ساخت کامپوننتهای اختصاصی در دلفی

توجه: این مقاله ترجمه‌ای است که از روی مجموعه مقالات مربوط به ساخت کامپوننت در سایت www.delphi.about.com انجام داده‌ام. دوستان اگر نظری دارند می‌توانند به من mail بزنند: mdehghanian@gmail.com

با تشکر: مسعود دهقانیان.

۱- چرا، چه وقت و چگونه باید یک کامپوننت جدید بسازید

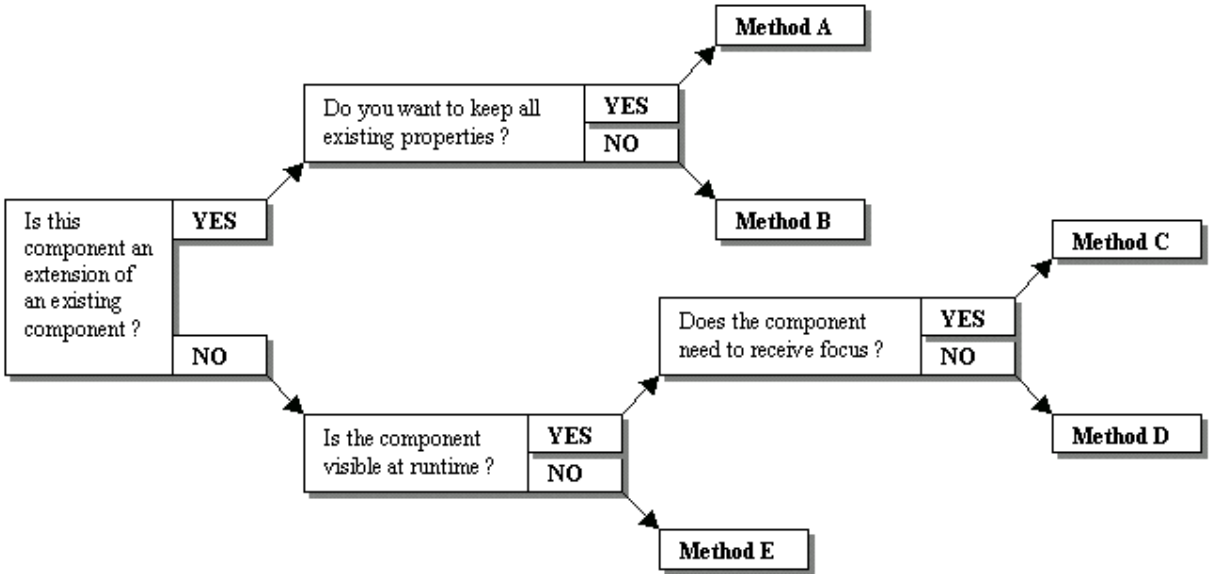
دو چیزی که در ابتدا باید از خودتان بپرسید عبارتند از اینکه چرا باید کامپوننت جدید بنویسید و چه وقت این کار لازم خواهد بود. جواب دادن به پرسش اول ساده است و در واقع جوابهای زیادی دارد.

- **سادگی استفاده**: کد کپسوله شده کامپوننت این مفهوم را با خود دارد که میتوانید به سادگی یک کامپوننت را بدون اینکه حتی یک خط کد تکراری بنویسید بارها و بارها روی فرمهای مختلف بگذارید و استفاده کنید.
- **اشکالزدایی**: با توجه به اینکه کد در یک کامپوننت متمرکز شده، با رفع خطا در یک کامپوننت واحد و کامپایل مجدد آن، برطرف کردن اشتباهات یک برنامه کامل (یا مجموعه‌ای از برنامه‌ها ساده میشود.
- **پول!**: شرکتهای زیادی هستند که فقط به خاطر اینکه "دوباره چرخ را اختراع نکنند" خوشحال میشوند که پولی پرداخت کنند (یعنی کامپوننتهایی که قبلاً ساخته شده‌اند را بخرند).

پاسخ دادن به پرسش دوم هم سخت نیست. هر وقت که فهمیدید مجبور هستید کد مشابهی را بیش از یکبار بنویسید، ایده خوبی است که یک کامپوننت بسازید، بویژه اگر کد بر اساس پارامترهای داده شده عملکردهای متفاوتی دارد.

کامپوننتها چطور ساخته میشوند

اولین گام این است که تصمیم بگیرید برای کامپوننتان از کدام کلاس پایه باید اشتقاق داشته باشید. وقتی از یک کلاس دیگر اشتقاق داشته باشید خصیصه، متد و رخدادهایی را که آن کامپوننت دارد به ارث میبرید. شکل زیر نمونه-ای است برای چگونگی تصمیم گیری در مورد اینکه وقتی کامپوننت خودتان را مینویسید از چه کلاسی باید ارث‌بری داشته باشید.



در مورد روشهای A و B، مثلاً با فرض اینکه کامپوننت مورد نظر TMemo باشد به جدول زیر میرسیم:

روش	راه حل
A	اشتقاق از TMemo
B	اشتقاق از TCustomMemo
C	اشتقاق از TCustomControl
D	اشتقاق از TGraphicControl
E	اشتقاق از TComponent

ممکن است کمی پیچیده به نظر برسد، پس اجازه دهید که کمی این روند را توضیح دهیم.

A: وقتی که میخواهید عملکرد اضافی به یک کامپوننت موجود اضافه کنید از آن کامپوننت اشتقاق انجام میدهید. این کار، به طور خودکار به کامپوننت جدید شما همه عملکردها و خصیصه‌های کامپوننت موجود را میدهد.

B: گاهی نه تنها میخواهید عملکردی را اضافه کنید، بلکه میخواهید همزمان عملکردهایی را هم حذف کنید. حالت استاندارد در نوشتن کامپوننت این است که در ابتدا یک کامپوننت TCustomXXXX بنویسید که در آن همه خصیصه‌ها، متدها و رخدادها در بخش protected کامپوننت اعلان شده‌اند. سپس کامپوننت اصلی از این کلاس پایه مشتق میشود. بنابراین لِم کار مخفی کردن عملکرد نیست، بلکه اشتقاق از نسخه "اختصاصی" کامپوننت و Publish اعلان کردن خصیصه‌های مورد نظر است (که در عمل خصیصه‌ها و رخدادهای ناخواسته را حذف میکند).

C: هر کامپوننتی که نیاز به گرفتن فوکوس دارد به یک هندل ویندوز احتیاج دارد. TWinControl نخستین جایی است که این هندل ارائه میشود. TCustomControl هم یک TWinControl است که خصیصه Canvas خودش را دارد.

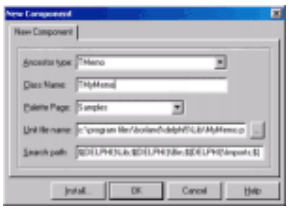
D: TGraphicControl هندل ویندوز ندارد و لذا نمیتواند فوکوس دریافت کند. کامپوننتهایی مثل TLabel از این کلاس پایه مشتق شده‌اند.

E: بعضی کامپوننتها برای توسعه قابلیت تعاملی GUI ساخته نشده‌اند بلکه برای ساده‌تر کردن کار برنامه‌نویسی ساخته شده-اند. هر چیزی که از TComponent مشتق شده باشد فقط در زمان طراحی قابل مشاهده است. کاربردهای متداول این نوع کامپوننتها برای اتصالات پایگاه داده، تایمرها و غیره است.

۲- ساخت اولین کامپوننت، کپسوله‌سازی

ایجاد اولین کامپوننت

گام بعدی، بعد از اینکه تصمیم گرفتید کلاس پایه چه کلاسی باید باشد، ایجاد کامپوننت است. از منوی کامپوننت، New Component را انتخاب کنید. فرم زیر را مشاهده خواهید کرد.



جدول زیر در مورد این فرم توضیح میدهد:

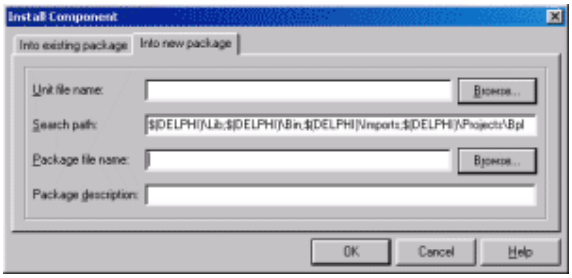
Ancestor type	این کلاس پایه‌ای است که میخواهیم از آن ارث‌بری داشته باشیم
Class name	این نام کلاس کامپوننت جدید ماست
Palette page	مشخص کننده قسمتی در صفحه کامپوننتهاست که میخواهیم کامپوننت ما آنجا ظاهر شود، وارد کردن نام پَلِتی که از قبل وجود ندارد به دلفی میگوید که یک بخش جدید در صفحه کامپوننتها ایجاد کند.

کد نویسی

حالا موقع آن است که مقداری کد بنویسیم. اولین مثال ما هیچ کاربرد عملی غیر از روشن کردن برخی مفاهیم پایه نوشتن کامپوننت نخواهد داشت.

در ابتدا از منوی اصلی دلفی Component و بعد New Component را انتخاب کنید. Tcomponent را بعنوان "نوع منشا"[1] و TfirstComponent را بعنوان نام کامپوننت وارد کنید. سپس روی دکمه Install کلیک کنید.

در این موقع شما میتوانید کامپوننت جدیدتان را در یک بسته[2] موجود (بسته‌ای که مجموعه‌ای از کامپوننتها را نگه میدارد) یا در یک بسته جدید نصب کنید. روی into new package و دکمه Browse کلیک کنید.



وقتی مسیر و نام فایل را برای بسته جدید انتخاب و توضیحی را برای آن وارد کردید دکمه OK را فشار دهید. در فرم بعدی (که سوال میکند آیا میخواهید بسته‌تان را دوباره کامپایل کنید) دکمه Yes را فشار بدهید. وقتی بسته کامپایل و نصب شد، بسته و Unit را ذخیره کنید.

به این ترتیب تا کنون ما کلاس پایه و نیز نام کلاس جدید را مشخص کرده‌ایم. برای نگهداری کامپوننت جدیدمان یک بسته جدید ساختیم و با اسکلت بندی ساختار کلاسها در دلفی آشنا شدیم.

اگر به کد ایجاد شده بوسیله دلفی نگاه کنید، قسمتهای Public،Private، Protected و Published را میبینید.

[1] Ancestor type
[2] Package

کپسوله‌سازی

کپسوله‌سازی مبحثی است که فهمیدنش ساده است، ولی در مورد نوشتن کامپوننتها دارای اهمیت زیادی است. کپسوله‌سازی با استفاده از چهار واژه پیاده‌سازی میشود: Private،Public، Protected و Published. خواهید دید که دلفی این بخشها را بطور خودکار به کد کامپوننت جدید شما اضافه کرده است.

بخش	قابلیت دسترسی
Private	متدها، خصیصه‌ها و رخدادهای اعلان شده در این بخش فقط در Unitی که کامپوننت در آن هست قابل دسترسی هستند. کامپوننتهای موجود در یک Unit میتوانند به آیتمهای Private همدیگر دسترسی داشته باشند.
Protected	متدها، خصیصه‌ها و رخدادهای اعلان شده در این بخش بوسیله کلاسهایی که از این کلاس ارث‌بری داشته باشند قابل دسترسی است.
Public	متدها، خصیصه‌ها و رخدادهای اعلان شده در این بخش از هر جایی قابل دسترسی هستند.
Published	این بخش به شما اجازه میدهد خصیصه‌ها و رخدادهایی اعلان کنید که در بازرس شئ[3] ظاهر میشوند. این خصیصه‌ها و رخدادها مقادیر زمان طراحی هستند که بهمراه پروژه شما ذخیره میشوند.

