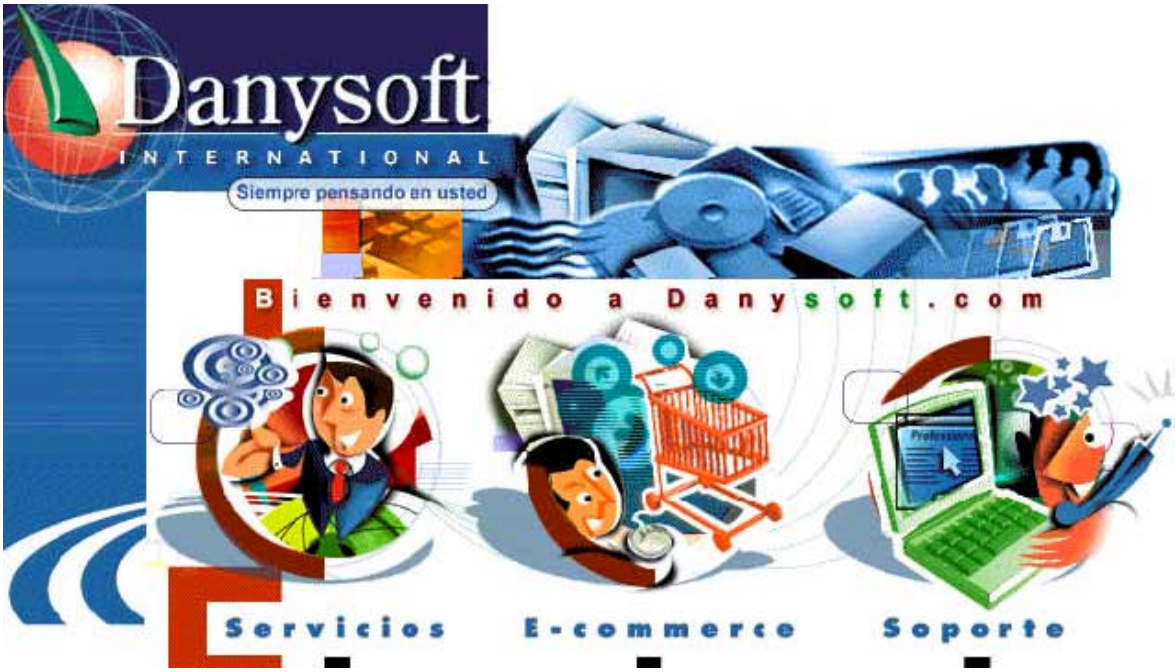
Guías técnicas Grupo Danysoft:

Creación de Componentes en Delphi

(parte I)



Por Pablo Reyes – Grupo Danysoft
mayo de 2001 - (902) 123146
www.danysoft.com

Primera parte

Este artículo es el primero de una serie de tres con los que pretendo explicar los fundamentos teóricos mínimos necesarios para la creación de componentes con Delphi.

En esta primera parte veremos aspectos básicos de la programación orientada a objetos con object pascal, el lenguaje de programación de Delphi.

Introducción a Clases y Objetos

Programación Orientada a Objetos

La programación orientada a objetos es una extensión de la programación estructurada que pone énfasis en la reutilización de código y en la encapsulación de datos con funcionalidades. Antes de la programación orientada a objetos los datos y las operaciones sobre los datos eran tratados como elementos separados.

El punto de partida en la programación orientada a objetos lo constituyen las clases. Una clase es un tipo de dato que encapsula datos y operaciones sobre esos datos en una sola unidad. Estas operaciones, llevadas a cabo por procedimientos y funciones, son llamadas métodos, mientras que los datos son llamados elementos o campos y pueden ser accedidos directamente o a través de propiedades. Muchas veces los métodos, elementos o campos y propiedades de una clase son llamados genéricamente miembros de la clase.

Cuando creamos un nuevo proyecto, Delphi crea un formulario nuevo para que podamos personalizarlo. En el editor de código, Delphi declara un nuevo tipo de clase para el formulario nuevo. El código generado es parecido al siguiente:

```
unit Unit1;

interface

uses Windows, Classes, Graphics, Forms, Controls, Dialogs;

type

  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

Implementation

{$R *.DFM}

end.
```

Código Fuente 1 - Ejemplo de un formulario

El nombre de la clase nueva es **TForm1** y desciende de la clase **TForm**, una de las tantas clases de la VCL. Inicialmente la clase **TForm1** parece no hacer nada, ya que no tiene métodos ni elementos declarados. Sin embargo, posee todos los métodos y elementos de la clase **TForm**, su clase ancestra. A medida que coloquemos componentes y escribamos manejadores de eventos, estaremos creando elementos y métodos para la nueva clase ampliando su clase ancestra.

Delphi también declara una variable del tipo de la clase nueva, es decir, **TForm1**. Esta declaración de variable no es diferente de la declaración de una variable, por ejemplo, de tipo integer. En general, la declaración de una variable significa que la variable declarada es capaz de alojar un elemento del tipo de dato de la declaración. Asi, una variable de tipo integer es una variable capaz de alojar a un dato de tipo integer, pero la declaración en sí no le asigna a la variable un dato de ese tipo. De la misma forma, la delcaración de la variable Form1 del tipo **TForm1** significa que la variable Form1 es capaz de alojar un dato del tipo **TForm1**, pero la declaración en sí no le asigna a la variable un dato de tipo **TForm1**.

