Private Investigations

Or: everything you ought to know about private, public, protected and published declarations but couldn't find in the manuals

by Bob Swart

Delphi's ObjectPascal model supports the three cornerstones of object oriented programming: encapsulation, inheritance and polymorphism. Encapsulation is the combination of data (slots or properties) and functions (methods or behaviour) to a class (or object). Inheritance is the process of building a hierarchy of classes, with a descendant class inheriting both data and functions from its ancestors. In polymorphism, a function declared in an class's ancestor is specialised in descendants of this ancestor, thereby refining the general action the ancestor might take with specialised actions for the derived descendant.

This article will focus on encapsulation. We'll delve into the class declaration syntax and semantics with some examples. I'll also propose and illustrate some guidelines for the use of `private`, `public`, `protected` and `published` declarations within Delphi classes.

Encapsulation

To illustrate encapsulation, let's consider a small example that will be used throughout this article. Consider a record `TPerson` with three fields; `Name`, `Birthday` and `Age` and a function `CalculateAge`. See Listing 1 for the code.

Of course, the field `Age` from the record `TPerson` can (and should) be derived from `BirthDay` (and not the other way around). That's why I have also written the function `CalculateAge`. In fact, I want to strongly express my feeling that the record `TPerson` and the function `CalculateAge` belong together, so I'd like to apply encapsulation to put them together in one class.

If you look at Listing 2, I've encapsulated the record declaration for `TPerson` and the `CalculateAge` function in a new class `TPerson`. I've changed the function `CalculateAge` into a procedure, since the `Age` can now be obtained from the class itself. From now on, I will call the field `Age` a *calculated* field: derived from other fields in the class.

The encapsulation made the internal fields `BirthDay` and `Age` visible and automatically accessible for the procedure `CalculateAge`. The new class is much like the previous record with a wrapper for joining several (in this case one) behavioural functions and data properties together. In Listing 2, I'd like to call `CalculateAge` right after every time we give `BirthDay` a (possibly new) value. So, we can be sure that the fields `BirthDay` and `Age` are always in sync.

Access

The class declaration of `TPerson` contains everything a class declaration should have, except access or visibility specifiers. While Borland Pascal, the ancestor of Delphi's Object Pascal, only distinguishes between `public` and `private` access specifiers, Delphi adds `published` and `protected`.

The four keywords `published`, `public`, `protected` and `private` are used within a class to denote special access constraints for a class declaration part. These keywords operate both on data and functions within a class.

Public

Class identifiers declared in the `public` part do not have any special

➤ *Listing 1*

```
Type
  TPerson = record
    Name: String;
    BirthDay: TDate; { has Year, Month and Day subfields }
    Age: Integer;
  end {TPerson};

function CalculateAge(BirthDay, Now: TDate): Integer;
var Age: Integer;
begin
  Age := Now.Year - BirthDay.Year;
  if (Now.Month < BirthDay.Month) or ((Now.Month = BirthDay.Month) and
    (Now.Day < BirthDay.Day)) then
    Dec(Age); { no birthday this year }
  CalculateAge := Age
end {CalculateAge};
```

➤ *Listing 2*

```
Type
  TPerson = class
    Name: String;
    BirthDay: TDate;
    Age: Integer;
    procedure CalculateAge(Now: TDate);
  end {TPerson};

procedure TPerson.CalculateAge(Now: TDate);
begin
  Age := Now.Year - BirthDay.Year;
  if (Now.Month < BirthDay.Month) or ((Now.Month = BirthDay.Month) and
    (Now.Day < BirthDay.Day)) then
    Dec(Age); { no birthday this year }
end {CalculateAge};
```

restrictions on their scope, that is everybody can access them freely. If you want to make all data fields and functions of the class TPerson available to everybody else, just use the public keyword

```
Type
  TPerson = class
  public
    Name: String;
    BirthDay: TDate;
    Age: Integer;
    procedure CalculateAge(
      Now: TDate);
  end {TPerson};
```

Now, if you have a variable Me of type TPerson you can access the field Age of Me via Me.Age. In fact, everybody could access the field Age of Me. This could result in a situation where Age and BirthDay are out of sync, even when I call CalculateAge after each possible update of BirthDay.

This leads to my first rule: ***never make your "calculated" class fields public!***

Fortunately, there are other access specifiers which come in handy for this particular situation!

Private

Class identifiers declared in the private part of a class cannot be accessed from the outside. Unfortunately, there is one exception: from within the same source module (unit or program) the private identifiers are just as visible as public identifiers. This is probably done to enforce backwards compatibility with Borland Pascal class declarations, which also used this rule. I prefer the C++ definition of private, in which private identifiers are truly invisible to anybody (except friends, but that's another story). So, here's our example:

```
Type
  TPerson = class
  public
    Name: String;
    BirthDay: TDate;
    procedure CalculateAge(
      Now: TDate);
  private
    Age: Integer;
  end {TPerson};
```

In Object Pascal, then, private component identifiers act like normal public component identifiers *within* the module that contains the class type declaration, but *outside* the module, any private component identifiers are unknown and inaccessible.

