Effective Delphi Class Engineering Part 4: The TObject Of My Affection

by David Baer

We have already examined one of the three foundation concepts of object oriented programming: *encapsulation*. In this part and the next, we'll be examining the other two: *inheritance* and *polymorphism*. Unlike encapsulation, which can be considered in isolation, inheritance and polymorphism are inextricable. We can't discuss one without the other.

It's also fair to say that inheritance is more straightforward and immediately comprehensible, and we're going to defer all but the essentials of polymorphism until next month.

It seems somehow fitting that during the time I've been developing the outline of these two parts, my pleasure reading has been the latest volume in the delightful Harry Potter series. It's not that there's anything magical about how inheritance and polymorphism are implemented by the compiler. I hope you'll agree this is all very straightforward when you've finished reading this article.

No, what makes this fitting is that, to me, there was wizardry afoot when these ideas were initially conceived. They are so inspired that it's difficult for me to view them as other than the creations of magical minds.

So, let's waste no more time: we've got a lot of ground to cover. Could somebody please hand me that wand? Thanks... Ready?

Accio TObject!

➤ *Listing 1*

```
TParent = class(TObject)
private
  ParNbr1: Integer;
  ParNbr2: Integer;
public
  procedure DoSomethingParental;
end;
TChild = class(TParent)
private
  ChldNbrA: Integer;
  ChldNbrB: Integer;
public
  procedure DoSomethingChildish;
end;
```

Basic Inheritance

Membership has its privileges: take advantage of the fact that your class is also a TObject.

In all our example class declarations so far, we've dodged a fundamental issue: `TObject` is a class that is the direct or indirect ancestor of all others in Object Pascal code. Stated another way: all classes, except `TObject` itself, inherit from a parent class and if that parent class is not explicitly named in the declaration then it's implicitly `TObject`.

But, what do these terms *ancestor* and *inherit* mean? *Ancestor* is easy to explain: class `TA` is the ancestor of class `TC` if `TA` appears somewhere in the chain of inheritance which extends back to `TObject`.

OK, so now what does *inherit* mean? Let's skip a formal definition for now and simply consider the consequences of having a class that is a descendant of `TObject`. So far, all our example class declarations have started something like:

```
TMyClass = class ...
```

But this form is the precise equivalent of:

```
TMyClass = class(TObject) ...
```

in which the name in parentheses denotes the parent class. So, what does having `TObject` as an ancestor buy us?

Quite a lot, as it turns out. `TObject` is a class itself; it's small but potent. And the good news is that all of the non-private member data items, properties and methods in an ancestor class are available to our class. Those members that are public are immediately available to clients of our class as well, with no extra work on our part. For properties in the ancestor class that are protected, we have two choices. We may utilise them in our class, but continue to keep them off limits to class clients, or we may 'promote' them to public for use by class clients.

Actually, there are no data members in `TObject`, but there are several handy methods. We can invoke the `ClassName` function, for example, to retrieve this information as a string (which can be immensely useful in constructing debug output). I would recommend that you peruse the `TObject` help to get a full accounting. When you do, pay attention to the `Dispatch` method. The messaging capability of `TObject` is a powerful mechanism which we'll be looking at closely a bit later.

More On Inheritance

Think of using inheritance as a mechanism for implementing class specializations.

All right, let's try a more formal approach to explaining inheritance. One excellent way to view it is as an extension capability. Let's say that you have a class that does many things you need, but not quite all that you need. You may write a descendant of that class which offers the extensions or customizations the parent class fails to deliver. As just stated, your descendant class has all of the non-private data and methods of the parent class available to it. Not only that, your class has all the non-private data and methods of *all* ancestor classes, going all the way up the chain to `TObject`.

The above statement is true when the descendant class resides in a unit that's separate from the parent. Recall from Part 2, how-

ever, that where classes are defined in the same unit, the descendant class has access to all data members, properties and methods of the parent (and of anything else defined in that unit as well).

