DelphiDabbler Console Application Runner Classes

User Guide

Contents

Overview

The Console Application Runner classes are intended to simplify working with child console application processes. There are three classes:

- `TPJCustomConsoleApp`
- `TPJConsoleApp`
- `TPJPipe`

These classes are briefly described in the following table:

Class	Description
TPJCustomConsoleApp	Base class for TPJConsoleApp. Provides all functionality but declares all properties protected. Use this class if creating sub-classes that do not want to make all properties public or need to execute custom code within the class rather than handling events.
TPJConsoleApp	Main console application class. Makes all properties public. Use instances of this class when working with console processes.
TPJPipe	Helper class to use when redirecting console application standard input, output and error channels using pipes.

This user guide provides a full explanation of the classes, including a review of the properties., methods and events.

Ten examples of using the classes are provided.

The document closes with two appendices, the first of which discusses the inheritable handles required for redirection and the second presents two console applications that can be used with the examples.

TPJConsoleApp

This class is designed to be used to control the execution of child console application processes. The class provides facilities to redirect the console process' standard input, output and error handles. The application's execution can be given a time-out and can be time sliced to allow progress to be reported and redirected output to be handled while the console application continues to execute. Security levels and environment blocks can be customised. Errant processes can be terminated. Finally, the program's exit code is made available.

TPJConsoleApp descends from TPJCustomConsoleApp. The only differences are that TPJCustomConsoleApp declares all properties and events as protected whereas TPJConsoleApp makes them all public.

Properties

```
property ConsoleTitle: string;
```

The title to be displayed in any new console window. If the property is set to the empty string then the console's default title is used. This property only has an effect if the console app has its own console window (see also: UseNewConsole).

Default = empty string.

```
property ElapsedTime: LongWord;
```

Read Only. Specifies the approximate time in milliseconds since the console application began executing. The clock stops when the application completes or times out. This property is updated only when the application yields control to this object. The frequency with which this happens depends on the TimeSlice property. If TimeSlice is INFINITE then ElapsedTime is only updated once the application terminates.

Examine ElapsedTime in a OnWork event handler to get the updated execution time while a process is executing. Examine the property after a process has completed to get the total execution time.

Times are approximate.

`property Environment: Pointer;`

Pointer to the environment block[1] used by the console application. The caller is responsible for allocating and freeing this memory, which must remain allocated while the console application is executing. If the property is nil then the console application inherits the parent process' environment block.

Default = nil.

`property ErrorCode: LongWord;`

Read Only. Provides information about any error that occurred while attempting to run a console application. The property is zero if the application was executed successfully. Error codes either correspond to Windows error codes or are set by this class. Error codes set by the class have bit 29 set[2]. The class specific codes are:

Error code (hex)	Constant defined in PJConsoleApp.pas	Description
$20000001	`cAppErrorTimeOut`	Application timed out (see the `MaxExecTime` property).
$20000002	`cAppErrorTerminated`	Application was forcibly terminated (see the `Terminate` method).

To check for application errors, use the constants from the table and **and** them with the `cAppErrorMask` constant, defined in `PJConsoleApp.pas` as `1 shl 29`. For example: `if (ErrorCode and cAppErrorTimeout) <> 0 then {Do something};`.

Note that this property only represents errors that prevent the console application from running. Errors that occur within the application are not caught here – they are usually reported via the program's exit code and will be stored in the `ExitCode` property.

`property ErrorMessage: string;`

Read Only. The error message corresponding the value of `ErrorCode`, or the empty string if `ErrorCode` is 0. If `ErrorCode` corresponds to a Windows error then `ErrorMessage` stores the error description provided by Windows.

`property ExitCode: LongWord;`

Read Only. Exit code returned by the console application on completion. The value of the property is application dependent[3].

The value of `ExitCode` is meaningless if the `ErrorCode` property is non-zero.

`property KillTimedOutProcess: Boolean;`

When this property is True the console application will be forcibly terminated should it time out or be terminated via the `Terminate` method. Setting the property to False causes a timed out or terminated application to be left running after the `Execute` method has returned, severing this object' connection with the application.

The Windows API `TerminateProcess` function is used to kill the application. This function does not perform a clean shut-down of the application. See Windows API help for details.

Default = True.

---

1) See http://www.delphidabbler.com/articles?article=6 for information about environment blocks and how to create them.
2) According to the Windows API documentation, error code with bit 29 set are reserved for application use.
3) Refer to the console application's documentation for details of any exit codes.

property `MaxExecTime: LongWord;`

Specifies the maximum permitted execution time of the console application, in milliseconds. If the application runs for longer than `MaxExecTime` then the `Execute` method returns once the time has elapsed. If the `KillTimedOutProcess` property is True then the console application is terminated, otherwise the process continues but no longer has any connection with this object.

`MaxExecTime` may be set to `INFINITE`, in which case the console application it will never time out until it has finished execution. In cases where the application is deemed to be running too long it may still be possible to force the application to terminate using the `Terminate` method.

Default = 60,000ms (one minute).

property `Priority: TPJConsoleAppPriority;`

Priority with which the console application is started. Possible values are:

Value	Description
`cpDefault`	Default priority. Normally `cpNormal` is used unless the parent process has priority `cpIdle` in which case `cpIdle` is used.
`cpHigh`	High priority. Use for time-critical tasks (processor intensive).
`cpNormal`	Normal priority for applications with no specific scheduling needs.
`cpIdle`	Idle priority. The process is run only when the system is idle.
`cpRealTime`	Real time priority. Highest possible priority (pre-empts all threads, including the operating system).

Default = `cpDefault`.

property `ProcessAttrs: PSecurityAttributes;`

Security and inheritance attributes for the console application process. Determines whether the process handle can be inherited by child processes. Setting to nil means the process handle can't be inherited.

Note that the caller is responsible for setting up the structure correctly. The caller need not maintain a copy of the provided structure once the property is set since `TPJConsoleApp` makes an internal copy.

Default = nil.

property `ProcessInfo: TProcessInformation;`

Read Only. Provides process information for the executing console application. All fields are zero when no application is executing.

`ProcessInfo` is only of use inside `OnWork` or `OnComplete` event handlers since these are the only way of accessing it when a process is running.

See the Windows API help for information about the fields of `TProcessInformation`[4].

property `StdIn: THandle;`

The console application's standard input handle. The program will read standard input from the specified handle. Leave as 0 if standard input is not to be redirected. Ensure that any handles used for this purpose are inheritable [See Appendix 1 for information about inheritable handles]

4) TProcessInformation is a Delphi type name. Look up `PROCESS_INFORMATION` in Windows API help.

Default = 0 (no redirection).

`property StdOut: THandle;`

The console application's standard output handle. The program will write standard output to the specified handle. Leave as 0 if standard output is not to be redirected. Ensure that any handles used for this purpose are inheritable. [See Appendix 1 for information about inheritable handles]

Default = 0 (no redirection).

`property StdErr: THandle;`

The console application's standard error handle. The program will write standard error output to the specified handle. Leave as 0 if standard error is not to be redirected. Ensure any handles used for this purpose are inheritable. [See Appendix 1 for information about inheritable handles]

Default = 0 (no redirection).

