Simulating Multiple Inheritance With Delphi Interfaces

by Marco Cantù

Contrary to what happens in C++, the Delphi inheritance model doesn't support multiple inheritance. This means that each class can have only a single base class. The usefulness of multiple inheritance is a topic of many heated debates. For this reason, in this article I'm not going to discuss whether the absence of this construct in Delphi is a pity (because you lose some of the power of C++) or if it is an advantage (because you get a simpler language and fewer problems).

I will assume that it is useful to view a single object from multiple 'perspectives', in other words to consider it as a generic object with different base classes. After introducing the role of interfaces, we'll build a complete example following this principle. Before we start, keep in mind that Java has the same problem: it has no multiple inheritance but it does have interfaces (this language construct is more integrated in Java than it is in Object Pascal).

From Abstract Classes To Interfaces

You probably know that in Delphi it is possible to create an *abstract class*, a class with one or more abstract virtual methods. An abstract virtual method is a virtual method which is declared but not defined. The definition of such a method becomes compulsory for derived classes (unless it is fine for these derived classes to be abstract as well). Actually, the implementation of abstract classes in Delphi differs from the implementation common in other object oriented languages: the compiler will allow you to create instances of these classes and only issues a warning. In general, it is meaningless to create an object of an abstract class.

Interfaces are a new feature introduced in Delphi 3. They allow the definition of a sort of *pure abstract class*, a class which has only virtual abstract methods. This definition is not precise though: interfaces are not classes and can have properties.

They are not classes because they are considered to be totally separate elements, with their own common base interface, `IUnknown`, which has the same role as `TObject` for classes. They not only have methods but can have properties mapped to those methods. Of course, interface properties cannot map to data (as class properties can) simply because interfaces cannot have any data. Besides this, they cannot have any code, or any implementation. As the name implies, they only provide an *interface*.

Advantages Of Interfaces

Borland introduced interfaces in Delphi to support COM programming. This is why interface types inherit from `IUnknown`. However, interfaces have some distinct advantages which can become useful for non-COM programming.

A class can inherit from a single base class, but can also implement multiple interfaces. The drawback is that a class which implements an interface must also provide the implementation for each of the methods of the interface.

Interface type objects are reference counted and automatically destroyed when there are no more references to the object. This mechanism is similar to how Delphi manages long strings and offers almost automatic memory management.

The VCL already provides a few base classes to implement the basic behavior required by the `IUnknown` interface. In a program which doesn't export COM objects you can use the `TInterfacedObject` class.

To function properly, each interface requires a numeric ID, like the one in Listing 1. In theory these should be unique GUIDs (generated in the Delphi editor by pressing `Ctrl+Shift+G`) but if you don't plan to export these objects any number will do.

Once you've declared an interface you can define a class which implements it, as in Listing 2.

As I mentioned, this class can derive from `TInterfacedObject` to inherit the implementation of the `IUnknown` methods. Although it is not compulsory to implement interface methods with virtual methods, this is the only approach you can use if you want to be able to modify these methods in further subclasses.

➤ *Listing 1*

```
type
  ICanFly = interface
    ['{10000000-0000-0000-0000-000000000000}']
    function Fly: string;
  end;
```

➤ *Listing 2*

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

Now that we have defined an implementation of the interface, we can write as usual:

```
var
  Airplane1: TAirplane;
begin
  Airplane1 := TAirplane.Create;
  Airplane1.Fly;
  Airplane1.Free;
```

But we can also use an interface type variable:

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

As soon as you assign an object to an interface variable, Delphi automatically checks to see if the object implements that interface, using a special version of the `as` operator. You can explicitly express this operation as follows:

```
Flyer1 :=
  TAirplane.Create as ICanFly;
```

In both cases, Delphi does one extra thing: it calls the `_AddRef` method of the object, increasing its reference count. At the same time, as soon as the `Flyer1` variable goes out of scope, Delphi calls the `_Release` method, which decreases the reference count, checks whether the reference count is zero, and eventually destroys the object. For this reason, in the code fragment above there is no code to free the object we've created.

In other words, in Delphi 3, objects referenced by interface variables are reference-counted and they are automatically deallocated when no interface variable refers to them any more.

There are some further rules for interfaces, but I'll cover them during the development of the example presented in this article. The only important thing to keep in mind is that the type checking for interface types looks at their interface IDs, not at the type names (as for a plain class).

