How to handle drag and drop in a TWebBrowser control (part 3 of 4)

Two case studies

Case studies

Case Study 1: Inhibiting Drag and Drop

I mentioned in the introduction that one of my goals was to stop web browser controls accepting drag and drop. This case study will show how to do that by creating a "do nothing" implementation of *IDropTarget* named *TNulDropTarget* in the unit `UNulDropTarget`. See *Listing 4*.

```
unit UNulDropTarget;

interface

uses
  Windows, ActiveX;

type

  TNulDropTarget = class(TInterfacedObject, IDropTarget)
  protected
    { IDropTarget methods }
    function DragEnter(const dataObj: IDataObject; grfKeyState: Longint;
      pt: TPoint; var dwEffect: Longint): HResult; stdcall;
    function DragOver(grfKeyState: Longint; pt: TPoint;
      var dwEffect: Longint): HResult; stdcall;
    function DragLeave: HResult; stdcall;
    function Drop(const dataObj: IDataObject; grfKeyState: Longint;
      pt: TPoint; var dwEffect: Longint): HResult; stdcall;
  end;

implementation

{ TNulDropTarget }

function TNulDropTarget.DragEnter(const dataObj: IDataObject;
  grfKeyState: Integer; pt: TPoint; var dwEffect: Integer): HResult;
begin
  dwEffect := DROPEFFECT_NONE;
  Result := S_OK;
end;

function TNulDropTarget.DragLeave: HResult;
begin
  Result := S_OK;
end;

function TNulDropTarget.DragOver(grfKeyState: Integer; pt: TPoint;
  var dwEffect: Integer): HResult;
begin
  dwEffect := DROPEFFECT_NONE;
  Result := S_OK;
end;

function TNulDropTarget.Drop(const dataObj: IDataObject;
  grfKeyState: Integer; pt: TPoint; var dwEffect: Integer): HResult;
begin
  dwEffect := DROPEFFECT_NONE;
  Result := S_OK;
end;
```

```
end.
```

Listing 4

All we do is set the cursor effect to *DROPEFFECT_NONE* in all relevant methods and return *S_OK* from each method.

We now need to make two minor modifications to the *boilerplate* code that was presented earlier: we must use `UNulDropTarget` and alter the *FormCreate* method to create a *TNulDropTarget* object. *Listing 5* shows the changes.

```
uses
  ..., UNulDropTarget, ...

procedure TForm1.FormCreate(Sender: TObject);
begin
  ...
  fWBContainer.DropTarget := TNulDropTarget.Create;
  ...
end;
```

Listing 5

If you run this code you will not be able to drop any dragged object on the web browser control. You can prove this works by commenting out the line in *FormCreate* where *fWBContainer.DropTarget* is assigned. The browser's default drag-drop will be enabled once more.

Case Study 2: Accepting Files and Text Dropped on the Browser Control

This case study will be more complex. We will start with a browser control containing an HTML document that displays some instructions along with a placeholder box where data extracted from dragged and dropped objects can be displayed. We will accept two kinds of data objects:

1. Text dragged and dropped from other applications. This text will be displayed as pre-formatted text.
2. Files dragged and dropped from Explorer. The names of the files will be listed.

Implementing *IDropTarget*

Let us begin by looking at how to implement *IDropTarget*. This code is more complicated than we've seen before for two reasons:

1. We actually have to handle some dropped data – in fact we handle two different kinds of data.
2. We have to communicate with the main form to enable the form to display the data. We could reduce this complexity by implementing *IDropTarget* in the main form, but I prefer to separate the code.