۳– کد اولیه کامپوننت، کلمات کلیدی Virtual، Dynamic، Abstract و Override

آغاز نوشتن کد کامپوننت

حالا دلفی باید همه چیز را در مورد کامپوننت ما بداند. کد زیر را در کد کامپوننت بنویسید:

```
  private
    { Private declarations }
     FStartTime،
    FStopTime   :DWord;
  protected
    { Protected declarations }
    function  GetElapsedTime :
        String; virtual;
  public
    { Public declarations }
    procedure Start; virtual;
    procedure Stop; virtual;

    property StartTime:DWord
      read FStartTime;
    property StopTime:DWord
      read FStopTime;
    property ElapsedTime:String
      read GetElapsedTime;
  published
    { Published declarations }
end;
```

[3] Object Inspector

کاری که انجام داده‌ایم به این صورت است که دو متغیر FstartTime و FstopTime را اضافه کردیم. (این یک استاندارد است که نام متغیرها را با F شروع کنیم ولی البته الزامی نیست.F اول کلمه Field است). دو متد برای کنترل این متغیرها وجود دارد، Start و Stop. یک تابع GetElapsedTime اضافه کرده‌ایم که FStopTime و FStartTime را بصورت رشته‌ای برمیگرداند. در نهایت سه خصیصه فقط خواندنی اضافه نمودیم.
کلیدهای SHIFT-CTRL-C را فشار دهید تا دلفی بطور خودکار کد کلاس شما را کامل کند (یا اینکه راست کلیک کنید و "Complete class at cursor" را انتخاب کنید). سپس کد زیر را برای هر یک از متدهای مربوطه وارد کنید:

```
{ TFirstComponent }

function TFirstComponent.GetElapsedTime: String;
begin
  Result := IntToStr(FStopTime - FStartTime);
end;

procedure TFirstComponent.Start;
begin
  FStartTime := GetTickCount;
end;

procedure TFirstComponent.Stop;
begin
  FStopTime := GetTickCount;
end;

end.
```

تست

یونیت‌تان را ذخیره کنید، و بسته را مجدداً باز کنید (File|Open Project و برای نوع فایل "Delphi Package" را انتخاب کنید)، وقتی بسته باز شد دکمه "Compile" را کلیک کنید. همچنین میتوانید بسته‌تان را با انتخاب Component از منوی اصلی و آنگاه Install Packages هم باز کنید. بسته را انتخاب کنید و بعد دکمه "Edit" را کلیک کنید.
حالا میتوانید یک TFirstComponent را روی فرم بگذارید. در واقع هر چند دفعه که بخواهید میتوانید این کار را انجام دهید. دو دکمه(btnStart و btnStop) به فرم اضافه کنید و کد زیر را بنویسید و سپس برنامه تست را اجرا کنید.

```
procedure TForm1.btnStartClick(Sender: TObject);
begin
  FirstComponent1.Start;
end;

procedure TForm1.btnStopClick(Sender: TObject);
begin
  FirstComponent1.Stop;
  Caption := FirstComponent1.ElapsedTime;
end;
```

کلیک کردن روی دکمه "Start" باعث نشانگذاری زمان شروع میشود (GetTickCount یک دستور WinAPI است که تعداد میلی ثانیه‌های سپری شده از زمان شروع بکار ویندوز را برمیگرداند).
کلیک کردن روی دکمه "Stop" زمان توقف را نشانگذاری میکند، و عنوان فرم را تغییر میدهد.

Virtual, Dynamic, Abstract و Override

ممکن است به اعلان Virtual بعد از Start, Stop و GetElapsedTime توجه کرده باشید. عملیاتی که در زیر می‌آید کاربرد آنها را توضیح میدهد.
یک کامپوننت جدید ایجاد کنید، آن را از TFirstComponent مشتق کنید (نام آنرا TSecondComponent بگذارید) و آنرا نصب کنید.
شناسه‌های Virtual و Dynamic روشی برای نویسنده کامپوننت هستند که با آن به دلفی میگوید که ممکن است در یک کلاس فرزند، این متد با متد دیگری جایگزین شود. اگر متدی را در کلاس Override کنیم، بجای کد اولیه، کد جدید ما اجرا خواهد شد.

```
protected
  { Protected declarations }
  function  GetElapsedTime : String; override;
```

بعد ما کد بالا را بصورت زیر پیاده‌سازی میکنیم:

```
function TSecondComponent.GetElapsedTime: String;
var
  S : String;
begin
  S := inherited GetElapsedTime;
  Result := S + ' milliseconds or ' +
    Format('%.2f seconds',
      [StopTime - StartTime) / 1000]);
end;
```

حالا بجای کد GetElapsedTime اولیه، کد جدید فراخوانده میشود. کد متد اولیه با استفاده از دستور Inherited قابل فراخوانی است.

توجه: اگر یک متد پایه را override نکنید (چون متد پایه بصورت Virtual اعلان نشده بود یا فراموش کرده بودید که متد را override کنید)، TSecondComponent کد جدید را فراخوانی خواهد کرد، در حالیکه کد ارائه شده در TFirstComponent همچنان به فراخوانی کد اصلی از TFirstComponent ادامه خواهد داد.

شناسه Abstract به دلفی میگوید که برای متد مورد نظر، انتظار کد نداشته باشد. نباید از کلاسی که متدهای انتزاعی[4] دارد (مثل Tstring) نمونه شئیی ایجاد کنید. روند استاندارد این است که از چنین کلاسی یک کلاس فرزند ایجاد کنید و همه متدهای انتزاعی را override کنید. (مثل کاری که کلاس TStringList میکند).
مقایسه Dynamic و Virtual پرسشی است درباره مقایسه سرعت و اندازه. یک متد Dynamic منجر به نیاز کمتری به حافظه در هر یک از نمونه‌های کلاس خواهد شد، در حالیکه یک متد Virtual منجر به اجرای سریعتر کد ولی نیاز به حافظه بیشتر خواهد شد.

[4] Abstract

۴ – اضافه کردن رخداد[5]ها

مراحل کمی برای اضافه کردن رخدادها به کامپوننت وجود دارد. رخدادها به کامپوننت امکان میدهند با برنامه ارتباط برقرار کند تا وقتی که چیز مهمی اتفاق افتاده، برنامه را از آن آگاه کنند. رخداد، یک خصیصه خواندنی / نوشتنی است، فقط به جای اینکه نوع آن یک نوع متغیر ساده (مثل String, Integer, ...) باشد یک روال یا تابع است.

یک کامپوننت جدید ایجاد کنید، آنرا از TSecondComponent مشتق کنید و نام آنرا TThirdComponent بگذارید. یونیت را ذخیره کنید، کامپوننت را نصب کنید، و کد زیر را اضافه نمایید:

```
type
  TState = (stStarted, stStopped);
  TStateChangeEvent = procedure
  (Sender : TObject; State : TState) of object;

  TThirdComponent = class(TSecondComponent)
  private
  { Private declarations }
    FState  : TState;
    FOnStart,
    FOnStop  : TNotifyEvent;
    FOnStateChange  : TStateChangeEvent;
  protected
    { Protected declarations }
  public
    { Public declarations }
    constructor Create(AOwner : TComponent); override;
    destructor Destroy; override;
    procedure Start; override;
    procedure Stop; override;
    property State : TState
      read FState;
  published
    { Published declarations }
    property OnStart : TNotifyEvent
      read FOnStart
      write FOnStart;
    property OnStateChange : TStateChangeEvent
      read FOnStateChange
      write FOnStateChange;
    property OnStop : TNotifyEvent
      read FOnStop
      write FOnStop;
  end;
```

رخدادها، روالها یا (ندرتاً) توابعی هستند که به یک کلاس وابسته هستند (به همین خاطر عبارت "of object" را در TStateChangeEvent میبینید). مثلاً، TNotifyEvent یک نوع استاندارد پیاده‌سازی شده بوسیله دلفی است که شیئی را که یک رخداد را فعال کرده ارسال میکند. خوب است که همیشه (Sender : TObject) "Self" را بعنوان اولین پارامتر هر رخداد ارسال کنیم چون ممکن است کد رخداد مشابهی بوسیله چند کامپوننت مورد استفاده قرار بگیرد. TNotifyEvent بصورت زیر تعریف شده:

[5] Event

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

فراخواندن یک رخداد داخل کامپوننت به این صورت است که ابتدا چک میکنیم آیا به رخداد چیزی نسبت داده شده و اگر اینطور بود آنرا فراخوانی میکنیم. من متدهای Start و Stop مربوط به TSecondComponent را override کرده‌ام تا این رخدادها را فعال کنیم، بصورت زیر:

```
procedure TThirdComponent.Start;
begin
  inherited; //This calls TSecondComponent.Start
  FState := stStarted;
  if Assigned(OnStart) then OnStart(Self);
  if Assigned(OnStateChange) then
    OnStateChange(Self, State);
end;

procedure TThirdComponent.Stop;
begin
  inherited;  //This calls TSecondComponent.Stop
  FState := stStopped;
  if Assigned(OnStop) then OnStop(Self);
  if Assigned(OnStateChange) then
    OnStateChange(Self, State);
end;

constructor TThirdComponent.Create(AOwner: TComponent);
begin
  inherited;
  //This is were you initialise properties, and create
  //and objects your component may use internally
  FState := stStopped;
end;

destructor TThirdComponent.Destroy;
begin
  //This is where you would destroy
  //any created objects
  inherited;
end;
```

دوباره بسته را کامپایل کنید (فراموش نکنید هر وقت کامپوننت جدیدی اضافه میکنید، بسته‌تان را ذخیره کنید). وقتی کامپوننت را روی فرم قرار دهید خواهید دید که سه رخداد وجود دارند. *OnStart, OnStop* و *OnStateChange*.

در کد زیر

OnStart عنوان را به "Started" تغییر میدهد.

OnStop زمان سپری شده را نشان میدهد.

OnStateChange دکمه‌های متناظر Start / Stop را فعال / غیرفعال میکند.

```
procedure TForm1.ThirdComponent1Start(Sender: TObject);
begin
  Caption := 'Start';
end;

procedure TForm1.ThirdComponent1Stop(Sender: TObject);
begin
  Caption := ThirdComponent1.ElapsedTime;
end;

procedure TForm1.ThirdComponent1StateChange
  (Sender: TObject; State: TState);
begin
  btnStart.Enabled :=
     ThirdComponent1.State = stStopped;
  btnStop.Enabled :=
     ThirdComponent1.State = stStarted;
end;
```

۵- استانداردهای نوشتن کامپوننت

استانداردهای نوشتن کامپوننت

در این بخش نکات اضافی در مورد نوشتن کامپوننتها، شامل برخی متدهای موجود کامپوننتهای پایه، و استانداردهای کدنویسی را بررسی میکنیم.

ایجاد و ازبین بردن[۶] کامپوننت

اشیاء با استفاده از یک سازنده[۷] ایجاد میشوند و با یک مخرب[۸] از بین میروند.

هدف از Override کردن یک سازنده این سه مورد است:

۱. برای اینکه فقط اشیایی که درون شی هستند (زیرشئ‌ها) ایجاد شوند
۲. برای مقداردهی اولیه مقادیر کلاس (خصیصه‌ها و غیره)
۳. برای ایجاد استثنا[۹] و توقف ایجاد کلاس

استاندارد این است که درون کد سازنده خودتان، سازنده به ارث رسیده (inherited) را فراخوانی کنید تا کلاس والد بتواند مقداردهی‌های اولیه خودش را انجام دهد، با وجود این چنین کاری برای ایجاد کامپوننت الزامی نیست(کامپوننت وقتی ایجاد میشود که کار متد سازنده شما به پایان رسیده باشد و با فراخوانی سازنده به ارث رسیده ایجاد نمیگردد).

هدف از override کردن یک مخرب، آزاد کردن هر گونه منبعی است که در مدت زمان وجود کامپوننت تخصیص داده شده است. فقط بعد از اینکه این منابع را آزاد کردید Inherited را فراخوانی کنید.

برخی از متدهای استاندارد کامپوننت

Paint: میتوانید این متد را Override کنید تا ترسیم اختصاصی خودتان را برای کامپوننت انجام دهید.

[6] Destroying
[7] Constructor
[8] Destructor
[9] Exception

Loaded: این متد در زمان اجرا بلافاصله بعد از آن فراخوانی میشود که خصیصه‌ها بعد از ایجاد فرم والد[10] مقدارگیری را به پایان برده‌اند. میتوانید این متد را override کنید تا عملیاتی را که به مقدار گرفتن گروهی از خصیصه‌ها وابسته هستند، انجام دهید.

Invalidate: هر وقت خصیصه‌ای تغییر میکند که بر نمای ظاهری یک کامپوننت تاثیر میگذارد، باید این متد را صدا بزنید.

ComponentState: این خصیصه وقتی میخواهید بفهمید که کامپوننت شما در زمان طراحی / اجرا وجود دارد یا نه، و اینکه آیا خصیصه‌های آن در حال خوانده شدن بوسیله یک فرایند جریانی[11] هستند یا نه بسیار مفید است.

کپسوله‌سازی صحیح کامپوننت

حالت استاندارد این است که کامپوننت را بصورت TCustomMyClass بنویسید و بعد کامپوننت را از آن کلاس پایه مشتق کنید. کامپوننت "custom" که مینویسید، بیشتر (اگر نه همه) خصیصه‌ها و متدهای اعلان شده را در بخش protected خودش خواهد داشت. وقتی از کلاس "custom" اشتقاق دارید به سادگی خصیصه‌ها را در بخشهای public یا published مجدداً اعلان میکنید.

```
type
  TCustomMyClass = class(TComponent)
  private
    FSomeString  : String;
  protected
    procedure SetSomeString(const Value : String);
      virtual;
    property SomeString  : String
      read FSomeString
      write SetSomeString;
  end;

  TMyClass = class(TCustomMyClass)
  published
    property SomeString;
  end;
```

این کار به بقیه امکان میدهد تا کامپوننتهای خودشان را در حالیکه میتوانند برخی خصیصه‌ها را حذف کنند بر اساس کامپوننت شما ایجاد کنند.

توجه کنید که چطور SetSomeString بصورت virtual در قسمت protected اعلان شده است. این روش خوبی است چون به کلاسهای فرزند اجازه میدهد تا با override کردن روالی که خصیصه‌ها را مقداردهی میکند به تغییرات مقدار خصیصه‌ها پاسخ بدهند. همچنین این روش برای رخدادها هم بکار میرود، جایی که یک رخداد OnStateChange را میبینید، معمولاً یک متد DoStateChange را هم پیدا میکنید، مثلاً:

```
type
  TCustomMyClass = class(TComponent)
  private
    FOnStateChange : TStateChangeEvent;
  protected
    procedure DoStateChange(State : TState); virtual;
```

[10] Parent
[11] Streaming process

```
  published
    property OnStateChange : TStateChangeEvent
      read FOnStateChange
      write FOnStateChange;
  end;

procedure TCustomMyClass.DoStateChange(State : TState);
begin
  if Assigned(OnStateChange) then
    OnStateChange(Self, State);
end;
```

بجای اینکه هر وقت وضعیت تغییر میکند کد "if assigned(OnStateChange) then" را بنویسید، خیلی راحت "DoStateChange(NewState)" را صدا میزنید. علاوه بر این، این کار به کلاسهای فرزند اجازه میدهد تا DoStateChange را Override کنند و در پاسخ به یک رخداد، کد متناسب را فعال کنند.

