SetMenu	Este método permite cambiar de menú del Form dinámicamente
Sql	Este método retorna el objeto SqlServer que tiene asociado el Form.
SqlErrMsg	Devuelve el string correspondiente al código de error actual del servidor SQL asociado al Form.
SqlError	Devuelve el código de error actual del servidor SQL asociado al Form.
Table	Este método retorna por referencia la tabla del Form que tiene de nombre el indicado como parámetro.

Hemos creado un módulo donde hemos creado una clase derivada de form llamada mi form.

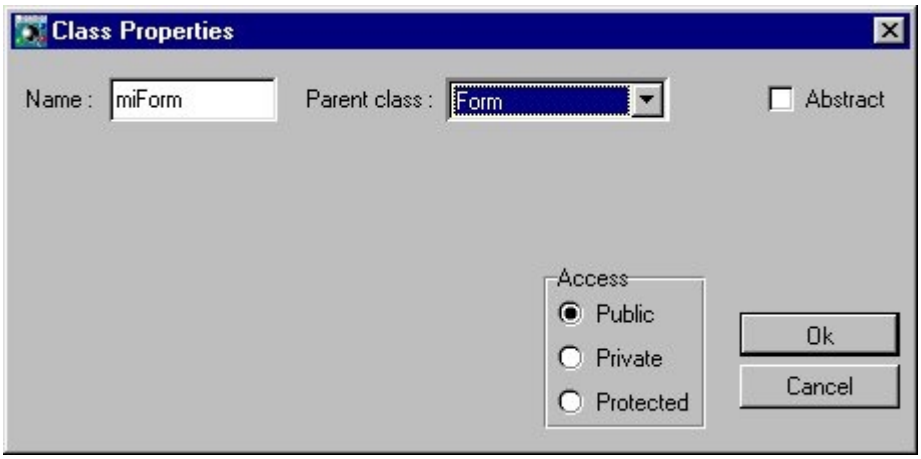


Figura 7.4. Creación de la clase miForm.

En la estructura de árbol del modulo aparece esta nueva clase dentro de la sección Classes:

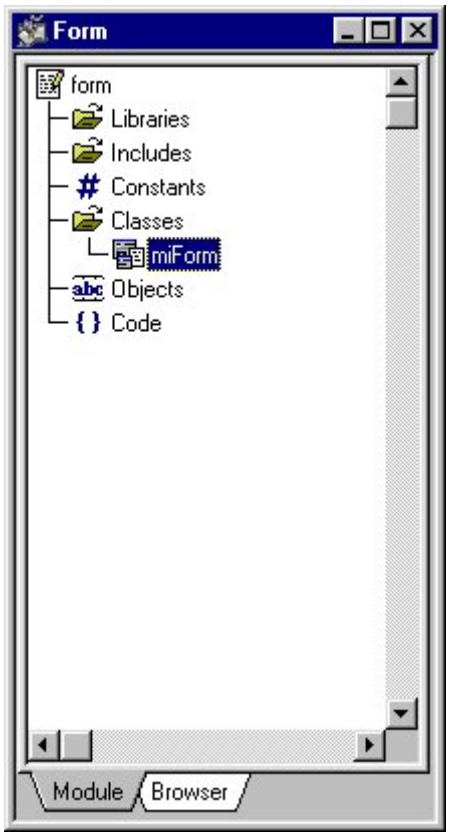


Figura 7.5. Nueva Clase del módulo.

Cuando hacemos doble click sobre esta nueva clase, miForm, se despliega y nos muestra los componentes que la forman:

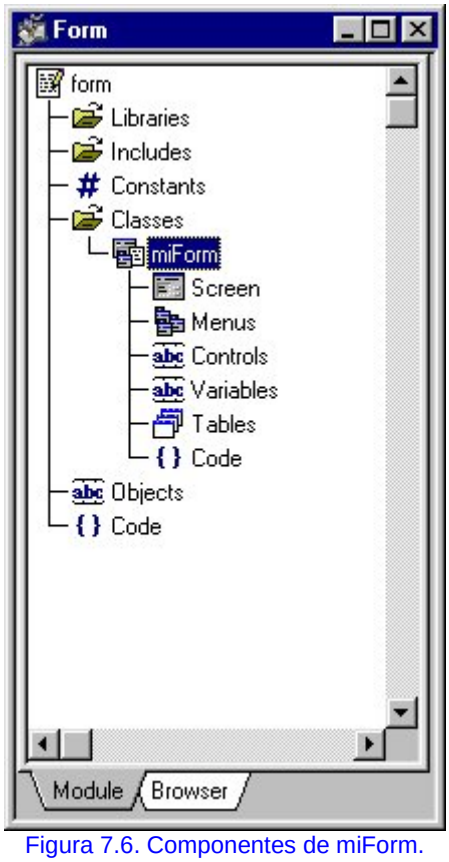


Figura 7.6. Componentes de miForm.

Cuando seleccionamos el componente Screen de la nueva clase nos aparece este componente y sobre el podemos componer toda la funcionalidad que deseamos posea nuestra Form:

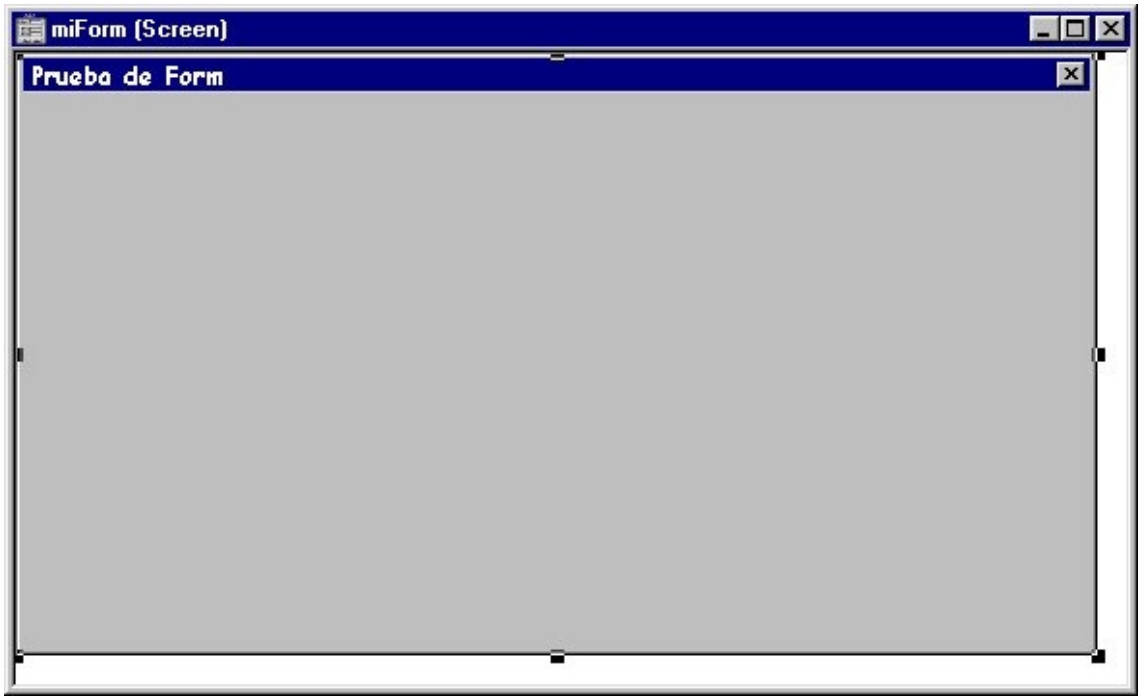


Figura 7.7. Componente Screen de la clase miForm.