`property ThreadAttrs: PSecurityAttributes;`

Security and inheritance attributes for the console application's primary thread. Determines whether the thread handle can be inherited by child processes. Leaving as nil means the thread handle can't be inherited.

Note that the caller is responsible for setting up the structure correctly. The caller need not maintain a copy of the provided structure once the property is set since `TPJConsoleApp` makes an internal copy.

Default = nil.

`property TimeSlice: LongWord;`

Execution of the console application is normally time-sliced. This property determines the length of each time slice, in milliseconds. At the end of each time slice execution of the console application is paused, the length of time left to run is recalculated and the `OnWork` event is triggered.

`TimeSlice` may be set to `INFINITE`, in which case the console application is left to run until completion without interuption. In this case the `OnWork` event will never be called and the application will never time out. Furthermore, it is not possible to force the application to terminate via the `Terminate` method. Setting `TimeSlice` to `INFINITE` is not recommended.

Default = 50ms ($1/20^{th}$ second).

`property TimeToLive: LongWord;`

Read Only. Indicates the amount of time, in milliseconds, the console application has left to run before timing out. The value will be `INFINITE` if `MaxExecTime` is `INFINITE`.

`TimeToLive` is only of any use if accessed within an `OnWork` event handler. The property will always be 0 once a process completes.

`property UseNewConsole: Boolean;`

Causes the console application to open a new console window when True. Setting the property to False causes the console application to use any existing console.

When `TPJConsoleApp` is hosted in a GUI program that has no console, a new console window is always used regardless of the value of this property.

Whether a new console window is actually displayed depends on the state of the `Visible` property.

Default = False.

`property Visible: Boolean;`

Determines whether the console application is to be displayed. The behaviour of this property depends on whether `TPJConsoleApp` is hosted in a GUI or console application, and on the value of the `UseNewConsole` property.

When `TPJConsoleApp` is hosted by a console application, behaviour depends on the value of the `UseNewConsole` property as follows:

- If `UseNewConsole` is True a console window is only displayed when `Visible` is True. When `Visible` is False no new console window is displayed and output from the process is not displayed.
- When `UseNewConsole` is False the program being executed displays its output in the host's console, regardless of the state of the `Visible` property.

When the host program is a GUI application the process uses a new console window regardless of the state of `UseNewConsole`. The console window is displayed only if the `Visible` property is True.

Default = False.

Methods

`constructor Create;`

Sets up the object. Initialises properties to default values.

`function Execute(const CmdLine, CurrentDir: string): Boolean;`

Executes the command line application.

- Parameter: `CmdLine` – Command line to execute. This should include the name of the application (with path if necessary) along with any required parameters. Paths and parameters containing spaces should be enclosed in double quotes.
- Parameter: `CurrentDir` – The directory that the application is to treat as its current directory. Pass the empty string to use the same current directory as the parent application.
- Return: True if the application was executed successfully and False if the application failed to run. When False is returned, the `ErrorCode` property will be store a non-zero error code.

`procedure Terminate;`

Attempts to terminate the process. Calling this method causes the `Execute` method to return after the next `OnWork` event. If `KillTimedOutProcess` is true the console application will be halted. The method has no effect when `TimeSlice` is `INFINITE`.

Events

`property OnComplete: TNotifyEvent;`

Event triggered when an application completes or times out. Always called and guaranteed to be called *after* the last `OnWork` event. Handle this event to tidy up after the console process is completed. `ProcessInfo` is available when this event is triggered. The `ErrorCode` property can be used to check how the application terminated – it will be zero if the application executed successfully, did not timeout and was not terminated.

property OnWork: TNotifyEvent;

Event triggered when the executing application yields control to this class. The frequency with which this event is triggered depends on the value of the TimeSlice property – the smaller the value the more frequently the event is triggered.

Handle the event to perform any required processing between time slices. Examples of the kind of processing you may wish to perform are:

- Updating a progress meter in the main application, or reporting time to live or elapsed time since the application began executing.
- Processing data written by the console application to redirected standard output or standard error. The most common way to do this is via a pipe.

GUI applications that update the user interface should call Application.ProcessMessages in the OnWork event handler to let the application refresh the interface.

This event is never triggered if TimeSlice = INFINITE.

TPJCustomConsoleApp

This is the base class for TPJConsoleApp. It implements all the functionality but declares all properties as protected. For details of the main properties, methods and events see TPJConsoleApp above.

If you intend to sub-class TPJConsoleApp you are recommended to subclass TCustomConsoleApp instead and to publish just those properties you wish to expose.

TPJCustomConsoleApp has two protected virtual methods that can be overidden in decendants if required:

procedure DoComplete;

Triggers the OnComplete event. Descendants can override to perform specific actions without triggering the event. If the event is still required **inherited** can be called.

procedure DoWork;

Triggers the OnWork event. Descendants can override to perform specific actions without triggering the event. If the event is still required **inherited** can be called.

TPJPipe

This is a helper class designed for use when redirecting output to and input from pipes. The class can create pipes with the inheritable handles that are required for redirecting the standard i/o of console applications.

TPJPipe also provides methods that make working with with pipes easier. It can check the number of bytes in a readable pipe, can read bytes from a pipe into either a buffer or a stream and write bytes to a pipe from buffers and streams. Lastly it can safely close a pipe's write handle, effectively signalling end-of-file to the process reading it.

Properties

property ReadHandle: THandle;

Read only. Windows handle used to read data from the pipe. If inheritable this handle can be used with TPJConsoleApp's StdIn property to redirect console application input from a pipe.

Do not pass this handle to any Windows API function that may alter it (e.g. `CloseHandle`).

`property WriteHandle: THandle;`

Read only. Windows handle used to write data to the pipe. If inheritable this handle can be passed to `TPJConsoleApp`'s `StdOut` and `StdErr` properties to redirect standard output and standard error to pipes.

Do not pass this handle to any Windows API function that may alter it (e.g. `CloseHandle`). If you need to close `WriteHandle` to signal end-of-file then use the `CloseWriteHandle` method. `CloseWriteHandle` closes the handle and sets it to 0.

`WriteHandle` should not be used when 0.

Methods

`function AvailableDataSize: LongWord;`

Peeks the pipe to get the size of data available for reading from a pipe.

- Return: Number of bytes of available data.
- Exception: Raises `EInOutError` if there is an error peeking the pipe.

`procedure CloseWriteHandle;`

Closes the pipe's write handle if it is open. This effectively signals end-of-file to any reader of the pipe. After calling this method no further data may be written to the pipe and `WriteHandle` will be zero.

`procedure CopyFromStream(const Stm: TStream; Count: LongWord = 0);`

Copies data from a stream into the pipe.

- Parameter: `Stm` – Stream from which to copy data.
- Parameter: `Count` – Number of bytes to copy. If 0 then all the remaining data from the stream is copied to the pipe.
- Exception: Raises `EInOutError` if the pipe's write handle has been closed or if the pipe can't be written to.
- Exception: Raises `EReadError` if the stream read fails.

`procedure CopyToStream(const Stm: TStream; Count: LongWord = 0);`

Copies data from the pipe to a stream.