۶- ارجاعات در کامپوننتها

ارجاعات در کامپوننتها

بعضی کامپوننتها نیاز دارند که به کامپوننتهای دیگر ارجاع داشته باشند. مثلاً TLabel یک خصیصه "FocusControl" دارد. وقتی یک علامت آمپرساند (&) در خصیصه "Caption" وارد کنید، حرف بعد از آمپرساند زیرخطدار میشود(&Hello میشود Hello)، و فشار دادن کلیدهای ALT-H رخدادی را در Label شما فعال میکند. اگر خصیصه "FocusControl" مقداردهی شده باشد، فوکوسِ فرم برنامه به کنترلی که در این خصیصه مشخص شده منتقل میشود.

داشتن از این قبیل خصیصه‌ها در کامپوننت ساده است. همه کاری که میکنید این است که یک خصیصه جدید اعلان میکنید و نوع خصیصه را برابر پایین ترین کلاس پایه‌ای که میتواند بپذیرد قرار میدهید (مثلاً TWinControl امکان میدهد که هر فرزند TWinControl بتواند استفاده شود)، البته نکاتی هم وجود دارند.

```
type
  TSimpleExample = class(TComponent)
  private
    FFocusControl : TWinControl;
  protected
    procedure SetFocusControl(const value : TWinControl);
      virtual;
  public
  protected
    property FocusControl : TWinControl
      read FFocusControl
      write SetFocusControl;
  end;

procedure TSimpleExample.SetFocusControl(const Value : TWinControl);
begin
  FFocusControl := Value;
end;
```

به مثال بالا توجه کنید. اگر چنین خصیصه‌ای در کامپوننتتان داشته باشید، بازرس شئ[12] یک کامبوباکس با لیستی از کامپوننتها را نشان میدهد که با معیار کلاس پایه مشخص شده سازگار باشند (در مثال ما همه کامپوننتهای مشتق شده از TWinControl).
کامپوننت ما میتواند مثلاً چنین کاری بکند:

```
procedure TSimpleExample.DoSomething;
begin
  if (Assigned(FocusControl)) and
     (FocusControl.Enabled) then
    FocusControl.Setfocus;
end;
```

ابتدا چک میکنیم که خصیصه مقدار گرفته است یا نه، اگر اینطور بود فوکوس را در آن مقداردهی میکنیم. ولی حالتهایی وجود دارد که خصیصه مقدار پوچ (Nil) ندارد و در عین حال کامپوننتی که به آن اشاره میکند هم دیگر معتبر نیست. این حالت معمولاً وقتی اتفاق می‌افتد که خصیصه به کامپوننتی ارجاع دارد که از بین رفته است.
خوشبختانه دلفی راه حلی را در این مورد به ما ارائه میکند. هر وقت کامپوننتی نابود میشود مالک[13] خودش (فرمی که کامپوننت در آن هست) را از پاک شدنش مطلع میکند. در این لحظه هر کامپوننت دیگری که به فرم متعلق است هم از این رخداد مطلع میشود. برای بدام انداختن این رخداد باید یک متد استاندارد TComponent به نام "Notification" را Override کنیم.

```
Type
  TSimpleExample = class(TComponent)
  private
    FFocusControl : TWinControl;
  protected
    procedure SetFocusControl(const value : TWinControl);
      virtual;
  public
  protected
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
    property FocusControl : TWinControl
      read FFocusControl
      write SetFocusControl;
  end;

procedure TSimpleExample.SetFocusControl(const Value : TWinControl);
begin
  FFocusControl := Value;
end;

procedure TSimpleExample.Notification(AComponent : TComponent;
  Operation : TOperation);
begin
  If (Operation = opRemove) and
     (AComponent = FocusControl) then
    FFocusControl := Nil;
end;
```

[12] Object Inspector
[13] Owner

حالا وقتی که کامپوننت مورد ارجاع قرار گرفته پاک میشود ما مطلع میشویم و در آن لحظه میتوانیم مقدار ارجاعمان را برابر Nil قرار بدهیم. با وجود این، به اینکه گفتیم "*هر کامپوننتی که متعلق به همان فرم است هم از این رخداد مطلع میشود*" توجه کنید.

این مورد ما را با مشکل دیگری مواجه میکند. ما فقط وقتی که کامپوننت متعلق به همان فرم باشد مطلع میشویم. ممکن است خصیصه ما به کامپوننتی در فرمهای دیگر اشاره داشته باشد (یا اصلاً مالکی نداشته باشد)، و وقتی این کامپوننتها پاک شوند ما مطلع نخواهیم شد. نگران نباشید! باز هم راه حل وجود دارد.

TComponent متدی ارائه میدهد به نام "FreeNotification". هدف متد FreeNotification این است که به کامپوننت (FocusControl) بگوید که در هنگام پاک شدن، ما را به خاطر داشته باشد!

پیاده‌سازی به این صورت خواهد بود:

```
type
  TSimpleExample = class(TComponent)
  private
    FFocusControl : TWinControl;
  protected
    procedure SetFocusControl(const value : TWinControl);
      virtual;
  public
  protected
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
    property FocusControl : TWinControl
      read FFocusControl
      write SetFocusControl;
  end;

procedure TSimpleExample.SetFocusControl(const Value : TWinControl);
begin
  if Assigned(FFocusControl) then
    FFocusControl.RemoveFreeNotification(Self);

  FFocusControl := Value;

  if Assigned(FFocusControl) then
    FFocusControl.FreeNotification(Self);
end;

procedure TSimpleExample.Notification(AComponent : TComponent;
  Operation : TOperation);
begin
  if (Operation = opRemove) and
     (AComponent = FocusControl) then
    FFocusControl := Nil;
end;
```

وقتی خصیصه FocusControl را مقداردهی میکنیم ابتدا چک میکنیم آیا قبلاً به یک کامپوننت مقداردهی شده یا نه. اگر از قبل مقداردهی شده باشد لازم است به کامپوننت اصلی بگوییم که دیگر نیاز نداریم بدانیم کامپوننت چه وقت پاک میشود. وقتی خصیصه مقدار جدیدی میگیرد، کامپوننت جدید را آگاه میکنیم که وقتی آزاد میشود به اعلان احتیاج داریم. بقیه کد مشابه قبل باقی میماند یعنی کامپوننت مورد ارجاع همچنان متد Notification ما را فراخوانی میکند.

۷- مجموعه‌ها[۱۴]

مجموعه‌ها

مطمئناً با ایجاد انواع ترتیبی[۱۵] آشنا هستید.

```
type
  TComponentOption = (coDrawLines,
                      coDrawSolid,
                      coDrawBackground);
```

خصیصه‌هایی از این نوع، کامبوباکسی را با لیستی از همه مقادیر ممکن نشان میدهند، ولی گاهی لازم است که ترکیبی از این مقادیر (یا همه آنها) را مقداردهی کنیم. این جایی است که مجموعه‌ها نقش خودشان را ایفا میکنند.

```
Type
  TComponentOption = (coDrawLines,
                      coDrawSolid,
                      coDrawBackground);
  TComponentOptions = set of TComponentOption;
```

Publish کردن خصیصه‌ای از نوع TComponentOptions باعث ظاهر شدن [+] در کنار نام خصیصه میشود. وقتی روی [+] کلیک میکنید تا آنرا باز کنید، لیستی از گزینه‌ها را مشاهده میکنید. برای هر عنصر در TComponentOption یک خصیصه Boolean مشاهده خواهید کرد که میتوانید با تغییر مقادیر true / false عناصر را در لیست شامل یا مستثنی کنید.
چک کردن و اصلاح عناصر مجموعه درون کامپوننت ساده است.

```
if coDrawLines in OurComponentOptions
  then DrawTheLines;
```

یا

```
procedure      TsomeComponent.SetOurComponentOptions(const      value      :
TComponentOptions);
begin
  if (coDrawSolid in Value) and
  (coDrawBackground in value) then
    {raise an exception}

  FOurComponentOptions := Value;
  Invalidate;
end;
```

[14] Sets
[15] Ordinal

۸- خصیصه‌های باینری

خصیصه‌های باینری

گاهی لازم است که روتینهای جریانی[۱۶] خودتان را برای خواندن و نوشتن انواع ویژه در خصیصه‌ها بنویسید (مثل کاری که دلفی برای خواندن و نوشتن خصیصه‌های Top و Left کامپوننتهای "غیر قابل مشاهده در زمان اجرا" بدون اینکه آنها را در بازرس شئ publish کند انجام میدهد).

مثلاً یکبار من کامپوننتی نوشتم که یک فرم را بر مبنای یک تصویر بیت مپ شکل بدهد. در آن موقع کد من برای تبدیل بیت مپ به یک ناحیه ویندوز بینهایت کند بود و احتمالاً هیچ استفاده‌ای در زمان اجرا نمیشد از آن کرد. راه حل من این بود که داده‌ها را در زمان طراحی تبدیل کنم، و داده‌های باینری که از تبدیل به دست آمده‌اند بصورت جریان (stream) دربیاورم.

ایجاد خصیصه‌های باینری دربردارنده سه گام است:

۱. نوشتن متدی برای نوشتن داده‌ها.
۲. نوشتن متدی برای خواندن داده‌ها.
۳. اگاه کردن دلفی از اینکه ما یک خصیصه باینری داریم و ارسال متدهای خواندن و نوشتن.

```
type
  TBinaryComponent = class(TComponent)
  private
    FBinaryData : Pointer;
    FBinaryDataSize : DWord;
    procedure WriteData(S : TStream);
    procedure ReadData(S : TStream);
  protected
    procedure DefineProperties(Filer : TFiler); override;
  public
    constructor Create(AOwner : TComponent); override;
  end;
```

وقتی دلفی میخواهد کامپوننت ما را جریانی (stream) کند، DefineProperties بوسیله دلفی فراخوانی میشود. کاری که باید انجام دهیم این است که این متد را override کنیم، و با استفاده از TFilter.DefineProperty یا TFilter.DefineBinaryProperty یک خصیصه اضافه نماییم.

```
procedure TFiler.DefineBinaryProperty(const Name: string;
  ReadData, WriteData: TStreamProc;
  HasData: Boolean);

constructor TBinaryComponent.Create(AOwner: TComponent);
begin
  inherited;
  FBinaryDataSize := 0;
end;

procedure TBinaryComponent.DefineProperties(Filer: TFiler);
```

[16] Streaming

```
var
  HasData : Boolean;
begin
  inherited;
  HasData := FBinaryDataSize <> 0;
  Filer.DefineBinaryProperty('BinaryData',ReadData,
    WriteData, HasData );
end;

procedure TBinaryComponent.ReadData(S: TStream);
begin
  S.Read(FBinaryDataSize, SizeOf(DWord));
  if FBinaryDataSize > 0 then begin
    GetMem(FBinaryData, FBinaryDataSize);
    S.Read(FBinaryData^, FBinaryDataSize);
  end;
end;

procedure TBinaryComponent.WriteData(S: TStream);
begin
  //This will not be called if FBinaryDataSize = 0

  S.Write(FBinaryDataSize, Sizeof(DWord));
  S.Write(FBinaryData^, FBinaryDataSize);
end;
```

در ابتدا DefineProperties را Override کردیم. وقتی این کار انجام شد یک خصیصه باینری با مقادیر زیر تعریف کردیم

BinaryData: نام خصیصه غیرقابل مشاهده برای استفاده.

ReadData: روال مسوول خواندن داده‌ها.

WriteData: روال مسوول نوشتن داده‌ها.

HasData: اگر برابر false باشد، روال WriteData صدا زده نمیشود.

ماندگاری[۱۷]

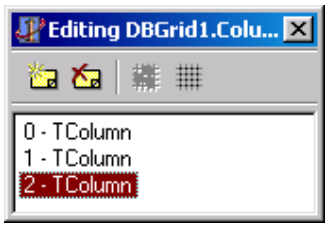
در اینجا ماندگاری تعریف میشود و ممکن است بعداً به این تعریف مراجعه کنیم. ماندگاری عاملی است که موجب میشود دلفی بتواند خصیصه‌های همه کامپوننتهایش را بخواند و بنویسد. TComponent از کلاسی به نام TPersistent مشتق میشود. TPersistent یک کلاس دلفی است که خصیصه‌هایش میتوانند بوسیله دلفی خوانده و نوشته شوند، یعنی هر فرزند TPersistent هم این قابلیت را دارد.

۹- Collectionها

هر چه در این مقاله جلوتر میرویم به مباحث پیچیده‌تری میپردازیم. Collectionها یکی از پیچیده‌ترین انواع خصیصه‌های استاندارد در دلفی هستند. اگر یک TDBGrid روی فرم بگذارید و به خصیصه‌هایش دقت کنید، خصیصه‌ای به نام "Columns" میبینید.

