The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project? Just email Brian Long, our Delphi Clinic Editor, on clinic@blong.com

RTTI And Module Boundaries

Q I have a project comprising an EXE and some DLLs. One of the DLLs exports a routine that takes a `TObject` parameter. This object can potentially be one of a number of different class types so I chose the most basic common ancestor type of `TObject`. The routine is called by code in the EXE, and from some of the other DLLs. To identify the type of object passed, the routine uses the `is` operator (Listing 1). However, in the cases where the object is not created by the DLL that implements the routine, the `is` operator always returns `False`. Why is this?

A Answering this question relies upon knowledge of what the `is` operator does to produce a result. With this knowledge, it should be fairly clear why it fails.

Rather than just looking at `is` in isolation, we will also look at the `as` operator, so we can see exactly what both of these operators do when they are called.

The `as` operator is used for safe typecasting of objects defined in terms of some generic object to a more specific type inherited from that generic type. For example, in an `OnClick` event handler, the `Sender` parameter is defined to be a `TObject`. `Sender` represents the component that triggered the event, and so to access that object and manipulate its specific properties, it needs to be cast to a descendant type. To ensure you get the right type, you employ the `as` operator, for example:

```
(Sender as TButton).Hide
```

If the object represented by `Sender` is a `TButton`, or inherits from `TButton`, the object is cast to a `TButton` and its `Hide` method is called. If this is not the case an exception is raised. The statement above is functionally equivalent to the code in Listing 2. This condition (or a brief assembler version of it) is executed by the `_AsClass` routine in the `System` unit.

This routine itself is not responsible for raising the exception. Instead, if the `SysUtils` unit has been used in the project, a global variable (`ErrorProc`) will point to its `ErrorHandler` procedure, which is called with a value of `reInvalidCast` (10). If `SysUtils` has not set up the `ErrorProc` pointer, `RunError(219)` is called instead.

Obviously, some of these details are completely irrelevant to most people. What is important here is that the `as` operator is implemented with a call to the `is` operator (or at least by executing the same code that the `is` operator executes).

Now let's look at the `is` operator. The first thing to note is that it needn't really exist at all. The `Boolean` expression `Sender is TButton` is functionally identical to this class method call:

```
Sender.InheritsFrom(TButton)
```

Most people have not encountered the `TObject` class function `InheritsFrom`. It does exactly the same as the `is` operator, but is slightly more flexible. The `is` operator can only determine the inheritance relationship of an object (you must place an object reference to the left of `is`) whereas `InheritsFrom` can test relationships between class references. For example, this is a valid `Boolean` expression which will evaluate to `True`:

```
TBitBtn.InheritsFrom(TButton)
```

The implementation of a call to `is` or `InheritsFrom` is straightforward. It involves checking the class type of the item in question and comparing it with the type passed in. If they don't match, you look at the ancestor class of the item. If there is no match, you look at that class's ancestor, and so on. Ultimately, there will either be a match (so `True` is returned) or you will get to type `TObject`, which has no ancestor (`False` is returned). Listing 3 shows the implementation of a function that does the same job.

It might seem like the original question has been long forgotten by now, but Listing 3 helps us to reach an understanding of why the `is` operator fails under the circumstances outlined in the question.

```
procedure ARoutine(Obj: TObject);
begin
  if Obj is TFoo then
    TFoo(Obj).Bar
end;
```

➤ *Listing 1: An 'is' test that fails for a reader.*

➤ *Listing 2: An expanded version of an 'as' expression.*

```
if Sender is TButton then
  TButton(Sender).Hide
else
  raise EInvalidCast.Create(
    'Invalid class typecast')
```

➤ *Listing 3: A function that does the same as 'is'.*

```
function IsA(Obj: TObject;
  Cls: TClass): Boolean;
var
  ObjCls: TClass;
begin
  Result := True;
  { Get class reference
    for this object }
  ObjCls := Obj.ClassType;
  while Assigned(ObjCls) do
    { Check class reference
      against that passed in }
    if ObjCls = Cls then
      Exit
    else
      { Get ancestor class }
      ObjCls := ObjCls.ClassParent;
  Result := False
end;
```

Remember that a DLL routine has a Delphi object passed into it from another module (EXE or DLL). The `is` operator is used to find out whether the object is of a certain type and fails, even when perhaps it shouldn't. This happens because of what is being compared. The code is comparing a `TClass` entity extracted either from a call to an object's `ClassType` method, or from a class's `ClassParent` method.