But let's return to the notion that a descendant class is a specialization of its parent class. There are two crucial parts here: 'specialization' and 'is a'. As far as the language is concerned, the descendant class formally *is a* parent class type as well. Let's see how this plays out in Object Pascal (OP) code.

Listing 1 shows two class declarations. We're going to use these throughout, adding methods, etc as we go, but we'll start with a bare-bones set. `TParent` is a specialization of `TObject`, and `TChild` is a specialization of `TParent`. So, we have three *is a* relationships here: `TParent` *is a* `TObject`, `TChild` *is a* `TParent` and also *is a* `TObject`. The OP `is` operator returns `True` in all cases where an *is a* relationship is present, and `False` otherwise.

Next, let's look at assignment compatibility. The compiler will allow assignment of a class instance to a class instance variable (ie, a class reference variable) of another type where the instance class type *is a* reference variable type. If that sounds a little confusing, take a look at the examples in Listing 2, which illustrates which assignments are legal and which are forbidden.

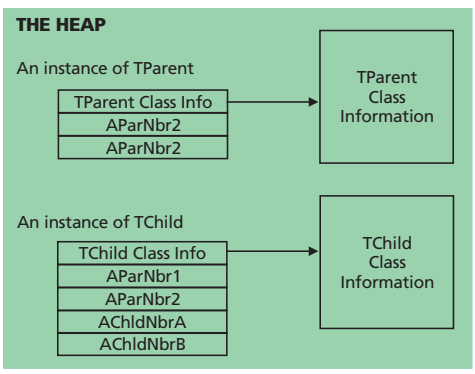
Finally, let's examine the storage allocation behavior that occurs with inheritance. We looked at basic object storage allocation in Part 2 and, hopefully, you found it to be straightforward. Fortunately, things are only marginally more complicated when we introduce inheritance into the picture.

When an instance of a class is created, a block of storage is allocated from the heap for instance data. For an instance `TObject`, the block will be four bytes. Those four bytes will contain a pointer to class information. Since `TObject` has no member data items, there's no need for additional storage. Of course, there is never a need to actually create an instance of `TObject`.

In our examples, `TParent` inherits from `TObject`. As such, the instance data block starts with a layout identical to that of its parent. This area is immediately followed by a layout that accommodates the data members of `TParent`.

The layout for the instance data of a `TChild` is organized in a similar fashion: parent data followed by `TChild` items. All of this is illustrated in Figure 1. One final note while we're on the subject: `TObject`'s public method `InstanceSize` may be used to discover the size of the instance data block of any object or class.

➤ *Figure 1*



Member Visibility

Use protected member methods and properties for those things you want to be accessible to descendant classes, but not accessible to the general public.

You might have noticed that I've used the term 'non-public' several times. This was basically a dodge. I wanted to avoid the equivalent phrase 'protected or public', an explanation of which would have slowed us down. Well, now it's time to address this subject. Hopefully, you'll recall from Part 2 that class members (data items, properties and methods) which are declared as public are available to code outside your class's methods. Private and protected items are not. Private members, on the other hand, are available only to code residing in the same unit in which the class declaration exists (which, of course, includes all methods of that class).

```
O1, O2: TObject;
P1, P2: TParent;
C1, C2: TChild;
...
// these assignments are legal
O1 := O2; // O2 is-a TObject
O1 := P2; // P2 is-a TObject
O1 := C2; // C2 is-a TObject
P1 := P2; // P2 is-a TParent
P1 := C2; // C2 is-a TParent
C1 := C2; // C2 is-a TChild
//  these produce compile errors
P1 := O2; // O2 is-a TParent - not!
C1 := O2; // O2 is-a TChild - not!
C1 := P1; // P1 is-a TChild - not!
```

➤ *Listing 2*

Protected status is less restrictive than private. Protected class members are available to descendant classes, but not to the general public. You may at some point find yourself confused when trying to decide which class members should be private and which protected. Time for a new guideline.

Protection Levels: Data

Declare member data items as private; allow derived classes access via protected properties.