Multiply Inherited Creatures

For this article I've heavily expanded an example from one of my books, *Mastering Delphi 3*, which is a sort of standard example for multiple inheritance. Suppose you have a hierarchy of living creature classes. You can arrange the animals following the standard classifications (with categories such as mammals, birds and insects) or you can arrange them by capability (with categories such as flying animals, quadrupeds, bipeds, meat-eaters and so on).

There is no easy way to express such a complex hierarchy with single inheritance. You can use multiple inheritance if the language you are using supports this feature, or you can use interfaces. This is what I've done in my example, which represents a rather common case study for multiple inheritance. As you can see in Table 1, the example has both a hierarchy of interfaces and a hierarchy of classes.

Classes:
```
TObject
  TInterfacedObject
    TAnimal
      TMammal
        TBat
        TMonkey
      TBird
        TEagle
        TPenguin
        TDuck
```
Interfaces:
```
IUnknown
  IAnimal
    IMammal
    IBird
    ICanFly
    ICanSwim
    ICanWalk
```

➤ *Table 1*

Both the hierarchy of classes and the hierarchy of interfaces actually use single inheritance. It is only if you look at how classes implement the various interfaces that the two hierarchies actually merge, as represented in Figure 1.

The declarations of these interfaces and their methods are quite long, but they are worth looking at, since they constitute the key element of the program. They are shown in Listing 3.

The implementation of these methods are totally trivial: they all return output strings with a description. Now that we have designed this infrastructure, how can we use it? How do we create objects of these classes and how is it possible to use polymorphism in classes implementing multiple interfaces?

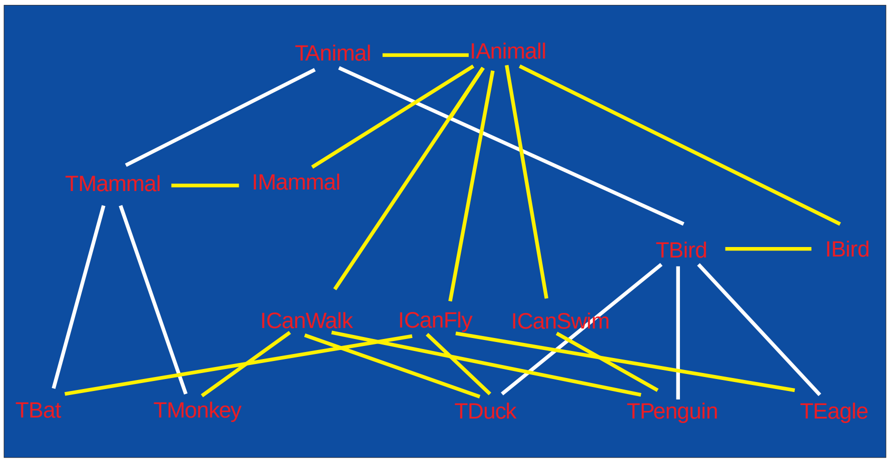
Approaches To Polymorphism

There are actually different ways to use polymorphism with interfaces. You can declare and initialize an array of objects and extract the various interfaces from an object, or you can directly use an array of interfaces.

In the example I have implemented both approaches, by declaring and filling two arrays inside a form:

```
private
  Animals:
    array [1..5] of TAnimal;
  AnimIntf:
    array [1..5] of IAnimal;
```

➤ *Figure 1*



```
type
  IAnimal = interface
    ['{248CC900-64CB-11D1-98D1-004845400FAA}']
    function Kind: string;
  end;
  ICanFly = interface (IAnimal)
    ['{248CC901-64CB-11D1-98D1-004845400FAA}']
    function Fly: string;
  end;
  ICanWalk = interface (IAnimal)
    ['{248CC902-64CB-11D1-98D1-004845400FAA}']
    function Walk: string;
  end;
  ICanSwim = interface (IAnimal)
    ['{248CC903-64CB-11D1-98D1-004845400FAA}']
    function Swim: string;
  end;
  IMammal = interface (IAnimal)
    ['{248CC904-64CB-11D1-98D1-004845400FAA}']
    function CarryChild: string;
  end;
  IBird = interface (IAnimal)
    ['{248CC905-64CB-11D1-98D1-004845400FAA}']
    function LayEggs: string;
  end;
  TAnimal = class (TInterfacedObject, IAnimal)
    function Kind: string; virtual; abstract;
    destructor Destroy; override;
  end;
  TMammal = class (TAnimal, IMammal)
    function CarryChild: string; virtual;
  end;
  TBird = class (TAnimal, IBird)
    function LayEggs: string; virtual;
  end;
  TEagle = class (TBird, ICanFly)
    function Kind: string; override;
    function Fly: string; virtual;
  end;
  TPenguin = class (TBird, ICanWalk, ICanSwim)
    function Kind: string; override;
    function Walk: string; virtual;
    function Swim: string; virtual;
  end;
  TDuck = class (TBird, ICanWalk, ICanFly, ICanSwim)
    function Kind: string; override;
    function Walk: string; virtual;
    function Fly: string; virtual;
    function Swim: string; virtual;
  end;
  TBat = class (TMammal, ICanFly)
    function Kind: string; override;
    function Fly: string; virtual;
  end;
  TMonkey = class (TMammal, ICanWalk)
    function Kind: string; override;
    function Walk: string; virtual;
  end;
```