`TClass` is a generic class reference type that can hold references to any class type, so the code is comparing class references. A class reference is implemented as a pointer to the VMT (virtual method table, where virtual method addresses are listed) of the class in question.

When an executable creates an object of some arbitrary type `TFoo`, it can only work if the executable's binary image contains the implementation of `TFoo`'s methods and also `TFoo`'s VMT. If a DLL has knowledge of a type called `TFoo`, the implication is that the implementation and VMT of `TFoo` are compiled into the DLL's binary image. So in the application as a whole, there are two implementations of `TFoo`, one in the EXE and one in the DLL.

Whenever DLL code refers to `TFoo`, it will be referring to its own implementation of it. The failing operation is one that asks an object whose type is the executable's `TFoo` if it is an object whose type is the DLL's `TFoo`. Since the two `TFoo` implementations have their own VMTs (as well as implementations), a match is never found.

In many cases this will be a very sensible outcome. If the executable is compiled in Delphi 2, but the DLL is compiled in Delphi 5, then the binary implementation of `TFoo` will differ wildly between the two compiled files. A result that says that the executable's `TFoo` object is not of the type represented by the DLL's `TFoo` is probably correct.

I assume that the questioner is compiling the DLL and EXE in the same version of Delphi and so, whilst the issue is technically valid, still wishes to avoid the problem.

Two ways of dealing with the problem occur to me. One would be to use packages instead of DLLs. Packages will help reduce the size of all the binary files, sharing any implementation of a routine or class with all other modules, and will make them all semantically one application, with no logical module divisions. Consequently, there will be only one implementation of `TFoo` present and the `is` operator will be forced to return `True` where expected.

```
procedure ARoutine(Obj: TObject);
begin
  if CompareText(
    Obj.ClassName, 'TFoo') = 0 then
    TFoo(Obj).Bar
end;
```

Listing 4: A class name comparison instead of a class type comparison.

The alternative is to recognise why the problem is arising, and perform an alternative test. Since `is` uses class type information (VMT addresses), which will always fail with an object created in a different module, perhaps you could just compare the class name instead. So, instead of something like Listing 1, you could try the alternative test in Listing 4.

In fact, there is a VCL routine which does just this type of class name comparison to overcome exactly the same problem. When implementing drag and drop in an application, you can make use of special `TDragObject` descendants (called drag objects) that are created in a dragged control's `OnStartDrag` event handler. In the target `OnDragOver` and `OnDragDrop` event handlers, the `Source` parameter will not be the dragged source control, but the custom drag object. The online help advises you to use `IsDragObject` to verify if `Source` represents a drag object or not.

At first glance, you'd expect this function to be implemented as shown below, but it is not:

```
Result := Source is TDragObject
```

Because drag objects can cross DLL/EXE boundaries, the implementation actually does class name comparison, as shown in Listing 5. For more on drag and drop in Delphi applications, look out for the first part of a series of articles on the subject next month.

```
function IsDragObject(
  Sender: TObject): Boolean;
var SenderClass: TClass;
begin
  SenderClass := Sender.ClassType;
  Result := True;
  while SenderClass <> nil do
    if SenderClass.ClassName =
      TDragObject.ClassName then
      Exit
    else
      SenderClass :=
        SenderClass.ClassParent;
  Result := False;
end;
```

Listing 5: The IsDragObject function which works across DLL/EXE boundaries.

Object Inspector Refuses To Make Event Handler

Q I've just made a component that has an event property of a type that didn't exist before. Everything works fine, except that I cannot get the Object Inspector to correctly create a new event handler for it in the editor. It creates the event handler, but then says it is incompatible with the event. I assume I need a property editor for it. I've done some of these before, but never one for an event and I can't seem to find any examples anywhere.

A The Object Inspector is all set up to manufacture event handlers for published properties which have method types (events, in other words). It relies upon the presence of the `TMethodProperty` property editor that is defined in the `DsgnIntf` unit and is used as the default editor for all published method properties. This property editor has built-in knowledge of how to manufacture an event handler for all published events.