En el código de esta nueva clase hemos escrito un método que ejecuta una serie de métodos de la clase form:

- Mostramos la form de forma no modal
- Creamos un MessageBox que inhabilita momentamente la form en ejecución y que muestra el handle asociado a la form.
- Mostramos el Handle
- Si contestamos afirmativamente a la pregunta de si se quiere cerrar, invocamos al método close.
- Mostramos un mensaje dependiendo si esta o no abierta la form

En el código del módulo, llamamos al método de la clase miForm y posteriormente ejecutamos de manera modal la form.

```
CONSTANTS  begin
end
CLASSES BEGIN
    miForm is Form begin
        objects begin
        end
        //interfaz de miForm
        INTERFACE
            POSITION 0 0 560 312
            FOREGROUND RGB 0 0 0
            BACKGROUND COLOR lightgray
            FONT "Comic Sans MS" 10 BOLD
            LABEL "Prueba de Form"
            SYSMENU
        BEGIN
```

```
            CONTROL AS BOX
                POSITION 0 0 552 287
                FOREGROUND RGB 0 0 0
                ATTACH ALL
                NOLABEL
                NOBORDER
            BEGIN
            END
        END
    end
END
OBJECTS BEGIN
    f AS miForm
END
CODE CLASS miForm BEGIN
// Inicio del código de la clase

//Se llama a una serie de metodos de la
//clase form sobre otro objeto form creado
public function metodos
begin
 self.Open;
 "Mostramos la form en no modal".Trace;
 MessageBox("Handle: "+(self.Hwnd).Char, "Información",0);
 if (Ok("¿Quiere cerrar la pantalla?" , "Información" )==true)
    then Close;
    else “No se cierra”.Trace;
 if self.IsOpen
   then begin
      MessageBox("Esta abierta la form ", "Información",0);
    end
   else MessageBox("Error la form no esta abierta", "Información",0);
end

// Fin del código de la clase
END

CODE BEGIN
// Inicio del código del módulo
main
begin
 f.metodos;
 MessageBox("Mostramos la form en forma modal","Información" ,0);
 f.Run;
end
// Fin del código del módulo
END
```

Una muestra de la ejecución de este módulo en el momento que se encuentra ejecutando dentro de métodos la segunda instrucción, es la siguiente:

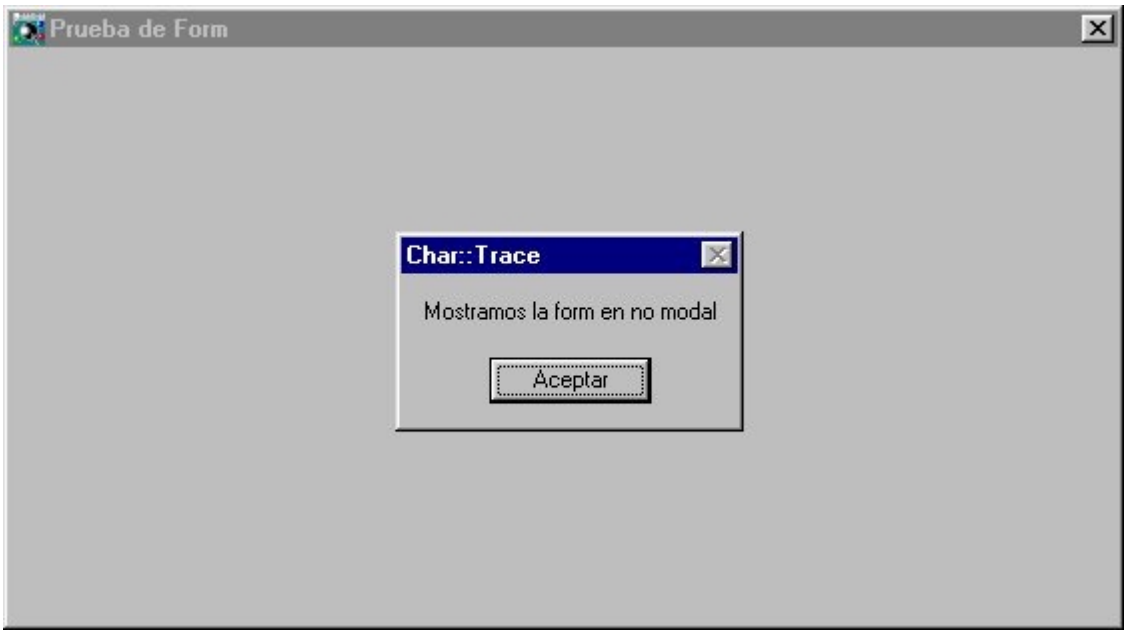


Figura 7.8. Ejecución del módulo.

En este momento tenemos abierta una ventana en forma no modal y sobre ella un mensaje que al ser no modal, deshabilita la form.

Figura 7.9. Ejecución del módulo.

Ahora tenemos la form creada en forma modal, mediante la instrucción f.Run que se encuentra en el código del módulo.

Clase Menu

Un menú Cosmos es un objeto gráfico que permite al usuario de una aplicación elegir entre una serie de opciones disponibles.

Método	Funcionalidad
Add	Permite añadir opciones o submenús a un menú existente mientras se ejecuta la aplicación, para proporcionar más información u opciones al usuario. Este método permite añadir ítems a un menú de forma dinámica.
Delete	Este método permite borrar una opción o submenú de un menú
Option	Este método retorna el submenú u opción que tiene un determinado identificador
Track	Este método muestra un menú en pantalla como “Popup Menu”.

Para explicar el funcionamiento de la clase Menu creamos un componente miMenu dentro de la sección Menús de la clase miForm. Aparece la ventana donde se especifican las propiedades de la clase.

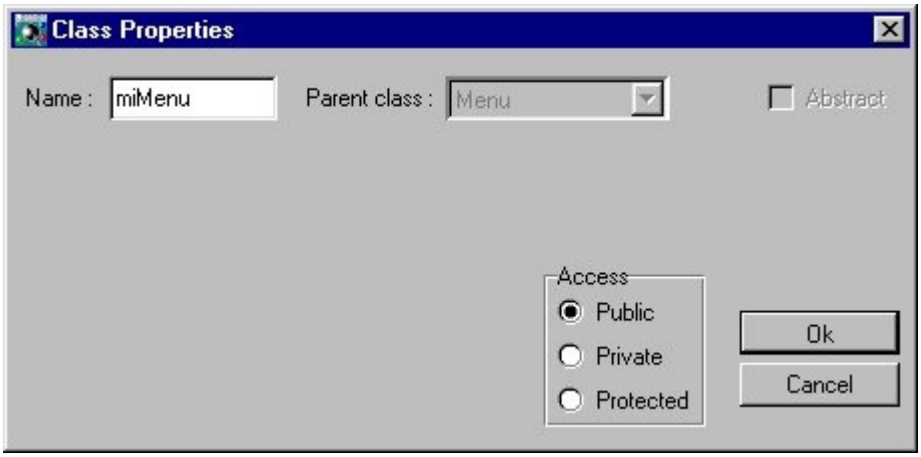
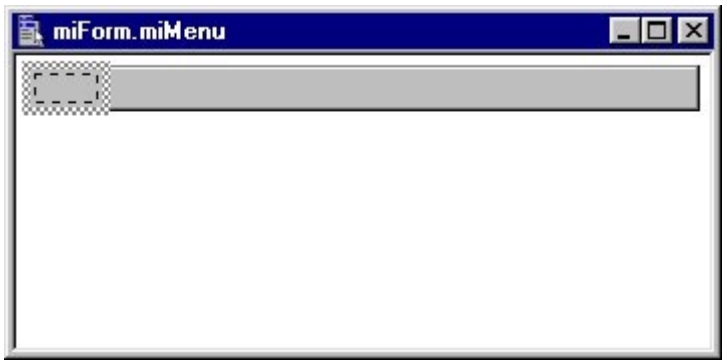


Figura 7.10. Propiedades de la clase miMenu.