Unless you're developing a class library for commercial distribution, don't sweat the small stuff in determining what should be private versus protected. You can expend a great deal of energy on trying to get this exactly right, but end up with very little that's beneficial to show for your efforts.

In practice, most of the classes you design will probably be 'final' classes (to borrow a Java term) in any case. By this I mean a class for which actual objects will be instantiated and which will not be used as the base class from which other classes will inherit. For a final class, all we need care about is encapsulation, ie deciding what class clients may access. From that perspective, private and protected accomplish the same thing.

But what if you are designing a class to be a base class, intended to be a parent in an inheritance scheme, for 'in-house' use? Even in this case, you or your team will usually be responsible for designing all the classes in the family. As such, you can normally apply a few 'rules of thumb', and if the rule doesn't work for a particular circumstance, then just ignore it for that case and get on with things.

It's only when you're producing a class library for external consumption that this issue starts to truly matter. On the one hand, the more you keep private, the more flexibility for code modifications you have in future releases without breaking existing client code (in client classes inheriting from those in your library). At the same time, being stingy with the protected designation can make effective inheritance anything from difficult to impossible. Fortunately, it's likely most of you will never find yourselves in this position.

In my own class coding, one of my 'rules of thumb' is stated in this guideline. The rationale behind this is the same as in never providing public access to data members (as explained in Part 3). Permit access from the outside as necessary, but retain control by using properties, which allows you to provide get and set methods as the need for such becomes evident.

While it's usually fairly easy to determine the proper amount of visibility for class data items and properties, the visibility of methods is quite another matter. Before we examine this, we need to consider the behavior of class methods in the context of inheritance.

Inheritance And Methods

Override base class methods to provide alternative behavior, but make calls to inherited methods to draw upon base class services when you need them.

The key to introducing specialization in class inheritance is in supplying methods that augment or replace services in base classes. For new behavior (ie new services), we may simply add methods to supply services not available from base classes. But when we want to alter existing behavior, we often need to substitute a method of our own for one in a base class. This replacement may involve totally supplanting the base class processing, it may involve subtle modifications in behavior, or anything in between.

Whatever the level of alteration, we're engaging in the practice of what most Delphi coders would call *overriding* base class methods. Actually, this terminology is not quite correct, but please briefly allow me some leeway (I'll clarify this point shortly).

Listing 3 shows a simple case added to our example class family. Normally, the same parameter types (and return type for functions) are used. For virtual methods, which we'll discuss in a moment, the `override` directive is required in the declaration.

So what can we do in our overridden method? Basically, anything we want, subject only to the limitation that we cannot call private methods in ancestor classes or access private data. But we can certainly call protected methods. In fact, one of the more common situations is where we call the ancestor version of the method being overridden. We do this by prefacing the method name with the keyword `inherited`.

➤ *Listing 3*

```
  TParent = class(TObject)
  public
    function ReturnOnInvestment: Double;
  end;
  TChild = class(TParent)
  public
    function ReturnOnInvestment: Double;
  end;
  ...
function TParent.ReturnOnInvestment: Double;
begin
  Result := <some calculation>;
end;
function TChild.ReturnOnInvestment: Double;
begin
  // get ROI from parent method; if news is good, make it
  // better;  if news is bad, make it less so
  Result := inherited ReturnOnInvestment;
  if Result > 0 then
    Result := Result * 2
  else
    Result := Result / 2;
end;
```

Listing 3 contains an example of this technique (admittedly, not a very realistic one). The compiler allows a shorthand notation as well. You may just call `inherited` without referencing the method name or specifying the parameters. This is interpreted as a call to the inherited version using the same parameters passed into the routine doing the calling. Look up `inherited` in the Delphi help for an explanation of all the nuances.

I tend to avoid this kind of notation because I like the explicitness of the fully written code. But the compiler behavior is well documented, and I'm not suggesting that the shorthand is bad practice.

If you're thinking that all of this is pretty straightforward so far, you're quite right. However, the subject gets more complicated when we introduce early versus late binding in method calls.