Por lo tanto, la variable Form1 es capaz de representar una instancia de la clase **TForm1** o, lo que es lo mismo, un objeto del tipo **TForm1**. Es posible crear tantas instancias u objetos como sea necesario y cada una de ellas tendrá su propia copia de los datos aunque todas compartirán los mismos métodos comportándose exactamente de la misma manera.

Clases y Objetos

En Delphi todas las clases descienden de la clase **TObject**. La clase **TObject** es una clase que encapsula el comportamiento fundamental común a todos los objetos de la VCL. Introduce métodos que proveen la habilidad de crear, mantener y destruir instancias de una clase mediante la asignación, inicialización y liberación de memoria.

Al declarar una clase nueva en Delphi es necesario que descienda de alguna de las clases existentes. Si en la delcaración no se especifica la descendencia, Delphi asume **TObject**. Asi, la siguiente declaración de clase:

```
type
  TMiClase = class
  end;
```

Código Fuente 2 - Declaración de una clase

declara la clase **TMiClase** descendiente de la clase **TObject**, lo que sería equivalente a:

```
type
  TMiClase = class(TObject)
  end;
```

Código Fuente 3 - Declaración de una clase

Visibilidad de los miembros de una clase

Cuando declaramos un campo, propiedad o método en una clase el nuevo miembro tendrá una visibilidad indicada por alguna de las palabras clave

private, ***protected***, ***public*** o ***published***. La visibilidad determina la accesibilidad del miembro desde otras clases y unidades.

Un miembro ***private*** puede ser accedido sólo dentro de la unidad en el que es declarado. Un miembro ***protected*** puede ser accedido dentro de la unidad en el que es declarado y dentro de cualquier clase descendiente aunque la misma esté declarada en otra unidad. Un miembro ***public*** puede ser accedido desde cualquier lugar en el que la clase o una instancia de la clase sea accesible. Un miembro ***published*** tiene la misma visibilidad que un miembro ***public*** con la diferencia que los miembros ***published*** aparecen en el Object Inspector en tiempo de diseño.

Estas palabras clave no delimitan secciones dentro de la declaración de una clase. Para determinar la visibilidad de un miembro de una clase Delphi utiliza la palabra reservada con la que fue declarado el miembro o la palabra reservada previa más cercana a la declaración del miembro o, si no encuentra ninguna palabra clave, asume ***published***. Las siguientes declaraciones de clases son todas válidas.

```
type
  TMiClase = class
    FUnMiembroPublished: string;
    procedure HacerAlgo;
  end;

type
  TMiClaseDos = class(TObject)
  private
    FUnMiembroPrivate: string;
  published
    FUnMiembroPublished: string;
    FOtroMiembroPublished: string;
  private FOtroMiembroPrivate: string;
    function HacerOtraCosa: boolean;
  public
    FUnMiembroPublic: string;
end;
```

Código Fuente 4 - Declaración de una clase

Es posible modificar la visibilidad de un miembro de la clase ancestra, pero esta modifición sólo es posible si aumenta la visibilidad que tenía el miembro en la clase ancestra. Para modificar la visibilidad sólo hay que re-declarar el miembro con la nueva visibilidad.

Campos

Los campos de una clase representan los datos de la clase. Todas las intancias de una clase (objetos) comparten el mismo código (métodos) pero cada una posee su propia copia de los datos.

Los campos de una clase pueden tener visibilidad private, protected o public (pueden tener visibilidad published si su tipo es una clase) aunque por lo general sólo tienen visibilidad private o protected debido a que, para cumplir con el concepto de encapsulación, los datos de una clase no deberían ser accedidos desde fuera de la clase.

Cuando Delphi crea una instancia de una clase (un objeto), inicializa todos los campos de la misma con el equivalente al valor nulo para el tipo de dato del campo. Asi, los campos de tipo entero son inicializados con el valor 0, los campos de tipo string con el valor ‘’, etc.

En una clase se pueden declarar campos con el mismo nombre que en clases ancestras. En ese caso el nuevo campo oculta al campo de la clase ancestra.

Los campos de una clase deberían ser accedidos desde fuera de la clase sólo por medio de métodos y/o propiedades.

Propiedades

Las propiedades son interfaces para acceder a los campos de una clase cumpliendo con el concepto de encapsulación. Por lo general, cada propiedad está directamente relacionada con un campo de la clase y por convención las propiedades tienen el mismo nombre que el campo relacionado pero sin la letra F. Asi, si una propiedad está directamente relacionada con el campo FUnValor, entonces el nombre de la propiedad debería ser UnValor.

Las propiedades son de un tipo de dato determinado el cual puede ser distinto del tipo de dato del campo relacionado. Las propiedades pueden dar acceso al valor de su campo relacionado de dos formas:

1. Directamente, en cuyo caso el tipo de dato de la propiedad debe ser igual al tipo de dato de su campo relacionado.
2. A través de un método, en cuyo caso el método debe ser implementado y el tipo de dato de la propiedad puede ser distinto al tipo de dato de su campo relacionado.