Aparece entonces el nuevo componente en el arbol de la estructura del módulo. Si hacemos doble click sobre miMenu, nos aparece la pantalla donde podemos ir definiendo las opciones del menú:



7.11. Pantalla del menú.

Debemos pulsar con el botón derecho del ratón y seleccionar del menú desplegable la opción Properties...

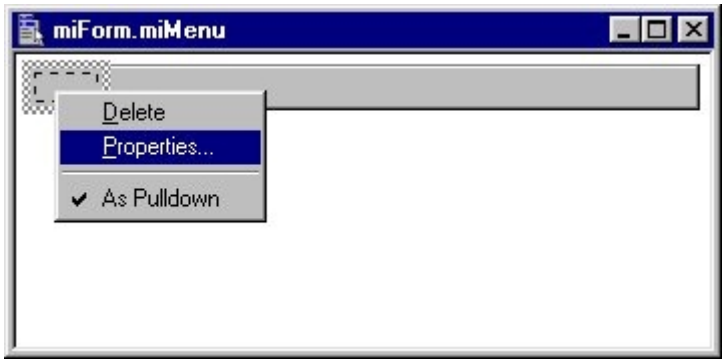


Figura 7.12. Seleccionar la opción Propiedades.

En la ventana de propiedades del nodo se introducen las propiedades que deseamos que cumpla el nodo introducido:

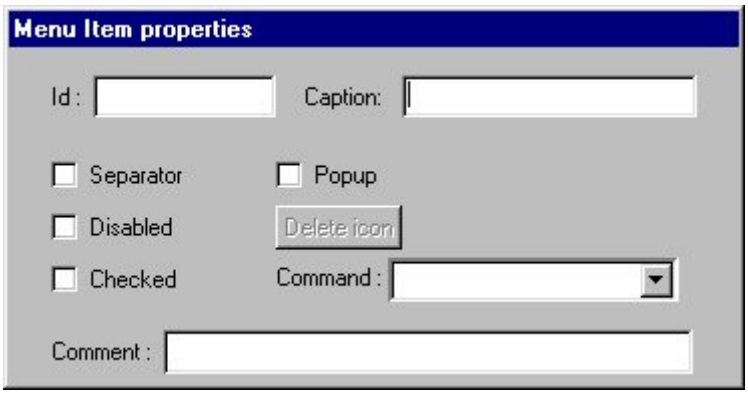


Figura 7.13. Propiedades del nodo.

Ahora debemos asociar el menú a la screen para lo que debemos seguir los siguientes pasos:

- Seleccionamos la opción “Screen layout” del menú “Layout

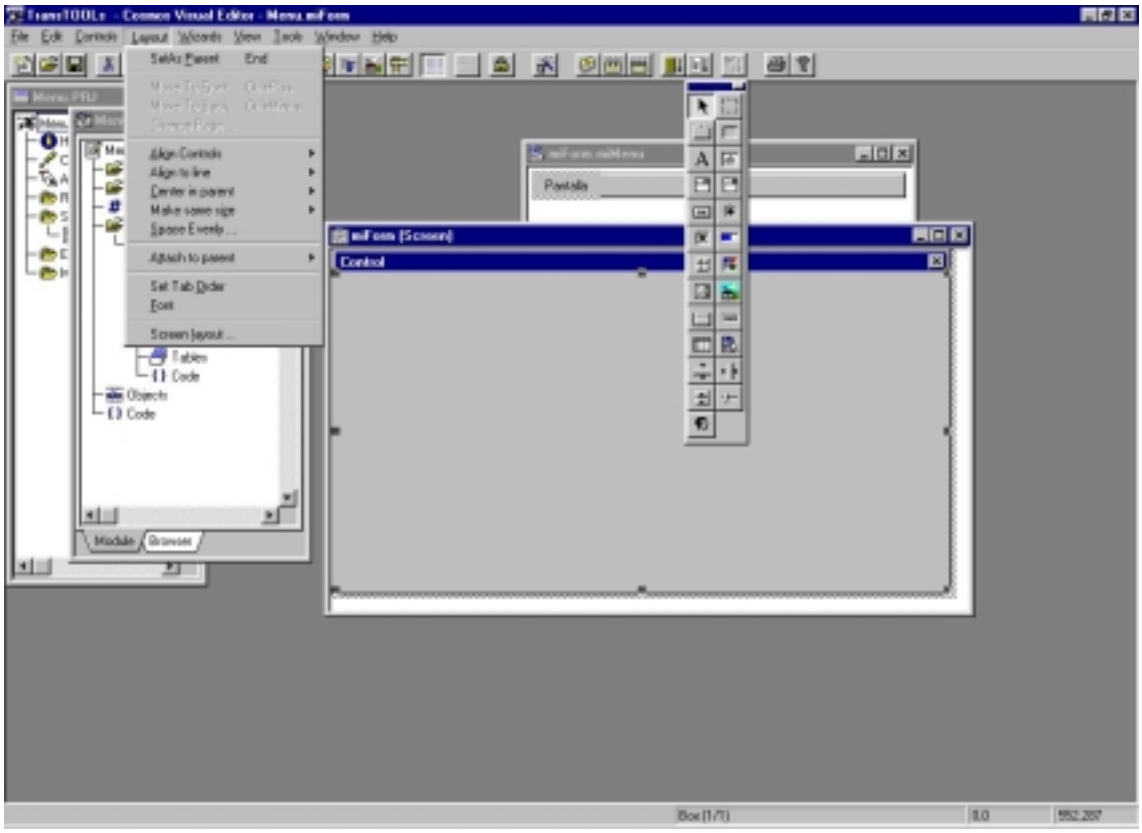


Figura 7.14. Seleccionar la opción Screen Layout.

- En el cuadro de diálogo “Frame layout” marcamos la casilla de verificación “Menu inside”.
- Seleccionamos en la lista desplegable el menú que desea asignar.
- Seleccionamos el tipo y la posición del menú.