- Parameter: `Stm` – Stream that receives the data.
- Parameter: `Count` – Number of bytes to copy. If 0 then all the remaining data in the pipe is copied to the stream.
- Exception: Raises `EInOutError` if there is an error peeking or reading the pipe.
- Exception: Raises `EWriteError` if the stream write fails.

```
constructor Create(const Size: LongWord; const Inheritable: Boolean = True);
  overload;
```

Creates a pipe object with specified size. The pipe may optionally have inheritable handles.

- Parameter: `Size` – Required size of the pipe. If `Size` is 0 then the Windows default pipe size used.
- Parameter: `Inheritable` – Indicates whether `ReadHandle` and `WriteHandle` are to be inheritable. Defaults to True.
- Exception: Raises `EInOutError` if the pipe can't be created. The exception message includes the reason returned from the operating system.

```
constructor Create(const Inheritable: Boolean = True);
  overload;
```

Creates a pipe object with default size. The pipe may optionally have inheritable handles.

- Parameter: `Inheritable` – Indicates whether `ReadHandle` and `WriteHandle` are to be inheritable. Defaults to True.
- Exception: Raises `EInOutError` if the pipe can't be created. The exception message includes the reason returned from the operating system.

```
constructor Create(const Size: LongWord; const Security: TSecurityAttributes);
  overload;
```

Creates a pipe object with specified size and security attributes.

- Parameter: `Size` – Required size of the pipe. If `Size` is 0 then the Windows default pipe size used.
- Parameter: `Security` – Required security for the pipe. If `ReadHandle` and `WriteHandle` are to be inheritable set the `TSecurityAttributes.bInheritHandle` field to True.
- Exception: Raises `EInOutError` if the pipe can't be created. The exception message includes the reason returned from the operating system.

```
constructor Create(const Security: TSecurityAttributes);
  overload;
```

Creates a pipe object with default size pipe and specified security attributes.

- Parameter: `Security` – Required security for the pipe. If `ReadHandle` and `WriteHandle` are to be inheritable set the `TSecurityAttributes.bInheritHandle` field to True.
- Exception: Raises `EInOutError` if the pipe can't be created. The exception message includes the reason returned from the operating system.

```
destructor Destroy; override;
```

Tears down the object.

```
function ReadData(out Buf; const BufSize: LongWord; out BytesRead: LongWord):
  Boolean;
```

Reads data from the pipe into a buffer.

- Parameter: `Buf` – Buffer that receives the data. `Buf` must have capacity of at least `BufSize` bytes.
- Parameter: `BufSize` – Size of buffer or the number of bytes requested.
- Parameter: `BytesRead` – Set by the method to the number of bytes actually read.
- Return: True if some data was read, false if not.
- Exception: Raises `EInOutError` if there is an error peeking or reading the pipe.

```
function WriteData(const Buf; const BufSize: LongWord): LongWord;
```

Writes data from a buffer to the pipe.

- Parameter: `Buf` – Buffer containing the data to be written. `Buf` must have capacity of at least `BufSize` bytes.
- Parameter: `BufSize` – Number of bytes to write from buffer.
- Return: Number of bytes actually written.
- Exception: Raises `EInOutError` if the pipe's write handle has been closed or if the pipe can't be written to.

Examples

The following examples show how to undertake various common tasks using `TPJConsoleApp`, `TPJCustomConsoleApp` and `TPJPipe`. Most examples require the use of a suitable command line application. Source code for two command line applications suitable for use with the examples is provided in Appendix 2.

Not all the features of each class are demonstrated.

Demo programs for each of the examples are included with the Console Application Runner Classes download.

Example 1: ExecAndWait

You will no doubt have seen a lot of code on the Internet that shows how to execute a console application and wait for it to complete. It will probably look something like this[5]:

```
function ExecAndWait(const CommandLine: string): Boolean;
var
  StartupInfo: Windows.TStartupInfo;     // start-up info passed to process
  ProcessInfo: Windows.TProcessInformation; // info about the process
  ProcessExitCode: Windows.DWord;           // process's exit code
begin
  // Set default error result
  Result := False;
  // Initialise startup info structure to 0, and record length
  FillChar(StartupInfo, SizeOf(StartupInfo), 0);
  StartupInfo.cb := SizeOf(StartupInfo);
  // Execute application commandline
  if Windows.CreateProcess(nil, PChar(CommandLine),
    nil, nil, False, 0, nil, nil,
    StartupInfo, ProcessInfo) then
  begin
    try
      // Now wait for application to complete
      if Windows.WaitForSingleObject(ProcessInfo.hProcess, INFINITE)
        = WAIT_OBJECT_0 then
        // It's completed - get its exit code
        if Windows.GetExitCodeProcess(ProcessInfo.hProcess,
          ProcessExitCode) then
          // Check exit code is zero => successful completion
          if ProcessExitCode = 0 then
            Result := True;
    finally
      // Tidy up
      Windows.CloseHandle(ProcessInfo.hProcess);
      Windows.CloseHandle(ProcessInfo.hThread);
    end;
  end;
end;
```

With the use if infinite waiting for the console application to complete, we can achieve the same result using `TPJConsoleApp` as follows:

```
function ExecAndWait2(const CommandLine: string): Boolean;
var
  App: TPJConsoleApp;
begin
  App := TPJConsoleApp.Create;
  try
```

5) Example taken from the DelphiDabbler Code Snippets Database.

```
    App.MaxExecTime := INFINITE;  // don't time out
    App.TimeSlice := INFINITE;    // timeout for WaitForSingleObject
    App.Visible := True;          // ensure we see the app
    if App.Execute(CommandLine) then
      Result := App.ExitCode = 0  // app executed OK - check its exit code
    else
      Result := False;            // app didn't execute
  finally
    App.Free;
  end;
end;
```

This is a little crude – the `TPJConsoleApp` class can do a lot better than this. Using code like this isn't recommended.

One problem with the above code is that, if you run the console app from a GUI application, your GUI locks up until the console app completes.

Try it. Using Delphi, create a new GUI application. First copy the above code for `ExecAndWait` and `ExecAndWait2` into the form unit's implementation section. Now drop two buttons on the main form and add the following `OnClick` event handlers:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if not ExecAndWait('Timed 3') then  // runs Timed.exe for 3 seconds.
    ShowMessage('Non-zero error code or application could not be executed');
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  if not ExecAndWait2('Timed 3') then // runs Timed.exe for 3 seconds.
    ShowMessage('Non-zero error code or application could not be executed');
end;
```

Note: We've used an application called `Timed.exe` here that takes a parameter that tells it how many seconds to run. You can use anything suitable. If you want to try `Timed.exe`, its source code is available in Appendix 2.

Run the application. Click either button. The console application will run. Try switching back to the application while the console application is running. You can't can you?

This behaviour may be what you want, but what if the console app freezes? Your app can't do anything about it. Example 2 shows how to let your GUI application "breathe" while the console application executes.

Example 2: ExecAndWait that lets a GUI remain interactive

As noted in Example 1, it's often worth letting a GUI program remain interactive – may be to display a progress bar or whatever.

To do this you need to handle `TPJConsoleApp`'s `OnWork` event.

Start a new Delphi GUI application and, in the form's private section, insert the following new method declaration:

```
procedure WorkHandler(Sender: TObject);
```

Implement the method as follows:

```
procedure TForm1.WorkHandler(Sender: TObject);
begin
  Application.ProcessMessages;
end;
```

Now drop a `TButton` and add the following code as its `OnClick` event handler:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  App: TPJConsoleApp;
begin
  App := TPJConsoleApp.Create;
  try
    App.MaxExecTime := INFINITE;  // don't time out
    App.TimeSlice := 100;         // yield to main app every 1/10 second
    App.Visible := True;          // ensure we see the app
    App.OnWork := WorkHandler;    // assign the event handler
    if not App.Execute('Timed 5') then  // run Timed.exe for 5 seconds
      ShowMessage('Failed to run Timed.exe');
  finally
    App.Free;
  end;
end;
```

The main code here is similar to Example 1 except that the `TimeSlice` property is now set to 100 rather than `INFINITE` and we have assigned a handler for the `OnWork` event. Setting `TimeSlice` forces the console app to yield to the GUI app every $1/10^{th}$ second. When this happens `TPJConsoleApp` triggers the `OnWork` event, and our event handler lets the GUI application receive messages.

We've used `Timed.exe` from Appendix 2 once again. Substitute another suitable program if you wish, providing it runs for a significant amount of time.

Run the program. Try to switch back to the GUI while the console app is running. Unlike in Example 1, you can now do it!

Example 3: Indicating Progress

It is common to want to display some kind of progress indicator while another application is running. This example shows how to do that using `TPJConsoleApp`. Our application needs to handle the `OnWork` event to get notified whenever the console application yields control in order to update a progress bar.

Start a new Delphi GUI application, drop a progress bar and a button on the form. Set the progress bar's `Max` property to 10. As in Example 2 we're going to need a `OnWork` event handler, so create the private method:

```
procedure WorkHandler(Sender: TObject);
```

This time implement the method as follows:

```
procedure TForm1.WorkHandler(Sender: TObject);
begin
  if ProgressBar1.Position = ProgressBar1.Max then
    ProgressBar1.Position := 0
  else
     ProgressBar1.Position := ProgressBar1.Position + 1;
  Application.ProcessMessages;
end;
```

Now create an `OnClick` event handler for the button as follows:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  App: TPJConsoleApp;
begin
  App := TPJConsoleApp.Create;
  try
    App.MaxExecTime := INFINITE;  // don't time out
    App.TimeSlice := 100;         // yield to main app every 1/10 second
    App.Visible := True;          // ensure we see the app
    App.OnWork := WorkHandler;    // assign the event handler
    if not App.Execute('Timed 5') then  // run Timed.exe for 5 seconds
      ShowMessage('Failed to run Timed.exe');
  finally
    App.Free;
  end;
end;
```

This is the same code we used in Example 2 – all the changes are in the `OnWork` event handler. Note we're using `Timed.exe` from Appendix 2 again – change it if you wish.

Click the button to run the console application, switch back to the main form and watch the progress bar.

Example 4: Timing Out

All the examples to date have allowed a console application to run for as long as it required. This was done by setting `TPJConsoleApp`'s `MaxExecTime` property to `INFINITE`. In this example we will look at how to set a maximum time for a console application to run and examine what happens when it times out.

The amount of time available for a console application to run is limited by setting the `MaxExecTime` property (in milliseconds). If this time expires before the console application completes then the `Execute` method returns False, the `ErrorCode` property is set to `$20000001` (`cAppErrorTimeOut`) and the `ErrorMessage` property stores an explanatory string. However, the console application may keep running. Whether this happens depends on the value of the `KillTimedOutProcess` property. If it is True the timed-out process is forcibly terminated. When the property is false the process is left executing, but the link between `TPJConsoleApp` and the process is broken.

To see all this, start a new Delphi GUI application project and drop a `TEdit`, a `TCheckbox`, a `TButton` and a `TMemo` control on the form. Give the `TMemo` a vertical scroll bar.. The edit control will be used to enter the maximum execution time of the program in milliseconds. The check box will set the `KillTimedOutProcess` property and the memo will be used to display output.

As we've done before add a private work handler method to the form class:

```
procedure WorkHandler(Sender: TObject);
```

and implement the method like this:

```
procedure TForm1.WorkHandler(Sender: TObject);
var
  App: TPJConsoleApp;
begin
  App := Sender as TPJConsoleApp;
  Memo1.Lines.Add(
    Format(
      'Elapsed time: %dms, Time to live: %dms',
      [App.ElapsedTime, App.TimeToLive]
    )
  );
  Application.ProcessMessages;  // keeps GUI interactive
end;
```

Now add the following `OnClick` event handler for the button:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  App: TPJConsoleApp;
begin
  App := TPJConsoleApp.Create;
  try
    App.MaxExecTime := StrToInt(Edit1.Text);
    App.KillTimedOutProcess := Checkbox1.Checked;
    App.TimeSlice := 200;
    App.Visible := True;
    App.OnWork := WorkHandler;
    if App.Execute('Timed 5') then
      Memo1.Lines.Add('Application completed normally')
    else
      Memo1.Lines.Add(App.ErrorMessage)
  finally
    App.Free;
  end;
end;
```

Again, `Timed.exe` from Appendix 2 is being used as the console application, set to run for five seconds.

Run the program and enter a maximum execution time that is less than the time the console program will run for (3000 is good if using `Timed.exe`). Leave the checkbox unchecked. Now click the button to execute the console application. Every 1/5th second the memo control will be updated with the time the program has been running and the maximum time left to run ("time to live"). Once the time to live value hits zero the program will time out and the resulting error message will be displayed in the memo control. Notice that the console application will continue to run.

Now check the check box and repeat the process. Once again the program should time out, but this time the console application will be terminated.

Finally, set the maximum execution time to be longer than the console application will run (5000 is good if using `Timed.exe`). Click the button to run the console application. The memo will be updated as before until the console application completes. Normal completion will be reported in the memo.

Warning: Windows may not clean up normally after forcible termination of an application. This is what happens when an application times out when `KillTimedOutProcess` is True. Look up `TerminateProcess` in the Windows API documentation for further information.

Example 5: Terminating an Application

In Example 4 we learned how to set a maximum time to run for a console application and how to forcibly terminate it if it timed out. Unfortunately it's not always possible to second guess how long an application will need to run, and so we may be forced to set the `MaxExecTime` property to `INFINITE`. In these cases it may be appropriate to give the user the choice to terminate the process.

`TPJConsoleApp` provides the `Terminate` method for just this purpose. Calling `Terminate` causes the `Execute` method to return false, the `ErrorCode` property to have value `$20000002` (`cAppErrorTerminated`) and the `ErrorMessage` property to store a suitable message. If the `KillTimedOutProcess` property is False the process is actually left executing, just the link between `TPJConsoleApp` and the process is broken. However when `KillTimedOutProcess` is True the application is forcibly terminated.