Listing 6 shows the declaration of *TCustomDropTarget*, our class that implements *IDropTarget*.

```
type

  IDropHandler = interface(IInterface)
    ['{C6A5B98C-9D4C-4205-B0C2-3F964938DF26}']
    procedure HandleText(const Text: string);
    procedure HandleFiles(const Files: TStrings);
  end;

  TCustomDropTarget = class(TInterfacedObject, IDropTarget)
  private
    fCanDrop: Boolean;
    fDropHandler: IDrophandler;
    function CanDrop(const DataObj: IDataObject): Boolean;
    function MakeFormatEtc(const Fmt: TClipFormat): TFormatEtc;
    function GetTextFromObj(const DataObj: IDataObject;
      const Fmt: TClipFormat): string;
    procedure GetFileListFromObj(const DataObj: IDataObject;
      const FileList: TStrings);
    procedure DisplayData(const DataObj: IDataObject);
    procedure DisplayFileList(const DataObj: IDataObject);
    procedure DisplayText(const DataObj: IDataObject);
  protected
    { IDropTarget methods }
    function DragEnter(const dataObj: IDataObject; grfKeyState: Longint;
      pt: TPoint; var dwEffect: Longint): HResult; stdcall;
    function DragOver(grfKeyState: Longint; pt: TPoint;
      var dwEffect: Longint): HResult; stdcall;
    function DragLeave: HResult; stdcall;
    function Drop(const dataObj: IDataObject; grfKeyState: Longint;
      pt: TPoint; var dwEffect: Longint): HResult; stdcall;
  public
    constructor Create(const Handler: IDropHandler);
  end;
```

Listing 6

For now, ignore the private methods of *TCustomDropTarget* and the *IDropHandler* interface – they will be explained later. We will start by looking at how the *IDropTarget* methods are implemented. *Listing 7* shows the methods.

```
function TCustomDropTarget.DragEnter(const dataObj: IDataObject;
  grfKeyState: Integer; pt: TPoint; var dwEffect: Integer): HResult;
begin
  Result := S_OK;
  fCanDrop := CanDrop(dataObj);
  if fCanDrop and (dwEffect and DROPEFFECT_COPY <> 0) then
    dwEffect := DROPEFFECT_COPY
  else
    dwEffect := DROPEFFECT_NONE;
end;

function TCustomDropTarget.DragLeave: HResult;
begin
  Result := S_OK;
end;

function TCustomDropTarget.DragOver(grfKeyState: Integer; pt: TPoint;
  var dwEffect: Integer): HResult;
begin
  if fCanDrop and (dwEffect and DROPEFFECT_COPY <> 0) then
    dwEffect := DROPEFFECT_COPY
  else
    dwEffect := DROPEFFECT_NONE;
  Result := S_OK;
end;

function TCustomDropTarget.Drop(const dataObj: IDataObject;
  grfKeyState: Integer; pt: TPoint; var dwEffect: Integer): HResult;
begin
  Result := S_OK;
  fCanDrop := CanDrop(dataObj);
  if fCanDrop and (dwEffect and DROPEFFECT_COPY <> 0) then
  begin
    dwEffect := DROPEFFECT_COPY;
    DisplayData(dataObj);
  end
  else
    dwEffect := DROPEFFECT_NONE;
end;
```

Listing 7

In *DragEnter* we check if we can handle the dragged data by calling the *CanDrop* method and recording its result for later use. If we can handle the data we intend to instruct Windows to display the copy cursor by setting *dwEffect* to *DROPEFFECT_COPY*. However before we can do this we must check that the caller supports copying by checking that *dwEffect* includes *DROPEFFECT_COPY*. Should we not be able to handle the data, or the caller doesn't support copying, we inhibit the drop by setting *dwEffect* to *DROPEFFECT_NONE*.

About *IDropTarget* & *IDataObject*

If you are not comfortable working with these interfaces please see "*How to receive data dragged from other applications*" which explains how to implement *IDropTarget* and use the methods of *IDataObject*. I won't spend much time explaining this here.

In *DragOver* we test *fCanDrop* and again check that the caller supports copying before setting *dwEfect*.

We need do nothing in *DragLeave*, so we simply return *S_OK*.

Finally, in *Drop* we re-check the data object and *dwEffect* to see if we should accept the drop and again set *dwEffect* accordingly. If we are accepting the drop we call *DisplayData* to interpret the data object and to display it.