Virtual Methods

Understanding virtual method behavior is the key to harnessing the power of inheritance (and avoiding some of its limitations).

Virtual methods are at the heart of effective inheritance. But they can be one of the more elusive topics of object orientation to fully understand (at least I remember that being the case when I was first studying OO).

We need to look at this subject now, because it plays an important role in method inheritance, but we're not going to really dig into it until the next instalment. When we do, I hope to show you an example of real life use where the value of virtual methods becomes immediately obvious. For now, I'm afraid we'll have to focus on the basic mechanics.

The overridden methods we've looked at so far provide for static calls. By static calls, I mean calls that are 'hard-wired' at compile-time (we'll look at an example shortly). Compilation into static calls is often referred to as early binding. But another possibility exists.

When a method is declared as `virtual`, overriding methods in descendant classes get different

treatment in the way calls to the method are compiled. In this case, the decision about which version of the method (eg the base class's version or the descendant class's version) is deferred until runtime. This is known as dynamic invocation, or late binding.

As it turns out, *only* virtual methods may be overridden, if we're being precise. Non-virtual methods are not overridden, but merely replaced. An additional consequence is that overriding methods must have the same parameter (and function return type) as the method being overridden. This is not a requirement for replaced methods (although it will typically be so, nevertheless).

If you spend any time reading the Delphi news groups, you'll realize that a fair number of otherwise expert participants can be a bit sloppy on this subject, and will nonchalantly speak of overriding non-virtual methods.

Let's return to a different version of our simple example class framework. In Listing 4 we see that `TParent` has two methods, one of which is `virtual`, and `TChild` has its own version of each. Note that the override for the virtual method has the directive `override` present in the declaration.

In the code, we create a `TParent` object, assigning it to a `TParent` reference. We also create two `TChild` objects, assigning one to a `TParent` reference and the other to a `TChild` reference. Remember, we can assign the `TChild` to a `TParent` reference variable because `TChild` *is a* `TParent`.

The comments in the code then tell the whole story. We see that the non-virtual (ie the statically called, or early bound) method calls are based on the reference type. The virtual (dynamically called, or late bound) method calls are based on the actual object type.

You may not realize it, but you've just encountered the wonderful power of polymorphism: that quality in objects that allows runtime responsiveness based on object type. Inheritance without polymorphism would be of limited use but, with it, inheritance offers extraordinary power.

```
TParent = class(TObject)
public
  procedure DecideEarly;
  procedure DecideLate; virtual;
end;
TChild = class(TParent)
public
  procedure DecideEarly;
  procedure DecideLate; override;
end;
...
Parent: TParent;
Child: TChild;
ParentalChild: TParent;
...
Parent := TParent.Create;
Child := TChild.Create;
ParentalChild := TChild.Create; // TChild assigned to TParent!
// early bound (non-virtual) calls
Parent.DecideEarly;          // TParent version called
Child.DecideEarly;           // TChild version called
ParentalChild.DecideEarly;   // TParent version called
// late bound (virtual) calls
Parent.DecideLate;           // TParent version called
Child.DecideLate;            // TChild version called
ParentalChild.DecideLate;    // TChild version called
```

➤ *Listing 4*

But this leads us back to the more mundane dilemma of deciding just how accessible a base class needs to be to descendant classes for method overriding.

Protection Levels: Methods

Determining private versus protected status on methods can lead to a major case of analysis-paralysis: use an arbitrary scheme and get on with more useful things.

Some would, no doubt, regard this guideline as heresy. This is the sort of topic over which 'holy wars' have been fought (at least in the contentious C++ community). But my motivations are purely pragmatic and assume, once again, that you're not engaged in developing a class library for commercial distribution, where these considerations can have an impact on the extensibility of the product.