Las propiedades pueden ser de lectura y escritura, de lectura solamente o de escritura solamente. Esto esta determinado por la presencia de las palabras clave **read** y **write** en la declaración de la propiedad. A continuación de las palabras clave **read** y **write** se indica la manera de acceder al valor del campo relacionado. Para que el acceso sea directo simplemente se debe indicar el nombre del campo relacionado. Para que sea a través de un método se debe indicar el nombre del método. Por convención el nombre del método para lectura es Get + el nombre de la propiedad y el nombre del método para escritura es Set + el nombre de la propiedad.

Por ejemplo, el siguiente código muestra la declaración de una clase con propiedades.

```
type
  TMiClase = class(TObject)
  private
    FUnValor: string;
    FOtroValor: integer;
    FOtroDato: integer;
    function GetOtroValor: integer;
    procedure SetOtroValor(Value: integer);
    function GetOtroDato: string;
    procedure SetOtroDato(Value: string);
  public
    property UnValor: string read FUnValor write FUnValor;
    property OtroValor: integer read GetOtroValor write SetOtroValor;
    property OtroDato: string read GetOtroDato write SetOtroDato;
```

```
  end;

implementation

procedure TMiClase.GetOtroValor: integer;
begin
  Result := FOtroValor;
end;

procedure TMiClase.SetOtroValor(Value: string);
begin
  FOtroValor := Value;
end;

procedure TMiClase.GetOtroDato: string;
begin
  Result := IntToStr(FOtroDato);
end;

procedure TMiClase.SetOtroDato(Value: string);
begin
  FOtroDato := StrToInt(Value);
end;
```

Código Fuente 5 - Ejemplo de propiedades

En el caso de la propiedad OtroDato, el tipo de dato del campo asociado, FOtroDato, es distinto al tipo de dato de la propiedad. En casos como estos es necesario crear métodos para las operaciones de lectura y escritura del valor de la propiedad para hacer las conversiones que sean necesarias.

De esta forma se cumple el concepto de encapsulación logrando que modificaciones dentro de la clase (por ejemplo modificar el tipo de dato de un campo) no afecten al contexto en el cual la clase es utilizada.

Métodos

Los métodos representan lo que un objeto puede hacer. Desde un punto de vista más técnico, los métodos son procedimientos y funciones construídos dentro de la estructura de una clase. Si bien prácticamente no hay restricciones en cuanto a lo que pueden hacer los métodos de una clase, Delphi utiliza un par de principios básicos que deberían ser tenidos en cuenta.

1. Evitar Dependencias: es deseable eliminar al máximo la dependencia de factores externos a la clase para que un método pueda ser utilizado. Si bien en algunos casos esto es muy difícil de lograr se debería intentar lograrlo lo más posible.
   Si en la clase se incluyen métodos que en determinado momento no pueden ser llamados o métodos que deban ser llamados en un orden determinado, como mínimo dichos métodos deberían implementar los mecanismos necesarios para reconocer dichas situaciones y no producir resultados inesperados.
2. El nombre de los métodos: Delphi no impone ningún tipo de restricciones a lo hora de ponerle nombre a los métodos. Sin embargo existen una serie de convensiones para ello que deberían ser tenidas en cuenta.
   Es deseable que el nombre de un método describa lo mejor posible lo que el

método hace sin que el nombre sea demasiado largo y tratando de utilizar palabras comunes. En el caso de las funciones, es importante que el nombre refleje el valor que devuelven.

La declaración de los métodos de una clase no difiere de la declaración de procedimientos y funciones estándar de Object Pascal.

Por lo general la visibilidad de los métodos debería ser protected o public. No es recomendable que un método tenga visibilidad private salvo que las características de la clase asi lo requieran.

Eventos

Un evento es un enlace entre un suceso en el sistema (como una acción realizada por el usuario) y una porción de código que responde a dicho suceso. El código en cuestión es normalmente llamado manejador de evento y es generalmente escrito por el desarrollador de la aplicación que utiliza el componente. Por medio de los eventos los desarrolladores de aplicaciones pueden personalizar el comportamiento de los componentes sin tener que modificar sus respectivas clases. Este concepto es conocido como delegación.

Los eventos para las acciones del usuario más comunes (como las acciones del ratón) están escritos dentro de los componentes estándar aunque siempre es posible crear nuevos eventos.

```
Type
  TNotifyEvent = procedure(Sender: TObject) of object;

  TMiClase = class(TComponent)
  private
    FOnClick: TNotifyEvent;
  published
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
  end;
```

Código Fuente 6 - Ejemplo de eventos

Los eventos son implementados como propiedades con la diferencia de que su tipo de dato es siempre un puntero a método. Para ello es necesario declarar un tipo de dato de acuerdo al método que deba ser ejecutado cuando el suceso relacionado con el evento ocurra.

En el ejemplo anterior se declara el tipo de dato TNotifyEvent como un procedimiento que recibe un parámetro del tipo TObject. Las palabras clave “of object” al final de la declaración son necesarias porque el tipo de dato debe ser un puntero a método y no un puntero a procedimiento o función.

