

OOP in Delphi

البرمجة غرضية التوجّه في دلفي

الفهرس ..

٣	مقدمة
		object-oriented in Delphi غرضية التوجه في دلفي
٤	الأصناف والأغراض
٦	تعريف الأصناف في دلفي
٨	التحميل الزائد للمناهج method overloading

نظرة أكثر تفصيلا في غرضية التوجه

٨	التغليف Encapsulation
١١	محددات الوصول Private, Protected, Public
١١	التغليف باستخدام الخصائص Properties
١٢	تعريف خاصية جديدة
١٤	مثال عملي
٢١	ملاحظات حول الخصائص
		المزيد عن التغليف
٢٢	التغليف مع الأشكال
٢٣	الباني Constructor
٢٤	بناء باني جديد
٢٥	الهادم Destructor والمنهج Free
٢٦	نموذج مرجعية أغراض دلفي
٢٦	نسب الأغراض
٢٧	الأغراض والذاكرة
٣٠	تحرير الأغراض مرة واحدة فقط
٣١	الوراثة من أنماط موجودة
٣٢	التغليف والحقول المحمية Protected Fields and Encapsulation
٣٣	دخول بيانات محمية لصنف آخر Protected Hack
٣٤	تطبيق عملي
٣٥	الوراثة والتوافق بين الأنماط
٣٦	التحديد المتأخر وتعددية الأشكال
٣٨	إعادة تعريف المناهج والمناهج المهمة
٣٩	المناهج الافتراضية مقابل الديناميكية Virtual versus Dynamic Methods
٤٠	المعاملان IS و AS
٤٢	إستخدام الواجهات Using Interfaces

تعتمد "بيئة التطوير دلفي" على البرمجة غرضية التوجُّه . دلفي هي توسيع للغة البرمجة باسكال ، والتي عرفت بإسم باسكال الغرضية (Object Pascal) .

عزمت بورلاند على الإشارة للغة بالإسم ("اللغة دلفي") أو ("The Delphi language") وإعتبارها لغة مستقلة بدلا من تسميتها بيئة التطوير دلفي ("Delphi development environment") .

ربما يكون ذلك لأن بورلاند تريد أن تصبح قادرة على القول يان كيلكس (Kylix) تستخدم اللغة دلفي ، كما أنها وفرت دلفي للعمل تحت منصة مايكروسوفت دوت_نيت (Microsoft .NET platform) ، وربما يسعنا القول أن دلفي معروفة كلغة برمجة مستقلة في صف اللغات الأكثر شيوعا في العالم وليست مجرد بيئة تطوير تعتمد باسكال الغرضية .

على كل حال سواء كانت "بيئة التطوير دلفي" أو "اللغة دلفي" ، فإن الروح هي نفسها لغة برمجة قوية ومستقرة تميزها إنتاجيتها الفريدة .

لا أظن أي ساقف طويلا أمام هذا الموضوع وسأستخدم التسمية التي تخرج معي أولاً .

غرضية التوجه في دلفي object-oriented in Delphi :

إن معظم اللغات الحديثة تدعم البرمجة غرضية التوجه (OOP) أو *object-oriented programming* ، حتى أن مقدار دعم اللغة للـ OOP أصبح ينظر إليه في كثير من الأحيان كمقياس يعبر عن أهمية اللغة .

تعتمد البرمجة غرضية التوجه على ثلاث مفاهيم أساسية سنعالجها في هذا الفصل :

- التغليف encapsulation .
- الوراثة inheritance .
- تعددية الأشكال polymorphism .

دلفي هي توسيع غرضي_التوجه للغة باسكال التقليدية . وما أريد التنويه إليه هنا أن الصياغة النحوية للغة الباسكال مشهورة بإنها صياغة أكثر وضوحا وقابلية للقراءة من معظم اللغات الأخرى (ولنقل مثلا لغة C) ، وبالتالي كمية أكبر من الحشو من أجل الحصول على شفرة مقروءة تشبه الكلام العادي بحيث يمكن فهمها وتذكرها بشكل سريع ومنتظم وهذا مما يقلل من الوقوع بالأخطاء .

كما أن التوسع الغرضي_التوجه لهذه اللغة بإعتراف الجميع لا يقل أهمية عن الموجود في النسل الحالي للغات البرمجة الغرضية من Java حتى C# .

الأصناف والأغراض :

بنيت دلفي أساساً وفق تصور البرمجة غرضية التوجه ، وبشكل خاص على تعريف أنماط لأصناف جديدة .
وعليك أن تعلم أن كل شيء في دلفي يتم التعامل معه وفق هذا المفهوم ، حتى لو كنت لاتشعر بذلك دائما . فإذا قمت
مثلاً بإنشاء شكل جديد (Form) ، فإن دلفي ستقوم أوتوماتيكيا بتعريف صنف جديد من أجله .
ومن المهم أن تفهم أن كل مكون تم وضعه على الشكل هو عبارة غرض ، وهذا الغرض خاص بصنف معين ...

توضيح :

المصطلحان غرض وصنف يستخدمان بكثرة ، وكثيرا ما يتم إساءة فهمهما ، أكاديميا نستطيع القول أن :
الصنف : هو نمط معطيات معرف يملك بعض التوصيفات ، وبعض العمليات " التصرفات " (أو ما يسمى مناهج) .
الغرض : هو منتسخ من الصنف ، (أو متغير من نمط معطيات صنف) . بحيث يملك قيم لكل التوصيفات التي يحددها
الصنف ، ويمكن إستعمال عملياته ويخضع لتصرفاته وخواصه ،..

دعنا نضرب مثلا من الطبيعة على هذا الموضوع :

في حالة الإنسان مثلا ، إننا عندما نقول إنسان ((بشكل عام)) فنحن ندل على الصنف إنسان . أي الإنسان هو الصنف ،
وأصبحنا نعرف بعض التوصيفات (الخصائص) المتعلقة به ، مثلا أن له إسم يميزه و طول ووزن ولون عيون وأنة يقوم
ببعض السلوكيات (المناهج) مثل الأكل والشرب والركض والكلام ، تذكر أنة عندما نتحدث عن الصنف إنسان فنحن
لأنحدد أي قيمة ثابتة لأي من خصائصه مثلا لايجوز القول أن طول الإنسان هو ١٧٥,٣ متر دوما أو حتما
إذا كان الإنسان هو الصنف فما هو الغرض إذا ، الغرض يجب أن يكون حالة من حالات الصنف ، حالة واحدة محددة
جيذا وتعطي الأجوبة والقيم الدقيقة لكل الصفات المتعلقة بالصنف التابعة له ،
الغرض في هذه الحالة هو إنسان ما ، أي إنسان محدد ، ولنقل عروة عيسى ،
الغرض عروة من الصنف إنسان يملك جميع خواص الإنسان ويأخذ قيم محددة لها ، الإسم عروة ، الطول ١٨٢,٣ متر ،
الوزن ٨٥ كغ ، لون العينين بني ... الخ ... ويستطيع تنفيذ كل من مناهجة (سلوكياتة) متى شاء من الركض والأكل
والكلام الخ ..

وبالتالي للصنف إنسان عدد كبير من الأغراض مثل عروة وكمال وعلاء وعصام وعلي الخ.... وكل منهم يملك صفات
الصنف الأساسية من أجل قيم محددة خاصة به هو

أتمنى أن يكون الأمر أصبح واضحا الآن .

- كثيراً ما يشار إلى الأغراض بالكيانات (entities) .

العلاقة بين الغرض (object) و الصنف (class) هي نفسها العلاقة بين المتحول (variable) والنمط (type) .

مثلا هي العلاقة بين Integer والمتحول A حيث A : integer ;

دعنا نوضح بعض المسائل المهمة هنا :

إن متحول من نمط صنف معين في دلفي كما هو الحال في معظم لغات البرمجة الغرضية التوجه لن يمثل تخزينا للغرض في الذاكرة ضمن هذا المتحول ، ولكنة فقط عبارة عن مؤشر إلى الغرض في الذاكرة .

وبناء على ذلك فإننا لا نستطيع التعامل مباشرة مع الأغراض كما نتعامل مع المتحولات العادية ، لأن الأغراض لن تخزن في ذاكرة المتحول المعرف لها كما يحدث في حالة متحول عادي ، بل هي بحاجة إلى حجز يدوي لمساحة ذاكرة خاصة بها ومن ثم يصبح المتحول الخاص بها مؤشر على هذه المساحة ، أي يخزن عنوان هذه المساحة فقط ، ولا يخزن بيانات الغرض ضمنه..

إذن وجدنا الآن فرقا مهما بين طريقة التعامل مع المتحولات العادية ومع الأغراض ، وقلنا أنه يجب حجز (إنشاء) ذاكرة للغرض قبل إستخدامه ،،، كيف يتم ذلك ؟

يتم ذلك بإحدى طريقتين :

١- إنشاء وحجز الذاكرة يدويا لغرض من صنف ما .. (باستخدام المنهج الباني Create)

٢- أو نسب الغرض إلى غرض موجود تم حجز الذاكرة له مسبقا .. (باستخدام النسب :=)

```
var
  Obj1, Obj2: TClass;
begin
  // assign a newly created object
  Obj1 := TMyClass.Create;
  // assign to an existing object
  Obj2 := Obj1;
```

في هذا المثال قمنا بتعريف غرضين Obj1, Obj2 من الصنف Tclass ، الغرض الأول obj1 قمنا بإنشاءه بالطريقة الأولى باستخدام المنهج المهم والشائع Create الذي يحجز الذاكرة له ويهيئها للإستخدام. والغرض الثاني obj2 قمنا بنسبة إلى غرض موجود مسبقاً وهو Obj1 ، وكلا الطريقتين صحيحتين ومستخدمتين .

مهم : بما أننا قمنا بحجز الذاكرة يدويا باستخدام المنهج Create ، فإنه يتوجب علينا تحريرها يدويا أيضاً ، ويتم ذلك باستخدام المنهج Free . سنلقي نظرة تفصيلية على هذين المنهجين في هذا البحث .

أنصحك باستخدام معالجة الإستثناءات (Try Finally) لتعريف الأصناف وتحريرها ، كذلك فإن معالجة الإستثناءات ستحدث عنها لاحقاً ، إن لم يكن لديك تصور واضح عنها بعد .

تعريف الأصناف في دلفي :

لتعريف صنف جديد في دلفي ، دعنا نتذكر أن الصنف يحوي شيئين مهمين هما الحقول (بيانات الصنف) والمناهج (عمليات الصنف) .

بناء صنف جديد موضوع سهل في دلفي .

إذا فكرنا بناءً على ما سبق ماذا نحتاج لنعرف صنف جديد ، لوحدنا أننا نريد تعريف الحقول التي يحويها هذا الصنف والعمليات التي يستطيع إنجازها ، وبالتأكيد نعرف له إسماً فريداً خاص به .