➤ *Listing 3*

Calling the methods described in the `IAnimal` interface and implemented in the `TAnimal` class is straightforward (Listing 4).

The two calls are executed when you press the top buttons in the left and right columns of the form, as shown in Figure 2. I won't really discuss the user interface of this program, because it is really as simple as it can be: many buttons for the user to press and a memo component for the program to display the output.

If accessing the base class methods is trivial in both cases, it is also quite easy to access the methods of specific interfaces supported by the objects. Of course, every time the program must check if the object actually supports that specific interface. There are several ways to accomplish this. Let me first focus on the techniques you can use when accessing the array of objects, then I'll discuss using the array of interfaces.

The `TObject` class has a method we can use to check if an object supports a given interface: `GetInterface` (Listing 5). If the interface is available, this function returns `true` and sets the value of the interface variable passed as the second parameter. The first parameter is the ID of the interface, or the interface type. As an alternative you can check the interface parameter after the call, as shown in Listing 6.

Actually there is an easier approach, which is to use the `as` operator to cast the object to the required interface (Listing 7). The problem is that, in this case, if the conversion fails, Delphi triggers an exception.

There's another problem: when you extract an interface from an object, Delphi activates reference counting for the object. Since reference counting is used only for interfaces referring to an object and not for plain variables, the first time you use an interface to call a method of an object, the reference count is first increased and then decreased. This surprisingly results in the object being destroyed! To solve the problem you should add an initial reference to an object after creating it. This can be done in the constructor or simply in the start- up code:

```
for I := 1 to 5 do
  Memo1.Lines.Add(Animals[I].Kind);
  // or:
  Memo1.Lines.Add(AnimIntf[I].Kind);
```

➤ *Listing 4*

```
var
  Swim1: ICanSwim;
...
  if Animals[i].GetInterface(
    ICanSwim, Swim1) then
    Memo1.Lines.Add(Swim1.Swim);
```

➤ *Listing 5*

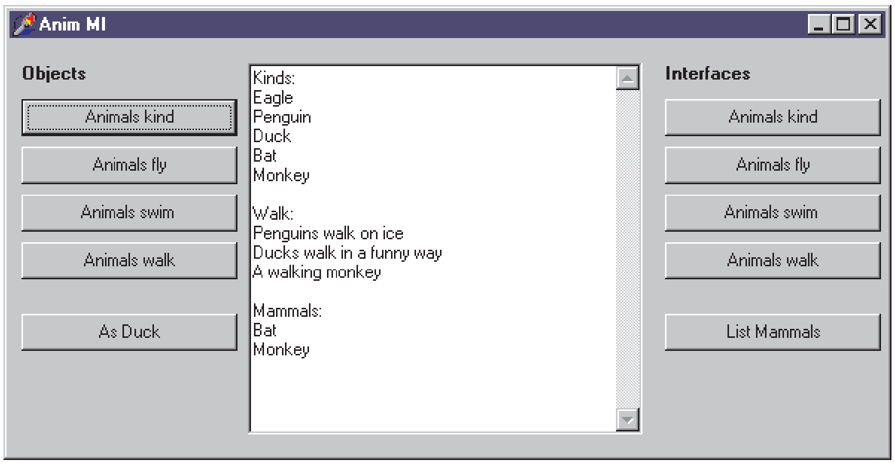
```
  Animals[i].GetInterface(
    ICanFly, Fly1);
  if Assigned (Fly1) then
    Memo1.Lines.Add (Fly1.Fly);
```