Constructores y Destructores

Constructor

Un constructor es un método especial que crea e inicializa instancias de una clase, es decir, objetos. La declaración de un constructor es similar a la declaración de un procedimiento con la diferencia que empieza con la palabra clave **constructor**. A pesar de que en la declaración de un constructor no se especifican valores de retorno, el constructor retorna una referencia al objeto creado. Por convención, el nombre del método constructor es **Create**. Así, para

crear una instancia de una clase, es decir, un objeto, debemos llamar al método constructor de la clase. Por ejemplo:

```
MiObjeto := TMiClase.Create;
```

Código Fuente 7 - Creación de un objeto

El constructor asigna memoria para el nuevo objeto e inicializa todos sus campos. Luego, el constructor retorna una referencia al objeto creado el cual es del tipo de la clase referenciada. Si ocurre un error durante la ejecución del método constructor el objeto no es creado y los recursos asignados son liberados.

A continuación se muestra un ejemplo de declaración de una clase con la implementación del método constructor.

```
type
  TShape = class(TGraphicControl)
  private
    FPen: TPen;
    FBrush: TBrush;
    procedure PenChanged(Sender: TObject);
    procedure BrushChanged(Sender: TObject);
  public
    constructor Create(Owner: TComponent); override;
    ...
  end;

constructor TShape.Create(Owner: TComponent);
begin
  inherited Create(Owner);
  Width := 65;
  Height := 65;
  FPen := TPen.Create;
  FPen.OnChange := PenChanged;
  FBrush := TBrush.Create;
  FBrush.OnChange := BrushChanged;
end;
```

Código Fuente 8 - El método constructor

La palabra clave "override" será tratada en el próximo artículo.

La primera acción que realiza el método constructor en la mayoría de los casos es llamar al método constructor de su ancestro para que se inicialicen los campos heredados. Luego el constructor inicializa los campos de la clase por lo que no hay necesidad de inicializarlos manualmente, salvo que los valores iniciales sean distintos a los asignados por defecto.

Destructor

Un destructor es un método especial que destruye al objeto para el cual es llamado y libera la memoria ocupada por el mismo. La declaración de un destructor es similar a la declaración de un procedimiento con la diferencia que empieza con la palabra clave **destructor**. Por convención el nombre del método destructor es **Destroy**. Así, para destruir un objeto se debe llamar al método destructor del objeto. Por ejemplo:

```
MiObjeto.Destroy;
```

Código Fuente 9 - El método destructor

Cuando el método destructor es llamado, primero se ejecutan las acciones implementadas en el método y luego se libera toda la memoria ocupada por el objeto.

A continuación se muestra un ejemplo de declaración de una clase con la implementación del método destructor.

```
type
  TShape = class(TGraphicControl)
  private
    FPen: TPen;
    FBrush: TBrush;
    procedure PenChanged(Sender: TObject);
    procedure BrushChanged(Sender: TObject);
  public
    destructor Destroy; override;
    ...
  end;

destructor TShape.Destroy;
begin
  FBrush.Free;
  FPen.Free;
  inherited Destroy;
end;
```

Código Fuente 10 - El método destructor

La palabra clave "override" será tratada en el próximo artículo.

Generalmente, la última acción del método destructor es llamar al método destructor de la clase ancestra.

Cuando ocurre un error durante la ejecución del método constructor, el método destructor es llamado automáticamente para liberar los recursos asignados al objeto que no pudo ser creado. Esto significa que el método destructor debe ser implementado para ser capaz de liberar parcialmente los recursos de un objeto.

El método Free

La clase **TObject** introduce e implementa el método Free el cual verifica que cada campo haya sido inicializado antes de liberar sus recursos ofreciendo una forma más segura para destruir un objeto. El método Free automáticamente llama al método Destroy si la referencia al objeto no es nil. De esta manera, si el objeto no fue inicializado, Free no genera un error.

Conclusión

Hemos visto algunos aspectos básicos de la programación orientada a objetos.

Sabemos lo que es una clase y un objeto. Sabemos cuales son los especificadores de visibilidad de los miembros de una clase y que dichos miembros pueden ser campos, propiedades, métodos y/o eventos. Y lo mas importante: sabemos como declarar una clase en object pascal.

También sabemos que en object pascal todas las clases deben tener un método constructor y un método destructor responsables de crear y destruir instancias de una clase o, lo que es lo mismo, objetos del tipo de la clase.