Columns یک خصیصه Collection است. وقتی روی دکمه [...] کلیک میکنید یک پنجره بازشونده کوچک مشاهده میکنید. این پنجره، ویرایشگرِ خصیصه استاندارد برای خصیصه‌های TCollection (و فرزندانش) است.

[17] Persistency



وقتی روی دکمه "New" کلیک کنید میبینید که یک آیتم جدید (یک آیتم TColumn) اضافه شده. کلیک کردن روی آن آیتم آنرا در بازرس شئ انتخاب میکند بطوریکه میتوانید خصیصه‌ها و رخدادهایش را تغییر دهید. این کار چطور انجام میشود؟ خصیصه Columns از TCollection مشتق میشود. TCollection شبیه به یک آرایه است، که لیستی از TCollectionItemها را دربر دارد. چون TCollection از TPersistent مشتق میشود، میتواند این لیست آیتمها را stream کند، بطور مشابه TCollectionItem هم مشتقی از TPersistent است و میتواند خصیصه‌هایش را stream کند. پس چیزی که ما در اختیار داریم یک آیتم شبیه آرایه است که میتواند همه آیتمهایش و خصیصه‌هایشان را stream کند.

اولین کاری که در هنگام ایجاد ساختار مبتنی بر TCollection / TCollectionItem باید انجام دهیم این است که CollectionItem را تعریف کنیم.

```
type
  TOurCollectionItem = class(TCollectionItem)
  private
    FSomeValue : String;
  protected
    function GetDisplayName : String; override;
  public
    procedure Assign(Source: TPersistent); override;
  published
    property SomeValue : String
      read FSomeValue
      write FSomeValue;
  end;
```

کاری که انجام دادیم این است که فرزندی از TCollectionItem ایجاد کرده‌ایم. یک خصیصه نشانه به نام "SomeValue" تعریف کرده‌ایم، تابع GetDisplayName را Override کردیم (تا متنی را که در ویرایشگر پیش‌فرض نشان میدهد تغییر دهیم)، و در نهایت متد Assign را Override کردیم تا به TOurCollectionItem امکان دهیم به یک TOurCollectionItem دیگر منتسب شود. اگر گام نهایی را انجام ندهیم آنگاه متد Assign کلاس Collection ما کار نخواهد کرد.

```
procedure TOurCollectionItem.Assign(Source: TPersistent);
begin
  if Source is TOurCollectionItem then
    SomeValue := TOurCollectionItem(Source).SomeValue
  else
    inherited; //raises an exception
end;

function TOurCollectionItem.GetDisplayName: String;
begin
```

```
  Result := Format('Item %d',[Index]);
end;
```

پیاده‌سازی TOurCollection پیچیده‌تر است و کار بیشتری میبرد.

```
  TOurCollection = class(TCollection)
  private
    FOwner : TComponent;
  protected
    function GetOwner : TPersistent; override;
    function GetItem(Index: Integer): TOurCollectionItem;
    procedure SetItem(Index: Integer; Value:
      TOurCollectionItem);
    procedure Update(Item: TOurCollectionItem);
  public
    constructor Create(AOwner : TComponent);

    function Add : TOurCollectionItem;
    function Insert(Index: Integer): TOurCollectionItem;

    property Items[Index: Integer]: TOurCollectionItem
      read GetItem
      write SetItem;
  end;
```

در مورد کد بالا به این مباحث توجه کنید:

GetOwner: متدی Virtual است که در TPersistent معرفی شد. این متد باید Override شود چون کد پیش فرض آن مقدار Nil برمیگرداند. ما در پیاده‌سازیمان سازنده را تغییر میدهیم تا فقط یک پارامتر دریافت کند (AOwner : TComponent). ما این پارامتر را در FOwner نگهداری میکنیم، که بعد بعنوان نتیجه GetOwner ارسال میشود (TComponent از TPersistent مشتق میشود، لذا این یک نوع برگشتی معتبر است).

```
constructor TOurCollection.Create(AOwner: TComponent);
begin
  inherited Create(TOurCollectionItem);
  FOwner := AOwner;
end;

function TOurCollection.GetOwner: TPersistent;
begin
  Result := FOwner;
end;
```

Create نه تنها Owner را ذخیره میکند (که برای عملکرد صحیح بازرس شئ ضروری است)، بلکه با فراخوانی "inherited Create (TOurCollectionItem)" به دلفی میگوید که CollectionItem ما چه کلاسی است.

GetItem / SetItem: به همان روشی که در TCollection وجود دارند اعلان شده‌اند، ولی بجای کار کردن در TCollectionItem آنها در کلاس جدید TOurCollectionItem کار میکنند. اینها بعداً در خصیصه "Items" استفاده خواهند شد.

Update: مثل حالت قبل جایگزینی است که در کلاس جدید کار میکند.

Add / Insert: هر دو مسوول اضافه کردن آیتمها به لیست هستند، و هر دو برای برگرداندن اشیاء از نوع کلاس متناسب تغییر یافته‌اند.

تهایتاً یک خصیصه "Items" معرفی گردیده است تا جایگزین خصیصه Items اولیه بشود، بگونه‌ای که بازهم نتیجه‌ای از نوع TOurCollectionItem بجای نوع TCollectionItem برگردانده باشیم تا ما را از مشکلات انجام هرباره type cast نجات دهد.

و بالاخره مثالی از پیاده‌سازی این نوع خصیصه در کامپوننت خودمان.

```
TCollectionComponent = class(TComponent)
  private
    FOurCollection : TOurCollection;
    procedure SetOurCollection(const Value:
      TOurCollection);
  public
    constructor Create(AOwner : TComponent); override;
    destructor Destroy; override;
  published
    property OurCollection : TOurCollection
      read FOurCollection
      write SetOurCollection;
  end;
```

وقتی کلاس TCollecton ما نوشته شده باشد قسمت سخت کار انجام گرفته است. سازنده ما کلاس Collection را ایجاد میکند، مخرب آنرا پاک میکند و SetOurCollection هم کار انتساب را انجام میدهد.

```
constructor TCollectionComponent.Create(AOwner: TComponent);
begin
  inherited;
  FOurCollection := TOurCollection.Create(Self);
end;

destructor TCollectionComponent.Destroy;
begin
  FOurCollection.Free;
  inherited;
end;

procedure TCollectionComponent.SetOurCollection(
  const Value: TOurCollection);
begin
  FOurCollection.Assign(Value);
end;
```

همانطور که قبلاً گفته شد، Self که به TOurCollectionItem.Create ارسال میگردد در متغیر Fowner کلاس TOurCollecton ذخیره میگردد، و بعنوان نتیجه GetOwner برگردانده میشود. یک نکته که در اینجا باید به آن توجه کنید این است که در SetOurCollection وقتی میخواهیم اشیاء را جایگزین کنیم نمی‌نویسیم := FOurCollection Value ، بلکه خصیصه را به مقدار منتسب میکنیم (چون اشیاء در واقع نوعی اشاره‌گر هستند).

نسخه‌های اخیر دلفی این کار را ساده‌تر میکنند. بجـای اجبـار بـه override کـردن GetOwner در کـلاس Collection خودمان، میتوانتیم از کلاس TOwnedCollection اشتقاق را انجام دهیم. این کلاس پوششی[18] برای TCollection است که این کار را برای ما انجام میدهد.

۱۰– زیرخصیصه‌ها[19]

زیرخصیصه‌ها

قبلاً دیدیم که چطور میتوانیم یک خصیصه قابل گسترش[20] ایجاد کنیم. محدودیت تکنیک قبلی در این بود که هر زیر ایتم به شکل یک خصیصه بولین ظاهر میشد. این بخش توضیح میدهد که چطور خصیصه‌های قابل گسترشی ایجاد کنیم کـه بتواننـد شامل هر نوع خصیصه‌ای باشند.

اگر یک کامپوننت به خصیصه‌ای از نوع رکورد نیاز داشته باشد، میتوان به سادگی با ارائه جداگانه هـر یـک از خصیصـه‌هـا آنـرا پیاده‌سازی کرد. ولی اگر کامپوننت ما بخواهد دو یا تعداد بیشتری خصیصه هم نوع معرفی نماید میبینیم که بازرس شئ بسیار پیچیده خواهد شد.

پاسخ در این است که یک ساختار پیچیده ایجاد نماییم (مثل یک رکورد یا شئ) و هر وقت لازم بـود ایـن سـاختار را بصـورت خصیصه publish نماییم. مشکل عمده این است که تا وقتی که به دلفی نگفته باشیم نمیداند چطور این خصیصـه را نمـایش دهد. ساخت یک ویرایشگر خصیصه[21] کامل مشکل خواهد بود، ولی خوشبختانه دلفی در ایـن مـورد راه‌حلـی را تـدارک دیـده است. همانطور که قبلاً گفته شد، streaming داخلی دلفی حول کلاس TPersistent قرار دارد. بنابراین گام اول ایـن اسـت که ساختار پیچیده‌مان را از این کلاس مشتق کنیم.

```
type
  TExpandingRecord = class(TPersistent)
  private
    FIntegerProp : Integer;
    FStringProp : String;
    FCollectionProp : TOurCollection;
    procedure SetCollectionProp(const Value:
      TOurCollection);
  public
    constructor Create(AOwner : TComponent);
    destructor Destroy; override;

    procedure Assign(Source : TPersistent); override;
  published
    property IntegerProp : Integer
      read FIntegerProp
      write FIntegerProp;
    property StringProp : String
      read FStringProp
      write FStringProp;
    property CollectionProp : TOurCollection
      read FCollectionProp
      write SetCollectionProp;
  end;
```

[18] Wrapper
[19] Sub-properties
[20] Expandable
[21] Property editor

در ساختار بالا ما مشتقی از کلاس TPersistent ایجاد کردیم و سه خصیصه نمونه: یک Integer، یک String و یک خصیصه از نوع Collection که قبلاً در این مقاله ساخته بودیم را به آن اضافه کردیم.
سازنده و مخرب در مورد ایجاد و از بین بردن شئ CollectionProp کار میکنند و SetCollectionProp برای این پیاده-سازی میشود که مانع از دست رفتن این ارجاع شئ شود (بجای اینکه بنویسیم FCollectionProp := Value مینویسیم Assign(Value)). Assign برای این پیاده‌سازی میشود که بتوانیم خصیصه‌های TExpandableRecord را به یک TExpandableRecord دیگر منتسب کنیم. (و این کار هم ضروری است چون لازم خواهیم داشت که در نهایت وقتی که بعنوان یک خصیصه کامپوننت پیاده‌سازی شد آنرا نسبت دهیم).

```
procedure TExpandingRecord.Assign(Source: TPersistent);
begin
  if Source is TExpandingRecord then
    with TExpandingRecord(Source) do begin
      Self.IntegerProp := IntegerProp;
      Self.StringProp := StringProp;

      //This actually assigns
      Self.CollectionProp := CollectionProp;
    end else
      inherited; //raises an exception
end;

constructor TExpandingRecord.Create(AOwner : TComponent);
begin
  inherited Create;
  FCollectionProp := TOurCollection.Create(AOwner);
end;

destructor TExpandingRecord.Destroy;
begin
  FCollectionProp.Free;
  inherited;
end;

procedure TExpandingRecord.SetCollectionProp(const Value: TOurCollection);
begin
  FCollectionProp.Assign(Value);
end;
```

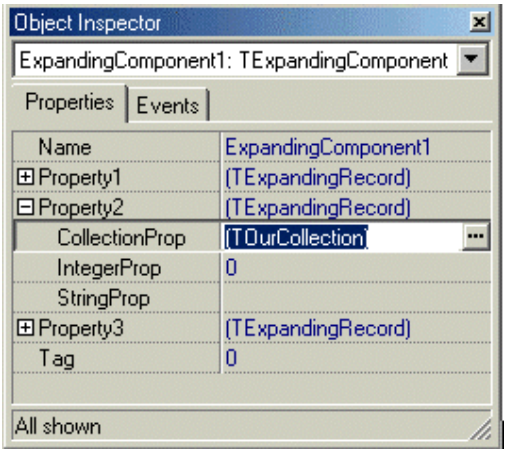
با پیاده‌سازی این کد، قسمت سخت کار انجام گرفته است. اکنون پیاده‌سازی این کلاس بصورت یک شئ ساده است.

```
TExpandingComponent = class(TComponent)
  private
    FProperty1,
    FProperty2,
    FProperty3 : TExpandingRecord;
  protected
    procedure SetProperty1(const Value :
      TExpandingRecord);
    procedure SetProperty2(const Value :
      TExpandingRecord);
    procedure SetProperty3(const Value :
      TExpandingRecord);
```

```
  public
    constructor Create(AOwner : TComponent); override;
    destructor Destroy; override;
  published
    property Property1 : TExpandingRecord
      read FProperty1
      write SetProperty1;
    property Property2 : TExpandingRecord
      read FProperty2
      write SetProperty2;
    property Property3 : TExpandingRecord
      read FProperty3
      write SetProperty3;
  end;
```

متد سازنده سه شیئی که بعنوان خصیصه مورد استفاده قرار گرفته‌اند را ایجاد میکند، و روشن است که متد مخرب هم برای آزاد کردن آنها استفاده میشود. متدهای SetPropertyX کار انتساب مقدار (Assign(Value)) را به شئ مناسب انجام میدهند.