To check this out, start a new Delphi GUI application and drop two buttons and a checkbox on the form. The first button will be used to execute a console application and the second button will be used to terminate it (set the seconds button's `Enabled` property to False – we will enable it when the

console application starts). The check box state will determine the value of the `KillTimedOutProcess` property.

It should come as no surprise we need a private work handler method of the form class:

```
procedure WorkHandler(Sender: TObject);
```

This time we simply need to keep the GUI interactive while the console application is running, so the implementation is simply:

```
procedure TForm1.WorkHandler(Sender: TObject);
begin
  Application.ProcessMessages;
end;
```

Now add this `OnClick` event handler for `Button1` (the button we use to execute the console application):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  fApp := TPJConsoleApp.Create;
  try
    Button2.Enabled := True;
    fApp.MaxExecTime := INFINITE;
    fApp.KillTimedOutProcess := Checkbox1.Checked;
    fApp.TimeSlice := 200;
    fApp.Visible := True;
    fApp.OnWork := WorkHandler;
    if fApp.Execute('Timed 5') then
      ShowMessage('Application completed normally')
    else
      ShowMessageFmt('Error %X: %s', [fApp.ErrorCode, fApp.ErrorMessage]);
  finally
    Button2.Enabled := False;
    FreeAndNil(fApp);
  end;
end;
```

We're using `Timed.exe` from Appendix 2 again as the long-running console application.

Finally, add the following `OnClick` event handler for `Button2` (that we will use to terminate the application):

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  if Assigned(fApp) then
    fApp.Terminate;
end;
```

Compile and run the application. Leave the check box clear and click the first button to start the console application. Press the terminate button while the the console application is running. You should get a message box showing the error code `$20000002` and a message noting the application has been terminated. The console application should continue to run, but the link to `TPJConsoleApp` will have been broken.

Now check the check box and run the console application again, pressing the terminate button. This time the same message should be displayed and the console application should be terminated.

Warning: The same warning that was given in Example 4 about terminating processes also applies here.

Example 6: Redirecting Standard I/O using Files

Console applications are often used with redirected input and output. It is quite common for a console application to process content read from standard input and to write the processed data to standard output. When we want to process files we must redirect the input file to standard input and redirect standard output to an output file.

Redirection from the command line is done using the < and > redirection symbols (and similar). For example, to get `MyApp` to read `In.txt` and write `Out.txt` we would use:

```
MyApp <In.txt >Out.txt
```

When redirecting programatically we can't use > and < on the command line, but must instead open handles to the files and pass those handles to Windows. Furthermore, on NT systems, the file handles must be inheritable. See Appendix 1 for details of how to make handles inheritable. `TPJConsoleApp` provides properties `StdIn`, `StdOut` and `StdError` to implement redirection. Simply set the required property to an inheritable handle to redirect standard input, standard output and standard error respectively.

To demonstrate this, we need a console application that processes text from standard input and writes the processed text to standard output. `Echoer.exe`, the description and source code of which is in Appendix 2, is a suitable application. If you have an alternative substitute it in the code below.

Start a new GUI application and drop a button and two memos on the main form.We will enter text to be processed in `Memo1`, write this text to a file, process the file using `Echoer.exe` which will write an output file, then load that file into `Memo2`.

First, copy `OpenInputFile` and `CreateOutputFile` from Appendix 1 into the form unit's implementation section. Next create an `OnClick` event handler for the button as follows:

```
procedure TForm1.Button1Click(Sender: TObject);
const
  cInFile = 'Eg6-in.txt';
  cOutFile = 'Eg6-out.txt';
var
  App: TPJConsoleApp;
begin
  // Save Memo1 to file
  Memo1.Lines.SaveToFile(cInFile);
  // Execute the application
  App := TPJConsoleApp.Create;
  try
    App.Visible := False;
    App.StdIn := OpenInputFile(cInFile);
    App.StdOut := CreateOutputFile(cOutFile);
    if not App.Execute('Echoer ">>> "') then
      raise Exception.CreateFmt(
        'Error %d: %s', [App.ErrorCode, App.ErrorMessage]
      );
  finally
    FileClose(App.StdIn);
    FileClose(App.StdOut);
    App.Free;
  end;
  // Load Memo2 from file
  Memo2.Lines.LoadFromFile(cOutFile);
end;
```

Note that we have not created an `OnWork` handler for this demo. This is because we expect the text processing application to run very quickly and therefore will have a negligible effect on our application responsiveness.

Run the application, enter some text in Memo1 and click the button. The resulting processed file will be displayed in Memo2. The files Eg6-in.txt and Eg6-out.txt will have been created.

Example 7: Redirecting Standard I/O using Pipes

As useful as redirecting files can be (see Example 6), it is not always very convenient for our application to have to exchange data with a console application by writing input data to a file and then reading a processed file. It is much more useful if we can pass the raw data directly to a console app for processing and to read the processed data back from the console application. We can do this using pipes. It's more convoluted than using files, but worth it.

To redirect a console application's input and output we need two pipes:

- The first pipe is used to send data to the console application's standard input. Our application uses the pipe's write handle[6] to write data to the pipe. We set TPJConsoleApp.StdIn to the pipe's read handle to enable the console application to read the data from the pipe.
- The second pipe is used to read data from the console application's standard output. We set TPJConsoleApp.StdOut to the pipe's write handle and we read the data using the read handle.

Because working with pipes can be quite complicated we will use the TPJPipe class to help simplify things.

To see the code working, create a new Delphi GUI application and drop a button and two memos on the form. As with Example 6, Memo1 will receive text to be processed and Memo2 will display the results. As before, this example uses the Echoer.exe console application from Appendix 2. As usual you can substitute another suitable program.

Add the following code to the form class' private section:

```
  private
    fOutPipe: TPJPipe;
    fOutStream: TStream;
    procedure WorkHandler(Sender: TObject);
```

Here, fOutPipe is a the TPJPipe object used to pipe processed data from the console application and fOutStream is a stream that receives data from fOutPipe. WorkHandler is an OnWork event handler that is implemented like this:

```
procedure TForm1.WorkHandler(Sender: TObject);
begin
  fOutPipe.CopyToStream(fOutStream, 0);
end;
```

This handler simply copies all available data from the output pipe to the output stream.

We can't just wait until the program is over before reading all the data from the pipe, because we don't know the size needed for the output pipe or the amount of data written to it in one time slice. It is possible that the data will overflow the pipe. The secret is to make the TimeSlice property of TPJConsoleApp small enough, and create the output pipe big enough, to ensure that the console application never fills the pipe between time slices. Experimentation is usually required.