The fact of the matter is that assigning method protection allocations that are nearly perfect can be extraordinarily difficult due to the number of factors that need to be considered for each case. For example, if a method absolutely needs access to a private data member to be functional (and where there's no protected property to access instead), then making the method protected (meaning it may be overridden) is pointless. You probably wouldn't be able to do anything useful in the overridden code anyway. For a big class with lots of methods and data members, attempting to apply this level of scrutiny in every case may leave your head spinning.

If you're designing a class framework for in-house use, you can start with a restrictive scheme and promote as the need arises. Alternatively, you can be more generous with the `protected` designation, knowing that you'll also be designing descendant classes and should therefore be able to manage the encapsulation at all levels. The point is to approach it pragmatically and find a scheme with which you're comfortable and which doesn't slow you down.

For example, property read and write methods should never need overriding. You can make these private without a moment's thought. Conversely, any method you know will be subject to overriding (or replacing) in descendant classes must be protected. Virtual methods are so designated because they are intended for overrides. As such, they should never be private.

But I must be honest and confess that I'm somewhat lazy in this area. My typical practice is simply to declare all non-public methods as protected (and I have yet to be burned by this approach). In my experience, if you properly limit the visibility of class data members, the rest pretty much takes care of itself. But if you regard this casual approach as undignified, I'll understand.

Class Parentage

Use an 'is a' versus 'has a' deliberation to determine if you are inheriting from the most appropriate base class.

Where a class is a specialization of an existing class, determining what to use as the base class is sometimes fairly obvious. But this is not always the case. In particular, it's important to distinguish between the cases where your class truly *is a* specialization of the base class and the one in which it simply requires an instance of that class to accomplish its goals.

Let's consider two examples. First, let's assume we need a class that does everything `TList` does, plus a little bit more. Those extensions might be the capability for a block deletion (starting at index `I`, delete `N` items). While we're at it, let's also provide a property, `High`, which returns `Count-1`, so that we can save a few keystrokes when writing code that iterates through the list with a `for` statement.

What to use as the base class here? It doesn't take a flash of brilliance to arrive at the answer. The base class should be a `TList`, no doubt about it.

Now, let's consider a different case. Suppose we need to implement a stack class (a list that offers last-in-first-out management of the stored items). Why re-invent the wheel here? Why not use a `TList` to manage the stored items and just provide the requisite methods: `Push`, `Pop` and `Peek`?

So, we once again use `TList` as the base class, right? Bzzzt! Wrong! This is a trap that neophyte class designers can easily fall into. `TList` has much of what we need so it's tempting to use it as a base class. There's one problem, though, and it's a major one. A stack *is not a* list, period!

As a stack, we want our clients to have access to the push, pop and peek services we provide, but they should not have the capability to insert or delete items at arbitrary locations in the stack. Those actions are not in the stack's charter. But by inheriting from `TList`, all of its public methods (`Insert`, `Delete`, `Exchange`, etc) are available to the users of our stack class.

So, what to do? We still would like to avoid re-inventing all the storage management services required. The answer is simple. We want our class to use a `TList` internally for storage management, but we just don't want it to inherit from `TList`. For the base class, `TObject` will do nicely.

As an internal helper class, we may expose as many or as few of the `TList` methods and properties as appropriate (in this case, not all that many). By employing a `TList`, but not *being* a `TList`, our stack class does not offer all of the public `TList` methods to our clients. While many classes will manage collections of items, very few will be such straightforward extensions of `TList` that inheriting from `TList` (or `TStringList`, either for that matter) is appropriate.

It all boils down to the *is a* criterion. If you can say that I could use the inherited class any place I needed the base class with no ill effect, you've made a valid case for

the proposed inheritance. But if the *is a* relationship is shaky, then rethink things. You're likely just looking at a *has-a* situation instead.

Under The Hood

To gain insight into the inner workings of a Delphi object, including how method invocations are handled, study the class information layout in system.pas.

I always feel more comfortable and assured about the code I write when I have some idea about what the compiler is generating from it, and I suspect most of you do as well. With this in mind, it's fortunate that the mechanisms in play for class method invocations are quite straightforward.