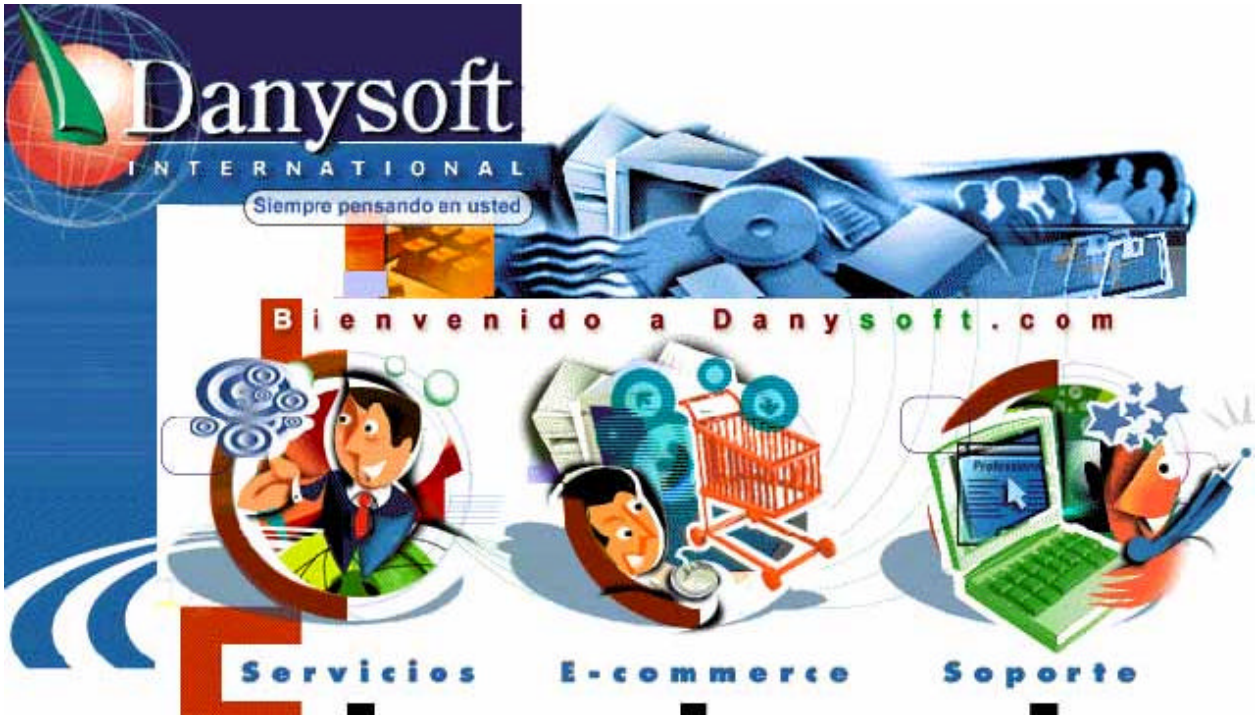
Lo que viene

En la segunda parte de este artículo veremos aspectos detallados de las propiedades y los métodos de una clase para saber de que manera object pascal implementa los conceptos fundamentales de la programación orientada a objetos, como herencia y polimorfismo.

Guías técnicas Grupo Danysoft:

Creación de Componentes en Delphi

(parte II)



Por Pablo Reyes – Grupo Danysoft
mayo de 2001 - (902) 123146
www.danysoft.com

Segunda parte

Este artículo es el segundo de una serie de tres con los que pretendo explicar los fundamentos teóricos mínimos necesarios para la creación de componentes con Delphi.

En esta segunda parte veremos más en profundidad las características principales de los métodos y las propiedades de una clase en object pascal, el lenguaje de programción de Delphi.

Sobre métodos y propiedades

Sobre métodos

Métodos estáticos (static)

Por defecto los métodos de una clase son estáticos. Los métodos estáticos son similares a procedimientos y funciones regulares. El compilador determina la dirección exacta del método y lo enlaza en tiempo de compilación.

```
type
  TClaseA = class
    procedure ProcedimientoA; static;
    procedure ProcedimientoB;
  end;

  TClaseB = class(TClaseA)
    procedure ProcedimientoA; static;
    procedure ProcedimientoB;
  end;
```

Código Fuente 1 - Métodos estáticos

La palabra clave “static” no es necesaria. En el ejemplo anterior ambos procedimientos son estáticos.

La ventaja principal que presentan los métodos estáticos es que su ejecución es sumamente rápida ya que, debido a que el compilador puede determinar la dirección de memoria exacta, el método se encuentra directamente enlazado a la clase.

Los métodos estáticos no cambian cuando son heredados por clases descendientes. Todas las clases descendientes ejecutan el mismo método en la misma dirección de memoria. Si en una clase descendiente el método es declarado nuevamente entonces la nueva declaración reemplaza al método ancestro.

```
var
  Objeto: TClaseA;
begin
  Objeto := TClaseA.Create;
  Objeto.ProcedimientoA;  // ProcedimientoA de TClaseA
  Objeto.Free;
  Objeto := TClaseB.Create;
  Objeto.ProcedimientoA;  // ProcedimientoB de TClaseA
```

```
  Objeto.Free;
end;
```

Código Fuente 2 - Métodos estáticos

Debido a que la variable Objeto es del tipo TClaseA no importa el tipo del objeto al cual haga referencia para saber que implementación del método ProcedimientoA será invocada ya que el método se encuentra enlazado a la clase y no a los objetos del tipo de la clase. A pesar de que la variable Objeto referencia primero a un objeto del tipo TClaseA y luego a un objeto del tipo TClaseB, en ambos casos será invocada la implementación de la clase TClaseA del método ProcedimientoA porque la variable es del tipo TClaseA y los métodos son estáticos.

Un método estático hace siempre exactamente lo mismo sin importar la clase desde la cual sea ejecutado.

Métodos dinámicos y virtuales (dynamic and virtual)

Los métodos dinámicos y virtuales son la mejor forma de implementar el concepto de polimorfismo (junto con los métodos sobre-escritos). Ambos tipos de métodos son semánticamente idénticos. La diferencia está dada por la manera con que Delphi implementa su invocación en tiempo de ejecución.

```
Type
  TClaseA = class
    procedure ProcedimientoA; virtual;
    procedure ProcedimientoB; dynamic;
  end;
```

Código Fuente 3 - Procedimientos dinámicos y virtuales

Los métodos dinámicos y virtuales no son enlazados directamente a la clase ya que el compilador no conoce su dirección de memoria. En su lugar, Delphi utiliza en mecanismo de referencia indirecta enlazando a la clase un tabla con la dirección de memoria de cada método dinámico y virtual.