Now create an OnClick event handler for the button and complete it as follows:

6) Pipes have both a read handle and a write handle. One application writes to the pipe using the write handle while another application reads data from the pipe using the read handle.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  App: TPJConsoleApp;
  Text: string;
  InPipe: TPJPipe;
begin
  fOutStream := nil;
  fOutPipe := nil;
  // Write memo 1 contents into read pipe
  Text := Memo1.Text;
  InPipe := TPJPipe.Create(Length(Text));
  try
    InPipe.WriteData(PChar(Text)^, Length(Text));
    InPipe.CloseWriteHandle;
    // Create out pipe and stream that receives out pipe's data
    fOutPipe := TPJPipe.Create;
    fOutStream := TMemoryStream.Create;
    // Execute the application
    App := TPJConsoleApp.Create;
    try
      App.TimeSlice := 2; // forces more than one OnWork event
      App.OnWork := WorkHandler;
      App.StdIn := InPipe.ReadHandle;
      App.StdOut := fOutPipe.WriteHandle;
      if not App.Execute('Echoer "--> "') then
        raise Exception.CreateFmt(
          'Error %d: %s', [App.ErrorCode, App.ErrorMessage]
        );
    finally
      App.Free;
    end;
    // Load data from output stream into memo 2
    fOutStream.Position := 0;
    Memo2.Lines.LoadFromStream(fOutStream);
  finally
    FreeAndNil(InPipe);
    FreeAndNil(fOutPipe);
    FreeAndNil(fOutStream);
  end;
end;
```

First of all we create the input pipe of the required size and write the contents of `Memo1` to it. We call the pipe's `CloseWriteHandle` method when all the data is written to it. This effectively signals end-of-file on the pipe. Without this the console application would endlessly wait for more data when all the pipe data had been read.

Next we create the output pipe (with default size) and the stream to receive output. As we have seen the pipe's data is copied to the stream in `WorkHandler`. It is because they are used in this separate method that we declare `fOutPipe` and `fOutStream` as fields of the class rather than as local variables.

Now we choose a small `TimeSlice`, and set the `StdIn` and `StdOut` properties to the appropriate pipe handles before executing the application.

Finally we load `Memo2` from the output stream.

Run the program. Type some text in `Memo1` and click the button to see the processed text appear in `Memo2`, this time without writing any intermediate files.

Example 8: Echoing Console Output to a GUI

You may have seen applications (*Inno Setup* springs to mind), that display output from a console application in real time in a GUI control. In this example we'll see how to use `TPJConsoleApp` and `TPJPipe` to do this. Be warned, this is the most complex example we have met.

Here is an overview of what we will do:

- Use a console application that processes text on standard input and writes processed output to standard output. We will redirect standard input to a lengthy text file. We will use `Echoer.exe` from Appendix 2 once more.
- Display the console applications' standard output in a memo control. To do this we will redirect standard output to a pipe and read the pipe in an `OnWork` event, passing output to a helper class that updates the memo control.
- Do the same thing with the program's standard error output, displaying that in another memo control.

Start a new Delphi GUI application and drop a button and two memo controls. We will display standard output in `Memo1` and standard error in `Memo2`.

Copy `OpenInputFile` from Appendix 1 into the form unit's implementation section. We need this to redirect the console application's standard input.

Now add the following declarations to the form class' private section:

```
  private
    fErrPipe, fOutPipe: TPJPipe;
    fOutLines, fErrLines: TTextToLines;
    procedure WorkHandler(Sender: TObject);
    procedure CompletionHandler(Sender: TObject);
```

`TTextToLines` is the class we'll use to provide the output text to the memo controls. Add a new type definition for it above the form declaration in the unit's interface section as follows:

```
type
  TTextToLines = class(TObject)
  private
    fRemainder: string;
    fLines: TStrings;
  public
    procedure AddText(const Text: string);
    procedure Flush;
    constructor Create(const Lines: TStrings);
  end;
```

Implement the class as follows:

```
procedure TTextToLines.AddText(const Text: string);
var
  EOLPos: Integer;
begin
  if Text = '' then
    Exit;
  fRemainder := fRemainder + Text;
  EOLPos := Pos(#13#10, fRemainder);
  while EOLPos > 0 do
  begin
    fLines.Add(Copy(fRemainder, 1, EOLPos - 1));
    fRemainder := Copy(fRemainder, EOLPos + 2, MaxInt);
    EOLPos := Pos(#13#10, fRemainder);
  end;
end;

constructor TTextToLines.Create(const Lines: TStrings);
begin
  inherited Create;
  fLines := Lines;
  fRemainder := '';
end;
```

```
procedure TTextToLines.Flush;
begin
  if fRemainder <> '' then
  begin
    fLines.Add(fRemainder);
    fRemainder := '';
  end;
end;
```

The constructor stores a reference to the `TStrings` object we are to update. `AddText` carves up the text passed to it into lines and adds each line to the string list. Any remaining text after the last end-of-line character is recorded in the `fRemainder` field. `Flush` is called when there is no more text to add and simply adds any remaining text to the string list. We need this method because we can't guarantee that the text output from the console application between time slices will be whole lines.

We need to create two instances of this class – one to handle standard output text and one for standard error. So, create `OnCreate` and `OnDestroy` event handlers for the form and add the following code:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  fOutLines := TTextToLines.Create(Memo1.Lines);
  fErrLines := TTextToLines.Create(Memo2.Lines);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  FreeAndNil(fErrLines);
  FreeAndNil(fOutLines);
end;
```

Having got the helper class out of the way it's time to look at the meat of the code. Create an `OnClick` handler for the button and add the following code:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  App: TPJConsoleApp;
begin
  fOutPipe := nil;
  fErrPipe := nil;
  try
    fOutPipe := TPJPipe.Create;
    fErrPipe := TPJPipe.Create;
    App := TPJConsoleApp.Create;
    try
      App.StdIn := OpenInputFile('InputFile.txt');
      App.StdOut := fOutPipe.WriteHandle;
      App.StdErr := fErrPipe.WriteHandle;
      App.OnWork := WorkHandler;
      App.OnComplete := CompletionHandler;
      App.TimeSlice := 1;
      if not App.Execute('Echoer') then
        raise Exception.CreateFmt(
          'Error %X: %s', [App.ErrorCode, App.ErrorMessage]
        );
    finally
      FileClose(App.StdIn);
      App.Free;
    end;
  finally
    FreeAndNil(fErrPipe);
    FreeAndNil(fOutPipe);
  end;
end;
```

Most of this code should by now be familiar. We redirect a file to standard input. Replace `InputFile.txt` with some suitably long text file. Two default sized pipes are created into which we

redirect the console application's standard output and standard error. We assign `WorkHandler` to the `OnWork` event. In this example we also need an `OnComplete` event handler – we use `CompletionHandler` for this. A very short `TimeSlice` is used to make output as smooth as possible. Next we execute the console application before closing the input file.

Finally, we need to implement the two event handlers – `WorkHandler` and `CompletionHander`. Implement them as follows:

```
procedure TForm1.WorkHandler(Sender: TObject);

  procedure ProcessPipe(const Pipe: TPJPipe;
    const LineHandler: TTextToLines);
  var
    Text: string;
    BytesToRead: Cardinal;
    BytesRead: Cardinal;
  begin
    BytesToRead := Pipe.AvailableDataSize;
    SetLength(Text, BytesToRead);
    Pipe.ReadData(PChar(Text)^, BytesToRead, BytesRead);
    if BytesRead > 0 then
    begin
      SetLength(Text, BytesRead);
      LineHandler.AddText(Text);
    end;
  end;

begin
  ProcessPipe(fErrPipe, fErrLines);  // Read from standard error
  ProcessPipe(fOutPipe, fOutLines);  // Read from standard output
  Application.ProcessMessages;       // Let the memo controls update
end;

procedure TForm1.CompletionHandler(Sender: TObject);
begin
  fOutLines.Flush;
  fErrLines.Flush;
end;
```

`WorkHandler` is where the meat of the code lies. There's a local procedure – `ProcessPipe` – that handles the pipes connected to standard output and standard error in the same way. Essentially `ProcessPipe` copies text from the pipe to the memo controls, via the appropriate `TTextToLines` instance. Firstly, all available data from the appropriate pipe is copied to a string. If any data was actually read (and it may be less than expected) we make sure the string is the correct size and pass it to the `AddText` method of the appropriate `TTextToLines` instance.

`CompletionHandler` is called after the console application has completed executing (or has timed out). We use it to flush any remaining text from the `TTextToLines` instances to ensure it is added to the memo controls.

Run the application and watch the output appear in the two memo controls.

Example 9: Subclassing TPJConsoleApp

Most of the previous examples have had to handle events to provide some of the needed functionality. In the later cases they have also had to manipulate pipes. This is all very well, but it does rather clutter up the main form code. There are a couple of alternatives that move the messy code out of the form:

1. Encapsulate all the code in another class that creates a `TPJConsoleApp` and supplies the required event handlers. This new class can then provide a simplified interface to the main form.

2. Derive a new class from `TPJCustomConsoleApp` and override methods to provide the required functionality.

There are benefits to both approaches. In this example we're going to look only at the latter case and derive a new class from `TPJCustomConsoleApp`.

Look again at Example 7. In essence what this code does is:

1. Read the data out of `Memo1`.
2. Pass that data to a pipe
3. Set up an output pipe and stream
4. Execute the console application
5. Write the resulting data to `Memo2`

We can replace this with the following:

1. Read data out of `Memo1` into an input stream
2. Create an output stream
3. Call a method of another object that uses the console app to process the input stream and write the output stream
4. Write the resulting data to `Memo2`

This has moved all knowledge of how to load streams into pipes and vice versa out of the form. We can create a descendant of `TPJCustomConsoleApp` that performs the task at item 3 above.

In this example we will recreate Example 7 by subclassing `TPJCustomConsoleApp`.

Start a new Delphi GUI project and add a new unit, named `UConsoleAppEx.pas`. Add the following to its interface section:

```
uses
  Classes, PJConsoleApp, PJPipe;

type
  TConsoleAppEx = class(TPJCustomConsoleApp)
  private
    fOutPipe: TPJPipe;
    fOutStream: TStream;
  protected
    procedure DoWork; override;
  public
    function Execute(const CmdLine: string; const InStream,
      OutStream: TStream): Boolean;
    // Make protected properties public
    property ErrorCode;
    property ErrorMessage;
    property TimeSlice;
  end;
```

The `Execute` method replaces the one in the base class. It receives the command we are to execute as well as an input stream containing data to be piped to the console process' standard input and a stream to receive data piped from standard output. We also make three required properties public.. (All properties of `TPJCustomConsoleApp` are protected.) The `DoWork` method is overridden to read the output pipe. This replaces the `OnWork` event handler in Example 7. In fact `DoWork` triggers the `OnWork` event handler in the base class.

`TConsoleAppEx` is implemented as follows:

```
uses
  SysUtils;

procedure TConsoleAppEx.DoWork;
begin
  fOutPipe.CopyToStream(fOutStream, 0);
end;

function TConsoleAppEx.Execute(const CmdLine: string; const InStream,
  OutStream: TStream): Boolean;
begin
  fOutStream := OutStream;
  InPipe := nil;
  fOutPipe := nil;
  try
    // Set up input pipe and associated with console app stdin
    InPipe := TPJPipe.Create(InStream.Size);
    InPipe.CopyFromStream(InStream, 0);
    InPipe.CloseWriteHandle;
    // Set up output pipe and associate with console app stdout
    fOutPipe := TPJPipe.Create;
    StdIn := InPipe.ReadHandle;
    StdOut := fOutPipe.WriteHandle;
    // Run the application
    Result := inherited Execute(CmdLine);
  finally
    StdIn := 0;
    StdOut := 0;
    FreeAndNil(fOutPipe);
    FreeAndNil(InPipe);
  end;
end;
```

`DoWork` is straightforward – it copies the current pipe contents to the output stream.

`Execute` now contains some of the set up code that was in the main form code in Example 7.

First we record a reference to the output stream for later use. Next we create the pipe associated with standard input and copy all the data from `InStream` to it. The output pipe is then created before associating pipe handles with the console application's standard input and output. Finally, the `Execute` method of the base class is called to execute the console application.

Now switch back to the main form, drop a button and two memos on it then create an `OnClick` event handler for the button, with the following code:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  App: TConsoleAppEx;
  InStream: TStream;
  OutStream: TStream;
begin
  // Set up i/o streams
  InStream := nil;
  OutStream := nil;
  try
    InStream := TMemoryStream.Create;
    Memo1.Lines.SaveToStream(InStream);
    InStream.Position := 0;
    OutStream := TMemoryStream.Create;
    // Run console app
    App := TConsoleAppEx.Create;
    try
      App.TimeSlice := 2;
      if not App.Execute('Echoer "-->"', InStream, OutStream) then
        raise Exception.CreateFmt(
          'Error %X: %s', [App.ErrorCode, App.ErrorMessage]
        );
    finally
      App.Free;
    end;
    // Read output stream
    OutStream.Position := 0;
    Memo2.Lines.LoadFromStream(OutStream);
  finally
    InStream.Free;
    OutStream.Free;
  end;
end;
```

First a stream is created and the contents of `Memo1` are copied into it. Next the empty output stream is created. Then we run the console application using our new class. And finally we display the data from the output stream in `Memo2`.

In the above code `Echoer.exe` from Appendix 2 has been used once again.

Run the application. It should perform exactly the same as Example 7.

Example 10: TPJConsoleApp & Console Applications

All our example applications so far have been Windows GUI applications. You shouldn't get the impression that you can only use `TPJConsoleApp` with GUI applications – it works perfectly well with console applications and this example will show.

Start a new Delphi console application. Now create a new unit in the project and save it as `UMain.pas`. Add the following class declaration to the new unit's interface:

```
type
  TMain = class(TObject)
  private
    fVisible: Boolean;
    fNewConsole: Boolean;
    fTitle: string;
    procedure WorkHandler(Sender: TObject);
    procedure ParseCommandLine;
  public
    procedure Execute;
  end;
```