الحقول هي عبارة عن متحولات عادية ، والعمليات هي عبارة عن مناهج (أي توابع أو إجراءات) . تنسيق هذا التعريف يتم بصورة بسيطة – يان نذكر في قسم Type إسماً للصنف محدداً بالكلمة المفتاحية **Class** وتتبعه مباشرة بتعريف الحقول الخاصة به ، ثم رؤوس المناهج التي يعرفها ، وبالتأكيد ننهي ذلك بـ **End;** :

```
Type
TDate = class
    Month, Day, Year: Integer;
    procedure SetValue (m, d, y: Integer);
    function LeapYear: Boolean; // يحدد إذا كانت السنة كبيسة
end;
```

ملاحظة :

إن البادئة T التي سبقنا بها إسم المتحول هي عبارة عن تقليد (عرف) للمترجم ، يتبعه مبرمجو الدلفي منذ ظهورها بإيعاز من شركة بورلاند نفسها . الحرف T هو إختصار لـ Type ، وهو مجرد حرف ولكن إتباع هذا العرف سيجعل شفرتك مفهومة أكثر من قبل بقية المطورين الذين اعتادوا على ذلك .

ربما لاحظت من تعريف الصنف أننا قمنا بتعريف أسماء التوابع والإجراءات فقط (رؤوس المناهج) ولم نقم بكتابة أجسامها هناك ، حيث نقوم بتعريف أجسام المناهج (توابع+إجراءات) في جسم الوحدة نفسها ، أي قسم الـ Implementation الخاص بها .

وبما أننا نستطيع تعريف أكثر من صنف في الوحدة (Unit) ولكل صنف مناهج خاصة به لذلك يجب تمييز جسم كل منهج لنعرف لأي صنف يتبع . من أجل ذلك فإن تعريف أجسام المناهج يسبق بإسم الصنف مفصولاً بنقطة عن إسم المنهج مثلاً TDate.SetValue :

Implementation

...

```

procedure TDate.SetValue (m, d, y: Integer);
begin
    Month := m;
    Day := d;
    Year := y;
end;

function TDate.LeapYear: Boolean;
begin
    // call IsLeapYear in SysUtils.pas
    Result := IsLeapYear (Year);
end;

```

فكرة :

إذا ضغطت Ctrl+Shift+C عندما يكون المؤشر ضمن تعريف الصنف ، فإن ميزة تكميل التعريف في دلفي سوف تقوم تلقائياً بمساعدتك وتوليد هيكل التعريف الخاص بالمناهج التي قمت بتعريفها في الصنف .

عرفنا الآن كيف نبي صنف جديد ، وأنها نستطيع أن ننشيء أغراضاً من هذا الصنف وإستخدامها في شفرتنا .

مثال على ذلك :

```
var
  ADay: TDate;
begin
  // create an object
  ADay := TDate.Create;
  try
    // use the object
    ADay.SetValue (1, 1, 2000);
    if ADay.LeapYear then
      ShowMessage ('Leap year: ' + IntToStr (ADay.Year));
  finally
    // destroy the object
    ADay.Free;
  end;
end;
```

لاحظ أن التعبير ADay.LeapYear مشابه تماماً للتعبير ADay.Year . مع أن إحداهما هو تابع للتنفيذ ، والآخر هو متحول (وصول بيانات مباشر) ، أي نصل لبيانات الغرض بنفس طريقة الوصول لمناهجة وذلك باستخدام النقطة .
- بإمكانك اختيارياً أن تضيف قوسان مغلقان بعد استدعاء تابع ليس له بارامترات ، وبإمكانك تجاهلهما على كل حال

مثال : التعريفان التاليان متكافئان :

```
GetCurrentDay();
GetCurrentDay;
```

التحميل الزائد للمناهج method overloading :

دلفي تدعم التحميل الزائد للمناهج (التوابع + الإجراءات) . ولكن ما هو التحميل الزائد ياترى ؟

التحميل الزائد يعني أن يكون لديك منهجان بنفس الاسم ، شريطة أن تقوم بإضافة الكلمة المفتاحية overloading اليهما وأن تكون قائمة البارامترات (المتغيرات) الخاصة بكل منهما مختلفة عن الآخر ، (إذا إتفقا بالإسم فإن دلفي بحاجة إلى شيء آخر للتمييز بينهما لذلك فإن منهجين متفقين بالإسم والبارامترات لا يمكن أن نحدد أي منهما نريد أن نستخدم) . بفحص البارامترات يستطيع مترجم اللغة (Compiler) تحديد أي واحد منهما نريد أن نستدعي ، ويقوم بالإستدعاء الصحيح للمنهج الصحيح . سنرى تطبيقات هذه الميزة لاحقاً

نظرة أكثر تفصيلا في غرضية التوجه :

- التغليف
- الوراثة
- تعددية الأشكال

التغليف Encapsulation :

تعتمد فكرة غرضية التوجه على إخفاء البيانات . وتستخدم الأصناف لتحقيق ذلك .

يتم إخفاء البيانات داخل الأصناف الخاصة بها ، أو نقول يتم تغليف البيانات داخل الأصناف .

عادة يتم توضيح هذه الفكرة باستخدام ما يسمى الصناديق السوداء (black boxes) ، حيث لا تضطر أن تعرف كيف تتم الأمور بالداخل وما هي المحتويات الداخلية ، وكل ما يهملك هو كيف تتعامل مع واجهة الصندوق الأسود وتعطية معطياتك وتأخذ النتائج بغض النظر عن ما يتم في الداخل . إن ما يهملك فعليا من الصندوق هو آلية التعامل معه (مع واجهته) ولا تعطي إهتماما كبيرا عن تفاصيل داخل الصندوق ، مثلا يهملك أن تفرج على البرامج المفضله على التلفزيون وأن تعرف تغيير المحطات وإطفاء وتشغيلة ، بغض النظر عن فهم الدارات الداخلية المكونة للتلفزيون

..

إذن نخزن البيانات داخل الأصناف وعندها يمكننا أن نكتفي بمعرفة كيفية إستخدامها من الخارج . إن كيفية الإستخدام تدعى واجهة الصنف (class interface) وهي التي تسمح للأجزاء الأخرى من البرنامج بإستخدام الأغراض المعرفة من هذا الصنف ، وبالتالي عندما تستخدم غرض ما فإن معظم شفرته تكون مخفية ، ونادرا ما تعرف ما هي البيانات الداخليه له حتى أنه قد لا توجد طريقة لدخول البيانات الخاصة به بشكل مباشر ما لم تستخدم المناهج المتاحة على الواجهة والتي تسمح لك بتغيير وقراءة البيانات ، وذلك يعتبر من أهم الفروق بين البرمجة غرضية التوجه و البرمجة الكلاسيكية والتي تكون البيانات فيها عامة لكل الأصناف غير تابعة لصنف محدد كما أنك تستطيع تغييرها مباشرة وبالتالي تقع في مطب عدم صلاحية القيمة لحالة أو لمجموعة حالات ،،،...

لنوضح هذه النقطة الهامة :

في مثال التاريخ السابق ،

- لو كنا نتبع الطريقة الكلاسيكية بالبرمجة فإن المتحولات (البيانات) ستكون متاحة للدخول والتغيير المباشر ، وبالتالي من الممكن إدخال قيم غير صالحة بدون وجود إمكانية للتأكد منها ، مثلا لو قمت بإدخال التاريخ February 30 (٣٠ شباط) والذي هو تاريخ خاطيء لأن شباط لايجوي ٣٠ يوم فإن البرنامج سيقبله لأننا عرفنا متحول اليوم من النوع الصحيح (Integer) الذي يقبل هذه القيمة ، وستحصل الأخطاء لاحقا عند العمليات الحسابية ، أو تعطي نتائج خاطئة تماما .

- أما في حال البرمجة الغرضية ، فإن الدخول المباشر للبيانات غير مسموح لأن البيانات مغلقة (مخبأة) في الصنف والوصول إليها يتم باستخدام المناهج التي خصصها الصنف لذلك ، أي أنك لن تستخدم المتحول مباشرة بل ستعامل مع إجراءات أو تابع لإدخال القيمة ، وبالتالي لن يظهر معنا النوع السابق من الأخطاء لأن هذه المناهج يمكن بسهولة تضمينها شفرات لفحص القيم والتأكد منها ، ورفض التعديلات في حال كانت القيمة غير صالحة . لاحظ أننا استخدمنا المنهج SetValue لضبط القيم ولم نقم بالضبط المباشر وهنا نستطيع إضافة الشفرة الخاصة بالتحقق من صلاحية القيمة ، كأن تكون أصغر من حد معين ، أو غير سالبة ، أو ...

• كما أن للتغليف ميزة سحرية للمبرمج نفسه هذه المرة ..

لإنها تسمح لة بتغيير التركيب الداخلي للصنف في التحديثات المستقبلية ، وبالتالي ستطبق التغييرات تلقائيا على بقية الأغراض التي إستخدمت هذا الصنف بإقل عناء ممكن ، دون الحاجة لتغيير شفرتنا في مناطق مختلفة من البرنامج .

ملاحظة :

بالإضافة إلى التغليف المعتمد-على-الصنف فإن دلفي تدعم التغليف المعتمد-على-الوحدة ، بحيث كل متغير تقوم بتعريفه في قسم الـ Interface للوحدة سيصبح مرثيا لباقي وحدات البرنامج عند إستخدامها في التعريف Uses في حين أن المتغيرات المعرفة في قسم الـ Implementation هي متغيرات محلية لهذه الوحدة فقط .

محددات الوصول Private, Protected, Public :

دلفي تملك ثلاث محددات وصول من أجل التغليف المعتمد-على-الصف .

وهي Private, Protected, Public ، تتحكم محددات الوصول بمجال الرؤية المسموح به من أجل حقن أو منهج ما

- التوجيه Private يدل على حقوق الصف ومناهجه التي تكون غير متاحة خارج الوحدة التي عرف فيها الصف ، ولا يمكن الوصول إليها سوى من داخل هذه الوحدة ، وبالتالي هي متغيرات محلية في هذه الوحدة تستخدم لإتمام عمل جزئي ما داخل الصف ولا داعي لظهورها لبقية العناصر .
- التوجيه Protected يستخدم لتحديد مجال رؤيه مقيّد لحقوق الصف ومناهجه ، حيث يستطيع الصف الحالي والأصناف المورثة منة فقط الوصول إلى البيانات المعرفة ضمنه ، وببساطة يستطيع الصف الأساسي والأصناف المشتقة منه بالإضافة إلى أي شفرة في نفس الوحدة الدخول إلى البيانات المحمية بـ Protected فقط هؤلاء هم من يملكون سماحية الدخول . هذا التوجيه مثل سابقة من ناحية أنه محلي ضمن الوحدة ، ولكن نضيف هنا إمكانية الرؤية من قبل الأصناف الجزئية المشتقة التي ربما تحتاج هذه البيانات .
- التوجيه Public يدل على حقوق ومناهج يمكن الدخول إليها بحرية من أي جزء من البرنامج كما لو أنها معرفة بنفس الوحدة ، حيث لا توجد قيود في الدخول إلى البيانات المعرفة بهذا التوجيه .

تحذير :

محددات الوصول السابقة تقوم بتحديد إمكانية دخول شفرات من خارج الوحدة إلى الصف المعرف فيها ، وبالتالي إذا وجد صنفان في نفس الوحدة فلا توجد حماية لدخول أحدهما إلى حقوق المعرفة Private من الصف الآخر ..

التغليف باستخدام الخصائص Properties :

الخصائص تعتبر من أروع تقنيات البرمجة الغرضية ، وتمثل فكرة التغليف بشكلها الأمثل .