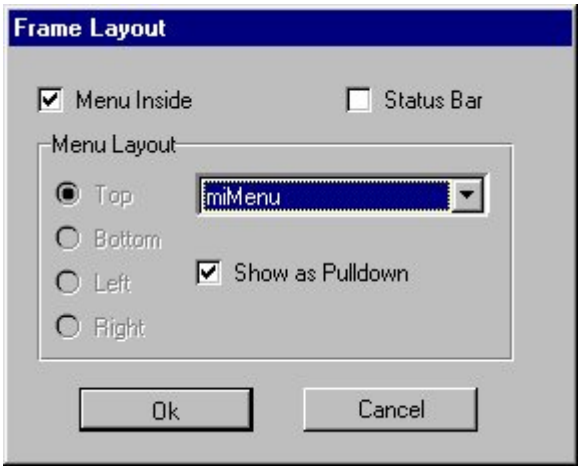


Figura 7.15. Selección del menú que queremos asignar a la Screen.

El aspecto que devuelve nuestro menú cuando ejecutamos el código del método es el siguiente:

Figura 7.16. Ejecución del módulo.

Clase Module

Un módulo es una unidad de ejecución que contiene componentes (clases, objetos, etc.) que definen el funcionamiento de la aplicación. Todos los módulos de programación de Cosmos tienen la misma estructura definida. Sin embargo podemos agrupar los módulos de un proyecto basándonos en su funcionalidad. Existen tres tipos de módulos Cosmos: Programas, librerías e includes.

- Library: Una librería es un módulo que exporta (permite su uso desde otros módulos) sus métodos públicos. Sus métodos privados y protegidos, no se exportan y por tanto no pueden ser utilizados en el módulo que incluya la librería.
- Include: Un Include es un módulo que exporta sus métodos públicos, clases públicas, objetos globales públicos y constantes. Así mismo un include exporta los métodos, clases, etc, públicos que incluye a su vez.
- Program: Un programa no exporta nada de su contenido. Desde otro módulo sólo se puede acceder a él cargándolo en memoria por medio de las instrucciones Run y Load. La comunicación entre programas se establece a través de los parámetros pasados a la función Main del módulo llamado.

En Cosmos los módulos en sí son objetos que pertenecen a una clase derivada de la clase predefinida Module. Esta clase es virtual. Los veremos con mas detalle en el próximo capítulo.

Clase Control

Cosmos ha sido diseñado con el fin de proporcionar al programador acceso a todos los controles de Windows de forma sencilla, sin requerir el conocimiento de los cientos de funciones de interfaz gráfico de las librerías de Windows ni de su complejo funcionamiento.

Esta clase derivada de la clase Complex es virtual y encapsula las propiedades y métodos genéricos para manejar:

- Los controles de una screen de un Form.
- Los controles de impresión de una página.

Métodos de la clase Control

Método	Funcionalidad
CoordToScreen	Este método convierte coordenadas relativas al control en coordenadas absolutas de pantalla
GetFramePos	Este método devuelve las coordenadas del control relativas al área de cliente de la ventana principal del Form
GetPos	Este método retorna las coordenadas del control respecto al área de cliente de su padre
GetScreenPos	Este método devuelve las coordenadas absolutas del control en la pantalla
GetSize	Este método retorna las dimensiones (altura y anchura) del control
Move	Este método permite cambiar la posición de un control dentro del área de cliente de su padre.
Name	Este método devuelve el identificador del control.
Next	Este método sirve para recorrer la estructura de controles de un Form o una página de impresión. Dependiendo del tipo de relación que se le pasa como parámetro, devuelve una referencia al control padre, hijo, siguiente o anterior.
ScreenToCoord	Este método convierte coordenadas absolutas de pantalla en coordenadas relativas al control que llama al método
Size	Este método permite modificar el tamaño de un control
Type	Este método retorna el tipo del control

Clase SimpleControl

Esta clase virtual derivada de la clase Control encapsula las propiedades y métodos para el manejo de controles gráficos de Form y Page.

Los diferentes controles gráficos son los que se nos muestran cuando accedemos a la sección Screen de Form o Template de Page.

Como ejemplo del uso de los controles gráficos, realizaremos una pequeña aplicación que consiste en diseñar una Screen que contiene un campo de edición y un botón.

En primer lugar creamos una nueva clase miForm derivada de Form, si hacemos doble click sobre la seccion Screen de miForm se nos muestra el aspecto de la Screen asociada a miForm:

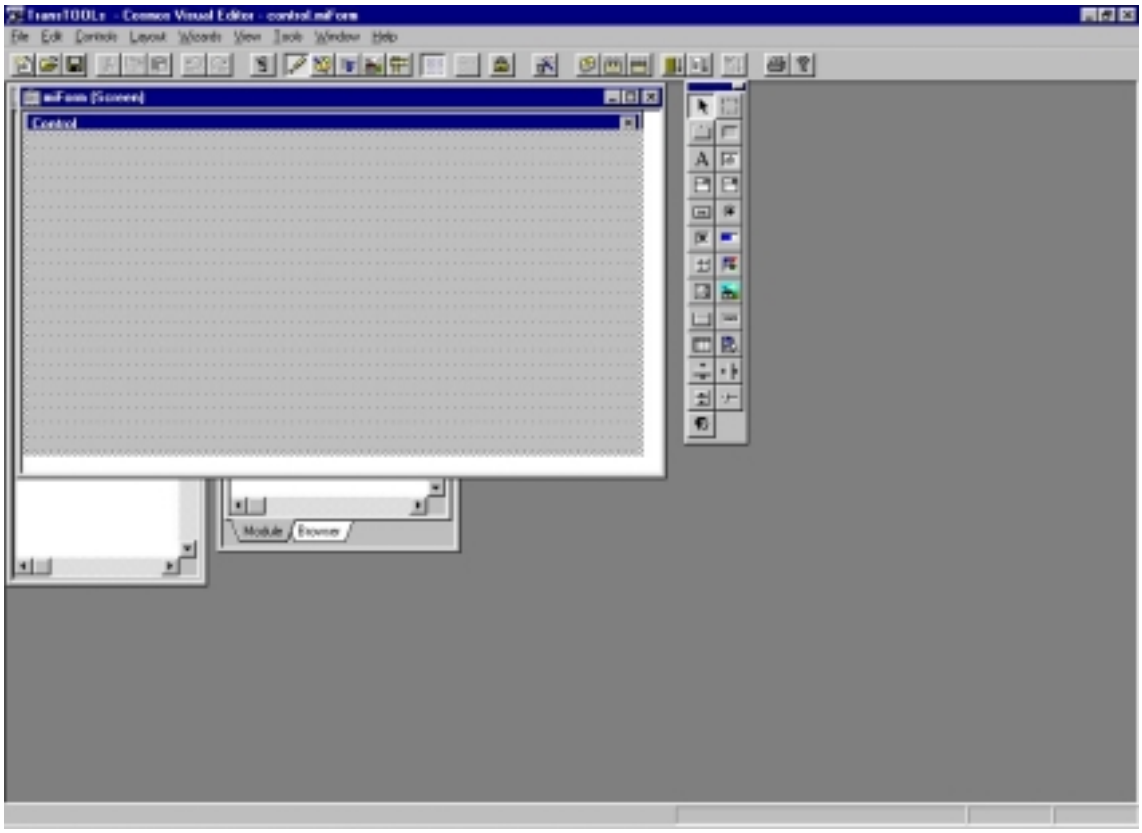


Figura 7.17. Aspecto de la Screen de miForm.