Para los métodos dinámicos, Delphi crea una tabla de métodos dinámicos por cada clase y coloca en ella una referencia a los métodos dinámicos de la clase. En una clase descendiente los métodos dinámicos de la clase ancestra no son incluidos en la tabla de métodos dinámicos de la clase salvo que el método sea sobre-escrito. En tiempo de ejecución Delphi debe buscar en la tabla de métodos dinámicos de la clase y de sus clases ancestras hasta encontrar la referencia al método invocado o hasta llegar a la clase TObject.

Para los métodos virtuales, Delphi crea un tabla de métodos virtuales por cada clase y coloca en ella una referencia a los métodos virtuales de la clase y de sus clases ancestras. En tiempo de ejecución Delphi debe buscar solamente en la tabla de métodos virtuales de la clase.

Una llamada a un método virtual es en promedio dos veces más rápida que una llamada a un método dinámico. Esto se debe a que el acceso a la tabla de métodos virtuales es directo por medio de un índice mientras que el acceso a la tabla de métodos dinámicos es secuencial. Además, la tabla de métodos virtuales de una clase posee una referencia a todos los métodos de la clase y de sus clases ancestras mientras que la tabla de métodos dinámicos sólo posee una referencia a los métodos dinámicos de la clase y a los métodos dinámicos de sus clases ancestras

que hayan sido sobre-escritos en la clase. Por otra parte, esta última característica hace que una clase con métodos virtuales ocupe más memoria que una clase con métodos dinámicos.

Si examinamos el código fuente de la VCL veremos que hay mayoría de métodos dinámicos frente a métodos virtuales. Esto se debe que las clases de la VCL son muy utilizadas (heredadas) y si hubiera mayoría de métodos virtuales las aplicaciones Delphi ocuparían mucha más memoria. Sin embargo esto no significa que debamos utilizar siempre métodos dinámicos. Como desarrolladores de aplicaciones puestos a desarrollar componentes es aconsejable que utilicemos métodos virtuales.

Métodos sobre-escritos (override)

Los métodos virtuales y dinámicos de una clase puede ser sobre-escritos en clases descendientes. Los métodos sobre-escritos deben ser implementados nuevamente en la clase que los sobre-escribe lo que permite modificar su comportamiento a través de una misma jerarquía de clases.

```
Type
  TClaseA = class
    procedure ProcedimientoA; virtual;
  end;

  TClaseB = class(TClaseA)
    procedure ProcedimientoA; override;
  end;
```

Código Fuente 4 - Métodos sobre-escritos

La clase TClaseA posee el método ProcedimientoA. La clase TClaseB desciende de la clase TClaseA y sobre-escribe el método ProcedimientoA. El método ProcedimientoA no está enlazado a ninguna de las dos clases. Los objetos del tipo TClaseA están enlazados a la implementación del método ProcedimientoA de la clase TClaseA y los objetos del tipo TClaseB están enlazados a la implementación del método ProcedimientoA de la clase TClaseB.

```
var
  Objeto: TClaseA;
begin
  Objeto := TClaseA.Create;
  Objeto.ProcedimientoA;  // ProcedimientoA de TClaseA
  Objeto.Free;
  Objeto := TClaseB.Create;
  Objeto.ProcedimientoA;  // ProcedimientoB de TClaseB
  Objeto.Free;
end;
```

Código Fuente 5 - Polimorfismo

A pesar de que la variable Objeto es del tipo TClaseA cuando el objeto que referencia es del tipo TClaseB Delphi llama a la implementación del método ProcedimientoA de la clase del objeto (TClaseB) y no de la clase de la variable que referencia al objeto (TClaseA). Asi es como Delphi implementa el concepto de polimorfismo.

Métodos abstractos (abstract)

Los métodos abstractos son métodos dinámicos o virtuales cuya implementación es delegada a clases descendientes.

```
Type
  TClaseA = class
    procedure ProcedimientoA; dynamic; abstract;
    procedure ProcedimientoB; virtual; abstract;
  end;
```

Código Fuente 6 - Métodos abstractos

La palabra clave “abstract” debe escribirse después de las palabras clave “dynamic” o “virtual”.

Los métodos abstractos permiten crear el esqueleto de una clase sin la necesidad de implementar cada uno de sus métodos. Las clases que poseen métodos abstractos son comunmente llamadas clases abstractas. El compilador muestra un mensaje de advertencia se creamos una instancia de una clase con métodos abstractos. Si un método abstracto es invocado en tiempo de ejecución Delphi muestra un mensaje de error.

Muchas clases de la VCL son clases abstractas, como por ejemplo la clase TDataSet.

Métodos sobre-cargados (overload)

Los métodos sobre-cargados son métodos que poseen el mismo nombre pero distintos parámetros haciendo que ambos métodos sean incompatibles. Dos métodos son compatibles cuando poseen la misma cantidad de parámetros, del mismo tipo de dato y en el mismo orden.

```
Type
  TClaseA = class
    procedure ProcedimientoA(A: string); overload;
    procedure ProcedimientoB(A: integer; B: string); overload;
  end;
```

Código Fuente 7 - Métodos sobre-cargados

Como los métodos sobre-cargados tienen el mismo nombre, en tiempo de ejecución Delphi examina los parámetros para saber cual método debe ser invocado. Los métodos sobre-escritos pueden pertenecer a una misma clase o distintas clases en una misma jerarquía.