والخصائص بشكل عام هي أول ما تعلمنا التعامل معه في مرحلة المبتدئ ، وللتبسيط فإن كل ما تراه في ضابط الكائنات عبارة عن خصائص ، والفكرة هي أنك تتعامل مع إسم ، والذي يخفي عنك بشكل كامل تفاصيل التنفيذ ، وتصبح مهمتك الحالية كمستخدم للصف هي قراءة القيم منة أو كتابتها إليه ، أعجبتني تعريف أحد الكتاب عندما قال أن الخصائص هي حقول افتراضية (virtual fields) .

ربما لاحظت في الفقرة السابقة أننا يجب أن ندخل للبيانات في حالة البرمجة الغرضية عن طريق المناهج بدلا من الدخول المباشر ، وهذا يبدو شيئا مربكا قليلا ، خاصة أن منهج القراءة سيكون مختلف عن منهج الكتابة ، إذا بنينا حقول إفتراضية تستخدم هذه المناهج وتخفيها عنا فإننا سنكسب سهولة الدخول المباشر للبيانات وقوة التغليف . هذه الحقول الإفتراضية هي الخصائص ، وتعامل معها مثلما نتعامل مع الحقول العادية .

تأمل الشفرة البسيطة التالية :

```
Edit1.Text := Button1.Caption;
```

لاحظ أننا إستخدمنا الخاصية Text المتعلقة بالعرض Edit1 للكتابة فيها ، والخاصية Caption من الغرض Button1 للقراءة منها . وببساطة أصبحنا بمذة الفكرة الرائعة نضيع الوقت بالتفكير بدلا من إضاعة الوقت بكتابة الشفرة ، بالتأكيد توجد مناهج خاصة للكتابة إلى الخاصية Text وللقراءة منها ، لكن الخاصية Text أخفت هذه الإرباكات عنا وسمحت لنا بإستخدامها بغض النظر عن معرفتنا بشفرتها المخفية محققة بذلك تغليفا مثالياً .

تعريف خاصية جديدة :

الفرقه السابقة تكلمت عن مستخدم الصنف الذي يستطيع إستخدام الخواص بسهولة ، هذه الفقرة ستتكم عن باي الصنف الذي يؤمن هذه السهولة .

عرفنا الآن أن للخاصية إزدواجية بالتعامل .. مرة قراءة ، ومرة كتابة

وبناء على ذلك لتعريف خاصية ما نحن نحتاج لتعريف قابلية القراءة وقابلية الكتابة أيضا ، ويتم ذلك ببساطة عن طريق الكلمتين المفتاحيتين Read , Write كما أننا نستخدم الكلمة المحجوزة property لتعريف خاصية جديدة

أمثلة:

```
1- property Month: Integer read FMonth write FMonth;
2- property Month: Integer read FMonth write SetMonth;
3- property Month: Integer read GetMonth write SetMonth;
```

حيث Fmonth متغير معرف كـ Private ، و SetMonth إجرائية و GetMonth تابع معرفان ضمن الصنف .

الحالة الأولى : وهي أبسط الحالات ، أن نقوم بتعريف متحول ما Fmonth مثلا من أجل القراءة والكتابة ، بدون إستخدام أي منهج ، القراءة تتم منه والكتابة إلية ، وطبعاً لا يمكن هنا التأكد من صحة الإدخال ، أو إرفاق إدخال أو إخراج القيمة بحدث ما .

الحالة الثانية : قمنا بالقراءة من متحول (Fmonth) بشكل طبيعي مثل الحالة الأولى ، حيث أنه في كثير من الأحيان لا نحتاج التأكد من صحة الإخراج طالما كنا قد تأكدنا من صحة الإدخال منذ البداية .

أما الكتابة فتتم بإستخدام الإجرائية SetMonth ، وهنا نستطيع التأكد من صلاحية الأذخال أو إرفاق الإدخال بأحداث ما (إلغاء تعطيل خواص معينة بعد الإدخال مثلا) ، وبالطبع هذه الحالة مستخدمة كثيراً على عكس الحالة الأولى .

الحالة الثالثة : إستخدمنا التابع GetMonth للإدخال والإجرائية SetMonth للإخراج ، وهي الحالة العامة .

ملاحظة : قراءة الخاصية ستعيد قيمة واحدة منها ، وبالتالي من المثالي هنا إستخدام تابع (Function) للقراءة .

الكتابة لن تعيد قيم ولكنها ستدخل قيمة ضمن بارامترات المنهج ، لذلك نستخدم إجرائية (Procedure) للكتابة .

عادة تكون حقول البيانات ومناهج الدخول السابقة Private (ومن الممكن أن تكون Protected)

بينما تكون الخصائص Public .

وهذا يعطي درجة مثالية من التغليف، لأنك تستطيع تغيير بيانات الصنف أو مناهج القراءة والكتابة (والتي هي غير مرئية لمستخدم الصنف) دون أن يتأثر بها مستخدم الصنف ولن يضطر لتغيير شفرة لإنة يستخدم أسماء الخواص فقط و التي بقيت ثابتة ، في حين أن كل التغيرات في طريقة القراءة والكتابة لن تؤثر عليه ..

تذكر الصندوق الأسود ، المستخدم يملك إسم الخاصية ويتعامل معها ، طالما بقي إسم الخاصية ثابتا فإن عملة لن يتأثر بأي تغيير .

مثال عملي

الهدف : التدريب على بناء أصناف جديدة ، وتطبيق ما تعلمناه في الفقرات السابقة من إنشاء المناهج والخصائص والتعامل مع محددات الوصول .

إذن هدفنا هو بناء صنف جديد للتاريخ ،

ماذا نريد من الصنف ،، القراءة منة والكتابة آلية ، بالإضافة إلى إجراء بعض التحكمات والعمليات المفيدة . ولكي يملك المثال صفة مسألة بحيث يمكنك تجريب حلها لوحدهك ، سأقوم بتفصيل ذلك :

المطلوب : بناء صنف جديد بالإسم Tdate مع مراعاة الخواص التالية:

- إمكانية ضبط القيمة بطريقتين . أولا :عن طريق إدخال ثلاث قيم لليوم والشهر والسنة ، ثانيا: عن طريق إدخال قيمة واحدة من النمط TdateTime
- إمكانية معرفة إذا كانت السنة الحالية كبيسة أو لا ؟
- إمكانية إخراج التاريخ بشكل نصي String .
- وجود منهج زيادة يوم ، بحيث يزيد يوم للتاريخ المخزن كلما تم تنفيذة .
- وجود ثلاث خصائص Year , Month , Day يمكن التعامل معها (قراءة وكتابة إلى كل منها) .

كيف نقوم بذلك ؟

تذكر أنه لا يوجد شيء في البرمجة يتم عملة دفعة واحدة ، لذلك سنقوم ببناء الصنف خطوة خطوة وتعديل شفرتنا كل مرة كأننا نعمل على حواسيبنا الشخصية منذ البداية .

طريقي بالعمل هي الطريقة التراجعية بحيث ننتقل من الآخر حتى نصل إلى البداية ، مثلا دعنا نفكر كيف سيصبح الصنف الجديد وما هي خصائصه ومناهجه قبل البدء بالعمل .

أولا علي أن أقوم ببناء صنف جديد بالإسم Tdate ، وقد تعلمنا ذلك سابقا وهو سهل :

```
type
```

```
TDate = class
```

- منهج "ضبط القيمة" لن يرجع أي قيم بل سنمرر لة القيم على شكل بارامترات ، لذلك من الأنسب أن يكون إجرائية Procedure وليس تابعا Function ،

نريد

نريد إجرائيتين مختلفتين لضبط القيمة ، ولكن تذكر أنه بإمكاننا إستخدام نفس الإسم لكلا الإجرائيتين وذلك بالإستفادة من خاصية التحميل الزائد في دلفي (حيث سيكون الفرق بالبرامترات) ، وسأختار إسما مناسباً لهما وليكن " SetValue" .

فإذا سيبدو إستدعاء كل من الإجرائيتين بعد الإنتهاء بالشكل :

```
TheDate.SetValue (2004, 7, 1); // الإجرائية الأولى
```

```
TheDate.SetValue(now); // الإجرائية الثانية
```

(*Now* يعيد قيمة التاريخ الحالي من النوع *TdateTime*)

- نريد أيضا منهج ليعرف إذا كانت السنة كبيسة أو لا ، من الملاحظ أنه سيعيد قيمة بوليانية واحدة ، لذلك من المناسب أن يكون تابعا Function وليس إجرائية Procedure وليكن إسمة " LeapYear" ، وسيبدو شكله بعد الإنتهاء كالتالي :

```
If (TheDate. LeapYear) then showMessage('Leap Year');
```

منهج إخراج التاريخ بشكل نصي مشابه لحالة تابع السنة الكبيسة لأنه سيعيد قيمة نصيه واحدة ، وبالتالي سيكون تابعا وسيكون شكله بعد الإنتهاء كالتالي :

```
showmessage(TheDate.GetText);
```

- منهج زيادة اليوم ليس بحاجة أصلا لبارامترات ، لأن مقدار الزيادة محددة سلفا حيث سيزيد التابع يوم واحد فقط عند كل مرة يتم إستدعاءه فيها ، وليكن إسمة " Increase"

وإستخدامة سيكون بمنتهى السهولة :

```
TheDate.increase;
```

إن كل من الأحداث السابقة يجب أن يكون مرثيا من كل الوحدات ويستطيع المستخدم إستخدامة بشكل طبيعي، وبالتالي يجب أن يعرف تحت التوجية **Public** .

وما أصبحنا نعرفه الآن أننا نملك صنف Tdate لة المناهج العامة السابقة وبالتالي أصبح التعريف سهلا ، وحتى هذه المرحلة يمكننا أن نكتب التعريف كالتالي :

```
type
  TDate = class
public
  procedure SetValue (y, m, d: Integer); overload;
  procedure SetValue (NewDate: TDateTime); overload;
  function LeapYear: Boolean;
  function GetText: string;
  procedure Increase;
end;
```

الخصائص المطلوبة Year و Month و Day تحتاج إلى قراءة وكتابة بالتأكيد ، وسأختار هنا الحالة العامة وأضع تابع للقراءة وإجرائية للكتابة .

فإذا سمينا تابع القراءة GetYear وإجرائية الكتابة SetYear ستصبح التعاريف الثلاثة كالتالي :

```
property Year: Integer read GetYear write SetYear;
property Month: Integer read GetMonth write SetMonth;
property Day: Integer read GetDay write SetDay;
```

بالتأكيد بما أن الخواص يجب أن تكون ظاهرة للمستخدم ولبقية الوحدات فهي في قسم **Public** كذلك ،

والشيء المهم هنا هو أن التتابع والإجراءات الخاصة بالقراءة والكتابة التي استخدمتها الخواص السابقة مثل `GetYear`، `SetYear` هي مناهج محلية ولا يجب على المستخدم أن يراها ويستعملها لإنه يستعمل الخاصة الأساسية مباشرة، وبالتالي سنقوم بتعريفها بشكل محلي ضمن التوجيهية `private`، حيث لن تكون مرئية خارج هذه الوحدة.