Seleccionamos de la paleta de controles el control botón y lo arrastramos sobre el tapiz de la Screen, hacemos lo mismo con el control de campo de edición.

Seleccionamos el control botón y haciendo doble click sobre él visualizamos sus propiedades. Añadimos en la pestaña Special el comando miOk, que significa que cuando se pulsa este botón se ejecuta ese comando.

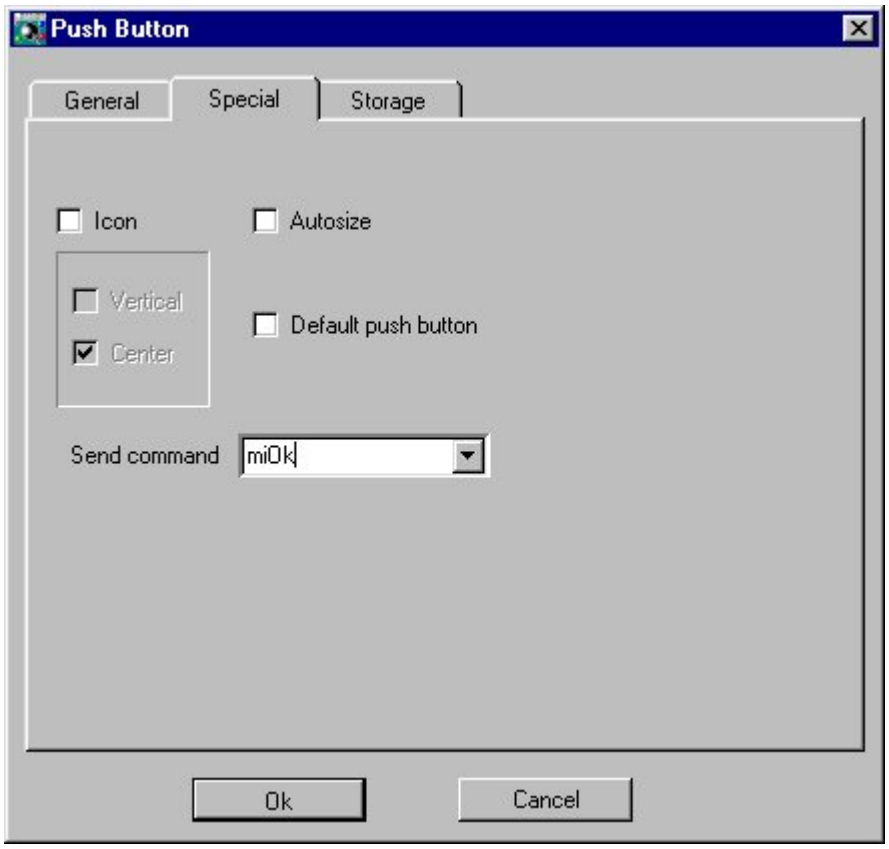


Figura 7.18. Comando que envia el control Botón.

En la pestaña Storage del control de campo de edición seleccionamos Variable para indicar que el valor que se escriba en este control va a otorgar valor a una variable que estará ya definida en la sección Variables o la definiremos en este momento pulsando el botón New:

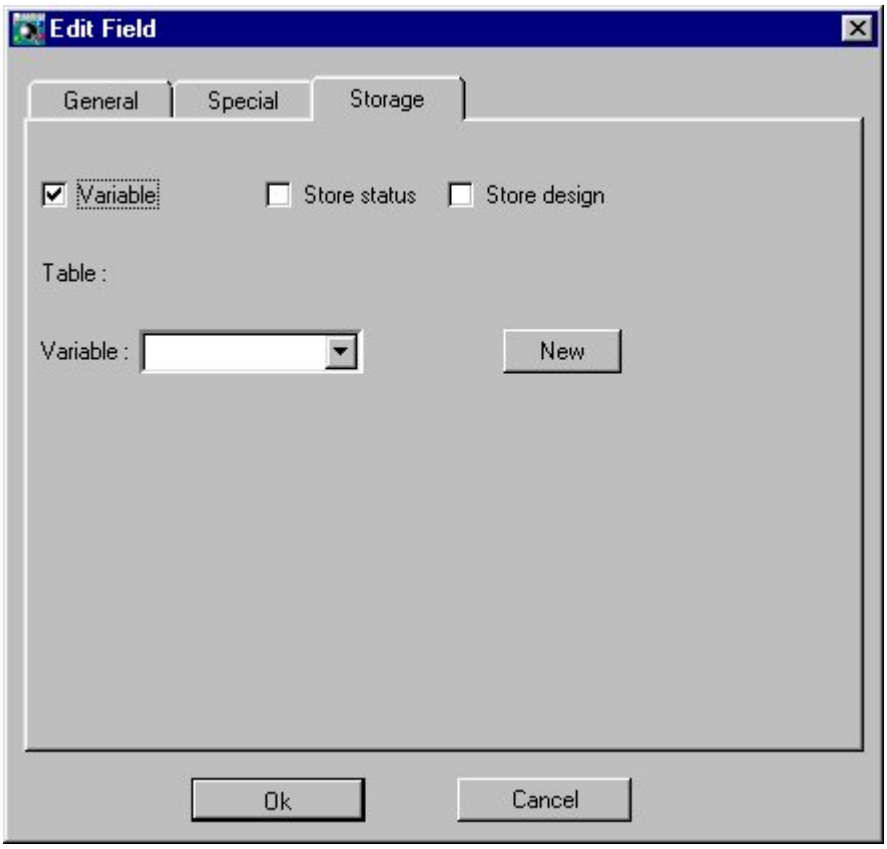


Figura 7. 19. Variable donde almacenar el valor del campo de Edición

Una vez realizados los pasos anteriores vamos a introducir el código de la clase miForm. Unicamente escribimos el código del comando que envía el control Botón. Mostraremos un mensaje y el valor actual de la variable s:

```
on command miOk
begin
 "Se ha enviado el comando ok al pulsar el botón ok".Trace;
 s.Trace;
end
```

El código del módulo únicamente consta de la definición de un objeto de la clase miForm, y la llamada del método Run de la clase Form sobre ese objeto:

```
main
objects
begin
 f as miForm
end
begin
 f.Run;
end
```

Cuando ejecutamos este módulo el resultado que obtenemos es el siguiente:

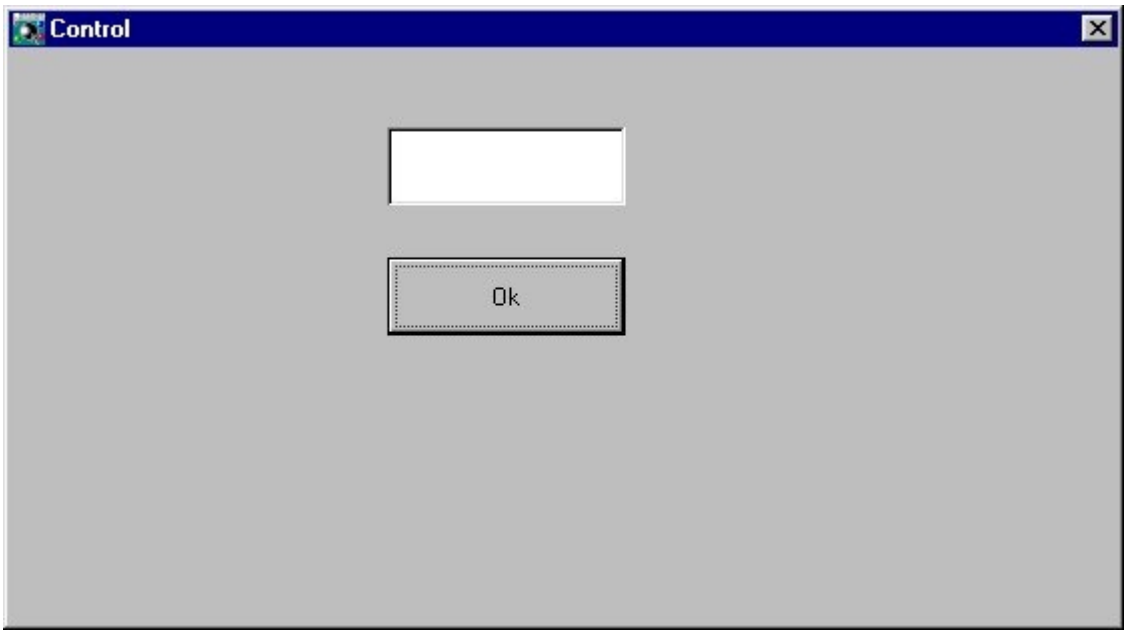


Figura 7.20. Ejecución del módulo.

Introducimos una cadena de caracteres en el campo de edición y pulsamos es botón Ok, esto provoca que se muestre un mensaje y posteriormente se muestra el contenido de la variable s que coincide con el valor del campo de edición:

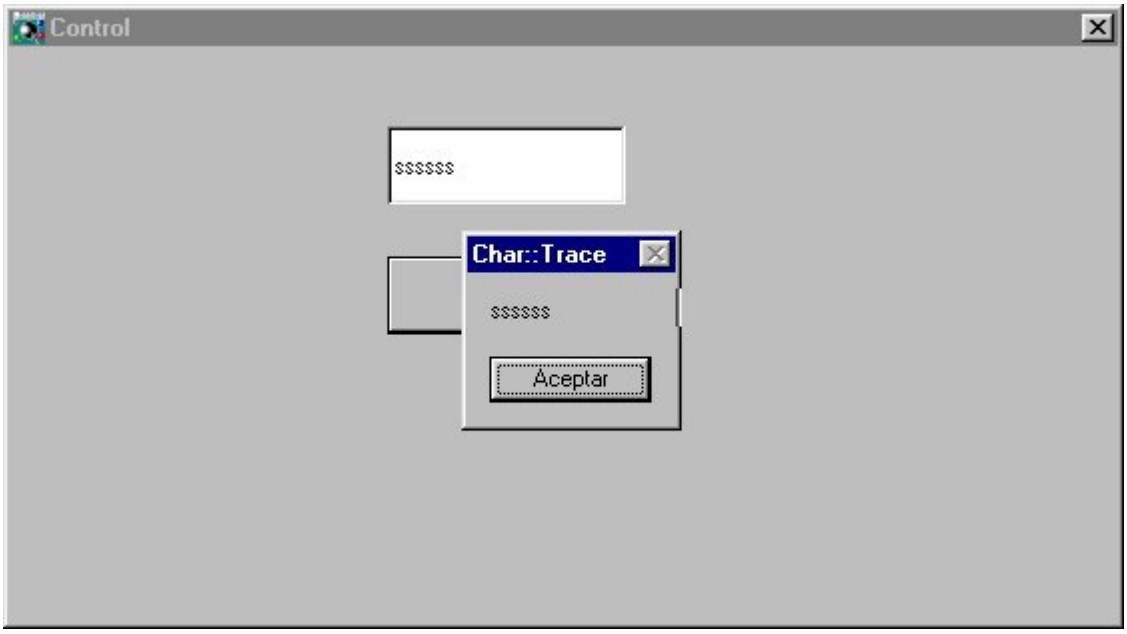


Figura 7.21. Pulsación del Botón Ok.

Capítulo 8

Los Módulos de un Proyecto

Introducción

En este capítulo vamos a estudiar los diferentes tipos de módulos que puede tener un proyecto de Cosmos. Un módulo es una unidad de ejecución que contiene componentes que definen el funcionamiento de la aplicación. Los módulos en Cosmos se dividen en Library, Includes y Program.

Library

Una librería es un módulo que exporta (permite su uso desde otros módulos) sus métodos públicos. Sus métodos privados y protegidos, no se exportan y por tanto no pueden ser utilizados en el módulo que incluya la librería.

Para definir una librería en un proyecto debemos haber definido previamente la librería, para ello añadimos un nuevo Módulo al proyecto y marcamos la casilla “Library” en la ventana de propiedades de nodo. La estructura de este nuevo modulo es:

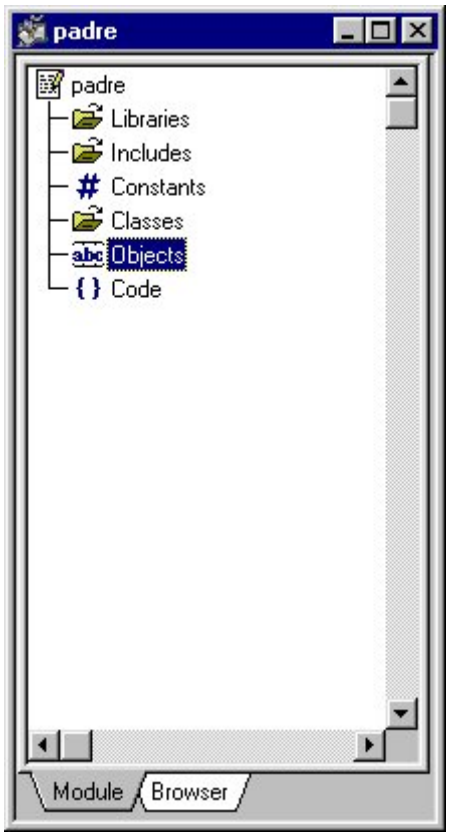


Figura 8.1. Estructura del Modulo Padre.

En el código de esta nueva librería vamos a escribir dos métodos, uno llamado ver que devuelve la cadena “abc” y el método cambiar que devuelve la cadena “cba”.

```
CONSTANTS  begin
end
CLASSES BEGIN
    miClase is Char (10)
END
CODE CLASS miClase BEGIN
//{{CODEBEGIN

//{{CODEEND
END
CODE BEGIN
//{{CODEBEGIN

public function ver return char begin
 return "abc" ;
end

public function cambiar return char begin
 return "cba" ;
end
//{{CODEEND
END
```

Crearemos otro módulo dentro del proyecto que será desde el que invocaremos estos métodos. Para poder acceder a esos módulos debemos añadir en la sección Libreries del módulo el nombre de la librería que los implementa.