So, how do we tell if we accept the dragged data object? It's simple. We accept data objects that store text (with format type *CF_TEXT*) or those that can list files dropped on the application (with format type *CF_HDROP*). The *CanDrop* method checks for these formats, as *Listing 8* shows:

```
function TCustomDropTarget.CanDrop(const DataObj: IDataObject): Boolean;
begin
  Result := (DataObj.QueryGetData(MakeFormatEtc(CF_HDROP)) = S_OK)
    or (DataObj.QueryGetData(MakeFormatEtc(CF_TEXT)) = S_OK);
end;
```

Listing 8

CanDrop uses *MakeFormatEtc* to populate the *TFormatEtc* structure required by the data object's *QueryGetData* method. *MakeFormatEtc*, which was explained in *article #24*, is shown in *Listing 9*.

```
function TCustomDropTarget.MakeFormatEtc(
  const Fmt: TClipFormat): TFormatEtc;
begin
  Result.cfFormat := Fmt;
  Result.ptd := nil;
  Result.dwAspect := DVASPECT_CONTENT;
  Result.lindex := -1;
  Result.tymed := TYMED_HGLOBAL;
end;
```

Listing 9

Let us move on to examine how we extract the information to be displayed from the data object. *Listing 10* presents all the relevant methods.

```
procedure TCustomDropTarget.DisplayData(const DataObj: IDataObject);
begin
  if DataObj.QueryGetData(MakeFormatEtc(CF_HDROP)) = S_OK then
    DisplayFileList(DataObj)
  else if DataObj.QueryGetData(MakeFormatEtc(CF_TEXT)) = S_OK then
    DisplayText(DataObj)
  else
    raise Exception.Create('Drop data object not supported');
end;

procedure TCustomDropTarget.DisplayFileList(const DataObj: IDataObject);
var
  Files: TStringList;
begin
  if Assigned(fDropHandler) then
  begin
    Files := TStringList.Create;
    try
      GetFileListFromObj(DataObj, Files);
      fDropHandler.HandleFiles(Files);
    finally
      FreeAndNil(Files);
    end;
  end;
end;

procedure TCustomDropTarget.DisplayText(const DataObj: IDataObject);
begin
  if Assigned(fDropHandler) then
    fDropHandler.HandleText(GetTextFromObj(DataObj, CF_TEXT));
end;

procedure TCustomDropTarget.GetFileListFromObj(
  const DataObj: IDataObject; const FileList: TStrings);
```

```
var
  Medium: TStgMedium;          // storage medium containing file list
  DroppedFileCount: Integer;  // number of dropped files
  I: Integer;                 // loops thru dropped files
  FileNameLength: Integer;    // length of a dropped file name
  FileName: string;           // name of a dropped file
begin
  // Get required storage medium from data object
  if DataObj.GetData(MakeFormatEtc(CF_HDROP), Medium) = S_OK then
  begin
    try
      try
        // Get count of files dropped
        DroppedFileCount := DragQueryFile(
          Medium.hGlobal, $FFFFFFFF, nil, 0
        );
        // Get name of each file dropped and process it
        for I := 0 to Pred(DroppedFileCount) do
        begin
          // get length of file name, then name itself
          FileNameLength := DragQueryFile(Medium.hGlobal, I, nil, 0);
          SetLength(FileName, FileNameLength);
          DragQueryFile(
            Medium.hGlobal, I, PChar(FileName), FileNameLength + 1
          );
          // add file name to list
          FileList.Add(FileName);
        end;
      finally
        // Tidy up - release the drop handle
        // don't use DropH again after this
        DragFinish(Medium.hGlobal);
      end;
    finally
      ReleaseStgMedium(Medium);
    end;
  end;
end;

function TCustomDropTarget.GetTextFromObj(const DataObj: IDataObject;
  const Fmt: TClipFormat): string;
var
  Medium: TStgMedium;
  PText: PChar;
begin
  if DataObj.GetData(MakeFormatEtc(Fmt), Medium) = S_OK then
  begin
    Assert(Medium.tymed = MakeFormatEtc(Fmt).tymed);
    try
      PText := GlobalLock(Medium.hGlobal);
      try
        Result := PText;
      finally
        GlobalUnlock(Medium.hGlobal);
      end;
    finally
      ReleaseStgMedium(Medium);
    end;
  end
  else
    Result := '';
end;
```