This leads to my second rule: ***always put each class declaration in its own unit!***

As long as we follow this rule, we can rest assured that private identifiers, such as Age, are indeed private and invisible to anyone except the class TPerson itself!

Ahem, didn't we forget something? If all private identifiers are indeed only visible to the class itself, how do we get at the value from the outside? Well, that's where *access functions* come in. We just have to add a function GetAge: Integer to the class TPerson that returns the value of the private field Age:

```
function TPerson.GetAge :
  Integer;
begin
  GetAge := Age
end {GetAge};
```

In general, for every private identifier whose value needs to be communicated to the outside world, you should write an access function. These GetXX functions guarantee that only the value of the private field is returned, while the field itself is safeguarded against improper updates!

Protected

Borland Pascal only offered the public and private access specifiers. With Delphi's Object Pascal, we now also have protected and published.

The protected keyword combines the advantages of the public and private keywords. As with private identifiers, you can hide implementation details from end users. However, unlike private identifiers, protected identifiers are still available to programmers who want to derive new classes from your classes without the requirement that the derived classes be declared in the same unit (which we don't want to do anyway, right?).

So, if you want the class TPerson to make the field Age available not only to the class TPerson but to descendants as well, you'd define Age as protected instead of private:

```
Type
  TPerson = class
  public
    Name: String;
    BirthDay: TDate;
    procedure CalculateAge(
      Now: TDate);
    function  GetAge: Integer;
  protected
    Age: Integer;
  end {TPerson};
```

Now, you can derive a new class, say TAdultPerson, from TPerson, which can still access the protected field Age! This is very useful, although a thorough analysis must be made between fields that can be private and fields that should be protected (ie accessible for descendant classes). Sometimes this involves a bit of trust...

Trust Me?

At this point, there are two choices we can make concerning the protection of our data fields in a class declaration. The first one (trusting) is to make all data fields public, except for the "calculated" data fields, which should be protected. The second (fail-safe) approach is to make all data fields at least protected and make the "calculated" fields private.

The first approach makes sure no accidental discrepancies will occur, but will allow any normal access to everything. This is a very open approach, which is best used when you're programming on your own or with a small group of people who have good communication between each other.

I don't say this without reason, as I've often seen mistakes happen from accidental misuse of data fields in classes. Therefore, if you really want to play it safe, or if you have to work together with a rather large or inexperienced group of programmers, I can only recommend the second (fail-safe)

approach: make everything at least protected, and all "calculated" fields private from the start. Combined with the second rule I gave earlier, this will safeguard all data fields in your classes at all times!

This leads to my third rule: ***always make your data fields at least protected!***

With all the data fields at least protected, we can rest assured that nobody but ourselves (or our descendants for protected data fields) can modify them. This should give us a safe feeling, as we know that any inconsistency within the class fields data will be caused by our code and our code alone!

In the class definition of TPerson, I've now included the methods SetName, GetName, SetBirthDay and GetBirthday to communicate the values of the protected data fields Name and BirthDay with the outside world. Since I've told you from the beginning that we'd only call CalculateAge whenever the value of BirthDay changes, we can integrate CalculateAge with the SetBirthDay procedure, which leads to the code in Listing 3.

This class definition gives me all I need: protection for my data and yet easy access for the outside world! I can even put my pre- and post-conditions inside the access procedures and functions if I want. Safety and convenience, all in one!

Properties

I've used the fail-safe approach described above for a long time now. The only thing I don't like about it is that I always have to call a procedure to set a new value or a function to get the current value. It may sound convenient at first, but after a while it's a real pain. Also, I can't help but wonder what the overhead is in calling a method for each access of an internal data field.

Delphi's Object Pascal now offers a feature called *properties* that can be used to extend the fail-safe approach and solve some of these problems. A property looks to the user just like a class data field, but internally can encapsulate methods which read or write the value of the field.

So, we can declare a property Name of type String for our class TPerson. The property definition declares the field Name and the actions associated with reading and writing the property Name:

```
Type
  TPerson = class
  private
    FName: String;
  protected
    procedure SetName(
      NewName: String);
    function  GetName: String;
  public
    property Name: String
      read GetName
      write SetName;
  end {TPerson};
```

Note that we need an internal (private!) field to store the actual contents of the property. This field is the name of the property with a letter 'F' as a prefix.

The property itself can now be seen as an alias to the user of the class. From outside the class, the property Name can be accessed as if it was a normal public field of type String. This time, we really do have safety (all access to Name goes through the SetName and GetName methods, in which we can put my consistency checking code) and convenience (now everybody can just assign to or from the property Name)!

And just in case we don't need any consistency checks, we can even skip the access methods and wire the property directly to the internal field itself. So we have no more consistency checks, but we don't have any overhead either! And the user of the class doesn't notice anything (they have just been using the same reference to the property Name):

```
Type
  TPerson = class
  private
    FName: String;
  public
    property Name: String
      read FName write FName;
  end {TPerson};
```

Which leads to the fourth possible access specifier: published.