بقي لدينا الآن شيء وحيد لم نأخذة بالحسبان وهو المتحول الذي سوف نخزن قيمة التاريخ فيه، لا تنسى أنك تضبط قيمة الصنف مرة واحدة ثم تستدعي المناهج السابقة للتاريخ المحفوظ ضمنه، وبالتالي نحن بحاجة لمتحول لحفظ التاريخ. وليكن هذا المتحول هو `fDate` من النمط `TDateTime`، وطبعاً لاحظت أنه يجب أن يعرف محلياً ضمن قسم `Private` كذلك.

أصبح الآن شكل التعريف النهائي الذي سنضعه في قسم الـ `Interface` كالتالي:

type

```
TDate = class
```

```
private
```

```
fDate: TDateTime;
```

```
procedure SetDay(const Value: Integer);
```

```
procedure SetMonth(const Value: Integer);
```

```
procedure SetYear(const Value: Integer);
```

```
function GetDay: Integer;
```

```
function GetMonth: Integer;
```

```
function GetYear: Integer;
```

```
public
```

```
procedure SetValue (y, m, d: Integer); overload;
```

```
procedure SetValue (NewDate: TDateTime); overload;
```

```
function LeapYear: Boolean;
```

```
function GetText: string;
```

```
procedure Increase;
```

```
property Year: Integer read GetYear write SetYear;
```

```
property Month: Integer read GetMonth write SetMonth;
```

```
property Day: Integer read GetDay write SetDay;
```

```
end;
```

وبقي علينا كتابة أجسام المناهج في قسم الـ `implementation`.

تذكير: لكتابة أجسام المناهج نضيف إسم الصنف مفصولا بنقطة عن إسم المنهج في الترويسة. مثال:

```
procedure TDate.SetValue (y, m, d: Integer);
```

```
begin
```

```
... // لاحظ إسم الصنف قبل إسم المنهج
```

```
end;
```

أما شفرة المناهج فهي بغاية السهولة ويمكن الإعتماد على بعض تعليمات التاريخ المعرفة في الوحدة `DateUtils`

<code>fDate := EncodeDate (y, m, d);</code>	يقوم بتحويل ثلاث متحولات صحيحة تدل على اليوم والشهر والسنة إلى متحول من النوع <code>TdateTime</code> .
<code>S:= DateToStr (fDate);</code>	يقوم بالتحويل من النمط <code>TdateTime</code> إلى النمط <code>String</code> ، (مشابهة لـ <code>InttoStr</code> مثلا)
<code>B:= IsInLeapYear(fDate);</code>	يعيد قيمة بوليانية إذا كانت السنة كبيسة أولا
<code>fDate := RecodeYear (fDate, Value);</code>	يضبط قيمة السنة في التاريخ عن طريق تمرير القيمة الجديدة للسنة بالمتحول <code>Value</code>
<code>Result := YearOf (fDate);</code>	يعيد قيمة السنة في متحول التاريخ <code>fDate</code>

تصبح شفرة الوحدة كاملة:

```
unit Dates;
```

```
interface
```

```
uses SysUtils;
```

```
type
```

```
  TDate = class
```

```
  private
```

```
    fDate: TdateTime;
```

```
procedure SetDay(const Value: Integer);
procedure SetMonth(const Value: Integer);
procedure SetYear(const Value: Integer);
function GetDay: Integer;
function GetMonth: Integer;
function GetYear: Integer;
public
  procedure SetValue (y, m, d: Integer); overload;
  procedure SetValue (NewDate: TDateTime); overload;
  function LeapYear: Boolean;
  function GetText: string;
  procedure Increase;
  property Year: Integer read GetYear write SetYear;
  property Month: Integer read GetMonth write SetMonth;
  property Day: Integer read GetDay write SetDay;
end;
implementation
uses
  DateUtils;
procedure TDate.SetValue (y, m, d: Integer);
begin
  fDate := EncodeDate (y, m, d);
end;
procedure TDate.SetValue(NewDate: TDateTime);
begin
  fDate := NewDate;
end;
function TDate.GetText: string;
begin
  Result := DateToStr (fDate);
end;
```

```
procedure TDate.Increase;
begin
    fDate := fDate + 1;
end;

function TDate.LeapYear: Boolean;
begin
    // from DateUtils
    Result := IsInLeapYear(fDate);
end;

procedure TDate.SetDay(const Value: Integer);
begin
    fDate := RecodeDay (fDate, Value);
end;

procedure TDate.SetMonth(const Value: Integer);
begin
    fDate := RecodeMonth (fDate, Value);
end;

procedure TDate.SetYear(const Value: Integer);
begin
    fDate := RecodeYear (fDate, Value);
end;

function TDate.GetDay: Integer;
begin
    Result := DayOf (fDate);
end;

function TDate.GetMonth: Integer;
begin
    Result := MonthOf (fDate);
```

```

end;

function TDate.GetYear: Integer;

begin
    Result := YearOf (fDate);
end;

end.

```

ملاحظات حول الخصائص :

يمكننا حذف الجزء الخاص بالكلمة المفتاحية Write من تعريف الخاصية لنحصل على خاصية للقراءة فقط (*read-only*) ، وعندها سيولد المترجم خطأ إذا حاول المستخدم تغيير قيمة خاصية للقراءة فقط .

نحوياً يمكنك حذف الجزء read والحصول على خاصية للكتابة فقط دون خطأ من المترجم ، ولكن ذلك لا يعتبر تفكيراً نظامياً وليس له إستخدامات حقيقية .

بيئة الدلفي تتعامل بطريقة خاصه مع خواص زمن-التصميم ، والتي تظهر في ضابط الكائنات أثناء التصميم . ويستخدم معها محدد الوصول published الذي ستعرف عنه في قسم بناء العناصر الجديدة في هذا الكتاب .

تسمى الخواص المعرفة بالمحدد Public خواص زمن-التنفيذ (design-time properties) ، وهي مخصصة للإستخدام ضمن شفرة البرنامج .

تنبيه : بعض التوجيهات الإضافية للخصائص (مثل stored و default) سنتعرض لها بالفصول القادمة .

ملاحظة : يمكن قراءة قيمة الخاصية ، أو الكتابة بها ، أو حتى إستخدامها ضمن التعابير المختلفة . ولكن ليس من الضروري أن يقبل تمريرها كمتغير إلى منهج ما ، لأنها ليست مواقع ذاكرة مثل المتحولات .

المزيد عن التغليف :

التغليف مع الأشكال

إحدى الأفكار المفتاحية للتغليف هي تقليل المتغيرات العامة المستخدمة من كامل البرنامج ، يمكن الدخول إلى المتغير العام من أي جزء من البرنامج ، لذلك فإن تغييرا في متحول عام سيؤثر على كامل البرنامج ، في حين أنك عندما تغير تمثيل أحد حقول صنف ما ، فإنك بحاجة لتغير شفرات بعض المناهج التي تستخدم فقط ، لذلك نستطيع القول أن إخفاء المعلومات يشير إلى تغليف المتغيرات .

ولتوضيح هذه الفكرة سأقترح مثلا ، لدينا برنامج له عدة أشكال (Multiple Forms) ، نستطيع أن نجعل بعض البيانات متاحة لكل الأشكال بتعريفها في قسم الواجهة (Interface) لإحدى الأشكال

```
var
  Form1: TForm1;

nClicks: Integer;
```

سيعمل البرنامج بشكل جيد بهذه الطريقة ، ولكن عليك أن تعرف بأن البيانات التي عرفناها هنا ليست تابعة للشكل الذي عرفناها فيه ، بل أصبحت تابعة لكامل البرنامج ، فإذا أنشأنا شكلان من نفس الصنف فإنهما سيتشاركان البيانات المعرفة في قسم الواجهة نفسها . مثلا لو قمنا بإنشاء عدة نسخ من الصنف *Tform3* في زمن التشغيل باستخدام الباني *Create* ، سيصبح لدينا عدة أشكال لنفس الصنف ، كيف نجعل بيانات ما خاصة بإحداها ومرئية من بقية البرنامج؟

فإذا أردت أن يكون لكل واحد منهما نسخة الخاصة من البيانات ، فعليك إضافة البيانات إلى صنف الفورم نسخة وبالتالي سينسخ كل شكل نسخة خاصة به .

```
type
  TForm1 = class(TForm)
  public
    nClicks: Integer;
  end;
```

ملاحظة : يستخدم ذلك بكثرة في تطبيقات MDI بحيث نحتاج متحولات خاصة بكل فورم ولا نضمن ما هو عدد الأشكال التي سينشئها المستخدم في زمن التصميم ، أو في تطبيقات الويب مثلا بحيث نحتاج جلسة لكل مستخدم موجود حاليا ، ولة إستخدامات كثيرة أخرى .

ملاحظة : الصنف السابق يستخدم البيانات على أنها عامة Public ومن أجل تغليف جيد عليك أن تجعل البيانات Private وأن تضيف لقسم Public مناهج خاصة للوصول إليها ، كما تعلمنا سابقا .

وببساطة دعنا نضيف خاصية جديدة إلى صنف الشكل ونريح بالنا أكثر ، بحيث كلمنا أحتجنا لتغيير أو قراءة البيانات من شكل آخر نستخدم الخاصية نفسها ، نعيد كتابة التعريف السابق ليصبح بالشكل :

```
type
TForm1 = class(TForm)
  private
    FClicks: Integer;
  public
    property nClicks: Integer read FClicks write SetClicks;
end;
```

راجع المثال المرفق مع الأمثلة ، ولاحظ أننا أنشأنا عدة أشكال من نفس الصنف ولكل واحد منها عداد نقرات خاص .

ملاحظة : خاصية زمن-التشغيل التي أضفناها لن تضاف إلى ضابط الكائنات ، بل يمكن إستخدامها داخل الشفرة فقط.

راجع أيضا : مثال المقارنة المرفق ، للتأكد من إتقان التعامل مع هذه الفكرة بشكل جيد ، وعدم الوقوع في مطباتها .

الباني Constructor :

الباني هو منهج خاص يستخدم لحجز الذاكرة لمتسخ من صنف ليصبح جاهزاً للإستخدام ، وكنا قد سمينا المنهج Create في ماضى ، الحقيقة التي أريدك أن تعرفها عن الباني أنه يعيد الغرض كقيمة جاهزة للإستخدام حيث نستطيع نسية إلى متحول موجود لكي نستخدمه لاحقا . أي كأنه تابع قيمة المعادة هي الغرض نفسه ، مثلا :

```
var B:Tbutton;

begin
B:=TButton.Create(Application);
```

لاحظ أن الباني يستخدم مع الصنف لكي يعيد الغرض ، مثلا هنا أستدعينا الباني من أجل الصنف Tbutton والنتيجة نضعها بالمتحول B ، ونستطيع إستخدام B لاحقا للتعامل مع الغرض الناتج كما نتعامل مع أي زر .