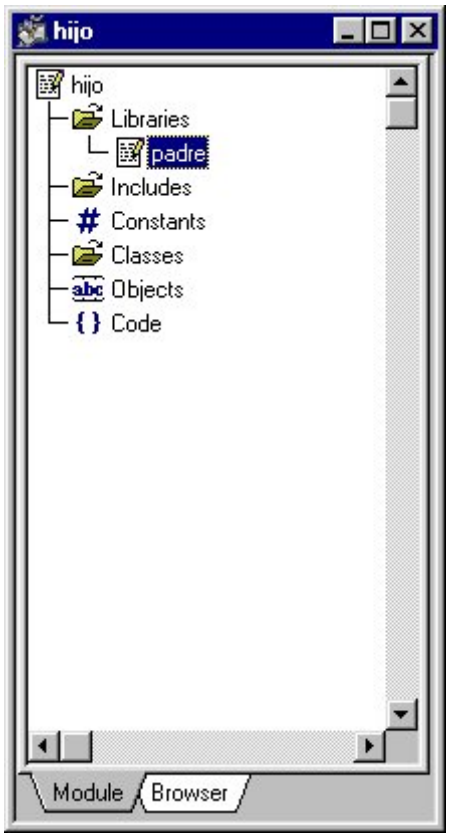


Figura 8.2. Estructura del módulo Hijo.

Entonces ya podemos escribir el código y tratar a los métodos de la librería como del propio módulo:

```
LIBRARIES BEGIN
    padre
END
CONSTANTS  begin
end
CODE BEGIN
//{{CODEBEGIN
main
objects
begin
 c1 as char
end
begin
 c1=ver;
 c1.Trace;
 c1=cambiar;
 c1.Trace;
end
//{{CODEEND
END
```

Includes

Un include exporta su declaración de constantes, su definición de clases públicas, objetos públicos y métodos públicos. Sus métodos y clases privados y protegidos, no se exportan y por tanto no pueden ser utilizados en el programa que lo incluya. Luego los módulos que utilicen ese include pueden utilizar además de los métodos, las clases.

Para demostrar el uso de los includes hemos realizado un pequeño módulo llamado Hijo que declara en su sección de Includes a “Padre”. Podrá utilizar los métodos y clases que se encuentren en Padre:

```
INCLUDES BEGIN
    padre
END
CONSTANTS  begin
end
CODE BEGIN
main
objects
begin
 c2 as miClase //Clase que se encuentra definida en Padre
end
begin
 c2=c2.ver; //Método de la clase miClase
 c2.Trace;
end
END
```

El código del modulo Padre es el siguiente:

```
CONSTANTS  begin
end
CLASSES BEGIN
   miClase is Char (10)
END
CODE CLASS miClase BEGIN
//{{CODEBEGIN
public ver return Char begin
  return "abc";
end

END
CODE BEGIN
END
```

Al tratarse de un include podemos crear una instancia de miClase en el modulo Hijo e invocar sus métodos.

La estructura del módulo que tenemos despues de incluir el Padre en la sección includes es:

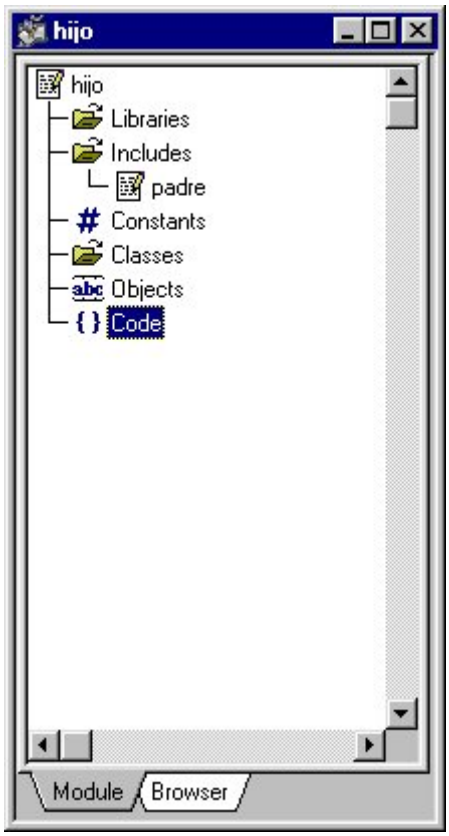


Figura 8.3. Estructura del módulo con su include.

Program

Un programa no exporta nada de su contenido. Desde otro módulo sólo se puede acceder a él cargándolo en memoria por medio de las instrucciones Run y Load. La comunicación entre programas se establece a través de los parámetros pasados a la función Main del módulo llamado.

Todos los módulos que hemos realizado a lo largo de esta guía eran de tipo Program.

Anexo A

Convenciones utilizadas

Las convenciones empleadas en Cosmos para presentar la sintaxis de cualquier elemento son las siguientes:

[ ]	Aquellos elementos que se encuentren entre corchetes son opcionales.
{ }	Indica la obligatoriedad de incluir uno de los elementos encerrados entre llaves. Dichos elementos estarán separados por el carácter de «barra vertical» (carácter ASCII 124).
|	Este carácter se utiliza para separar los diferentes elementos opcionales u obligatorios de la sintaxis, dependiendo de si éstos van entre corchetes o entre llaves, respectivamente.
...	El elemento anterior a los puntos puede aparecer más veces en la sintaxis.
,...	Como el caso anterior, pero cada nueva aparición del elemento debe ir precedida por una coma.
palabra	Las palabras en minúsculas corresponden con identificadores COOL o expresiones, etc.
PALABRA	Las palabras en mayúsculas y en negrita son reservadas del lenguaje COOL por tanto invariables.
subrayado	Si no se especifica otra opción, la palabra o palabras subrayadas se toman por defecto.
negrita	Cualquier signo de las convenciones que se encuentre en negrita es obligatorio en la sintaxis.

El resto de signos se utilizan con su valor. Así, en aquellos lugares en los que se pueden especificar comillas, éstas podrán ser simples o dobles (", '), con la obligatoriedad de cerrar el texto con el mismo tipo con el que se abrió. Asimismo, se pueden incluir comillas como parte de un texto, en cuyo caso, dichas comillas deberán ser necesariamente del tipo contrario a las que sirven de delimitadores del texto. Es decir, en el caso de definir un literal entre comillas que incluya una parte también entrecomillada, habrá que combinar ambos tipos.

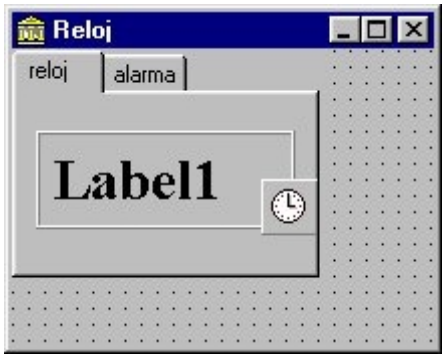
Anexo B

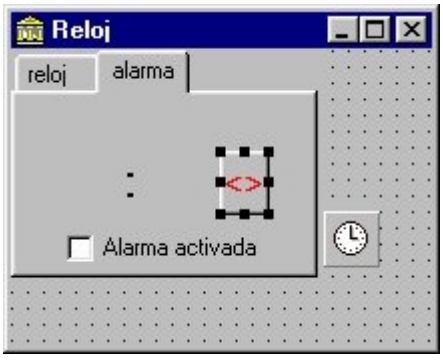
Código de la DLL

Código de Uti.dpr

```
library Uti;
uses

  SysUtils,
  Classes,
  Reloj in 'Reloj.pas' {Form2};

exports
 ver_reloj;
begin
end.
```