Listing 10

Recall that *DisplayData* is called when a data object is dropped. All it does is test the format of the dropped data and call *DisplayFileList* if files were dropped or *DisplayText* if text was dropped. No other data format should have been passed to *DisplayData*, so we raise an exception if that is the case.

DisplayFileList simply creates a string list to receive the names of the dropped files and calls *GetFileListFromObj* to extract the file list from the data object. Similarly, *DisplayText* calls *GetTextFromObj* to extract the dropped text from the data object.

GetTextFromObj & GetFileListFromObj

Similar methods are explained in detail in article #24. See "*Example 1: Text* *stored in global memory*" and "*Example 2: File list stored in global memory*" for details.

You will have noticed that both *DisplayFileList* and *DisplayText* only get data from the data object if the *fDropHandler* field is set. They also call methods on *fDropHandler* when it is set. So what is *fDropHandler*?

Put simply *fDropHandler* is the means by which we communicate with the main form. *fDropHandler* is an instance of an object that supports *IDropHandler*, the interface we declared in *Listing 6*. A reference to such an object is passed to *TCustomDropTarget*'s constructor and recorded in *fDropHandler*. The relevant methods of *IDropHandler* are called by *DisplayFileList* and *DisplayText*. *Listing 11* shows the constructor, which needs no further explanation.

```
constructor TCustomDropTarget.Create(const Handler: IDropHandler);
begin
  inherited Create;
  fDropHandler := Handler;
end;
```

Listing 11

And where is *IDropHandler* implemented? It could be any object that knows how to display the data, so where better than the main form?

The Main Form

The main form contains a web browser control to display the HTML page. From what we've said above, the form must also implement *IDropHandler*.

Using a blank form, drop a *TWebBrowser* control on it, aligned to fill all the form's client area. Create *onCreate*, *OnDestroy* and *OnShow* event handlers and change the form's interface section to look like *Listing 12*:

```
uses
  ..., ActiveX, MSHTML, ...;

type
  TForm1 = class(TForm, IDropHandler)
    WebBrowser1: TWebBrowser;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormShow(Sender: TObject);
  private
    fWBContainer: TWBDragDropContainer;
    procedure SetBoxInnerHTML(const HTML: string);
  protected
    { IDropHandler methods }
    procedure HandleText(const Text: string);
    procedure HandleFiles(const Files: TStrings);
  end;
```

Listing 12

Now implement *FormCreate*, *FormDestroy* and *FormShow* as shown in *Listing 13*.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  OleInitialize(nil);
  fWBContainer := TWBDragDropContainer.Create(WebBrowser1);
  fWBContainer.DropTarget := TCustomDropTarget.Create(Self);
  WebBrowser1.Navigate(ExtractFilePath(ParamStr(0)) + 'Page.html');
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  fWBContainer.DropTarget := nil;
  OleUninitialize;
  FreeAndNil(fWBContainer);
end;

procedure TForm1.FormShow(Sender: TObject);
begin
  while WebBrowser1.ReadyState <> READYSTATE_COMPLETE do
  begin
```

```
    Sleep(5);
    Application.ProcessMessages;
  end;
end;
```

Listing 13

In *FormCreate* we once again initialise OLE. We then create a web browser container object and use it to assign the drop target, now implemented by *TCustomDropTarget*. A reference to this form is passed to *TCustomDropTarget*'s constructor, because our form implements *IDropHandler*. Instead of navigating to `about:blank` we navigate to an HTML page named `Page.html` in the same directory as the executable program. We will discuss this file later.

FormDestroy simply tidies up, while *FormShow* pauses until the HTML document has fully loaded.

We still have to implement the methods of *IDropHandler*, so let us do that now. *Listing 14* shows our implementation, along with that of the helper method, *SetBoxInnerHTML* that is used update the display.