If you have published a property that represents an event, then event handlers should be manufactured and assigned automatically in all but a small subset of cases. The problem is likely to arise from using untyped `var` and `const` parameters. The IDE code that checks published methods for compatibility with events relies upon method types to do its job.

Untyped parameters cause it to assume (perhaps wrongly) that it cannot make a full enough check against other methods and so it produces the error. Of course, there is nothing to stop you assigning the method to the event property in code, which will work fine.

A possible way of avoiding this problem would be to use generic pointer parameters (of type `Pointer`) instead of `var` parameters. This way, you are supplying a type, but you are still not committing yourself to what type of data is being referred to. The component user who uses such an event handler will still need to typecast the parameter to an appropriate type. So, for example, an event handler that was intended to look like Listing 6 can instead look like Listing 7.

That said, it is usually a good idea to ensure that information you pass to event handlers is unambiguous and leaves no room for error from uneducated users. Untyped `var` and `const` parameters (as well as untyped pointers) have a potential for misuse, due to their inherently ill-defined nature.

This requirement for event handlers to be sort of foolproof is why the Delphi manual advises against defining event properties that use function methods (as opposed to procedures). Functions return a value, but their default return value is undefined (and typically garbage). A user who forgets to assign a return value will inadvertently cause garbage to be returned. Better to use a typed `var` parameter, as many of the supplied component events do.

TStrings And Object Owners

Q I have a `TStringList` object that I am using for storage of strings and objects (I use the `AddObject` method). I have a loop that runs through the records in a table, and each iteration in the loop sets fields in an instance of a custom object I have. At the end of each loop iteration, I call the stringlist's `AddObject` method to add a string value and the object into the list, then go onto the next iteration.

After the table has been read, I start another loop running through each entry in the stringlist, adding the string and information from the object to cells in successive rows of a stringgrid component. Can you tell me why I only see the last entry made into the list, repeated across the rows in the grid?

A The problem in this case is a lack of understanding of Delphi objects, object references and the `Objects` property of the `TStrings` class (and descendants such as `TStringList`). To start with, when you have an instance of any Delphi object, the variable that you use is (despite its appearance) a pointer to the object. So, a declaration such as:

```
var
  Foo: TFoo;
```

is actually declaring a four-byte pointer variable that can hold the address of an object. Without any typecasting, this particular pointer (or *object reference* as it is called) can be told to refer to an instance of `TFoo` or any class inherited from `TFoo`. When you create an object for the object reference to point to, using a statement like

```
Foo := TFoo.Create;
```

it gets assigned a value. The right-hand side of the expression is evaluated, causing the `TFoo` instance to be created. The expression evaluates to the address of the object (an object reference value) and this value is assigned to the object reference variable on the left-hand side of the assignment. The programmer must free this object.

Despite the fact that the object reference is a pointer to an object instance, Borland made sure that programmers do not have to use pointer syntax for accessing objects. When you refer to, for example, `Button1.Caption`, the compiler causes the object reference to be de-referenced so that the `Caption` property of the object itself can be evaluated. The same is true of 32-bit Delphi strings, interface references (Delphi 3 and later) and dynamic arrays (Delphi 4 and later). These are also internally represented as pointers which are automatically de-referenced.

All objects must be freed in some way. Components (objects of type `TComponent` or classes inherited from `TComponent`) provide a mechanism for the programmer to delegate responsibility of freeing the component to another component. This other component (which is specified as a parameter to the constructor of the component in question) becomes the component's owner and, during its own destruction, will destroy (or free) all components it owns.

Components placed on a form at design-time are owned by the form. When the form is destroyed, all the components on the form are also destroyed. All of the auto-created forms are owned by

➤ *Listing 6: An event with an untyped var parameter (and problems for the Object Inspector).*

```
TDataType = (dtShortString, dtDouble, dtInteger);
...
procedure TForm1.TheComponentEventHandler(DataType: TDataType; var Data);
begin
  case DataType of
    dtShortString: ShowMessage(ShortString(Data));
    dtDouble: ShowMessage(FloatToStr(Double(Data)));
    dtInteger: ShowMessage(IntToStr(Integer(Data)));
  end
end;
```