این کامپوننت را به شکل یک بسته (Package) کامپایل کنید و یک TExpandableComponent روی فرم قرار دهید. با مشاهده بازرس شئ خواهید دید که خصیصه‌های TExpandingRecord همگی یک [+] در کنارشان دارند، کلیک کردن روی این دکمه باعث گسترش خصیصه و آشکار شدن همه زیرخصیصه‌ها میشود.



در نگاه اول همه چیز خوب کار میکند، خصیصه‌ها قابلیت گسترش و ارائه زیرخصیصه‌ها را دارند ولی اگر روی دکمه [...] مربوط به "CollectionProp" کلیک کنیم، ویرایشگر استاندارد TCollection ظاهر نمیشود، در واقع هیچ اتفاقی نمی-افتد.

ممکن است بپرسید "چه کاری را اشتباه انجام دادیم؟"، جواب این است "هیچ کاری را!". خطایی که در اینجا وجود دارد از ناحیه ما نیست بلکه متوجه توسعه دهندگان بورلند دلفی است. توسعه دهندگان دلفی هم مصون از خطا نیستند.

وقتی یک ویرایشگر خصیصه را ثبت کنید میتوانید حوزه‌ای از کامپوننت را که روی آن کار میکند محدود نمایید. میتوانید اینطور مشخص کنید که باید فقط روی خصیصه‌هایی با اسامی خاص کار کند، و یا فقط روی کامپوننتهای خاصی کار کند. با وجودی که معماری دلفی اینطور تعریف میکند که پایین‌ترین شکل شیئی قادر به Streaming کلاس TPersistent است، یک نفر در بورلند ویرایشگر خصیصه را اینطور ثبت کرده که فقط با اشیاء مشتق شده از TComponent کار کند.

راه حل مشکل این است که TExpandableRecord را بجای اینکه از TPersistent مشتق کنیم، از TComponent مشتق کنیم. مشکل این است که ویرایشگر خصیصه پیش‌فرض دلفی برای خصیصه‌های نوع TComponent (و فرزندانش) بجای نمایش یک نمای قابل گسترش از زیرخصیصه‌ها یک کامبوباکس نمایش میدهد.
راه‌حل جامع این مساله در ویرایشگر خصیصه است که در ادامه به آن پرداخته میشود.

۱۱- ویرایشگرهای ویژه، استانداردهای کدنویسی

ویرایشگرهای کامپوننت ویژه

با وجودیکه بازرس شئ دلفی میتواند بیشتر انواع خصیصه را شناسایی کند ولی نمیتواند با هر نوعِ ویژه‌ای که ممکن است در کامپوننت نوشته باشیم کار کند. بنابراین ممکن است لازم شود که ویرایشگرهای خصیصه / کامپوننت بنویسیم. دلفی تعداد زیادی ویرایشگر پیش‌فرض دارد. این ویرایشگرها در فایل DsgnIntf.pas در مسیر $(Delphi)\Source\ToolsAPI قرار دارند، بنابراین باید این یونیت را در ویرایشگر کامپوننت / خصیصه‌ای که مینویسید با دستور uses ذکر کنید. همچنین خوب است در حین نوشتن ویرایشگر این فایل را باز کنید تا به آن مراجعه نمایید.

استانداردهای کدنویسی

در این قسمت استانداردهایی را که در نوشتن کد ویرایشگر استفاده میشوند شرح میدهیم. تعداد این استانداردها کم است ولی بکار بردن آنها کار خوبی است چون درک ویرایشگر شما را برای دیگران ساده‌تر میکند.

- وقتی یک ویرایشگر خصیصه مینویسید، نام ویرایشگر را با کلمه "Property" به پایان برسانید؛ مثلاً TAngleProperty.
- وقتی ویرایشگر کامپوننت مینویسید، نام آنرا با کلمه "Editor" به پایان برسانید. مثلاً TPieChartEditor.
- وقتی ویرایشگر مینویسید، حتماً آنرا در یک یونیت جدا از کد کامپوننت قرار دهید. جدا کردن کد زمان اجرا از کد زمان طراحی کار خوبی است، همچنین این کار اندازه فایل اجرایی بدست آمده را کاهش میدهد (در بعضی نسخه‌های دلفی ممکن است جدا نبودن کد ویرایشگر مانع عمل کامپایل شود).
- فایل یونیت ویرایشگر را به همان نام یونیت کامپوننت نامگذاری کنید، فقط کلمه "reg" را به انتهای آن اضافه کنید. مثلاً کامپوننتی با نام یونیت "MyComponent.pas" منجر به نام فایل "MyComponentReg.pas" برای ویرایشگر میشود.
- نهایتاً اینکه، وقتی ویرایشگر کامپوننت / خصیصه مینویسید، عبارت *RegisterComponents* را از یونیت کامپوننت خارج کنید، و آنرا در یونیت ویرایشگر کامپوننت بنویسید. به این ترتیب کامپوننت شما بدون ویرایشگر به ثبت نمیرسد.

۱۲- ویرایشگر کامپوننت، اضافه کردن امکانات به ویرایشگرهای دیگران

ویرایشگر کامپوننت

ویرایشگرهای کامپوننت وقتی فعال میشوند که برنامه‌نویس در زمان طراحی روی کامپوننت راست کلیک میکند، درست قبل از اینکه context menu ظاهر شود دلفی بدنبال یک ویرایشگر کامپوننت مرتبط میگردد. اگر ویرایشگری پیدا شود دلفی متدهای ویرایشگر کامپوننت را اجرا میکند تا امکان دهد آیتمهای منوی اختصاصی آن کامپوننت اضافه گردند.
با وجودیکه امکان نوشتن ویرایشگرهای خصیصه با عملکردی ساده‌تر نسبت به یک ویرایشگر کامپوننت (مثلاً به طرقی که قبلاً گفته شد) وجود دارد، ولی چون فقط پنج متد و دو خصیصه هستند که برای نوشتن ویرایشگر باید با آنها کار کنیم، نوشتن ویرایشگر کامپوننت هم میتواند قابل بررسی باشد.

```
procedure Edit; virtual;
function GetVerbCount: Integer; virtual;
function GetVerb
  (Index: Integer): string; virtual;
procedure ExecuteVerb(Index: Integer); virtual;
procedure PrepareItem
  (Index: Integer; const AItem: TMenuItem); virtual;

property Component: TComponent read FComponent;
property Designer: IFormDesigner read GetDesigner;
```

Edit

وقتی کاربر روی کامپوننت دبل کلیک کند فراخوانده میشود. بسیاری از کامپوننتها از این متد استفاده میکنند تا یک رخداد OnClick ایجاد نمایند، ولی مثلاً TForm هم هست که از آن برای رخداد OnCreate استفاده میکند. میتوان این عملکرد استاندارد را override کرد تا IDE دلفی کار دلخواه ما را انجام دهد.

GetVerbCount

وقتی توسط دلفی فراخوانی میشود که دلفی میخواهد بداند ما چه تعداد آیتم میخواهیم به context menu اضافه کنیم.

GetVerb

برای هر آیتم منو فراخوانده میشود. اگر اینطور مشخص کنیم که میخواهیم چهار آیتم اضافه کنیم، چهار بار فراخوانده میشود (با مقادیر 0, 1, 2, 3). هدف این متد این است که IDE را از عنوان[۲۲]هایی که در context menu ظاهر میشوند آگاه نماید.

```
case Index of
  0: Result := 'Item 0';
  1: Result := 'Item 1';
end;
```

ExecuteVerb

وقتی برنامه‌نویس یکی از آیتمهای منوی اختصاصی را انتخاب کند فراخوانده میشود.

PrepareItem

برای هر آیتمی که اضافه کرده‌ایم یکبار فراخوانی میشود. دلفی بلافاصله بعد از اینکه ایجاد آیتم منو را به پایان برد این متد را فرامیخواند. میتوان آیتم را مخفی یا غیرفعال کرد ویا زیرآیتم اضافه نمود.

توجه: در هنگام استفاده از PrepareItem مراقب باشید که نباید آیتم منو را free کنید. اگر قصد ندارید که آیتم را نمایش دهید باید Visible را روی مقدار false قرار دهید. همچنین مطمئن شوید که Menus را به قسمت Uses اضافه میکنید.

[22] Caption

Component

این خصیصه باید برای دسترسی به کامپوننتی که روی آن راست کلیک شده مورد استفاده قرار گیرد. میتوانید به سادگی آنرا به نوع صحیح کلاس typecast نمایید.

Designer

این اینترفیس طراحی کننده دلفی است و میتواند کارهای زیادی انجام دهد که بیشترشان در حوزه بحث این مقاله نیستند، ولی در اینجا برای مطلع کردن IDE از اینکه ما چیزی را تغییر داده‌ایم بکار میرود. در نتیجه دلفی خصیصه‌ها را در بازرس شئ بِروز میکند و نشان گذاری میکند که پروژه تغییر کرده است. بعد از اینکه همه متدهای مربوط را override کردیم، کاری که باقی میماند رجیستر کردن آن است:

```
procedure Register;
begin
  RegisterComponents( [ComponentClass], 'Tab name');
  RegisterComponentEditor( ComponentClass, ComponentEditor);
end;
```

مثال زیر نشان میدهد که چطور دو آیتم (open, close) را به context menu یک TTable اضافه کنیم. TTableEditor ما از TComponentEditor مشتق میشود که کلاس پایه همه ویرایشگرهای کامپوننت محسوب میشود.

```
type
  TTableEditor = class(TComponentEditor)
  public
    { Public declarations }
    procedure Edit; override;
    procedure TTableEditor.ExecuteVerb(Index: Integer);
    function GetVerb(Index: Integer): string; override;
    function GetVerbCount: Integer; override;
    procedure PrepareItem
    (Index: Integer; const AItem: TMenuItem); override;
  end;

procedure TTableEditor.Edit;
begin
  with TTable(Component) do
    if Active then
      Close
    else
      Open;
end;

procedure TTableEditor.ExecuteVerb(Index: Integer);
begin
  case Index of
    0: TTable(Component).Open;
    1: TTable(Component).Close;
  end;
  //Tell the IDE something changed
  Designer.Modified;
end;

function TTableEditor.GetVerb(Index: Integer): string;
```

```
begin
  case Index of
    0 : Result := 'Open';
    1 : Result := 'Close';
  end;
end;

function TTableEditor.GetVerbCount: Integer;
begin
  Result := 2;
end;

procedure TTableEditor.PrepareItem
  (Index: Integer; const AItem: TMenuItem);
begin
  case Index of
    0: AItem.Enabled := not TTable(Component).Active;
    1: AItem.Enabled := TTable(Component).Active;
  end;
  //Tell the IDE something changed
  Designer.Modified;
end;
```

اضافه کردن آیتمهای بیشتر

اگر مثال بالا را کامپایل کنید و آنرا نصب نمایید، هر وقت که روی TTable کلیک کنید دو آیتم اضافی "Open" و "Close" را میبینید. وقتی میگوییم "اضافی" منظورمان اضافه فقط نسبت به context menu استاندارد TComponent است.

توجه: به رجیستر کردن ویرایشگرهای کامپوننت / خصیصه در ادامه خواهیم پرداخت.

ترتیب اولویت

اگر برای کامپوننتی که ویرایشگر خاص خودش را دارد یک ویرایشگر دیگر بنویسیم چه اتفاقی می‌افتد؟ فرض کنید ویرایشگر مثال قبل را برای TDateSet مینوشتیم تا هم در TTable و هم در TQuery قابل فراخوانی باشد. مشکل در این است که TTable و TQuery هر دو از قبل ویرایشگرهای خاص خودشان را دارند که باعث توقف فراخوانی ویرایشگر رجیستر شده در TDataSet میشود. اتفاقی که می‌افتد به این ترتیب است:

۱. برنامه‌نویس روی کامپوننت راست کلیک میکند.
۲. دلفی کلاس کامپوننتی که روی آن کلیک شده را میگیرد.
۳. دلفی چک میکند که آیا ویرایشگر اختصاصی برای این کلاس رجیستر شده است.
۴. اگر اینطور نباشد، دلفی کلاس والد را تا زمان پیدا کردن یک ویرایشگر چک میکند.
۵. اگر ویرایشگری پیدا شود، یک نمونه ایجاد میگردد و اجرا میشود.

چون دلفی ویرایشگر کامپوننت را در سطحی بعد از TDataSet رجیستر کرده است، حتی نگاهی هم به ویرایشگر ما نمیکند! در ابتدا جواب مساله خیلی ساده به نظر میرسد، اگر ما دوبار ویرایشگر کامپوننتمان را رجیستر کنیم (یکبار برای TTable و یکبار برای TQuery) آنگاه ویرایشگر کامپوننتمان اجرا میشود (دلفی ویرایشگری را که جدیدتر نصب شده استفاده میکند). البته این کاملاً درست است، و همینطور میشود. مشکل این است که حالا ما ویرایشگر استاندارد را حذف کرده‌ایم، لذا عملیات استانداردی مثل

· Fields editor
· Explore
· Execute (TQuery)
· Delete table (TTable)
· Rename table (TTable)
· Update table definition (TTable)

دیگر در لیست ظاهر نمیشوند و البته نمیتوان نبود آنها را پذیرفت. حالا آیا باید این عملیات را خودمان اضافه کنیم؟ من شخصاً از چنین کاری خوشم نمی‌آید!