```
procedure TForm1.HandleFiles(const Files: TStrings);
var
  Idx: Integer;
  HTML: string;
begin
  HTML := '<ul>'#13#10;
  for Idx := 0 to Pred(Files.Count) do
    HTML := HTML
      + '<li>'
      + MakeSafeHTMLText(Files[Idx])
      + '</li>'#13#10;
  HTML := HTML + '</ul>';
  SetBoxInnerHTML(HTML);
end;

procedure TForm1.HandleText(const Text: string);
begin
  SetBoxInnerHTML('<pre>' + MakeSafeHTMLText(Text) + '</pre>');
end;

procedure TForm1.SetBoxInnerHTML(const HTML: string);
var
  BoxDiv: IHTMLElement;
  Doc: IHTMLDocument3;
begin
  if Supports(WebBrowser1.Document, IHTMLDocument3, Doc) then
  begin
    BoxDiv := Doc.getElementById('box');
    if Assigned(BoxDiv) then
      BoxDiv.innerHTML := HTML;
  end;
end;
```

Listing 14

SetBoxInnerHTML examines the displayed HTML document to find an element with an id of "box". If such an element exists its inner HTML is set to the code passed to method as a parameter. This causes the required HTML to be displayed in the document. The HTML document presented below contains a **<div>** with the required id.

HandleFiles accepts a string list containing the names of the dropped files. It creates a HTML unordered list of file names. *HandleText* simply encloses the text passed to it in **<pre>** tags.

Both *HandleFiles* and *HandleText* make use of a helper function named *MakeSafeHTMLText* that ensures the text contains only valid HTML characters. *MakeSafeHTMLText*, which was taken from the *Code Snippets Database*, appears in *Listing 15* below.

```
function MakeSafeHTMLText(TheText: string): string;
var
  Idx: Integer; // loops thru the given text
begin
  Result := '';
  for Idx := 1 to Length(TheText) do
    case TheText[Idx] of
```

```
      '<':  // opens tags
        Result := Result + '&lt;';
      '>':  // closes tags
        Result := Result + '&gt;';
      '&':  // begins char references
        Result := Result + '&amp;';
      '"':  // quotes (can be a problem in quoted attributes)
        Result := Result + '&quot;';
      #0..#31, #127..#255:  // control and special chars
        Result := Result + '&#'
          + SysUtils.IntToStr(Ord(TheText[Idx])) + ';';
      else  // compatible text: pass thru
        Result := Result + TheText[Idx];
    end;
end;
```

Listing 15

HTML of Displayed Page

All that remains now is to show the content of `Page.html`, the HTML page displayed in the web browser control, which we do in *Listing 16*.

```
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <title>Article#25 Demo 4</title>
    <style type="text/css">
      #box {
        background-color: #ff9;
        border: 1px silver solid;
        padding: 0.25em;
        margin: 6px 0;
        height: 160px;
        width: 420px;
        overflow: auto;
      }
    </style>
</head>
<body>
  <h1>
    Drag Drop Test
  </h1>
  <p>
    Drag and drop one or more files or a text selection on this
    browser control.
  </p>
  <div id="box">
    <div style="text-align:center;vertical-align:bottom;">
      <em>Dragged and dropped text or list of files will be
      displayed here.</em>
    </div>
  </div>
</body>
</html>
```

Listing 16

The main thing to note is the **<div>** with id of "box" which is where the dropped text or file list is displayed. We style the **<div>** with a silver border and pale yellow background, and give it a fixed height and width so that scroll bars will appear for any oversized content.

Exercising the Code

Run this code and experiment by dragging single and multiple files from Explorer and by selecting and dragging text from a text editor or word processor that supports text drag and drop.

This completes our investigation of how to handle drag and drop in the *TWebBrowser* control.

In the *final section* you'll find a summary and a link to the article's demo source code.