والشفرة الكاملة لذلك هي :

```
var B:Tbutton;

begin

b:=TButton.Create(Application);

b.Parent:=form1;

b.Left:=10;

b.Top:=10;

b.Caption:='Hi';

end;
```

بناء بائي جديد :

إن بيانات أي غرض جديد تكون مضبوطة للصفر ، فإذا أردت أن تبدأ هذه البيانات بقيم محددة عليك كتابة بائي جديد لها تحدد فيه القيم الافتراضية التي يبدأ الغرض بها ،

لبناء بائي جديد نستخدم الكلمة المفتاحية `constructor` في بداية تعريف منهجنا الجديد (بدلاً من `Function` أو `Procedure`) . في الحقيقة تستطيع أن تستخدم أي إسم لهذا البائي ولن يعطيك المترجم أي خطأ نحوي ، ولكن إذا أردت رأيي لا تستخدم أبداً إسم غير الإسم القياسي له ، والذي أصبحت تعرفه جيداً وهو "Create" . لأن أي مبرمج آخر لن يتعرف على البائي الخاص بك ويحسن إستخدامه ،، تذكر نحن هنا لتعلم كتابة برامج قياسية كما يكتبها الخبراء والمحترفون .

ملاحظة : عندما تعرف بائي جديد لصنف موجود أو لصنف مستق ، فإن البائي القديم لن يصبح متاحاً وسيصبح إستخدام البائي الجديد إلزامياً ، ولتغلب على هذه الحالة وإبقاء البائي الأساسي متاحاً بالإضافة إلى البائي الجديد عليك إستخدام ميزة التحميل الزائد لأسماء المناهج والتي سبق ذكرها . مثال :

```
type
TDate = class
public
    constructor Create; overload; // the old constructor
    constructor Create (y, m, d: Integer); overload; // the new constructor
```

حيث نستطيع ضبط قيمة التاريخ عند الإنشاء في الحالة الثانية ، أما الحالة الأولى ستأخذ قيمة التاريخ الافتراضي (أو الحالي مثلاً) .

تجربة :

إذا لم نستخدم التحميل الزائد فإن الباني الجديد سيحل تماماً محل الباني القديم ، ولن يصبح بالإمكان إستخدام الباني القديم ، مثلاً جرب الشفرة التالية ولاحظ الخطأ الناتج عند محاولة إستدعاء الباني الأصلي :

```

type
  TDate = class
  public
    constructor Create (y, m, d: Integer);
.....
var
  ADay: TDate;
begin
  // Error, does not compile:
  ADay := TDate.Create;
  // This one is OK:
  ADay := TDate.Create (1, 1, 2000);

```

ملاحظة مهمة : إن قواعد كتابة الباني من أجل عناصر جديدة تختلف قليلاً كما ستلاحظ في القسم الخاص ببناء عناصر جديدة . حيث ستلاحظ إستخدام الباني القديم ضمن الباني الجديد بعملية " override " .

الهادم Destructور والمنهج Free :

بنفس الطريقة ، بالإضافة إلى إمكانية كتابة باني خاص للصف ، يمكن كذلك كتابة هادم خاص له . ويقوم بالعمل المعاكس حيث يحرر أي ذاكرة قام الباني بحجزها ولم تحرر فيما بعد . ويبدأ بالكلمة المفتاحية destructor و يكون إسمه Destroy .

سترى لاحقاً إمكانية إستخدام الهادم القديم ضمن الجديد من أجل أن يقوم الصف ببعض عمليات التنظيف دون أن نشغل بالنا فيها من جديد .

لا تستخدم أبداً إسم للهادم غير الإسم Destroy ، لإن الأغراض يتم تحريرها عادة بالمنهج الشهير Free ومبدأ هذا المنهج (Free): هو إختبار إذا كانت قيمة الغرض Nil قبل أن يستدعي المنهج الهادم Destroy ، وبالتالي فإن Free لن يتعرف على الهادم الجديد الذي قمنا ببناءه إذا لم يكن إسمه Destroy وستنتج أخطاء .

إن Free هي منهج للصف الأب Tobject والذي ترثه كل الأغراض الأخرى .

والجدير بالذكر هنا أن Free لن تقوم بضبط قيمة الغرض إلى Nil بعد تحرير الذاكرة الخاصة به. لذلك عليك أن تقوم بذلك بنفسك والسبب بسيط أن الغرض لن يعرف ما هي المتحولات التي نسبت إليه ، ولن يستطيع التأكد من أنه ضبط قيمة جميع هذه المتحولات إلى Nil ، لذلك تركت العملية إلى المستخدم ، ولكن دلفي تملك بعض الدوال المساعدة هنا مثل الدالة FreeAndNil التي قد تساعد أحيانا . حيث أنها تحرر الغرض وتلغي المرجع الذي يشير إليه (يصبح المؤشر معرّفاً مثل أي متحول ولكنه لا يشير إلى أي قيمة بالذاكرة) . على كل حال تستطيع القيام بذلك يدويا باستخدام السطرين التاليين :

```
Obj1.Free;
Obj1 := nil;
```

نموذج مرجعية أغراض دلفي :

إن تعريف متغير من صنف ما ، في بعض اللغات الغرضية التوجه الأخرى، سينشيء منتسحا من هذا الصنف تلقائياً . ولكن دلفي مبنية على نموذج مرجعية الغرض بدلا من ذلك ، والفكرة في ذلك أن تعريف متغير في دلفي من نمط صنف ما لن يخزن الغرض داخله ، ولكنه يخزن مرجع لموقع الغرض في الذاكرة ، أي عبارة عن مؤشر (Pointer) يشير إلى موقع خاص بالذاكرة حيث يتم تخزين جسم الغرض هناك ، وليس ضمن المتغير نفسه ، وبناء على ذلك كما لاحظت معي في الصفحات السابقة إن تعريف متغير لا يعني إنشاء الغرض في الذاكرة (مما يربك مستخدم دلفي الجدد) ، ولكنك تكون حجزت موقع ذاكرة صغير يحوي عنوان موقع الذاكرة الآخر الذي يخزن الغرض ، . والأصناف التي نعرّفها يدويا تحتاج إلى إنشاء يدوي (المنهج Create) ، أما العناصر التي تضعها على الشكل (Form) ستقوم دلفي بإنشائها آليا .

وبالطبع تحتاج إلى تحرير الأغراض التي إنتهيت من إستخدامها ، لأنها أصبحت تشغل ذاكرة غير مفيدة .

طالما تقوم بإنشاء الأغراض وتحريرها عند الإنتهاء فإن نموذج مرجعية الغرض يعمل بدون أدنى مشكلة ويعتبر متماسكا جدا ، أما الخبر السعيد الآن أنه يملك أهمية خاصة وقدرة عالية على إدارة الذاكرة ولة الكثير من الفوائد ، وأليك بعضاً من ذلك :

نسب الأغراض :

بما أن المتغير الذي يحوي الغرض يدل فقط على العنوان الأساسي للغرض في الذاكرة ، فإننا نستطيع تعريف أكثر من متحول تدل جميعها على الغرض نفسه ، فإذا قمنا بنسب متحول جديد ما من نفس الصنف إلى آخر تمت تهيئة فإننا نستطيع التعامل مع المتحول الجديد دون أدنى مشكلة .

مثال :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NewDay: TDate;
begin
  NewDay := TDate.Create;
  TheDay := NewDay;
  Label1.Caption := TheDay.GetText;
end;
```

ماذا حصل ... ؟ يقوم هذا الكود بنسخ موقع ذاكرة الغرض الذي يشير إليه المتحول NewDay ووضعها في المتحول TheDay ، إن ذلك ليس نسخ بيانات غرض ما إلى غرض آخر ، إن هذه العملية سهلة وسريعة التنفيذ لأنها تنقل عناوين مواقع الذاكرة فقط .

ولكن عليك أن تحذر من ذلك ، وتحسن إستخدامه ، لاحظ أننا سنقوم بإنشاء الغرض من جديد كل مرة يتم الضغط على الزر button1 ولم نقم بتحريره بإستخدام Free مثلا ،

ولتجنب ذلك تستطيع ببساطة تحرير الغرض القديم قبل بناء الغرض الجديد :

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
begin
  TheDay.Free;
  TheDay := TDate.Create;
```

إن هذه الميزة تبدو ميزة عادية بداية ولكن عليك أن تعرف أن لها تطبيقات مهمة ، إن إمكانية تعريف متحول جديد يمكن أن يشير إلى غرض ما يجعل من الممكن إستخدام المتحول لتخزين الغرض الناتج من قراءة إحدى الخصائص ، وإستخدامه فيما بعد ، مثلا :

```
var
  ADay: TDate;
begin
  ADay := UserInformation.GetBirthDate;
  // use a ADay
```

إننا نكسب ديناميكية خاصة للتعامل مع الأغراض ونستطيع تشبيه ذلك بأي متحول عادي .

كما نستطيع تمرير الغرض بهذه الطريقة على شكل بارامتر خاص بتابع ما ، أي أننا نعامل الغرض كمتحول ونمرره إلى مناهج تقوم بمعالجة والتعديل فيه من مكانة .

مثلا نفرض إجرائية لها بارامتر وحيد من الصنف TButton ، نمرر لها زر ما فتقوم بتغيير إسمه مثلا أو أي بيانات يملكها :

```
procedure ChangeCaption (B: TButton);
begin
  B.Caption := B.Caption + ' was Modified';
end;
.....
// call...
ChangeCaption (Button1)
```

المتحول الذي قمنا بتمريره كزر أعطى عنوان الذاكرة للإجرائية التي دخلت إليه وقامت بالتعامل معه مباشرة ، . هذا يعني أن الغرض تم تمريره بالمرجع بلا استخدام الكلمة المفتاحية Var التي نستخدمها بالحالة العادية، وبدون أي من التقييدات الأخرى التي تفرضها حالة التمرير بالمرجع (pass-by-reference) .

لكن ماذا لو كنا نريد أن نقوم بنسخ البيانات فعليا من غرض إلى آخر ، هذه المرة لانريد التعامل مع متغيرات وعناوين ، لإننا نريد نسخ فعلي للغرض . ربما علينا أن نقوم بنسخ كل حقل من حقول الغرض ، ولكن ربما لا تكون كل الحقول معرفة Public أي لانستطيع الوصول إليها جميعها ، .

في هذه الحالة نستخدم منهج خاص يقوم بنسخ البيانات الداخلية كلها ، أصناف مكتبة العناصر المرئية في دلفي والتي تم توريثها من الصنف Tpersistent تملك المنهج Assign والذي يقوم بهذه العملية .

(ملاحظة هذا المنهج غير متوفر في جميع أصناف الـ VCL حتى المورثة من Tpersistent أحيانا) .