➤ *Listing 7: An event using a pointer parameter.*

```
procedure TForm1.TheComponentEventHandler(DataType: TDataType; Data: Pointer);
begin
  case DataType of
    dtShortString: ShowMessage(ShortString(Data^));
    dtDouble: ShowMessage(FloatToStr(Double(Data^)));
    dtInteger: ShowMessage(IntToStr(Integer(Data^)));
  end
end;
```

```
type
  TRecordDesc = class(TObject)
  public
    CustNo,
    State: String;
  end;
procedure TForm1.FormCreate(Sender: TObject);
var
  List: TStrings;
  RecordDesc: TRecordDesc;
  Loop: Integer;
begin
  List := TStringList.Create;
  try
    tblCustomer.Open;
    { Loop through table records }
    while not tblCustomer.Eof do begin
      { Create new object instance }
      RecordDesc := TRecordDesc.Create;
      try
        { Set up object data fields }
        RecordDesc.CustNo :=
          tblCustomer.FieldByName('CustNo').AsString;
        RecordDesc.State :=
          tblCustomer.FieldByName('State').AsString;
        { Let string list look after object for a while }
        List.AddObject(tblCustomer.FieldByName(
          'Company').AsString, RecordDesc);
      except
        RecordDesc.Free;
      end;
      tblCustomer.Next
    end;
    tblCustomer.Close;
    Grid.RowCount := List.Count + 1;    { Initialise grid }
    Grid.ColCount := 3;
    Grid.Cells[0, 0] := 'Customer No.';
    Grid.Cells[1, 0] := 'Company';
    Grid.Cells[2, 0] := 'State';
    { Loop through string list setting up grid rows }
    for Loop := 0 to List.Count - 1 do begin
      Grid.Cells[0, Loop + 1] :=
        TRecordDesc(List.Objects[Loop]).CustNo;
      Grid.Cells[1, Loop + 1] := List[Loop];
      Grid.Cells[2, Loop + 1] :=
        TRecordDesc(List.Objects[Loop]).State;
    end;
  finally
    for Loop := 0 to List.Count - 1 do
      List.Objects[Loop].Free;  { Delete objects in list }
    List.Free    { Delete list }
  end
end;
```

➤ *Listing 8: Adding objects to a string list and using them.*

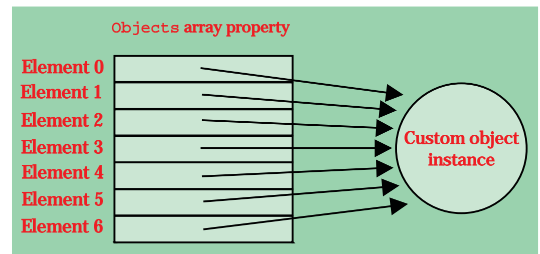
the `Application` object. When the program closes and the `Application` object gets destroyed, all forms are destroyed, which then destroys the components on them.

Apart from the cases of owned components (and forms), all objects created by the programmer must be freed by the programmer. This is one of the cardinal rules of Delphi programming.

As the questioner has seen, a `TStringList` has a way of storing an object in association with each string in the list. This is done by calling the `AddObject` method instead of `Add`. `AddObject` takes a string value and an object reference (declared as type `TObject`, so any Delphi object reference can be passed along). The string is stored in the `Strings` array property and the object reference is stored in the `Objects` array property.

The key point about the `Objects` array property is that it is an array of `TObject` references. In other words, it is an array of addresses. It keeps a record of the object reference you pass in, and that is all it stores. It does not 'copy' the object passed in, in any way. It merely records that object's address.

The question suggests that there is an instance of a custom object, with various data fields set with values on each iteration of a loop. After all the data fields are set, the object is passed (with a string) to the `AddObject` method of a string



➤ *Figure 1: Multiple object references all referring to the same object.*

list. This causes the address of the custom object to be stored in the `Objects` array; this object reference will still refer to the original instance of the custom object.

The next iteration of the loop replaces the data fields in the custom object with new values describing the next record in the table and passes its address to `AddObject`. The remaining iterations do the same again.

The end result is that the `Objects` array is filled with duplicate values. Each entry will hold the address of the original custom object instance whose data fields have information describing (since the table-reading loop has finished) the last record in the table. Figure 1 tries to show what is going on with the `Objects` array property.