سه راه‌حل برای این مساله پیشنهاد میشوند، متاسفانه هیچکدام کار نمیکنند و مساله حل نشده باقی میماند.

راه حل ۱: وقتی GetVerbCount و بقیه را برمیگردانید، متدهای inherited را صدا بزنید و بعد یک نتیجه اصلاح شده، یعنی:

```
function TMyEditor.GetVerbCount: Integer;
begin
  //We want to add 2 additional items
  Result := inherited GetVerbCount +2;
end;
```

مشکل: ویرایشگر ما از TComponentEditor مشتق میشود، یعنی هیچ verb برگردانده نمیشود.

راه حل۲: وقتی ویرایشگر ساخته شد، به لیست ویرایشگرهای رجیستر شده نگاه کنید و آنی که قبل از مال ما رجیستر شده را بیابید. نمونه‌ای از آن کلاس ایجاد کنید تا بتوانیم بفهمیم چند verb دارد.

مشکل: تابع GetComponentEditor در DsgnIntf.pas ویرایشگر خود ما را برمیگرداند، و "ComponentClassList" که استفاده میکند پایین بخش implementation یونیت اعلان شده و لذا قابل دسترسی نیست.

راه حل۳: ویرایشگر کامپوننتمان را از ویرایشگر کامپوننت موجود مشتق کنیم و آنگاه فراخوانی متدهای Inherited مقادیر درست را برمیگرداند.

مشکل: همه نسخه‌های دلفی دایرکتوری "$(Delphi)\Source\Property Editors" را ندارند (نسخه استاندارد) و همه نسخه‌های دلفی که این دایرکتوری را دارند توانایی کامپایل محتویات آنرا ندارند (نسخه حرفه‌ای).

این مشکلات وقتی بروز میکنند که میخواهیم با ویرایشگرهای استاندارد دلفی تعامل کنیم، با وجود این وقتی کامپوننتهای ساخته شده از جانب اشخاص دیگر که ویرایشگر خودشان را دارند نصب میکنیم این تضمین وجود دارد که این ویرایشگرها کامپایل خواهند شد (در غیر اینصورنصب نمیشدند).

توجه: به یاد داشته باشید که این ویرایشگر را uninstall کنید وگرنه نخواهید توانست به عملکردهای پیش‌فرض این شئ دسترسی داشته باشید.

اضافه کردن آیتم به ویرایشگرهای ساخت دیگران

اضافه کردن آیتم به ویرایشگر کامپوننتی که بوسیله دیگران ثبت شده ساده است. همه کاری که باید بکنیم ارثبری کلاس ویرایشگر خودمان از آن ویرایشگر است.

مثال زیر نشان میدهد چطور یک TPanelEditor بسازیم که روش سریعی را برای پاک کردن عنوان یک TPanel فراهم میکند. بعد ما یک ویرایشگر جدید از این کلاس مشتق میکنیم و دو ویژگی اضافی را پیاده‌سازی میکنیم.

```
  TPanelEditor = class(TComponentEditor)
  public
    { Public declarations }
    procedure ExecuteVerb(Index: Integer); override;
    function GetVerb(Index: Integer): string; override;
    function GetVerbCount: Integer; override;
  end;

procedure TPanelEditor.ExecuteVerb(Index: Integer);
begin
  TPanel(Component).Caption := '';
end;

function TPanelEditor.GetVerb(Index: Integer): string;
begin
  Result := 'Clear caption';
end;

function TPanelEditor.GetVerbCount: Integer;
begin
  Result := 1;
end;
```

با مشتق کردن ویرایشگر جدید از ویرایشگر اصلی حتی اگر بعداً ویرایشگر اصلی توسط نویسنده آن تغییر کند ما نیازی به تغییر کد خودمان نداریم. زیرا ویرایشگر خودمان را مستقل کرده‌ایم.

```
TAdvancedPanelEditor = class(TPanelEditor)
    { Public declarations }
    procedure ExecuteVerb(Index: Integer); override;
    function GetVerb(Index: Integer): string; override;
    function GetVerbCount: Integer; override;
  end;
```

گام اول:

IDE باید بداند در ویرایشگر ما چند آیتم وجود دارند. بنابراین از کلاس inherited میپرسیم که چند آیتم وجود دارند و بعد تعداد آیتمهای "جدید" را به نتیجه اضافه میکنیم. در مثال بعدی دو آیتم اضافه میشوند.

```
function TAdvancedPanelEditor.GetVerbCount: Integer;
begin
  Result := inherited GetVerbCount + 2;
end;
```

گام دوم:

حالا IDE باید بداند که هر آیتم لیست را چطور نامگذاری کند. در اینجا ما تصمیم میگیریم که باید عنوان یکی از آیتمهای جدید را برگردانیم یا عنوان یکی از آیتمهای اصلی را.

```
function TAdvancedPanelEditor.GetVerb
                (Index: Integer): string;
var
  NewIndex: Integer;
begin
  if Index < inherited GetVerbCount then
    Result := inherited GetVerb(Index)
  else begin
    NewIndex := Index - inherited GetVerbCount;
    case NewIndex of
      0: Result := 'Align client';
      1: Result := 'Align bottom';
    end;
  end;
end;
```

گام سوم:

در نهایت، باید کد مناسب را در هنگام انتخاب یکی از آیتمها ویرایشگر اجرا کنیم. در اینجا هم باید تعیین کنیم که کد کلاس والد اجرا شود یا کد ما.

```
procedure TAdvancedPanelEditor.ExecuteVerb
                  (Index: Integer);
var
  NewIndex: Integer;
begin
  if Index < inherited GetVerbCount then
    inherited ExecuteVerb(Index)
  else begin
    NewIndex := Index - inherited GetVerbCount;
    case NewIndex of
      0: TPanel(Component).Align := alClient;
      1: TPanel(Component).Align := alBottom;
    end;
  end;
end;
```

۱۳- ویرایشگر خصیصه؛ رجیستر کردن ویرایشگرهای خصیصه

ویرایشگر خصیصه

IDE برای اینکه امکان ویرایش خصیصه‌های کامپوننت را بوجود آورد از ویرایشگرهای خصیصه استفاده میکند. بعضی ویرایشگرها ساده و برخی هم پیچیده هستند. خود دلفی دارای چند ویرایشگر استاندارد میباشد، بعضی از آنها عبارتند از:

- *TIntegerProperty*: برای وارد کردن مقادیر صحیح.
- *TCharProperty*: برای وارد کردن یک کاراکتر تکی.
- *TEnumProperty*: برای یک عنصر انتخاب شده در یک نوع شمارشی (مثل alTop, alClient, ...).
- *TBoolProperty*: برای انتخاب "True" یا "False" برای مقادیر بولین.
- *TFloatProperty*: برای وارد کردن اعداد ممیز شناور (متغیرهای از نوع Float و Extended و غیره. نوع "Real" نباید برای خصیصه‌های کامپوننت بکار رود).
- *TStringProperty*: برای وارد کردن رشته‌های بطول حداکثر ۲۵۵ کاراکتر.
- *TSetProperty*: برای شامل یا مستثنی کردن عناصر یک خصیصه مجموعه‌ای. هر عنصر به شکل یک زیرخصیصه بولین نمایش داده میشود. مقدار "True" عنصر را شامل و مقدار "False" عنصر را مستثنی میکند.
- *TClassProperty*: این کلاس پایه‌ای است که وقتی میخواهید یک ویرایشگر اختصاصی برای خصیصه‌های یک کلاس خاص فعال شود باید از آن اشتقاق را انجام دهید (وقتی که نوع خصیصه یک کلاس است مثلاً TImage.Picture).

همه این ویرایشگرهای خصیصه مستقیم یا غیرمستقیم از TPropertyEditor مشتق میشوند. این کلاس خصیصه‌ها و متدهای زیادی دارد، مهمترینها عبارتند از:

```
function AllEqual: Boolean; virtual;
function GetAttributes: TPropertyAttributes; virtual;
procedure Edit; virtual;
function GetValue: string; virtual;
procedure GetValues(Proc: TGetStrProc); virtual;
```

AllEqual

وقتی چند کامپوننت انتخاب میشوند بازرس شئ لیست خصیصه‌هایش را طوری فیلتر میکند که فقط خصیصه‌هایی که در همه کامپوننتهای انتخاب شده مشترک هستند ظاهر شوند. اگر مقدار هر یک از این خصیصه‌ها (مثل width) در همه کامپوننتها یکی باشد همان مقدار نشان داده میشود وگرنه مقدار خصیصه برای کامپوننتی که زودتر به فرم اضافه شده نشان داده میشود. AllEqual روتینی است که در این مورد عمل میکند.

```
function TStringProperty.AllEqual: Boolean;
var
  I: Integer;
  V: string;
begin
  Result := False;
  if PropCount > 1 then
  begin
    V := GetStrValue;
    for I := 1 to PropCount - 1 do
      if GetStrValueAt(I) <> V then Exit;
```

```
  end;
  Result := True;
end;
```

در مثال بالا TStringProperty هر یک از مقادیر را (با استفاده از GetStrValueAt) با مقدار اولین کامپوننت لیست مقایسه میکند (با استفاده از GetStrValue، GetStrValueAt(0) هم همین کار را انجام میداد). اندازه لیست با استفاده از PropCount تعیین میشود، که تعداد کل کامپوننتهای انتخاب شده را برمیگرداند.

GetAttributes

GetAttributes وقتی بوسیله IDE فراخوانی میشود که IDE میخواهد اطلاعاتی درباره ویرایشگر خصیصه جمع‌آوری کند. بازرس شئ ویرایشگر متناسبی را بر اساس اطلاعات مهیا شده نمایش میدهد. حاصل GetAttributes (TPropertyAttributes) یک مجموعه است، لذا میتواند حاوی ترکیبی از مقادیر زیر باشد (البته این لیست کامل نیست)

paDialog

به بازرس شئ میگوید که یک دکمه [...] بعد از نام خصیصه نمایش بدهد، وقتی کاربر روی این دکمه کلیک کند متد Edit فعال میشود.

paSubProperties

به بازرس شئ میگوید که یک دکمه توسعه [+] قبل از نام خصیصه نمایش دهد، کلیک کردن روی این دکمه لیست گسترش یافته‌ای از زیر خصیصه‌ها (معمولاً خصیصه‌های publish شده یک خصیصه از نوع کلاس) را نمایش میدهد.

paValueList

بازرس شئ کامبوباکسی را بهمراه لیستی از مقادیر نمایش میدهد، این لیست توسط IDE با فراخوانی متد GetValues مشخص میشود.

توجه: متد GetValues، نه متد GetValue که کاملاً متفاوت است.

paSortList

اگر با paValueList ترکیب شود، مقادیر نمایش داده شده بطور الفبایی مرتب میشوند.

paMultiSelect

به IDE میگوید که وقتی چند کامپوننت انتخاب شده باشند کامپوننت مجاز به نشان داده شدن هست. این آیتم برای ویرایشگرهایی مثل TClassProperty وجود ندارد.

paAutoUpdate

باعث میشود بجای منتظر ماندن برای آنکه کاربر خصیصه دیگری را فشار دهد یا ویرایش کند، هر وقت مقدار در بازرس شئ تغییر کرد متد SetValue صدا زده شود. این حالت برای خصیصه‌های "Caption" و "Text" استفاده میشود، تا ورودی کاربر حالت "زنده" داشته باشد.

paReadOnly

اگر این عنصر در نظر گرفته شده باشد، مقدار موجود در بازرس شئ فقط خواندنی خواهد بود. این عنصر معمولاً بهمراه paDialog بکار میرود. متد GetValue، override میشود تا نمایشی توصیفی از خصیصه ارائه نماید.

Edit

وقتی دکمه [...] خصیصه فشار داده شود این متد فراخوانی میشود. این دکمه وقتی نمایش داده میشود که عنصر paDialog در نتیجه‌ی حاصل از متد GetAttributes موجود باشد.

GetValue

وقتی بازرس شئ میخواهد بداند که چطور خصیصه را به شکل یک رشته نمایش دهد این متد فراخوانی میشود. این حالت وقتی بکار گرفته میشود که در نتیجه‌ی متد GetAttributes، مقادیر [paDialog, paReadOnly] وجود داشته باشند.

GetValues

این متد وقتی فراخوانی میشود که بازرس شئ میخواهد لیستی از مقادیر را، در هنگامی که paValueList در نتیجه‌ی حاصل از GetAttributes وجود دارد، برای نمایش دادن دریافت کند.

GetValues پارامتری به نام "Proc" ارسال مینماید که از نوع TGetStrProc است. GetStrProc بصورت TGetStrProc = procedure(const S: string) of object; اعلان گردیده است.

IDE انتظار دارد که "Proc" برای هر مقداری که باید برای این خصیصه در بازرس شئ نشان داده شود، یکبار فراخوانده شود.

```
procedure THintProperty.GetValues(Proc: TGetStrProc);
begin
  Proc('First item to display');
  Proc('Second item to display');
end;
```

مثال بعدی نشان میدهد که چطور لیستی از مقادیر پیش‌فرض را برای خصیصه "Hint" کامپوننتها تهیه کنیم، ضمن اینکه همچنان به کاربر اجازه میدهیم مقداری که در لیست وجود ندارد را وارد کند.

```
type
  THintProperty = class(TStringProperty)
  public
    function GetAttributes: TPropertyAttributes; override;
    procedure GetValues(Proc: TGetStrProc); override;
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterPropertyEditor(TypeInfo(String), nil, 'Hint', THintProperty);
end;

{ THintProperty }

function THintProperty.GetAttributes: TPropertyAttributes;
begin
  Result := inherited GetAttributes + [paValueList, paSortList];
end;

procedure THintProperty.GetValues(Proc: TGetStrProc);
begin
  Proc('This is a required entry');
  Proc('Press F1 for more information');
  Proc('This value is read-only');
end;
```

در ابتدا GetAttributes، override میشود و [paValueList, paSortList] در نتیجه خروجی قرار میگیرند. بعد GetValues، override میشود و با فراخوانی روال "Proc"، سه مقدار به لیست پایین افتادنی اضافه میشوند.