Los métodos utilizados para leer o escribir el valor de una propiedad no pueden ser sobre-cargados.

Métodos re-introducidos (reintroduce)

Si en una clase se delcara un método virtual de un clase ancestra y no se lo declara como sobre-escrito entonces el compilador muestra un mensaje de advertencia indicando que se está ocultando el método virtual de la clase ancestra.

Para evitar el mensaje de advertencia se debe indicar que el método es un método re-introducido colocando la palabra clave “reintroduce” al final de la declaración del método.

```
Type
  TClaseA = class
    procedure ProcedimientoA; virtual;
  end;

  TClaseB = class(TClaseA)
    procedure ProcedimientoA; reintroduce;
  end;
```

Código Fuente 8 - Métodos re-introducidos

El método ProcedimientoA existe en ambas clases aunque en la clase TClaseB es estático y sus clases descendientes no pueden sobre-escribirlo.

```
procedure TClaseB.ProcedimientoA;
begin
  inherited ProcedimientoA;  // ejecuta ProcedimientoA de TClaseA
end;
```

Código Fuente 9 - Métodos re-introducidos

Sin embargo en la clase TClaseB el método ProcedimientoA de la clase TClaseA sigue existiendo y puede ser invocado utilzando la palabra clave “inherited”.

La palabra clave Inherited

La palabra clave “inherited” permite invocar un método de las clases ancestras. Si inherited es seguido por un nombre de método entonces dicho método es invocado de la manera indicada (con los parámetros especificados). Si inherited no es seguido por un nombre de método entonces es invocado un método con el mismo nombre y de la misma manera (los mismos parámetros) que el método dentro del cual inherited fue invocado.

Sobre propiedades

Propiedades que referencian un array

Cuando un campo de una clase es un array la propiedad para acceder al valor de dicho campo desde fuera de la clase tiene una sintaxis especial.

```
Type
  TClaseA = class
  private
    FObjetos: array[1..10] of TObject;
    function GetObjetos(Indice: integer): TObject;
    procedure SetObjetos(Indice: integer; Objeto: TObject);
  public
    property Objetos[Indice: integer]: TObject
        read GetObjetos write SetObjetos;
  end;
```

Código Fuente 10 - Propiedades que referencian un array

En primer lugar, se debe indicar una lista de índices (por lo general es uno pero pueden ser más y de distintos tipos de dato) que permitan ubicar un elemento del array. El tipo de dato de la propiedad debe ser igual al tipo de dato de los elementos del array.

Los identificadores de acceso de la propiedad deben ser métodos. El identificador de acceso de lectura debe ser una función que reciba tantos parámetros como índices tenga la propiedad, en el mismo orden y del mismo tipo de dato y debe devolver un valor del mismo tipo de la propiedad. El identificador de acceso de escritura debe ser un procedimiento que reciba tantos parámetros como índices tenga la propiedad, en el mismo orden y del mismo tipo de dato y un parámetro adicional del tipo de dato de la propiedad.

La implementación de los métodos de la clase puede ser como sigue:

```
function TClaseA.GetObjetos(Indice: integer): TObject;
begin
  Result := nil;
  if (Indice >= 1) and (Indice <= 10) then
    Result := FObjetos[Indice];
end;

procedure TClaseC.SetObjetos(Indice: integer; Objeto: TObject);
begin
  if (Indice >= 1) and (Indice <= 10) then
    FObjetos[Indice] := Objeto;
end;
```

Código Fuente 11 - Implementación de una clase con propiedades que referncian un array

Una propiedad que referencia a un campo que es un array puede ser declarada como propiedad por defecto de la clase lo que permite acceder a la propiedad con solo escribir el nombre de la variable que referencia a un objeto del tipo de la clase sin tener que escribir el nombre de la propiedad.

```
    property Objetos[Indice: integer]: TObject
        read GetObjetos write SetObjetos; Default;
```

Código Fuente 12 - Propiedades que referencian un array y la palabra clave "Default"

El código anterior indica que la propiedad Objetos es la propiedad por defecto de la clase TClaseA. De esta forma es posible acceder a la propiedad sin tener que escribir el nombre de la propiedad.

```
var
  Objeto: TClaseA;
begin
  Objeto := TClaseA.Create;
  Objeto[1] := Objeto;  // es igual que la siguiente línea
  Objeto.Objetos[2] := Objeto;  // es igual que la línea anterior
  Objeto.Free;
end;
```

Código Fuente 13 - Acceder a la propiedad por defecto

En el código anterior se muestra como es posible acceder a la propiedad por defecto con sólo escribir el nombre del objeto que referencia a la clase que posee una propiedad que referencia a un array que fue declarada como propiedad por defecto.