Looping through the stringlist's entries and accessing the value of the `Objects` array will simply return you the custom object instance. Consequently, if you are using the object returned from the `Objects` array property as a basis for filling the cells in a string grid row, you will find that all the rows look the same, because you are reading from the same object for each row.

So the initial problem is that if you want to store multiple objects in a `TStringList`, you must create as many instances as you want to store. Each instance should be passed to `AddObject`, so the string list can hold object references for each individual object instance. Listing 8 shows some sample code that does this (from project GridEg.Dpr on this month's disk).

When you do this, you must remember to add code to loop through the stringlist destroying any objects you have given it. The stringlist takes no responsibility for destroying these objects. Since the programmer created them, the programmer must free them. It is a pity that the help for the `Objects` property of a `TStringList` doesn't mention this: it's a common mistake to leave objects unfreed in stringlists. There's some sample code at the end of Listing 8.

The `TStringGrid` component actually has an `Objects` array property of its own. The content of each cell is normally accessed through the `Cells` array property. The `Objects` property allows each cell to have an associated object (again, by storing object references). The help for this property does correctly emphasise the situation: *'The string grid does not own the objects in the Objects array. Objects added to the Objects array still exist even if the string grid is*

destroyed. They must be explicitly destroyed by the application.'

Because of this lack of ownership, the `Objects` array property is often used for other purposes. You can consider the `Objects` array property as an array of 4-byte storage locations (each object reference is 4 bytes in size). If you want to hold, for example, a `Longint` in association with each string then you can, using a typecast:

```
List.AddObject('A string',
  TObject(57));
```

will add an integer value in the place of an object reference. This statement extracts an `integer` and assigns it to an `integer` property:

```
Tag:=Longint(List.Objects[0]);
```

ListView Checkbox Changes

Q I seem to be having a problem detecting when the check status changes for an item in the `ListView`. I have tried looking at the `OnChange` event and interrogating the `TItemChange` parameter but that doesn't help me.

A When a `ListView` (Delphi 3 onwards) has its `Checkboxes` property set to `True`, each list item has a checkbox on its left. The user can check these checkboxes as they like, and program code can examine the `Checked` property of the `TListItem` objects in the `ListView` to see which ones are checked. However, the component does not have an event which triggers when a checkbox is checked, so it's difficult to write code that reacts immediately to this.

The `OnChange` or `OnChanging` event handlers can be used to detect checkbox changes. If the `Change` parameter (type `TItemChange`) has a value of `ctState`, it indicates that either the `Cut`, `Selected`, `Focused` or `Checked` property of the list item passed as the `Item` property has changed or it is about to change. But, the `OnChange` event handler doesn't tell you which of the four properties has changed and the `OnChanging` event handler doesn't tell you which property is about to change. The `ListView` component fires the `OnSelectItem` event after the `OnChange` event, when the `Selected` property of the list item changes, but there is no dedicated indication for the other properties.

However, you could record the state of the `Checked` property in the `OnChanging` event handler and do a comparison in the `OnChange` event handler. A `Boolean` form data field can be used to record this information. You can use code like that in Listing 9 (from project ListViewEg.Dpr on the disk).

Instead of doing this recording and comparison in the event handlers of a `ListView` component, you could move the extra code into a new component with a dedicated event for the job. As a component user, you customise a component's functionality by making event handlers for available events. As a component writer, you need to take a step back. You don't make event handlers, but override the polymorphic routines which trigger events in the first place. The `OnChange` event is triggered from within the polymorphic `Change` method. `OnChanging` is triggered from the `CanChange` method. With this information, we

➤ *Listing 9: Detecting when a ListView's checkboxes are toggled.*

```
TForm1 = class(TForm)
  ...
private
  FListItem: TListItem;
  FChecked: Boolean;
end;
...
procedure TForm1.ListView1Changing(Sender: TObject; Item: TListItem;
  Change: TItemChange; var AllowChange: Boolean);
begin
  if Change = ctState then begin
    FListItem := Item;
    FChecked := Item.Checked
  end
end;
procedure TForm1.ListView1Change(Sender: TObject; Item: TListItem;
  Change: TItemChange);
const
  BooleanIdents: array [Boolean] of string = ('False', 'True');
begin
  if (Change = ctState) and (Item = FListItem) and
    (Item.Checked <> FChecked) then begin     //A checkbox has been toggled
    ListBox1.Items.Add(Format('Item %d (%s): Checked = %s',
      [Item.Index, Item.Caption, BooleanIdents[Item.Checked]]));
    ListBox1.ItemIndex := ListBox1.Items.Count - 1
  end
end;
```