ملاحظة : نستطيع تزويد الأصناف التي نقوم بكتابتها بمنهج شبيهه بالمنهج Assign بسهولة من داخل شفرة الصنف . لإننا نستطيع الولوج إلى جميع بيانات الصنف مثلا لإضافة المنهج Assign إلى الصنف Tdate الذي قمنا ببناءه سابقا :

```
procedure TDate.Assign (Source: TDate);
begin
    fDate := Source.fDate;
end;
```

(لاحظ أن المتحول fDate الذي تخزن ضمنه قيمة التاريخ لا يكون متاحا خارج الوحدة لإنه معرف private)

وسيكون إستخدامه بالطريقة التالية مثلا :

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
var
    NewDay: TDate;
begin
    NewDay := TDate.Create;
    TheDay.Assign(NewDay);
    LabelDate.Caption := TheDay.GetText;
    NewDay.Free;
end;
```

الأغراض والذاكرة :

توجد ثلاث قواعد لإدارة الذاكرة في دلفي ، على الأقل لتكون واثقا أن النظام يعمل بتناغم من دون ظهور رسائل إنتهاك الذاكرة ، أو من دون ترك مساحات غير مستخدمة محجوزة دون تحريرها .

- كل غرض يجب أن يتم إنشائه قبل أن يتم إستخدامه .

- كل غرض يجب أن يتم تحريره بعد الإنتهاء من إستخدامه.

- كل غرض يجب أن يتم تحريره مرة واحدة فقط .

إذا كنت ستقوم بذلك يدويا ضمن شفرتك أو ستترك دلفي تقوم بذلك عوضا عنك ، فإن ذلك يعتمد على النموذج الذي سوف تعتمدة من بين نماذج إدارة الذاكرة التي تقدمها دلفي .

تدعم دلفي ثلاث أنواع من إدارة الذاكرة للعناصر الديناميكية :

- كلما قمت بإنشاء غرض يدويا ضمن شفرتك ، عليك تحريره يدويا أيضا ، وإذا لم تقم بذلك فإن الذاكرة التي يستخدمها لن تحرر كي تستفيد منها بقية عناصر تطبيقك حتى يتم إنهاء تنفيذ البرنامج .

- تستطيع تحديد عنصر مالك (owner component) للعناصر التي تقوم بإنشاءها ، بتمرير المالك إلى باي العنصر الجديد . ويصبح المالك مسؤولا عن تحرير ذاكرة كل العناصر التي يملكها ، بعبارة أخرى عند تحرير شكل (Form) فإن كل العناصر التي تتبع له سيتم تحريرها معه . وبالتالي في حالة العناصر (Components) عندما تقوم بتحديد عنصر مالك لعنصرك ، لاداعي لتذكر تحريره من الذاكرة. وهذا هو التصرف القياسي للعناصر التي قمنا بوضعها على الشكل Form في زمن التصميم ، حتى الشكل والذي يعتبر مالكا لمعظم عناصر التطبيق يكون مملوكا من قبل أغراض Application والتي تحرر آليا عند إنهاء التطبيق .

- عندما تقوم مكتبة RTL بتخصيص الذاكرة من أجل السلاسل والمصفوفات الديناميكية ، فإنها ستقوم آليا بتحرير الذاكرة عندما يخرج المرجع من مجال الرؤيا ، لن تحتاج لتحرير سلسلة محرفية ، عندما تصبح غير قابلة للوصول سيتم تحريرها .

تحرير الأغراض مرة واحدة فقط :

إذا قمت بإستدعاء المنهج Free أو الهادم Destroy أكثر من مرة ، فإن ذلك سيولد خطأ بلا شك . ولكن إذا ضبطت متحول الغرض إلى Nil فإنك تستطيع إستدعاء المنهج Free أكثر من مرة دون أخطاء .

ملاحظة : ربما تتساءل لماذا تستطيع إيمان أن تستدعي Free إذا كان مرجع الغرض Nil ، ولاتستطيع إستدعاء Destroy . السبب أن Free هي منهج معرف على موقع ذاكرة معطى . في حين أن الإستدعاء الإفتراضي Destroy يتم تحديده في زمن التشغيل بالنظر إلى صنف الغرض ، وهي تعليمة خطيرة جدا في حال أستخدمت ولم يكن الغرض موجودا .

ولتجميع الأمور بشكل جيد ، إليك هذه الخطوط العريضة :

- دائما إستخدم المنهج Free لتحرير الأغراض بدلا من الهادم Destroy .
- إستخدم الدالة FreeAndNil ، أو إضبط مرجع الغرض إلى Nil بعد إستدعاء المنهج Free .

تستطيع إختبار إذا كانت قيمة مرجع غرض ما Nil بإستخدام التابع Assigned .

العبارتان التاليتان متكافئتان في معظم الحالات :

```
if Assigned (ADate) then ...
if ADate <> nil then ...
```

تذكر أنه حتى لو كانت القيمة ليست Nil فهذا لا يعني أن المؤشر صالح للتعامل . مثال إن إستخدام المنهج Free سيحرر الغرض ولكنة لن يضبطة إلى Nil وبالتالي التعليمة التالية ستسبب خطأ .

```
ToDestroy.Free;
if ToDestroy <> nil then
  ToDestroy.DoSomething;
```

الوراثة من أنماط موجودة :

غالباً ما نحتاج لبناء نموذج مختلف قليلاً من صنف موجود ، بدلاً من بناء صنف جديد من البداية ، ربما نحتاج إضافة مناهج جديدة أو خصائص أو تعديل أخرى موجودة .

نستطيع فعل ذلك بطريقتين ، نسخ الشفرة من هناك ولصقها هنا . (يدل على ضعف خبرة بالبرمجة ما لم توجد غاية مبررة له) ، وبذلك ستضعف شفرتك مرتين ، ناهيك عن الأخطاء ، والغرق في تفاصيل تبعثك عن مشروعك الأساسي ببساطة ، لا تكن من الذين يتبعون هذا النوع من الحلول كحلول أساسية . لماذا لا تقوم بدلاً من ذلك باستخدام إحدى أروع ميزات البرمجة الغرضية : ألا وهي الوراثة (inheritance) .

وراثة صنف موجود عملية سهلة التحقيق باستخدام دلفي ، عليك فقط أن تذكر إسم الصنف الموروث في ترويسة تعريف الصنف الجديد . لاحظ أن شفرة مشروع جديد في دلفي تحوي التعريف التالي :

```
type
  TForm1 = class(TForm)
  end;
```

هذه هي الوراثة يا صديقي ، التعريف السابق يدل على أن TForm1 يرث كل صفات الصنف TForm ، الحقول ، المناهج ، الأحداث .. كل شيء ، تستطيع أن تستدعي أي منهج عام Public معرف في الصنف TForm من غرض من الصنف الوارث TForm1 ، والذي بدوره ورث بعض الصفات من صنف أب له حتى نصل إلى الصنف السلف TObject . بإمكانك إضافة بياناتك الخاصة للصنف الجديد ، أو تعديل البيانات الموروثة من الصنف الأب بإعادة تعريفها لكن في نفس الوحدة .

مثلاً إذا أردنا أن نبني صنف جديد مشتق من الصنف Tdate الذي سبق وبنينا ، ونعدل في المنهج GetText الخاص به

```
type
  TNewDate = class (TDate)
  public
    function GetText: string;
  end;
...
function TNewDate.GetText: string;
begin
  GetText := FormatDateTime ('dddddd', fDate);
end;
```

يكفي إعادة تعريف المناهج الموجودة بنفس الإسم حتى نحصل على نسختنا الخاصة منها، هذا العمل يوفر الجهد والتعب ، وستقوم دلفي بإستبدال التعريف القديم بالجديد وسيستخدم في كل مرة يتم إستدعاء فيها .

ملاحظة : في المثال السابق نضع تعريف TnewDate ضمن نفس الوحدة التي عرفنا بها Tdate لأن المنهج GetText يستخدم المتحول الخاص fDate المعروف كـ Private ضمن Tdate ولا يمكن الوصول إليه من خارج الوحدة .

ولتوضيح ذلك أكثر دعنا ننتقل للفقرة التالية :

التغليف والحقول المحمية Protected Fields and Encapsulation :

كما لاحظت أن شفرة المنهج GetText الخاصة بالصنف TnewDate سترجم بلا أخطاء فقط إن تمت إضافتها في نفس وحدة الصنف الأساس Tdate . لأنها كما وضحنا تحاول دخول المتحول fDate والذي هو متحول موضعي Private ، إذا أردنا أن نضع الصنف المشتق في وحدة جديدة بدون أن نجعل المتحول fDate متحول عام Public يمكن الوصول إليه من أي مكان ، فإننا سنجد طريقتين لتحقيق ذلك

- تعريف المتحول fDate كمتحول محمي أي (Protected) بدلا من المتحول العام أو المحلي ، إذا كنت تتذكر إن هذا النوع يسمح فقط للأصناف المشتقة من الصنف الأساسي بالدخول للبيانات .
- ترك المتحول Private وإضافة منهج محمي Protected يؤمن الوصول له .

ربما لاحظت معي أن الطريقة الأولى هي الأفضل لأنها أكثر عمومية عندما نورث الصنف لعدد كبير من الأصناف الفرعية ، وأنصحك بإتباعها دائما ، حتى لو لم تجد حاجة حاليا لتعريف معطيات محمية قم بذلك من أجل توريث أصناف جديدة مستقبلا ، البيانات المحمية تجعل الصنف قابل للوصول بشكل مناسب لتقنية الوراثة .

ربما تقول إن ذلك خروج عن قاعدة التغليف الكامل في البرمجة الغرضية (التي تقول بيانات محلية مناهج عامة) ، الجواب نعم إلى حد ما ، ولذلك علينا أن نكون متبهين أننا لن نحصل على كامل ميزات التغليف ما لم نتبعه بشكل جيد ، لاحظ مثلا في حالة قمنا بتوريث عشرات الأصناف من صنف ما ، إن تغيير البيانات المحمية Protected في هذا الصنف ربما يضطرنا لتغيير مايقابلها في كل من الأصناف المشتقة .

بكلمة أخرى المرونة ، قابلية التوسع ، التغليف ، غالبا ما تكون أهداف متنازعة ، ومن الصعب تحقيقها جميعا ، عندما يحصل ذلك عليك أن تفضل التغليف من بينها . إذا كان من الممكن تحقيق ذلك بدون التضحية بالمرونة فإن

ذلك سيكون ممتازا . غالبا ما يتم تحقيق هذا الحل الوسطي باستخدام ما يعرف بالمناهج الافتراضية والتي سنأتي على ذكرها قريبا تحت عنوان التغليف المتأخر وتعددية الأشكال . وربما إذا لم تختتر التغليف من هذه الحلول من أجل تحقيق سرعة في كتابة الشفرة مثلا ، فإنك عندها لن تكون متبع لقواعد البرمجة الغرضية بشكل جيد .

دخول بيانات محمية لصنف آخر Protected Hack :

محددات الوصول Private و Protected تسمح بالدخول إلى بياناتها من نفس الوحدة فقط ، الجدير بالذكر هنا أنه من الممكن الدوران على الموضوع في حالة المحدد Protected والدخول إلى البيانات المحمية الخاصة بصنف ما . الطريقة تعتمد على ما شرحناه سابقا ، أنه يمكن الوصول إلى البيانات المحمية لصنف من الأصناف المشتقة منه . لذلك نقوم بإشتقاق صنف جديد من الصنف الذي نريد دخول بياناته مثلا نشق الصنف TtestHack من الصنف Ttest ، وعلى إفتراض المتغير ProtectedData معرف كمتغير محمي ضمن الصنف Ttest كالتالي :

```
type
  Ttest = class
    protected
      ProtectedData: Integer;
    end;
```

فإننا نستطيع أن نستخدم الطريقة التالية لدخول المتغير ProtectedData :

```
type
  TtestHack = class (Ttest);

  ...

