How to write filters that extend the functionality of the TStream classes

Why do it?

In Java there are various predefined stream classes that provide filters for other stream classes – the filter classes essentially "wrap" the streams they operate on. The filters can often be applied to further filters. This article demonstrates how we can do this in Delphi in a way that is extendable – i.e. we can wrap filters around other filters.

How it's done

First of all, let's look at why we want to do this. Well say you want to write some data primitives as text to a stream and the text to be formatted to fit on a page, word wrapping properly. Then if we can wrap a filter that formats the primitives around another that formats the text and this filter is wrapped round a file stream object, then all we have to do is access the methods of the first class and the rest of the process happens automatically.

The approach I've taken is to define a class, *TStreamWrapper*, that provides a base class for any filters that we want to define. Any *TStreamWrapper* performs it's i/o using another *TStream* object – the wrapped object. The key point is that *TStreamWrapper* is itself derived from *TStream*, so that it can also wrap other *TSteamWrapper* objects – giving the extensibility we need. *TSteamWrapper* can also cause a wrapped stream to be freed when it is itself freed – allowing the wrapped streams to be created "on the fly" when the *TSteamWrapper* constructor is called.

There is no additional functionality built in to *TSteamWrapper* – this is to be provided by derived classes. A small example class is demonstrated here.

First to *TSteamWrapper*. Here's the class declaration:

```
type
  TStreamWrapper = class(TStream)
  private
    FBaseStream: TStream;
      {The wrapped stream}
    FCloseStream: Boolean;
      {Free wrapped stream on destruction?}
  protected
    procedure SetSize(NewSize: Longint); override;
      {Sets the size of the stream to the given value
      if the operation is supported by the underlying stream}
    property BaseStream: TStream read FBaseStream;
      {Gives access to the underlying stream to descended
      classes}
  public
    constructor Create(const Stream: TStream;
      const CloseStream: Boolean = False); virtual;
      {If CloseStream is true the given underlying stream is
      freed when this object is freed}
    destructor Destroy; override;
    // Implementation of abstract methods of TStream
    function Read(var Buffer; Count: Longint): Longint;
      override;
    function Write(const Buffer; Count: Longint): Longint;
      override;
    function Seek(Offset: Longint; Origin: Word): Longint;
      override;
  end;
```

Listing 1

and the implementation is:

```
constructor TStreamWrapper.Create(const Stream: TStream;
  const CloseStream: Boolean);
begin
  inherited Create;
  // Record wrapped stream and if we free it on destruction
  FBaseStream := Stream;
  FCloseStream := CloseStream;
end;

destructor TStreamWrapper.Destroy;
begin
  // Close wrapped stream if required
  if FCloseStream then
    FBaseStream.Free;
  inherited Destroy;
end;

function TStreamWrapper.Read(var Buffer;
  Count: Integer): Longint;
begin
  // Simply call underlying stream's Read method
  Result := FBaseStream.Read(Buffer, Count);
end;

function TStreamWrapper.Seek(Offset: Integer;
  Origin: Word): Longint;
begin
  // Simply call the same method in the wrapped stream
  Result := FBaseStream.Seek(Offset, Origin);
end;

procedure TStreamWrapper.SetSize(NewSize: Integer);
begin
  // Set the size property of the wrapped stream
  FBaseStream.Size := NewSize;
end;

function TStreamWrapper.Write(const Buffer;
  Count: Integer): Longint;
begin
  // Simply call the same method in the wrapped stream
  Result := FBaseStream.Write(Buffer, Count);
end;
```

Listing 2

We can now derive a small filter class – *TStrStream*. As it stands it's not particularly useful, but does demonstrate the techniques. The class reads and writes strings (which are preceded by their lengths) from and to any stream. The declaration is:

```
type
  TStrStream = class(TStreamWrapper)
  public
    procedure WriteString(AString: string);
    function ReadString: string;
  end;
```

Listing 3

The class is implemented as follows:

```
function TStrStream.ReadString: string;
var
  StrLen: Integer;     // the length of the string
  PBuf: PChar;         // buffer to hold the string that is read
begin
  // Get length of string (as 32 bit integer)
  ReadBuffer(StrLen, SizeOf(Integer));
  // Now get string
  // allocate enough memory to hold string
  GetMem(PBuf, StrLen);
  try
    // read chars into buffer and set resulting string
    ReadBuffer(PBuf^, StrLen);
    SetString(Result, PBuf, StrLen);
  finally
    // deallocate buffer
    FreeMem(PBuf, StrLen);
  end;
end;

procedure TStrStream.WriteString(AString: string);
var
  Len: Integer;     // length of string
begin
  // Write out length of string as 32 bit integer
  Len := Length(AString);
  WriteBuffer(Len, SizeOf(Integer));
  // Now write out the string's characters
  WriteBuffer(PChar(AString)^, Len);
end;
```

Listing 4

The following code should demonstrate how to write a string to a file and read it back in again. Here we use a file stream that is created on the fly and automatically closed when we are done. Of course you could use any stream type and handle its lifetime yourself.

```
procedure WriteText(const Txt: string);
var
  TS: TTextStream;
begin
  // This opens stream on a file stream that will
  // be closed when this stream closes
  TS := TTextStream.Create(
    TFileStream.Create('test.dat', fmCreate), True);
  TS.WriteString(Txt);
  TS.Free;  // this also closes wrapped file stream
end;

function ReadText: string;
var
  TS: TTextStream;
begin
  TS := TTextStream.Create(
    TFileStream.Create('test.dat', fmCreate), True);
  Result := TS.ReadString;
  TS.Free;
end;
```

Listing 5

The filter in this example provides additional methods to those in *TSteamWrapper*. We can also provide filters that override the *Read* and *Write* methods to alter the way that files are written.

Worked Example

You can download a *worked example* of the above code. The zip file contains a Delphi 4 project that writes out some user provided strings to a file using the *TStrStream* class, and reads them back in again.

Stream Extension Classes

My *Stream Classes* provide an example of the techniques described here.