Especificadores de índices

Los especificadores de índices permiten que varias propiedades compartan los mismos identificadores de acceso aún cuando dichas propiedades representen valores distintos. Un especificador de índice es una constante de tipo integer. Si una propiedad tiene un especificador de índice entonces los identificadores de acceso deben ser métodos.

```
Type
  TNombre = record
    Nombre: string;
    Apellido: string;
  end;

  TPersona = class
  private
    FNombre: TNombre;
    function GetNombre(const Index: Integer): string;
    procedure SetNombre(const Index: Integer; const Value: string);
  public
    property Nombre: string index 0 read GetNombre write SetNombre;
    property Apellido: string index 1 read GetNombre write SetNombre;
    property ApellidoNombre: string index 2 read GetNombre;
  end;
```

Código Fuente 14 - Especificadores de índice

Los identificadores de acceso de una propriedad que tiene un especificador de índice deben recibir un parámetro adicional de tipo integer que representa el índice de la propiedad. Cuando una aplicación accede al valor de la propiedad el parámetro adicional es pasado automáticamente.

La implementación de los métodos de la clase anterior puede ser como sigue.

```
function TPersona.GetNombre(const Index: Integer): string;
begin
  case Index of
    0: Result := FNombre.Nombre;
    1: Result := FNombre.Apellido;
    2: Result := FNombre.Apellido + ', ' + FNombre.Nombre;
  end;
end;

procedure TPersona.SetNombre(const Index: Integer; const Value: string);
begin
  case Index of
    0: FNombre.Nombre := Value;
    1: FNombre.Apellido := Value;
  end;
end;
```

Código Fuente 15 - Implementación de una clase con propiedades con especificadores de índice

Mediante el uso de especificadores de índice el código de una clase es más corto y simple.

Especificadores de almacenamiento

Las propiedades pueden tener asociados especificadores de almacenamiento para indicar si su valor debe ser almacenado o no. Estos especificadores no modifican el comportamiento de la clase ya que sólo son utilizados para almacenar y cargar el valor de la propiedad en el archivo .DFM de un formulario.

La palabra clave "Stored" permite indicar si el valor de una propiedad debe ser almacenado. La palabra clave "Stored" debe ser seguida por una expresión lógica que puede ser la palabra clave "False", la palabra clave "True", el nombre de un campo cuyo tipo de dato sea boolean o el nombre de una función sin parámetros que devuelva un valor de tipo boolean. Si la palabra clave "Stored" no es indicada el compilador asume "Stored True".

La palabra clave "Default" permite indicar cual es el valor por defecto de una propiedad para evitar que el mismo sea almacenado. La palabra clave "Default" debe ser seguida por una constante del mismo tipo que la propiedad. El valor por defecto debe ser asignado durante la creación de una instancia de la clase (en el método constructor, por ejemplo).

El valor de una propiedad es almacenado si:

- La expresión que sigue a la palabra clave "Stored" es evaluada como verdadera (True).
- La palabra clave "Default" no está presente o la constante que la sigue es distinta al valor de la propiedad.

En cualquier otro caso el valor de la propiedad no es almacenado.

Propiedades sobre-escritas

La declaración de una propiedad que no incluye un tipo de dato es una propiedad sobre-escrita. Mediante la sobre-escritura de propiedades se puede lograr varias cosas.

Cambiar la visibilidad

Lo más simple es aumentar la visibilidad de una propiedad. Si una propiedad es declarada con visibilidad protected, en una clase descendiente puede ser sobre-escrita con visibilidad public con solo escribir el nombre de la propiedad con la nueva visibilidad.

```
Type
  TClaseA = class
  private
    FNombre: string;
  protected
    property Nombre: string read FNombre write FNombre;
  end;

  TClaseB = class(TClaseA)
  public
    property Nombre;
  end;
```

Código Fuente 16 - Propiedades sobre-escritas. Cambiar la visibilidad.

En el código anterior la propiedad Nombre tiene visibilidad protected en la clase TClaseA y visibilidad public en la clase TClaseB.

Modificar especificadores

Cuando se sobre-escribe una propiedad sus especificadores pueden ser sobre-escritos también o pueden agregarse especificadores faltantes.

```
Type
  TClaseA = class
  private
    FNombre: string;
  public
    property Nombre: string read FNombre;
  end;

  TClaseB = class(TClaseA)
  public
    property Nombre write FNombre;
  end;
```

Código Fuente 17 - Propiedades sobre-escritas. Agregar especificadores.

En el código anterior la propiedad Nombre es de sólo lectura en la clase TClaseA. En la clase TClaseB, que desciende de la clase TClaseA, la propiedad Nombre es sobre-escrita agregándosele el especificador de escritura y conviertiendola en una propiedad de lectura y escritua.

Conclusión

Hemos visto la mayoría de las características de los métodos y las propiedades de una clase.

También vimos como implementa object pascal conceptos del paradigma de la programación orientada a objetos, como herencia y polimorfismo. Estos conceptos son muy utilizados a la hora de construir las clases que forman un componente. Seguramente muchas de las palabras claves que vimos resultan familiares para aquellos que han curioseado por el código fuente de la VCL.

Lo visto hasta aquí tiene que ver con objetct pascal solamente que no necesariamente tienen una relación directa con la creación de componentes. Sin embargo, a la hora de escribir componentes, los conceptos vistos en la primera y la segunda parte deben estar bien claros en la mente del programador.

Lo que viene

En la tercera parte nos meteremos de lleno en la creación de componentes. Veremos las facilidades que nos ofrece Delphi (lease asistentes) para la creación de componentes ya sea para crear un componente desde cero, extender un componente existente, proveer una imagen para un componente, registrar un componente y demás.