var
  Obj: Ttest;
begin
  Obj := Ttest.Create;
  TtestHack (Obj).ProtectedData := 20;
```

تطبيق عملي :

كلنا نتعامل مع العنصر الجميل DBNavigator ، ولكن هذا العنصر لا يجوي خاصية لضبط اللون ، إذا علمت أن المتحول Color متحول محمي للصف TDBNavigator إكتب شفرة تحويل اللون إلى الأحمر .

الحل :

أولا لضبط الخاصية Flat للنافيغيتور إلى True .

ثانيا أضف شفرة مشابهة للآتية :

```
type NewNav=class(TDBNavigator) ;

procedure TForm1.Button1Click(Sender: TObject);
begin
NewNav(DBNavigator1).Color:=clred;
end;
```

لابد من الإشارة أن هذه العملية ليست قياسية ، مع أنها تصلح ويمكن إستخدامها في كثير من الأحيان ، ولكن لماذا جعل كاتب الصنف الأساسي البيانات محمية لو أنه يريد مشاركتها ، لاحظ مثلا في المثال السابق أننا أضطررنا لضبط الخاصية Flat ولن تظهر التغيرات بلا ذلك ، الخلاصة يمكنك إستخدام هذه الطريقة لتحقيق غاية ما، ولكن أشدد عليك أن تتأكد من أنها الطريقة الأخيرة لذلك .

ملاحظة : يجب أن يكون تعريف الصنف المشتق و شفرة الدخول للصف الأب في نفس الوحدة . أي السطر

```
NewNav(DBNavigator1).Color:=clred;
```

والتعريف ; type NewNav=class(TDBNavigator)

يجب أن يكونا بنفس الوحدة ، وعند فصلهما في وحدات مختلفة فإن الشفرة السابقة لن تترجم .

الوراثة والتوافق بين الأنماط :

الباسكال لغة نموذجية ومثالية بشكل صارم ، وهذا يعني أنك لن تستطيع مثلا أن تنسب قيمة رقمية إلى قيمة نصية أو العكس بدون إجراء تحويل بدوال التحويل المناسبة التي تزودك دلفي بها ،

وللتبسيط فإن دلفي تسمح لك في بعض الأحيان نسب نمطين مختلفين ، في حال كان أحدهما يستوعب الآخر ، مثلا لاحظ أننا نستطيع نسب متحول Integer إلى متحول Real بدون أخطاء ، وبالطبع الحالة المعاكسة غير صحيحة .

```
var i:integer;
    r:real;

begin
r:=i;    // ok
i:=r;    //error!!!

End;
```

وهذا الإستثناء مهم في حالة أنماط الأصناف ، فإذا قمت مثلا بتعريف صنف جديد مثل Tanimal وأشتقت منه صنف ولنقل مثلا Tdog ، تستطيع عندها أن تنسب غرض من الصنف Tdog إلى متحول من الصنف Tanimal ، لأن الكلب حيوان ، ولكن ليس بالضرورة أن يكون كل حيوان كلب ، وبناء على ذلك العكس غير ممكن :

```
var
MyAnimal: Tanimal;
MyDog: Tdog;

begin
MyAnimal := MyDog; // OK
MyDog := MyAnimal; // error!!!
```

التحديد المتأخر وتعددية الأشكال :

تعتمد توابع الباسكال وإجراءاتها عادة على التحديد الساكن أو المبكر (*static or early binding*) ، وهذا يعني أن المترجم سيقوم بتحليل الإستدعاء وإستبدال الطلب بإستدعاء لموقع الذاكرة الحاوي على التابع أو الإجراء (عنوان الروتين) ، وهذا يعني التحديد المسبق للتابع أو الإجراء الذي سوف يتم إستدعاءه ، ويقوم المترجم بهذا التحديد لحظة ترجمة المشروع وبالتالي التصرف في وقت التنفيذ معروف ومحدد منذ ترجمة المشروع .

لغات البرمجة الغرضية التوجه (OOP) تسمح بنوع آخر من التحديد يسمى التحديد الديناميكي أو المتأخر (*dynamic or late binding*) ، وفي هذا الحالة فإن العنوان الفعلي الذي سوف يستدعى لن يتم تحديده حتى وقت التشغيل ، ويعتمد التحديد على نمط المتسخ الذي قام بالطلب .

تسمى هذه التقنية تعددية الأشكال (*polymorphism*) باليونانية تعني (many forms) بالإنكليزية ، تعددية الأشكال تعني أنك تقوم بإستدعاء منهج ، تنسبة لمحول ، ولكن ماهو المنهج الذي قامت دلفي فعليا بإستدعاءه فإن ذلك يعتمد على نمط الغرض الذي نسب إليه المتغير ، ودلفي لن تستطيع تحديد صنف غرض هذا المتغير حتى وقت التشغيل .

ميزة تعددية الأشكال أنها تسمح بكتابة أكواد أبسط ، والتعامل مع أغراض متباينة النمط كما لو أنها متشابهة بحيث يتم الحصول على التصرف المناسب في زمن التشغيل .

لتوضيح ذلك دعنا على سبيل المثال نفترض أن صنف ما وصنف مشتق منه يعرفان نفس المنهج ، ولهذا المنهج تحديد متأخر ، مثلا الصنفان Tdog و Tanimal يعرف كل منهما المنهج Voice والذي يخرج صوت الحيوان المختار ، فإذا عرفنا المتغير MyAnimal والذي سيشير في زمن التشغيل إما لغرض من النوع Tanimal أو لغرض من النوع Tdog ، بالتالي فإن المنهج الذي سيتم إستدعاءه فعليا يتم تحديده وقت التشغيل إذا كان منهج Voice الخاص ب Tanimal أو Voice الخاص ب Tdog ، حسب النمط الذي يشير إليه MyAnimal .

لتحقيق ذلك نعرف المنهج Voice على شكل Virtual (إفتراضي) في الصنف الأساسي ، وعلى شكل OverRide (مهيمن) في الصنف المشتق ، بإستخدام الكلمتان الافتاحيتان *virtual* و *override* .

```
Type
Tanimal = class
public
function Voice: string; virtual;

Tdog = class (Tanimal)
public
function Voice: string; override;
```

فإذا عرفنا المتغير `MyAnimal` من الصنف `Tanimal` فإن نستطيع بناء كغرض من `Tanimal` أو `Tdog` ، حسب قواعد التوافق بين الأنماط التي رأيناها في الفقرة السابقة وبالتالي نستطيع إستدعاء كلا من :

```
MyAnimal := Tdog.Create;
MyAnimal := Tanimal.Create;
```

وبالتالي فإننا نستطيع في زمن التشغيل أن نشير إلى غرض من الصنف `Tdog` أو من الصنف `Tanimal` وكل منهما له المنهج `Voice` ، وعندما نحاول إستدعاء هذا المنهج من الغرض `MyAnimal` الذي قد يشير إلى أي منهما ، فإن تصرف مترجم اللغة `Compiler` يكون مختلفا عن الحالة الساكنة للإستدعاء التي يتم تحديد فيها عنوان الإستدعاء سلفا ، وسيعتمد على نوع الغرض الحالي لتحديد أي منهج يجب أن يستدعي :

```
MyAnimal := Tdog.Create;
MyAnimal.Voice // Tdog.Voice
```

// OR ..

```
MyAnimal := Tanimal.Create;
MyAnimal.Voice // Tanimal.Voice
```

إن ذلك يحدث فقط لأن المنهج إفتراضي `Virtual` كما عرفناه ، وبكلمة أخرى فإن هذا الإستدعاء لـ `MyAnimal.Voice` متوافق مع كل الأصناف المستقبلية المشتقة من الصنف `Tanimal` ، حتى التي لم تتم كتابتها بعد .

ملاحظة :

هذا هو السبب التقني لكون البرمجة غرضية التوجه تفضل القدرة على التوسع وإعادة الإستعمال (`reusability`) ، حيث أنك تستطيع كتابة شفرات تستخدم أصناف من بنية وراثية معقدة دون أي معرفة بالأصناف المحددة التي تشكل جزء من هذه البنية ، وبكلمة أخرى تبقى هذه البنية الوراثةية وبرامجك التي تستخدم هذه البنية قابلة للتوسع والتغيير ، حتى بوجود آلاف السطور من الشفرة التي تستخدمها . ولكن بشرط واحد أساسي : أن يكون الصنف السلف لهذه الشجرة الوراثةية مصمم بعناية فائقة .

إعادة تعريف المناهج والمناهج المهيمنة:

كما رأينا ، لجعل منهج متأخر التحديد مهيمنًا في صنف مشتق نقوم باستخدام الكلمة المفتاحية `override` ، تذكر أنك ذلك يمكن أن يحدث فقط إذا كان المنهج معرفًا كمنهج افتراضي (`virtual`) في الصنف السلف ، أي ديناميكي (`Dynamic`) أما إذا كان معرفًا كمنهج ساكن (`static`) فلا توجد طريقة عندها لتفعيل التحديد المتأخر (`late binding`) إلا بتغيير شفرة الصنف السلف نفسها .

قواعد هذه العملية ليست صعبة : المنهج المعرف كمنهج ساكن (`Static`) سيبقى ساكنًا في كل الأصناف الموروثة ، حتى تقوم بإخفاء منهج افتراضي جديد يحمل نفس الاسم . المنهج المعرف كمنهج افتراضي (`Virtual`) يبقى متأخر التحديد في كل الأصناف الموروثة (إلا إذا قمت بإخفاء منهج ساكن ، وذلك يعتبر عملاً غيبياً جداً) . لا توجد طريقة لتغيير هذا التصرف ، لأن المترجم سيولد شفرة مختلفة من أجل مناهج التحديد المتأخر .

لإعادة تعريف منهج ساكن ، أضف تعريف المنهج في الصنف المشتق بدون أي إضافات ، وبإمكانة أن يملك بارمترات مختلفة عن المنهج الأصل في هذه الحالة . لجعل منهج افتراضي ما منهجًا مهيمنًا ، يجب أن تستخدم الكلمة المفتاحية `override` ، وفي هذه الحالة يجب أن يملك المنهج نفس بارمترات المنهج الأصل .

```
type
TmyClass = class
  procedure One; virtual;
  procedure Two; {static method}
end;

TmyDerivedClass = class (MyClass)
  procedure One; override;
  procedure Two;
end;
```

تستطيع جعل منهج ما مهيمنًا بطريقتين : الأولى هي إستبدال منهج الصنف الأصل بواحد جديد .

الثانية هي إضافة شفرات إضافية إلى المنهج الموجود ، ولتحقيق ذلك نستخدم الكلمة المفتاحية `inherited` لإستدعاء المنهج الخاص بالصنف الأصل ضمن شفرة منهج الصنف المشتق .