Published

The visibility rules for published identifiers in a class are identical to those of public identifiers. The only difference is that run-time type information is generated for fields, methods and properties that are declared in a published part. This run-time type information enables an application to dynamically query the fields, methods and properties of an otherwise unknown class type. Furthermore, the Delphi IDE uses a component's run-time type information to determine the list of properties shown in the Object Inspector.

Since properties can be used as a kind of developer and user interface to our class, it makes sense to always use properties for any data fields in classes. These properties will be published, so anybody can easily access them from either outside the class or the Object Inspector (if you should want to make your class a real component).

➤ *Listing 3*

```
Type
  TPerson = class
  public
    procedure SetName(NewName: String);
    function  GetName: String;
    procedure SetBirthDay(NewBirthDay: TDate);
      { calls previous CalculateAge }
    function  GetBirthDay: TDate;
    function  GetAge: Integer;
  protected
    Name: String;
    BirthDay: TDate;
  private
    Age: Integer;
  end {TPerson};
```

This leads to my fourth rule: ***always use*** `published` ***properties for data fields, based on*** `private` ***internal fields and*** `protected` ***access methods!***

Which leads to the class declaration for `TPerson` in Listing 4 (note that we need to derive from `TComponent` in order to use the `published` keyword – a little undocumented 'feature' of Delphi).

Note that for the `Age` property we only have a read method, as `Age` is a read-only property! The one thing that's missing from this approach is the "Now" date, which must be available when we calculate the value of the `FAge` field. Fortunately, we can ask the operating system for the current date, so we'll just step over this little problem and focus on the design task itself again.

Dependencies

The last step you can take is to dissolve all hidden dependencies or calculated fields. Since `Age` depends on the value of `BirthDay`, why store the value of `Age` inside the class? Why not calculate the value of `Age` each time we access the property `Age` (and make an implicit call to `GetAge`)? That means the internal field `FAge` can be omitted and the access method `GetAge` has to be modified to contain the original `CalculateAge` code. The final version of `TPerson` is defined as in Listing 5.

Properties are a natural extension of fields in a class. Both can be used to express attributes of a class, but whereas fields are merely storage locations which can be examined and modified at will, properties provide greater control over access to attributes, they provide a mechanism for associating actions with the reading and writing of attributes and they allow attributes to be computed.

➤ *Listing 4*

```
Type
  TPerson = class (TComponent)
  private
    FName: String;
    FBirthDay: TDate;
    FAge: Integer;
  protected
    procedure SetBirthDay(NewBirthDay: TDate);
     { calls previous CalculateAge }
    function  GetBirthDay: TDate;
    function  GetAge: Integer;
  published
    property Name: String read FName write FName;
    property BirthDay: TDate read GetBirthDay write SetBirthDay;
    property Age: Integer read GetAge;
  end {TPerson};
```

➤ *Listing 5*

```
Type
  TPerson = class (TComponent)
  private
    FName: String;
    FBirthDay: TDate;
  protected
    procedure SetBirthDay(NewBirthDay: TDate);
      { calls previous CalculateAge }
    function  GetBirthDay: TDate;
    function  GetAge: Integer;
  published
    property Name: String read FName write FName;
    property BirthDay: TDate read GetBirthDay write SetBirthDay;
    property Age: Integer read GetAge;
  end {TPerson};
{ The implementation of the property access methods is as follows: }
procedure TPerson.SetBirthDay(NewBirthDay: TDate);
begin
 { we can do some checks here }
 FBirthDay := NewBirthDay;
end {SetBirthDay};
function TPerson.GetBirthDay: TDate;
begin
  GetBirthDay := FBirthDay;
end {GetBirthDay};
function TPerson.GetAge: Integer;
var Age: Integer;
    Now: TDate;
begin
  { somehow, get the value of Now: TDate }
  Age := Now.Year - FBirthDay.Year;
  if (Now.Month < FBirthDay.Month) or ((Now.Month = FBirthDay.Month) and
    (Now.Day < FBirthDay.Day)) then
    Dec(Age); { no birthday this year }
  GetAge := Age
end {GetAge};
```

Conclusion

We've explored the Delphi class declaration syntax and semantics with some examples. We've also examined class access specifiers and some rules and guidelines for the usage of access specifiers within Delphi classes. Finally, we've seen the great use of Delphi class properties. The rules I've given are (in summary):

- Never make your "calculated" class fields `public`;
- Always put each class declaration in its own unit;
- Always make your data fields at least `protected`;
- Always use `published` properties for data fields, based on `private` internal fields and `protected` access methods.

My final recommendation is to use Delphi and properties as much as possible, and have fun!

Bob Swart (you can email him at 100434.2072@compuserve.com) is a professional 16- and 32-bit software developer using Borland Delphi and sometimes a bit of Pascal or C++. In his spare time, he likes to watch video tapes of Star Trek Voyager with his almost two year old son Erik Mark Pascal.