رجیستر کردن ویرایشگرهای خصیصه

نهایتاً ویرایشگر خصیصه با استفاده از RegisterPropertyEditor رجیستر میشود. RegisterPropertyEditor چهار پارامتر میگیرد:

PropertyType: PTypeInfo
به اشاره‌گری به یک رکورد TTypeInfo نیاز دارد. کاری که باید انجام دهیم این است که TypeInfo را به قسمت uses اضافه کنیم و تابع TypeInfo را بکار ببریم تا اشاره‌گر را بدست آوریم: TypeInfo(SomeVariableType)

ComponentCalss: TClass
این کلاس پایه‌ای است که این ویرایشگر باید برای آن بکار رود. ویرایشگر برای این کلاس و همگی فرزندانش بکار گرفته میشود. اگر مقدار nil مشخص شده باشد، این ویرایشگر برای هر کلاسی بکارمیرود.

const PropertyName: String
اگر این ویرایشگر باید فقط برای یک خصیصه ویژه بکار برود آنگاه نام خصیصه باید در اینجا مشخص شود. اگر ویرایشگر باید برای هر خصیصه‌ای از نوع مشخص شده در PropertyType بکار رود، آنگاه این مقدار باید برابر "" باشد.

EditorClass: TPropertyEditorClass
این کلاسی است که برای سروکار داشتن با خصیصه ایجاد شده. در مثال بالا کلاس THintProperty است.

استفاده نادرست از RegisterPropertyEditor

مهم است که در هنگام استفاده از RegisterPropertyEditor اطلاعات درست را فراهم کنید. تهیه اطلاعات نادرست میتواند به معنای باشد که ویرایشگر شما بر روی خصیصه‌های نادرست تاثیر میگذارد (مثلاً همه خصیصه‌های رشته‌ای) و یا اینکه روی کامپوننتهای نادرستی کار میکند.
از طرف دیگر، مقداردهی نادرست پارامترها میتواند به معنای این باشد که فقط یک خصیصه خاص در یک کامپوننت خاص (و فرزندانش) به ویرایشگر شما مرتبط است. این مساله در ابتدا خیلی مثل یک مشکل به نظر نمیرسد، ولی کامپوننتهای فرزند ممکن است بخواهند که خصیصه‌های بیشتری از همان نوع را پیاده‌سازی کنند. این خصیصه‌ها با وجودیکه نام متفاوتی خواهند داشت ولی ویرایشگر خصیصه‌ای به آنها منتسب نخواهد بود.
یک نمونه از ویرایشگر بد رجیستر شده در VCL وجود دارد. ویرایشگر استاندارد برای TCollection برای همه کلاسهای مشتق شده از TComponent رجیستر شده بود. مشکل این است که پایین‌ترین کلاس قابل نمایش در بازرس شئ TPersistent است (کلاسی که TComponent از آن مشتق میشود).
اگر کامپوننتی خصیصه‌ای از نوع TPersistent داشته باشد (که بطور پیش‌فرض زیرخصیصه‌هایش را در یک لیست بازشدنی نشان میدهد)، و یکی از خصیصه‌هایش از نوع TCollection باشد، نتیجه یک دکمه [...] در بازرس شئ است که وقتی روی آن کلیک شود کاری انجام نمیدهد (قبلاً در این مورد توضیح داده شد).
راه‌حل این مشکل کاملاً ساده به نظر میرسد. بجای اینکه زیرخصیصه ما از TPersistent مشتق شود میتوانیم آنرا از TComponent مشتق کنیم. با وجود این، رفتار پیش‌فرض خصیصه‌ای از نوع TComponent (بگونه‌ای که توسط

ویرایشگر TComponentPropertyEditor مشخص گردیده) این است که بجای نشان دادن زیرخصیصه‌های یک کامپوننت تعبیه شده، لیستی از کامپوننتهای دیگر را نشان دهد.
راه حل واقعی واقعاً ساده است, به شرطی که بدانید چطور یک ویرایشگر خصیصه بنویسید.

گام اول:

```
type
  TExpandingRecord = class(TPersistent)
```

باید اینطور تغییر کند:

```
type
  TExpandingRecord = class(TComponent)
```

گام دوم:

یک ویرایشگر خصیصه بصورت زیر ایجاد کنید:

```
type
  TExpandingRecordProperty = class(TClassProperty)
  public
    function GetAttributes : TPropertyAttributes; override;
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Article', [TExpandingComponent]);
  RegisterPropertyEditor(TypeInfo(TExpandingRecord),
    nil, '', TExpandingRecordProperty);
end;

{ TExpandingRecordProperty }

function TExpandingRecordProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paReadOnly, paSubProperties];
end;
```

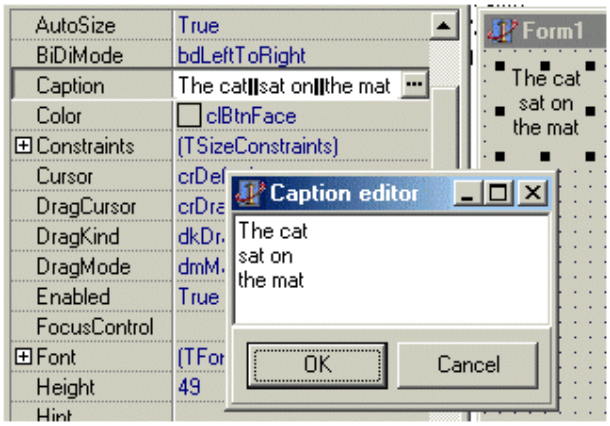
گام سوم:

فراخوانی RegisterComponents را از یونیت کامپوننت حذف کنید و در عوض آنرا در یونیت ویرایشگر رجیستر کنید.
حالا خصیصه از نوع TExpandingRecord بصورت یک خصیصه بازشدنی نشان داده میشود (چون ما paSubProperties را از GetAttributes برمیگردانیم)، و ویرایشگر پیش‌فرض TCollection کار خواهد کرد چون مالک خصیصه TCollection یک TComponent است.

۱۴– ویرایشگرهای خصیصه دیالوگ؛ ویرایشگرهای خصیصه پیشرفته

ویرایشگرهای خصیصه دیالوگ

اغلب هنگام ساخت یک ویرایشگر خصیصه اختصاصی، هدف فراهم آوردن ابزاری برای تعامل گرافیکی با خصیصه است.
مثال اول روشی بسیار ساده است به هدف امکان دادن به کاربر برای وارد کردن "Caption" چندخطی برای یک TLabel. با وجودیکه این مثال خیلی پیچیده نیست، به شما نشان میدهد که چطور فرمی را در ویرایشگرتان قرار دهید.



گام اول:

از منوی اصلی File|New Application را انتخاب کنید. این کار باعث ایجاد یک فرم میشود که نام آنرا "frmLabelEdit" میگذاریم، یک TMemo با نام memoCaption و دو دکمه "Ok" و "Cancel" که خصیصه‌های ModalResult آنها به ترتیب mrOk و mrCancel هستند به فرم اضافه نمایید.

گام دوم:

به قسمت uses عبارتهای DsgnIntf و TypInfo را اضافه کنید.

گام سوم:

کد ویرایشگر کامپوننت زیر را به یونیت اضافه کنید.

```
TCaptionProperty = class(TStringProperty)
public
  function GetAttributes: TPropertyAttributes; override;
  procedure Edit; override;
end;
```

و ویرایشگر خصیصه را به این صورت رجیستر کنید:

```
procedure Register;

implementation
{$R *.DFM}

procedure Register;
begin
  RegisterPropertyEditor(TypeInfo(TCaption), TLabel,
    'Caption', TCaptionProperty);
end;
```

گام چهارم:

کد زیر را اضافه کنید تا بازرس شئ دکمه ویرایشی [...] را در کنار نام خصیصه نشان دهد.

```
function TCaptionProperty.GetAttributes: TPropertyAttributes;
begin
  Result := inherited GetAttributes + [paDialog];
end;
function TCaptionProperty.GetAttributes: TPropertyAttributes;
begin
  Result := inherited GetAttributes + [paDialog];
end;
```

گام پنجم:

نهایتاً از فرم ویرایشگر یک نمونه ایجاد میکنیم، محتویات Memo را برابر عنوان موجود قرار میدهیم، و بعد فرم را به شکل modal نشان میدهیم.

```
procedure TCaptionProperty.Edit;
var
  I: Integer;
begin
  with TfmLabelEdit.Create(Application) do
  try
    memCaption.Lines.Text := GetStrValue;
    ShowModal;

{If the modal result of the form is mrOK, we
need to set the "Caption" property of each TLabel.}

    if ModalResult = mrOK then
      for I:=0 to PropCount-1 do
        TLabel(GetComponent(I)).Caption := memCaption.Lines.Text;
  finally
```

```
    Free;
  end;
end;
```

گام ششم:

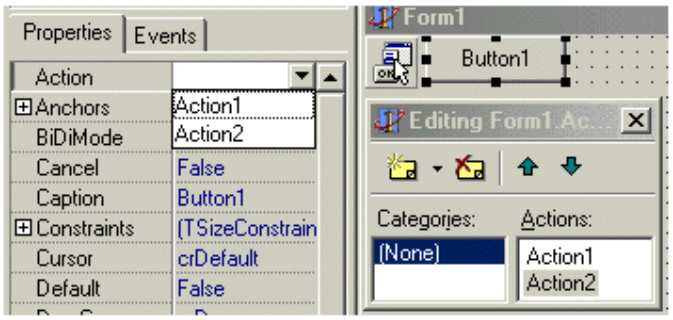
یونیت را در بسته نصب و بعد آنرا امتحان کنید!

ویرایشگرهای خصیصه پیشرفته

اگر با TActionList یا TDataSet(TTable / TQuery) کار کرده باشید مثال بعدی برایتان آشناست.
ویرایشگر ActionList یک ویرایشگر ویژه است که امکان گروهبندی actionها را میدهد، در مقابل FieldEditor در TDataSet در ابتدا ممکن است شبیه یک ویرایشگر استاندارد به نظر برسد، ولی اگر دقیقتر نگاه کنیم دارای منوی popup با آیتمهایی مثل "Add fields" میباشد. علیرغم این، جالبترین ویژگی هر دوی این ویرایشگرها در این نیست که ویرایشگرهای دیالوگ اختصاصی هستند (شبیه مثال قبلی)، بلکه این واقعیت است که آیتمهایی که ایجاد میکنند در اعلان کلاس اصلی یونیت جاری قرار میگیرد.

```
type
  TForm1 = class(TForm)
    ActionList1: TActionList;
    Action1: TAction;
    Action2: TAction;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

مزیت این مورد در این است که IDE از وجود این آیتمها آگاه گردیده، بنابراین میتوان وقتی خصیصه کامپوننتی به آنها نیاز دارد، آنها را از لیست اشیا انتخاب کرد.



در مثال بالا، دو action به یک TActionList اضافه شده‌اند، کلیک کردن خصیصه "Action" در Button1 لیستی از actionهای اضافه شده را نشان میدهد. همچنین دو action به اعلان کلاس فرم اضافه شده‌اند، و لذا میتوان با نام (Action1, Action2) به آنها مراجعه کرد.

لم کار در اینجا کاملاً در ویرایشگر خصیصه قرار دارد و نه در کامپوننت. وقتی یک ویرایشگر خصیصه فعال میشود (یعنی متد Edit فراخوانده میشود) خصیصه Designer دربردارنده یک ارجاع معتبر به یک IFormDesigner است. البته بسیاری از توابع این اینترفیس در حوزه این مقاله نیستند.
بعضی از متدها اینها هستند:

```
function MethodExists(const Name: string): Boolean;
procedure RenameMethod(const CurName, NewName: string);
procedure SelectComponent(Instance: TPersistent);
procedure ShowMethod(const Name: string);
function GetComponent(const Name: string): TComponent;
function CreateComponent(ComponentClass: TComponentClass;
  Parent: TComponent;
  Left, Top, Width, Height: Integer): TComponent;
```

بعضی از این متدها مقدماتی محسوب میشوند. مثلاً MethodExists برحسب اینکه آیا نام یک متد از قبل در یونیت جاری هست یا (مثلFormCreate, Button1Click و غیره) مقدار درست یا نادرست برمیگرداند. ShowMethod نشانگر (curser) را به متدنامگذاری شده میبرد، و RenameNethod نام یک متد را تغییر میدهد.
دو متدی که کاربردشان در اینجا جالب توجه است عبارتند از:

CreateComponent

با دادن کلاس کامپوننت، یک والد برای نگهداری آن، و مکان و اندازه، نمونه‌ای از کلاس را به شکلی که انگار برنامه‌نویس کامپوننت را از پلت انتخاب کرده باشد و خودش آنرا به فرم اضافه نموده باشد ایجاد میکند.

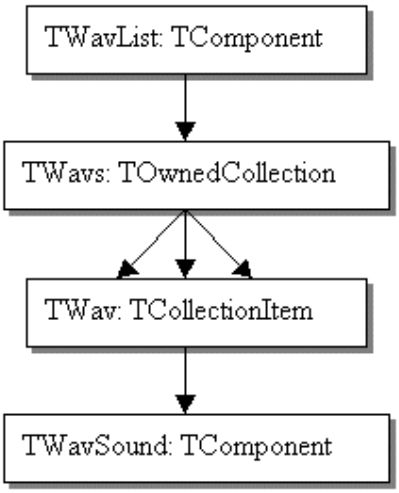
Modified

به طراح اطلاع میدهد که چیزی تغییر کرده است (خصیصه و ...). این حالت وضعیت یونیت را تغییر میدهد طوری که IDE میداند باید قبل از بسته شدن، یونیت ذخیره گردد (دکمه save در IDE را هم فعال میکند).
وقتی آیتمها را به آرایه‌مان اضافه میکنیم کاری که باید انجام دهیم این است TMyProperty.Designer را بگیریم تا از جانب خودمان یک کامپوننت ایجاد نماییم. بعداً این کامپوننت به فرم اضافه میشود و هر خصیصه‌ای که به کلاسی از از این نوع ارجاع داشته باشد بطور خودکار از این اتفاق آگاه میشود. در مورد TActionList و TDataSet کامپوننتهایی که به فرم اضافه میشوند در زمان طراحی قابل مشاهده نیستند، کامپوننت مالک بعنوان نوعی "مدیر (manager)" برای این کامپوننتها عمل میکند.
شما در زمان طراحی یک کامپوننت TAction یا TField روی پلت کامپوننتها نمیبینید طوری که ممکن است شک کنید آنها رجیستر نشده‌اند، ولی IDE میتواند نمونه‌هایی از این کامپوننتها ایجاد کند (و البته آنها غیرقابل مشاهده هم هستند). این کامپوننتها رجیستر شده‌اند، مساله در اینجا چگونگی رجیستر شدن است.
در مثال بعدی ما کامپوننتی به نام TWavSound میسازیم. TWavSound داده‌های یک فایل WAV را نگهداری میکند، و در صورت نیاز آنرا پخش مینماید. با وجودی که میتوان برای هر صوت WAV یک TWavSound روی فرم گذاشت ولی با این کار بعد از مدتی فرم شلوغ میشود و مدیریت آن مشکل. بنابراین ما یک اداره کننده به نام TWavList هم میسازیم.

توجه: در انتهای یونیت در قسمت initialization باید کد زیر اضافه شود:

```
initialization RegisterClass(TWavSound);
```

دلیلش این است که ظاهراً RegisterNoIcon کارش را کامل انجام نمیدهد. با وجودیکه به ما اجازه ایجاد نمونه‌های کامپوننت رجیستر شده از ویرایشگر خصیصه را میدهد ولی ظاهراً در هنگام بارگذاری مجدد پروژه‌ای که این کامپوننتها را داشته باشد، مشکل پیش می‌آید. پیغام "Class not registered" نمایش داده شده و پروژه خراب میشود. رجیستر کردن کلاس به این روش مشکل را حل میکند.



TWavSound

```
type
  PWavData = ^TWavData;
  TWavData = packed record
    Size: Longint;
    Data: array[0..0] of byte;
  end;

  TWavSound = class(TComponent)
  private
    FWavData: PWavData;
    FWav: TWav;
    procedure ReadWavData(Stream: TStream);
    procedure WriteWavData(Stream: TStream);
  protected
    procedure DefineProperties(Filer: TFiler); override;
  public
    destructor Destroy; override;
    procedure Clear;
    procedure LoadFromFile(const Filename: TFilename);
    procedure LoadFromStream(Stream: TStream);
    procedure Play;
  published
  end;
```

FWavData

برای ذخیره کردن محتویات فایل WAV وقتی که از stream یا فایل لود شود بکار میرود.

Clear

حافظه نگهدارنده FWavData را آزاد میکند.

Play

از تابع sndPlaySound API در MMSystem.pas برای پخش داده موجود در FWavData.Data استفاده میکند.

ReadWavData و WriteWavData

توسط IDE وقتی میخواهد داده‌ها را از FWavData بخواند یا در آن بنویسد، بکار برده میشود.

DefineProperties

یک خصیصه "مخفی" به نام WavData مشخص میکند و به IDE میگوید که برای stream کردن داده‌ها باید از ReadWavData و WriteWavData استفاده شود.

FWav

وقتی TWav.WavSound به کامپوننت ما مقداردهی میشود، این خصیصه بطور درونی توسط کلاس TWav مقدار میگیرد. دلیل این است که این آیتم Collection باید وقتی کامپوننت TWavSound آزاد میشود، آزاد شود تا از اشاره کردن به یک شئ نامعتبر اجتناب شود.

TWavSound

```
type
  TWav = class(TCollectionItem)
  private
    FWavSound: TWavSound;
    procedure SetWavSound(const Value: TWavSound);
  protected
  public
    procedure Play;
  published
    property WavSound: TWavSound read FWavSound write SetWavSound;
  end;
```

SetWavSound

این تضمین را ایجاد میکند که WavSoundی که به آن اشاره میکند خصیصه FWav خودش را درست مقداردهی کرده.

TWavSound

TWavs

یک پیاده‌سازی استاندارد از TCollection است.

TWavList

یک کامپوننت است که یک خصیصه TWavs را publish میکند تا به ما امکان دهد لیست wavها را در زمان طراحی ویرایش کنیم.

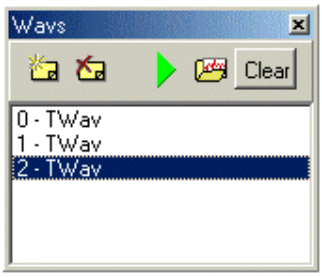
TWavsProperty

ویرایشگر خصیصه‌ای است که برای اداره این کلاس طراحی شده. این ویرایشگر امکان پخش و حذف wavها را در زمان طراحی دارد.

در ابتدا یک یونیت جدید با یک فرم ایجاد میکنیم و چند TSpeedButton و یک TListBox به آن اضافه مینماییم.

همچنین این آیتمها هم به قسمت اعلان کلاس فرم اضافه شده‌اند.

```
FWavs: TWavs;
 FComponent: TComponent;
 TheDesigner: IFormDesigner;
```



FWavs

ارجاعی را به TCollectionِی که ویرایش میکنیم نگه میدارد.

FComponent

ارجاع به کامپوننتی را نگه میدارد که مالک collection است. چون فرم ما به شکل modal نشان داده نمیشود باید در صورتی که این کامپوننت نابود شد فرممان را ببندیم (با کمک متد Notification فرممان).

TheDesigner

ما یک ارجاع ا به شئ Designer فعلی که به ویرایشگر خصیصه ما ارسال گردیده نگهداری میکنیم. این ارجاع برای فراخوانی CreateComponent و برای انتخاب TWavSound مخفی در بازرس شئ در هنگامی که آیتمی در لیست باکس انتخاب شده باشد بکار خواهد رفت.

در واقع ویرایشگر خصیصه بسیار ساده است.

```
type
  TWavsProperty = class(TClassProperty)
  public
    function GetAttributes: TPropertyAttributes; override;
    function GetDisplayName: string;
    procedure Edit; override;
  end;
```

تنها متدی که باید درباره‌اش توضیح دهیم متد Edit است. پیاده‌سازی Edit به شکل زیر است:

```
procedure TWavsProperty.Edit;
begin
  if fmWavsEditor = nil then
    fmWavsEditor := TfmWavsEditor.Create(Application);

  with fmWavsEditor do begin
    TheDesigner := Self.Designer;  //Don't forget SELF !!
    Caption := Self.GetName;

    //Setup the display, and then show the form
    Edit(TComponent(GetComponent(0)), TWavs(GetOrdValue));
  end;
end;
procedure TWavsProperty.Edit;
begin
  if fmWavsEditor = nil then
    fmWavsEditor := TfmWavsEditor.Create(Application);

  with fmWavsEditor do begin
    TheDesigner := Self.Designer;  //Don't forget SELF !!
    Caption := Self.GetName;

    //Setup the display, and then show the form
    Edit(TComponent(GetComponent(0)), TWavs(GetOrdValue));
  end;
end;
```

در ابتدا درصورتی که فرم ویرایشگر قبلاً ایجاد نشده باشد، ایجاد میگردد.

"TheDesigner" فرم مقدار Self.Designer میگیرد. فراموش نکنید که در اینجا "self" بعنوان TForm هم یک خصیصه Designer دارد که برابر nil خواهد بود.

GetComponent(0) برای گرفتن کامپوننتی که مالک خصیصه است بکار میرود. برای این کامپوننت FreeNotification فراخوانده میشود تا تضمین کند که اگر کامپوننت نابود شده باشد فرم ما مطلع میگردد (بطوریکه ما میتوانیم فرممان را ببندیم).

GetOrdValue برای گرفتن شئ کلاس (خصیصه "Wavs") که باید ویرایش شود بکار میرود، نتیجه بصورت TWavs، Typecast میشود.

متد Editی که فراخوانده میشود بخشی از TfrmWavsEditor است، و متدی است که لیست باکس را پاک میکند و آیتمها را بر اساس اسامی درایه‌های FWavs در لیست باکس قرار میدهد و سپس فرم را نمایش میدهد.

توجه: نسخه‌های اخیر دلفی نوع TPersistent را از تابع GetComponent برمیگردانند، بنابراین نتیجه باید به TComponent، Typecast شود.

ارتباط برقرار کردن با IFormDesigner

دو بخش اصلی این ویرایشگر (بجز پاک کردن WAV و پخش WAV) جاهایی هستند که "TheDesigner" کار میکند.

در ابتدا، جایی که دکمه "New" کلیک میشود یک آیتم جدید بـه Collection اضـافه میشـود، یـک TWavSound بـه اعلان کلاس فرم ما اضافه میشود، و در نهایت TWavSound در بازرس شئ انتخاب میشود.

```
procedure TfmWavsEditor.sbNewClick(Sender: TObject);
var
  Wav: TWav;
  WavSound: TWavSound;
begin
  //Add an item to the collection
  Wav := FWavs.Add;

  //Ask TheDesigner to create a new TWavSound component for us
  WavSound := TWavSound(TheDesigner.CreateComponent(TWavSound,
    nil, 0, 0, 0, 0));

  //Set the Wav (CollectionItem) to point to our new TWavSound component
  Wav.WavSound := WavSound;

  //Select our new TSoundComponent into the object inspector
  //so that it may be renamed if so desired
  TheDesigner.SelectComponent(WavSound);

  //Internally refresh the items in the listbox
  RefreshList;
  lbItems.ItemIndex := FWavs.Count-1;

  //Tell the IDE that something has changed
  TheDesigner.Modified;
end;
```

بخش دوم جایی است که وقتی در لیست باکس کلیک میشود، TWavSound صحیح در بازرس شئ انتخاب شده است.

```
procedure TfmWavsEditor.lbItemsClick(Sender: TObject);
begin
  with lbItems do
    if ItemIndex >=0 then
      TheDesigner.SelectComponent(FWavs[ItemIndex].WavSound);
end;
```

پرهیز از نقایص دسترسی[۲۳]

نهایتاً باید مطمئن شویم که ارجاع به شیئی که دیگر معتبر نیست باقی نگذاشته باشیم. این مورد با پیروی از گامهـای زیـر بـه سادگی ممکن است:

۱. مطمئن شوید که فرم ما در هنگام نابودی کامپوننتی که مالک کلاس خصیصه ماست مطلع میشود.
۲. متد Notification فرم را override کنید و اگر کامپوننت مربوطه نابود شد فرم را ببندید.

برای اطمینان پیدا کردن از اطلاع از نابودی کامپوننت:

[23] Access violations

```
procedure TfmWavsEditor.Edit(AComponent: TComponent; AWavs: TWavs);
begin
  //First we need to remove notification for the current component
  if FComponent <> nil then
    FComponent.RemoveFreeNotification(Self);

  //Now we need to add notification for the current component
  AComponent.FreeNotification(Self);
  FComponent := AComponent;

  FWavs := AWavs;
  lbItems.ItemIndex := -1;
  RefreshList;

  Show;
end;
```

کاری که در هنگام نابودی کامپوننت باید انجام شود:

```
procedure TfmWavsEditor.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  inherited;
  if Operation = opRemove then begin
    //If the owner component is destroyed
    //we should close our form
    if (AComponent = FComponent) then
      Close
    else
    //If the component that is destroyed
    //we refresh our list just incase it affects our component
    if (AComponent is TWavSound) then
      RefreshList;
  end;
end;
```

توجه: تکنیکهای بکار رفته در این مقالات در ساخت کامپوننت و ویرایشگرهای خاص در کامپوننتهایی با نـام DIB(Device Independent Bitmap) کـــه بوســـیله نویســـنده مقالـــه نوشـــته شـــده‌انـــد، در آدرس http://delphi.about.com/gi/dynamic/offsite.htm?site=http://www.howtodothings.com قابل دریافت است.