```
procedure TmyDerivedClass.One;
begin
  // new code
  ...
  inherited One; // call inherited procedure MyClass.One
end;
```

المناهج الافتراضية مقابل الديناميكية Virtual versus Dynamic Methods :

نستطيع في دلفي تفعيل التحديد المتأخر بطريقتين . إما تعريف المنهج كمنهج إفتراضي كما رأينا سابقا ، أو تعريفه كمنهج ديناميكي . العبارة النحوية للكلمتان المفتاحيتان (virtual) و (dynamic) هي نفسها تماما ، ونتيجة إستخدامهما هي نفسها أيضا ، والإختلاف بينهما فقط هو الآلية الداخلية التي يتبعها المترجم لتنفيذ التحديد المتأخر .

المناهج الإفتراضية تعتمد على جدول المناهج الإفتراضية VMT أو (virtual method table) ، والذي يخزن ضمنه عناوين المناهج في الذاكرة ، ويسمى أحيانا Vtable ، حيث سيقوم المترجم بتوليد شفرة القفز إلى عنوان الذاكرة المخزن في السجل n لجدول المناهج الإفتراضية ، تسمح جداول المناهج الإفتراضية بتنفيذ سريع للإستدعاء ، ولكنها تتطلب مدخلا خاصا من أجل كل منهج إفتراضي لكل صنف مشتق ، حتى لو لم يكن المنهج مهيمننا (overridden) في الصنف المورث .

إستدعاءات المناهج الديناميكية بالمقابل تنجز بإستخدام رقم فريد يحدد المنهج ، ويكون مخزنا في الصنف فقط في حال كان معرفا في أصلا أو كان منهجا مهيمننا ، البحث عن المنهج المطابق يكون عادة أبطأ من البحث بخطوة واحدة في جدول المناهج الإفتراضية السابق ، والميزة الأساسية له أن المناهج الديناميكية تولد في الأصناف المشتقة فقط عندما يكون منهج الصنف المشتق مهيمننا .

فكرة : المناهج المتأخرة التحديد يمكن أن تستخدم لمعالجة رسائل ويندوز ، وهذه التقنية يمكن أن تكون مفيدة جدا للمبرمج ويندوز الذين يملكون الخبرة الكاملة برسائل ويندوز وتوابع API .

ملاحظة :

كذلك تستخدم الكلمة المفتاحية Abstract للتصريح عن مناهج ستعرف فقط في الأصناف المشتقة من الصنف الحالي ، والفرق بينها وبين المناهج الإفتراضية أنه يجب تعريفها في كل الأصناف المشتقة . مثلا إذا كان الصنف TAnimal يملك المنهج الإفتراضي Voice فإن أي صنف مشتق منه يمكن أن يعرف منهج Voice خاص به ، أما إذا كان يملك المنهج Voice كمنهج مستخلص (Abstract) فإنه يتوجب على كل صنف مشتق إعادة تعريفه .

المعاملان IS و AS :

لقد تكلمنا في الفقرات السابقة عن توافقية الأنماط في دلفي ، وإمكانية نسب أصناف مشتقة إلى أصنافها الجذر بدون مشاكل . حسناً ..

دعنا نفرض حالة خاصة هنا ، وليكن الصنفان TAnimal و Tdog .

أستطيع أن أنسب Tdog إلى متحول من الصنف الأب TAnimal كما رأينا سابقا . لكن لنفرض أن للصنف Tdog منهج جديد هو Eat مثلا ، إذا قمت بالنسب السابق وقبل المتغير من النوع TAnimal أن يدل على الغرض من النوع TDog يفترض أن أستطيع إستدعاء المنهج Eat الخاص ب Tdog ، .. ولكن ذلك لن يحدث في الحقيقة ، لأن المتحول لازال من الصنف TAnimal والعملية برمتها هي أن المتحول قبل أن يحتوي غرض Tdog لكنه غير قادر على تلبية كافة متطلبات Tdog (الغير موجودة في TAnimal أصلا) .

لحل هذا الإشكال نستطيع إستخدام تقنية من تقنيات RTTI (*run-time type information*) ، والتي تعتمد على فكرة أن كل غرض يعرف ماهو نمطة وماهو الصنف الأب له .

تستطيع السؤال عن هذه المعلومات بإستخدام المعامل IS ، الشبيه إلى حد ما بالمساواة (=) . بارامترا is هما غرض من اليسار وصنف من اليمين ، ويحدد Is إذا كان الغرض الأيسر من الصنف الأيمن أم لا .

if MyAnimal is TDog then ...

الآن وبعد أن تأكدت أن الصنف هو Tdog يمكنك إجراء تحويل آمن للصنف TAnimal إلى Tdog كالاتي :

```
var
  MyDog: TDog;
begin
  if MyAnimal is TDog then
  begin
    MyDog := TDog (MyAnimal);
    Text := MyDog.Eat;
  end;
```

بإمكاننا إنجاز نفس العملية السابقة بالمعامل الهام الثاني الذي توفرة لنا RTTI وهو المعامل AS ، والذي يقوم بعملية التحويل السابقة فقط إذا كان الصنف المطلوب متوافق مع الصنف الأصلي . بارامترا المعامل هما أيضا غرض يساري وصنف يميني والنتاج هو غرض تم تحويله إلى الصنف الجديد المعطى :

```
MyDog := MyAnimal as TDog;
Text := MyDog.Eat;
```


إذا كنت تريد فقط إستدعاء Eat بإمكانك إستخدام حالة أبسط :

```
(MyAnimal as TDog).Eat;
```

نتيجة التعبير ستكون غرض من الصنف Tdog وبإمكانك إستخدام إي منهج من هذا الصنف ، الفارق بين التحويل السابق والتحويل بإستخدام as ، أن as سيختبر الغرض ويرفع إستثناء إذا كان الغرض غير متوافق مع الصنف الذي نحاول تحويله إليه ، وهذا الإستثناء هو ("EInvalidCast") .

إذا أردت تجنب حدوث الإستثناء بإمكانك إستخدام الطريقة الأولى

```
if MyAnimal is TDog then
  TDog(MyAnimal).Eat;
```

لكني لا أرى داع لإستخدام is و as وراء بعضهما والقيام بعملية المقارنة مرتين .

هذان المعاملان أساسيان و مفيدان جدا في دلفي ، لإنك عادة ما تريد كتابة شفرة عامة تستخدم من عدة عناصر بنفس الطريقة .

أهم مثال يخطر ببالي هو إستخدام المتحول الوسيطي Sender في كل أحداث دلفي ، Sender هو متحول من الصنف Tobject الذي يقبل جميع الأغراض ، وبالتالي دائما تحتاج لردة إلى الغرض الأصلي لمعرفة من هو هذا الغرض الذي طلب الحدث :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Sender is TButton then
    ...
end;
```

إنها تقنية شائعة في دلفي ، وتستخدم في كثير من الحالات ، معاملا معلومات نمط زمن التشغيل RTTI هذين قويان للغاية ، ويعتبران من طرق البرمجة القياسية التي تتبعها بورلاند في شفرة عناصر VCL ، ولكن رغم ذلك توجد بعض الحالات الخاصة لتفضيل إستخدام طرق أخرى عليهما عندما تريد حل مشكلة معقدة تستخدم أصناف متنوعة ومتداخلة أنصحك بالعمل مع تعددية الأشكال ، والقاعدة الآن هي أن لا تستخدم RTTI مكان تعددية الأشكال ،

مشكله ال RTTI أنها يجب أن تسير في الشجرة الوراثة للتأكد من أن الصنف مطابق . ربما يسبب ذلك أثرا سيئا للأداء

إستخدام الواجهات Using Interfaces :

تكمن أهمية الواجهات في تقديم بعض ميزات الوراثة المتعددة (multiple inheritance) بسهولة وبدون الغرق في التفاصيل النحوية لها ، كما أن للواجهات أهمية أساسية من أجل إستخدام نماذج الأغراض الموزعة (مثل CORBA و SOAP) والتفاعل مع أغراض تمت كتابتها في لغات أخرى ++C أو Java مثلا .

معظم اللغات الغرضية التوجه الحديثة ، من Java حتى #C ، تملك فكرة الواجهات هذة .

إذا قمت بتعريف صنف جديد من أجل تمثيل صنف آخر بشكل مختصر ، يمكن لهذا الإختصار أن يصل إلى درجة أن الصنف يقوم بسرد مجموعة من المناهج فقط حتى بدون تعريف لإجسام هذة المناهج في قسم Implementation ، في هذه الحالة يمكنك إستخدام تقنية الواجهات Interfaces ، وكأنا نعبّر عن هذا الصنف المختصر بواجهة تحوي تعريف لإسماء المناهج التي يمكن إستخدامها .

- الأغراض من النمط واجهة تحرر تلقائيا عند إنتهاء الصلات معها ، بشكل مشابه لمعالجة السلاسل الطويلة حيث تقوم دلفي بإدارة الذاكرة بالكامل تقريبا .
- مع أن الصنف يمكن أن يرث صنفا أساسيا واحدا فقط ، لكنه يمكن أن يمثل أكثر من واجهه معا .
- كما أن كل الأصناف تنحدر من السلف Tobject ، كذلك تنحدر كل الواجهات من السلف Interface متبعة شجرة وراثية مستقلة .

يتم تعريف الواجهات بالشكل العام التالي :

```
type interfaceName = interface (ancestorInterface)
    ['{GUID}']
    memberList
end;
```

مثلا :

```
type
    ICanFly = interface
        ['{EAD9C4B4-E1C5-4CF4-9FA0-3B812C880A21}']
        function Fly: string;
    end;
```

حيث أن تعريف كل من (ancestorInterface) و ['{GUID}'] اختياري حسب الحالة .

ancestorInterface هو الجد الذي سنورث عنه صنفنا .

GUID إختصار لـ (Globally Unique Identifier) هو رقم معرف يستخدم لتعريف الواجهة بشكل فريد .

تستطيع تعريف هذا الرقم تلقائيا بضغط Ctrl+Shift+G في محرر دلفي .

كما تلاحظ فإن تعريف الواجهه شبيهة بتعريف الصنف في كثير من الحالات ، ولكن توجد بعض الفروقات لوضعها في الحسبان :

- قائمة العناصر (التي سميناها memberList في الشكل العام للتعريف) يمكن أن تحوي فقط إما مناهج أو خصائص . تعريف حقول غير مسموح به في الواجهات .
- بما أنه لا توجد للواجهه حقول ، فإن محددات Read و Write الخاصتان بتعريف الخصائص يجب أن يكونا مناهج حتما .
- كل عناصر الواجهة تكون عامة (Public) ، ولا مجال لتقييد الرؤيا .
- ليس للواجهة بائي أو هادم (constructors or destructors) يعرفان ضمنها .

وهذا مثال على تعريف واجهة مأخوذ من دلفي:

```

type
  IMalloc = interface(IInterface)
    ['{00000002-0000-0000-C000-000000000046}']
    function Alloc(Size: Integer): Pointer; stdcall;
    function Realloc(P: Pointer; Size: Integer): Pointer; stdcall;
    procedure Free(P: Pointer); stdcall;
    function GetSize(P: Pointer): Integer; stdcall;
    function DidAlloc(P: Pointer): Integer; stdcall;
    procedure HeapMinimize; stdcall;
end;

```

وبعدما تقوم بتعريف الواجهة تستطيع تعريف صنف من أجل إستخدامها بواسطة .

مثلا :

```

type
  TAirplane = class (TInterfacedObject, ICanFly)
    function Fly: string;
end;

```

الآن أصبحنا جاهزين لإستخدام غرض من هذا الصنف الذي يمثل الواجهة ، طريقة الإستخدام هي نفسها لإننا نتعامل

مع صنف من الواجهة وليس مع الواجهة مباشرة :

```

var
  Flyer1: ICanFly;
begin
  Flyer1 := TAirplane.Create;
  Flyer1.Fly;
end;

```

رؤيه وإعداد :

عروة عيسى

المراجع :

Marco Cantu Mastring Delphi للمؤلف