We have discussed memory allocation issues of object instances several times. You may recall that I showed how the first four bytes of any object instance data is a pointer to some class information, but I was quite vague as to what that information was. Well, now is the time to take a closer look.

Each class has a block of data in the load module that contains essential information about the class, information that's used in a variety of ways. The layout of this storage changes from time to time (although no changes were evident between Delphi 4 and 5).

In any case, the Borland engineers did what any good developer would do with arbitrarily changing offsets. They declared these as constants in System.pas, and a study of them can be enlightening.

➤ *Listing 5*

```
{ Virtual method table entries }
  vmtSelfPtr          = -76;
  vmtIntfTable        = -72;
  vmtAutoTable        = -68;
  vmtInitTable        = -64;
  vmtTypeInfo         = -60;
  vmtFieldTable       = -56;
  vmtMethodTable      = -52;
  vmtDynamicTable     = -48;
  vmtClassName        = -44;
  vmtInstanceSize     = -40;
  vmtParent           = -36;
  vmtSafeCallException = -32;
  vmtAfterConstruction = -28;
  vmtBeforeDestruction = -24;
  vmtDispatch         = -20;
  vmtDefaultHandler   = -16;
  vmtNewInstance      = -12;
  vmtFreeInstance     = -8;
  vmtDestroy          = -4;
  vmtQueryInterface   = 0;
  vmtAddRef           = 4;
  vmtRelease          = 8;
  vmtCreateObject     = 12;
```

These declarations are shown in Listing 5.

There's just a little misinformation here, but it's easily cleared up. You can see that the comment identifies the offsets as belonging to the virtual method table (more frequently, the VMT). Actually, there's a good deal more here than just virtual method call support. In fact, only some virtual method information appears with these constants, but we'll get back to that in a moment.

The class instance data pointer to the VMT truly does point to the VMT portion of the class information, but there's much other information at negative offsets to that pointer, as can be seen in Listing 4. For example, at `vmtInstanceSize` is a 4-byte integer specifying the size of the instance data required by objects of the class. The parent class VMT may be found at offset `vmtParent`.

The addresses of the routines used to do the allocation (for a create) and de-allocation (for a destroy) are found in `vmtNewInstance` and `vmtFreeInstance`. These are declared as virtual in `TObject`, but appear before the official start of the virtual methods beginning at offset zero of the VMT. In fact, this is true of all the virtual methods declared in `TObject`.

A complete discussion of what is found at these negative VMT locations is more than we have space for here. But you can learn much just by examining the names of the constants.

The real VMT of `TObject` descendants, that is to say, the part of the class information that supports virtual method invocations, begins at offset zero. Don't be distracted by the declarations of `vmtQueryInterface` etc (which would appear to occupy space starting at offset zero). The first virtual method in an immediate descendant of `TObject` will appear at that offset.

Each virtual method in a class will have a pointer to the start of the code for the method listed in the VMT in successive 4-byte slots. The key to why this works is that this pointer will appear in the *same* offset from the start of the VMT for every class descendant as well. That's all there is to it!

If a descendant class overrides a virtual method, the address of the overriding method will now occupy the slot. If it does not override a method, the slot will contain the address of the most recent class in the hierarchy that did override it (or of the original virtual method if no overrides have been made anywhere in the inheritance chain).

The machine code for calling a virtual method requires a lookup the address of the method before executing the jump to the routine. There's a small amount of overhead in doing it this way compared to jumping to an early-bound ('hard-wired') method address. But the few extra instructions aren't all that expensive, and one should never shy away from using virtual methods over concerns of execution efficiency.

Next Time

We've still got a lot of ground to cover in the exploration of inheritance and polymorphism. Next month we're going to pick right up where we left off here and look at some of the more subtle aspects of these topics.

David Baer is Senior Architectural Engineer at StarMine in San Francisco. He'll turn 53 right around the time you'll receive this issue and he's grateful for his inheritance: the genes that, thus far, have decided that hair on his head should continue to grow in reasonable profusion. Contact him at dbaer@starmine.com