➤ *Listing 6*

```
try
  Walker1 := Animals[i] as ICanWalk;
  Memo1.Lines.Add (Walker1.Walk);
except;
end;
```

➤ *Listing 7*

➤ *Figure 2*



```
Kinds:
Eagle
Penguin
Duck
Bat
Monkey

Walk:
Penguins walk on ice
Ducks walk in a funny way
A walking monkey

Mammals:
Bat
Monkey
```

```
for I := 1 to 5 do
  (Animals[I] as IAnimal)._AddRef;
```

Writing this code in the constructor seems a better idea at first sight, but we would have problems using the objects through interfaces, so I've decided not to use the constructor in the example.

To debug these kind of problems (which usually result in nasty memory errors and system crashes) you can un-comment the code in the destructor of the TAnimal class, which shows a message box for every object being destroyed. I've commented out this code in the final version of the program because getting ten message boxes when the program terminates is really annoying.

Using An Array Of Interface Variables

As I've already discussed, the alternative to using an array of objects is to use an array of interfaces. The program extracts the IAnimal interface from newly created objects to initialize this array. This is done automatically by Delphi when you write:

```
AnimIntf[1] :=
  TEagle.Create;
```

which is the same as writing:

```
AnimIntf[1] :=
  TEagle.Create as IAnimal;
```

This time the program can check whether an interface is supported by calling the QueryInterface method instead of GetInterface (Listing 8).

Again, the alternative is to check the return value of the method (Listing 9).

We can also use the as statement with a try-except block, exactly as we did before with the array of objects. What is different is that if we use interfaces we cannot simply use the is statement even to test if an object inherits from the TMammal class. Also in this case we will have to use an interface-related approach. The code shown in Listing 10 is used to list the mammals (inside a for loop).

In other words, if you decide to use interfaces you might have to add some extra features to them, since you won't be able to directly access the actual object which the interface relates to, unless you use one of the interfaces implemented by the object.

On the other hand, interfaces have the clear advantage of being reference counted and requiring no support for freeing unused objects. In a complex program this can be really helpful.

Is This Multiple Inheritance?

The code fragments I've discussed show that we can use an object and cast it to the multiple interfaces it supports. In other words, we can consider an eagle as a bird or as a flying animal and call methods of both interfaces for a single object. We can also cast an object to two different base types. So, this really is like multiple inheritance.

What we don't get is the inheritance of the actual implementation of the methods. There is no code in the ICanFly interface and some eventual common code must be re-implemented in each class that supports this interface. However, this also solves many typical problems with multiple inheritance in C++: the example uses a diamond shaped inheritance graph, there is no need to figure out how each class inherits from the repeated ancestors (and no need to use what C++ calls virtual base classes). This is not only simpler for programmers, but also for the compiler.

As I mentioned at the beginning, Borland added interfaces to Delphi 3 to support Microsoft's COM, but they can really be used as an extra language feature. The only annoying element is that interfaces must have an ID, even for internal objects, because the type checking of interfaces depends on these numbers.

The other minor problem is that there isn't an is operator to check whether an object supports a given interface, but as we've seen it is very simple to mimic this behavior with a single method call.

Summing up, does it really make sense to use interface types and variables in a program that doesn't require COM support? My personal opinion is that, if the program is designed around a complex hierarchy which might benefit from multiple inheritance, then the answer is yes. If, after considering the extra complexity of this design, you disagree, however, I'll fully understand!

Marco Cantù is the author of *Mastering Delphi 3* and the advanced *Delphi Developer's Handbook*. When he's not writing books or articles, he teaches Delphi training classes and speaks at many conferences worldwide. You can contact Marco through his website at www.MarcoCantu.com

➤ *Listing 8*

```
AnimIntf[i].QueryInterface (ICanFly, Fly1);
if Assigned (Fly1) then
  Memo1.Lines.Add (Fly1.Fly);
```

➤ *Listing 9*

```
if AnimIntf[i].QueryInterface (ICanSwim, Swim1) <> E_NoInterface then
  Memo1.Lines.Add (Swim1.Swim);
```

➤ *Listing 10*

```
AnimIntf[i].QueryInterface (IMammal, Mam1);
if Assigned (Mam1) then
  Memo1.Lines.Add (Mam1.Kind);
```