➤ *Listing 10: An enhanced ListView component.*

```
unit ListViewEx;
interface
uses Classes, ComCtrls;
type
  TItemCheckEvent = procedure (Sender: TCustomListView;
    Item: TListItem; Checked: Boolean) of object;
  TListViewEx = class(TListView)
  private
    FOnCheck: TItemCheckEvent;
    FChecked: Boolean;
    FListItem: TListItem;
  protected
    function CanChange(Item: TListItem; Change: Integer):
      Boolean; override;
    procedure Change(Item: TListItem; Change: Integer);
      override;
  published
    property OnCheck: TItemCheckEvent read FOnCheck
      write FOnCheck;
  end;
procedure Register;
implementation
uses CommCtrl;
procedure Register;
begin
  RegisterComponents('Clinic', [TListViewEx]);
end;
function TListViewEx.CanChange(Item: TListItem; Change:
  Integer): Boolean;
begin
  Result := inherited CanChange(Item, Change);
  if Result and (Change = LVIF_STATE) then begin
    FListItem := Item;
    FChecked := Item.Checked
  end
end;
procedure TListViewEx.Change(Item: TListItem; Change:
  Integer);
begin
  inherited;
  if (Change = LVIF_STATE) and (Item = FListItem) and
     (Item.Checked <> FChecked) and Assigned(FOnCheck) then
    FOnCheck(Self, Item, Item.Checked)
end;
end.
```

```
unit ListViewEx2;
interface
uses
  Windows, Messages, Classes, Controls, ComCtrls, CommCtrl;
type
  //This is a variation on the wm_Notify message record
  //that is used for certain ListView notification messages
  TWMListViewNotify = packed record
    Msg: Cardinal;
    IDCtrl: Longint;
    NMLV: PNMListView;
    Result: Longint;
  end;
  TItemCheckEvent = procedure (Sender: TCustomListView;
    Item: TListItem; Checked: Boolean) of object;
  TListViewEx2 = class(TListView)
  private
    FOnCheck: TItemCheckEvent;
  protected
    procedure CNNotify(var Msg: TWMListViewNotify);
      message cn_Notify;
  published
    property OnCheck: TItemCheckEvent
      read FOnCheck write FOnCheck;
end;
procedure Register;
```

```
implementation
procedure Register;
begin
  RegisterComponents('Clinic', [TListViewEx2]);
end;
procedure TListViewEx2.CNNotify(var Msg: TWMListViewNotify);
const
  OldChecked: Boolean = False;
  OldItem: Integer = -1;
begin
  with Msg.NMLV^ do
    case hdr.code of
      LVN_ITEMCHANGING:
      begin
        Olditem := iItem;
        OldChecked := Items[OldItem].Checked;
      end;
      LVN_ITEMCHANGED:
        if (iItem = OldItem) and (Items[iItem].Checked <>
          OldChecked) and Assigned(FOnCheck) then
          FOnCheck(Self, Items[iItem], Items[iItem].Checked)
    end;
  inherited;
end;
end.
```

➤ *Listing 11: A message-based enhanced ListView.*

can override these two methods and use similar code to that in Listing 9. Listing 10 shows a component implementation, where you can see that the methods have raw Windows constants passed in, rather than Delphi `TItemChange` values like the event handlers.

This component is written in a Delphi-esque fashion, overriding VCL methods. You could take a more Win32-oriented approach, if you know how the underlying Windows `ListView` control deals with having checkboxes toggled.

When a checkbox has its state toggled, the control sends a special notification message (`wm_Notify` passed along with some `ListView`-specific parameters) to its parent. In a Delphi application, if a control sends `wm_Notify` to its parent, a `cn_Notify` message will be sent back to the control with the same parameters. This message we can catch and do what we need to in the message handler.

The notification message is generated just before an item changes (this is how the `CanChange` method gets called, and how the `OnChanging` event arises) and just after an item changes (giving rise to the `Change` method and `OnChange` event). Listing 11 shows the alternative implementation. The two units which implement these enhanced ListView components are included on this month's disk.