Aspecto del Form utilizado





Código de Reloj.pas

```
unit reloj;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, ExtCtrls, ComCtrls, Mask, Buttons;

type
  TForm2 = class(TForm)
    PageControl1: TPageControl;
    TabSheet1: TTabSheet;
    TabSheet2: TTabSheet;
    Panel1: TPanel;
    Label1: TLabel;
    Timer1: TTimer;
    CheckBox1: TCheckBox;
    SpeedButton1: TSpeedButton;
    Label2: TLabel;
    Edit1: TEdit;
    Edit2: TEdit;
    procedure Timer1Timer(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure SpeedButton1Click(Sender: TObject);

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

 var alarm,ahora:TdateTime;
    Halarm,Malarm,Salarm,MSalarm,Hnow,Mnow,Snow,MSnow:word;

procedure ver_reloj;Export;StdCall;

implementation

{$R *.DFM}

procedure TForm2.Timer1Timer(Sender: TObject);
begin
label1.caption:=timetostr(time);
if checkbox1.checked=true
 then
  begin
    DecodeTime(alarm, Halarm, Malarm, Salarm, MSalarm);
    ahora:=now;
    DecodeTime(ahora, Hnow, Mnow, Snow, MSnow);
  if (halarm=Hnow)and (Malarm=Mnow)
   then
    begin
    beep;beep;beep;
    end;
  end;
end;
```

```
procedure ver_reloj;
 var ficha:Tform2;
 begin
  ficha:=TForm2.Create(Nil);
  ficha.Show;
 end;
procedure TForm2.FormCreate(Sender: TObject);
begin
label1.caption:=timetostr(Time);
end;

procedure TForm2.FormClose(Sender: TObject; var Action: TCloseAction);
begin
release;
end;

procedure TForm2.SpeedButton1Click(Sender: TObject);
begin
alarm:=encodetime(strtoint(edit1.text),strtoint(edit2.text),0,0);
end;

end.
```

Títulos de la Colección

Ingeniería Informática

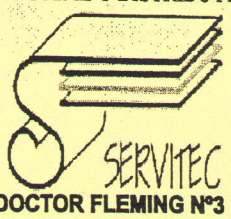
Nº CUADERNO	TITULO	ISBN
1	ANÁLISIS Y DISEÑO ORIENTADO A OBJETOS	84-8497-802-8
2	DISEÑO DE PÁGINAS WEB USANDO HTML	84-8497-803-6
3	FORMATOS GRÁFICOS	84-8497-804-4
4	PROGRAMACIÓN EN LENGUAJE C	84-8497-805-2
5	ADMINISTRACIÓN DE WINDOWS N 4.0	84-8497-899-0
6	GUÍA DE USUARIO DE DERIVE PARA WUIDOWS	84-8497-919-9
7	GUÍA DE REFERÊNCIA DE MULTIBASE COSMOS	84-8416-001-7
8	GESTIÓN DE UN TALLER MULTIBASE CÓSMOS	84-8416-034-3
9	EJERCICIOS DE LÓGICA INFOMÁTICA	84-8416-357-1
10	CONEPTOS BÁSICOS DE PROCESADORES DE LENGUAJE	54-8416-889-1
11	INTRODUCCIÓN A LA ADMINISTRACIÓN DE UNIX	84-8416-570-1
12	LÓGICA PROPOSICIONAL PARA LA INFOMÁTICA	84-8416-613-9
13	PROGRAMACIÓN PRÁCTICA EN PROLOG	84-8416-612-0
14	LA HERRAMIENTA CASE META COSMOS	84-699-0054-4
15	GUIA DE LENGUAJE COOL DE MULTIBASE COSMOS	84-699-0053-6
16	GUÍA DE REFERÊNCIA PROGRESS	84-699-2083-9
17	GESTIÓN DE DEPOSITO DENTAL CON PROGRESS	84-699-2082-0
18	GUIA DE DISEÑO Y CONS. REPOSICIÓN MUL. COSMOS	84-699-2081-2
19	GESTIÓN Y ADMIN. DE UN COLEGIO MAYOR MUL-COSMOS	84-699-2080-4
20	MODELADO DE SOFWARE EN UML	84-699-2079-0
21	EL LENGUAJE DE PROGRAMACIÓN JAVA	84-699-3880-0
22	PRINCIPIOS DE ALGORITMIA	84-699-6537-9
23	LISTAS, PILAS Y COLAS	84-699-6536-0
24	RECURSIVIDAD	84-699-6535-2
25	ARBOLES	84-699-6849-1
26	CONJUNTOS Y TABLAS DE DISPERSIÓN	84-699-7398-3
27	ALGORITMOS DE ORDENACIÓN	84-699-7397-5
28	TEORÍA DE GRAFOS	84-699-8362-8
29	IINTR. AL PROCES. EFEC. DE TEXTOS CON MICROSOFT WORD	84-699-9618-5
30	ORACLE SQL	84-688-0420-7
31	TÉCNICAS DE DISEÑO DE ALGORITMOS	84-688-1764-3
32	LA PLATAFORMA . NET	84-688-3449-1
33	EJERC. DE HOJAS DE CÁL. EXCEL APLICADOS A LA GES.EMPRES.	84-688-3450-5
34	TIPOS ABSTRACTOS DE DATOS	84-688-4209-5
35	INTERPRETES T DISEÑO DE LENGUAJES DE PROGRAMACIÓN	84-688-4210-9
36	LENGUAJES Y AUTOMATAS EN PROCESADORES DE LENGUAJE	84-688-4211-7
37	METODOLOGIA DE LA PROGRAMACIÓN: GUIA DEL ALUMNO	84-688-5901-4
38	ANÁLISIS SEMÁNTICO EN PROCESADORES DE LENGUAJE	84-688-6208-8
39	ARQUITECTURA WEB EN APLICACIONES JAVA/J2EE	84-688-6580-9
40	ALGORITMICA CON FORTRAN 90	84-688-7250-4
41	TABLAS DE SÍMBOLOS EN PROCESADORES DE LENGUAJES	84-688-7631-3
42	INTRODUCCIÓN A LA COMINICACIÓN PERSONA MÁQUINA	84-688-8362-X
43	INTRODUC. A LA PROG. ORIENTADA A OBJETOS CON JAVA	84-688-8826-6
44	DESARR. APLIC. EN SISTEM. DISTRIBUIDOS E INTERNET	84-689-3379-1
45	LÓGICA DE PREDICADOS	84-689-3380-5
46	LENGUAJE C#	84-689-5893-X
47	INTEGRACIÓN DE APLICACIONES OFIMÁTICAS	84-689-5892-1
48	INFORMÁTICA GENERAL	84-689-7033-6
49	PROYECTOS INFORMÁTICOS	

IMPRIME Y DISTRIBUYE

C/DOCTOR FLEMING Nº3
33005 OVIEDO
TLF-FAX 985 250581
E-mail:copisteriaservitec@fade.es

Consultor Editorial
Juan Manuel Cueva Lovelle
cueva@lsi.uniovi.es

IMPRIME Y DISTRIBUYE



C/DOCTOR FLEMING Nº3
33005 OVIEDO
TLF-FAX 985 250581
www.fade.es/copisteriaservitec
E-mail:copisteriaservitec@fade.es