This class parses the program command line then uses `TPJConsoleApp` to run the `Timed.exe` program from Appendix 2. Change this program if you wish, but note we need a program that runs for at least 2 or 3 seconds. Add the following implementation of `TMain`:

```
uses
  SysUtils, PJConsoleApp;

procedure TMain.Execute;
var
  App: TPJConsoleApp;
begin
  ParseCommandLine;
  App := TPJConsoleApp.Create;
  try
    // Set properties based on parameters
    App.Visible := fVisible;
    App.UseNewConsole := fNewConsole;
    App.ConsoleTitle := fTitle;
    // Set OnWork handler
    App.OnWork := WorkHandler;
    // Run the application
    WriteLn('Starting Timed.exe');
    if not App.Execute('Timed 3') then
      raise Exception.CreateFmt(
        '%X: %s', [App.ErrorCode, App.ErrorMessage]
      );
    WriteLn('Timed.exe completed with exit code: ', App.ExitCode);
  finally
    App.Free;
  end;
end;

procedure TMain.ParseCommandLine;
var
  Param: string;
  Idx: Integer;
begin
  for Idx := 1 to ParamCount do
  begin
    Param := ParamStr(Idx);
    if Param = '-v' then
      fVisible := True
    else if Param = '-n' then
      fNewConsole := True
    else if (Param <> '') and (Param[1] <> '-') then
      fTitle := Param
    else
      raise Exception.CreateFmt('Unknown parameter "%s"', [Param]);
  end;
end;

procedure TMain.WorkHandler(Sender: TObject);
var
  App: TPJConsoleApp;
begin
  App := Sender as TPJConsoleApp;
  if App.UseNewConsole then
    WriteLn('Waited ', App.ElapsedTime, ' ms for Timed.exe');
end;
```

The `Execute` method creates a `TPJConsoleApp` instance then sets properties based on the parameters passed on the command line. An `OnWork` event handler is set before executing the application.

`WorkHandler` checks to see if the console application is sharing the main program's console. If so the handler does not write any output – this would interfere with text written by the child process. If the child process is displaying in its own console (which could be hidden if the `Visible` property is False), then the event handler writes out the elapsed time to the main console.

The `ParseCommandLine` method checks the command line for valid commands and sets the appropriate field values. Valid parameters are:

Parameter	Description
`-v`	The child process is visible. If this switch is not provided the child process is not visible. If the child process is displayed in the host program's console this switch is ignored.
`-n`	Displays the child process in its own console window. If this switch is not provided the child process outputs to the host's console window.
`<title>`	Text to be displayed in the console window's title bar. If not supplied the child process' console window has its default title. Any title is ignored if the child process does not have its own console. Quote any title that contains spaces.

Now switch back to the project file and enter the following code in the program body:

```
begin
  try
    WriteLn('TPJConsoleApp Example 10');
    with TMain.Create do
      try
        Execute;
      finally
        Free;
      end;
    WriteLn('TPJConsoleApp Example 10: Done');
  except
    on E: Exception do
      WriteLn('ERROR: ', E.Message);
  end;
end.
```

This code simply displays some text before executing the main code in `TMain` and signing off. The code also handles any exceptions.

Run the application with various parameters and note its behaviour. See the table above for details of the command line parameters.

Appendices

Appendix 1: Inheritable Handles

To make a handle inheritable we must create it with security attributes that specify that it is inheritable.

For example, we can open a file with an inheritable handle as follows:

```
function OpenInputFile(const FileName: string): THandle;
var
  Security: TSecurityAttributes;  // file's security attributes
begin
  // Set up security structure so file handle is inheritable (NT)
  Security.nLength := SizeOf(Security);
  Security.lpSecurityDescriptor := nil;
  Security.bInheritHandle := True;
  // Open the file for reading
  Result := CreateFile(
    PChar(FileName),        // file name
    GENERIC_READ,           // readable file
    FILE_SHARE_READ,        // share read access
    @Security,              // security attributes (NT only)
    OPEN_EXISTING,          // file must exist
    FILE_ATTRIBUTE_NORMAL,  // just the normal attributes
    0                       // no template file (NT only)
  );
  if Result = INVALID_HANDLE_VALUE then
    raise Exception.CreateFmt('Can''t open file "%s"', [FileName]);
end;
```

The crucial pieces of code are highlighted. Regardless of whether you are opening a file or a pipe for reading or writing you always need to set up the security attributes as shown and pass the structure to the relevant API call.

Here is another function, this time to create new files for writing:

```
function CreateOutputFile(const FileName: string): THandle;
var
  Security: TSecurityAttributes;  // file's security attributes
begin
  // Set up security structure so file handle is inheritable (NT)
  Security.nLength := SizeOf(Security);
  Security.lpSecurityDescriptor := nil;
  Security.bInheritHandle := True;
  // Create new empty file
  Result := CreateFile(
    PChar(FileName),        // file name
    GENERIC_WRITE,          // writeable file
    0,                      // no sharing
    @Security,              // security attributes (NT only)
    CREATE_ALWAYS,          // always create the file, even if exists
    FILE_ATTRIBUTE_NORMAL,  // just the normal attributes
    0                       // no template file (NT only)
  );
  if Result = INVALID_HANDLE_VALUE then
    raise Exception.CreateFmt('Can''t create file "%s"', [FileName]);
end;
```

Appendix 2: Console Application Source Code

This appendix provides source code for two console applications that can be used with the various examples presented above.

Timed.exe

Usage:

```
Timed [<seconds>]
```

Where:

- `<seconds>` = valid positive whole number.

This program runs for the number of seconds passed on the command line, or for 5 seconds if no such parameter is provided. It is useful for examples that require an application that runs for a reasonable amount of time. The program outputs a sign-on message, then a full-stop character approx every 1/10th second and writes "Done" when it completes.

Here is the source code:

```
program Timed;

{$APPTYPE CONSOLE}

uses
  SysUtils, Windows;

var
  TimeToRun: Integer; // time program is to run for in ms
  StartTick: Integer; // tick count when program starts
  TickNow: Integer;   // tick count during each program loops
begin
  TimeToRun := 1000 * StrToIntDef(ParamStr(1), 5);
  ExitCode := 0;
  WriteLn('TIMED: Running for ', TimeToRun div 1000, ' seconds');
  StartTick := GetTickCount;
  repeat
    TickNow := GetTickCount;
    Sleep(100);
    Write('.');
  until TickNow - StartTick >= TimeToRun;
  WriteLn;
  WriteLn('Done');
end.
```

Echoer.exe

Usage:

```
Echoer [<prefix>]
```

Where:

- `<prefix>` = Prefix text. (Quote the text if it contains spaces).

This program copies text read from standard input, prepends a prefix passed on the command line to each line of text. It then writes the modified text to standard output.

If no parameter is provided then lines of text are preceded by the '>' character. The program also writes some sign-on and sign-off messages to standard error, which is useful for examples that redirect standard error output.

Here is the source code:

```
program Echoer;

{$APPTYPE CONSOLE}

uses
  Windows, SysUtils;


{ Emulates C std lib stderr value by returning appropriate Windows handle }
function StdErr: Integer;
begin
  Result := Windows.GetStdHandle(STD_ERROR_HANDLE);
end;

{ Writes a single character or string to a file }
procedure WriteStr(Handle: THandle; const S: string);
var
  Dummy: DWORD;
begin
  Windows.WriteFile(Handle, PChar(S)^, Length(S), Dummy, nil);
end;

{ Writes a single character or string to a file followed by a newline }
procedure WriteStrLn(Handle: THandle; const S: string);
begin
  WriteStr(Handle, S + #13#10);
end;

var
  Prefix: string;
  Line: string;
  Count: Integer;
  ProgName: string;
begin
  ProgName := ExtractFileName(ParamStr(0));
  WriteStrLn(StdErr, ProgName + ' - Starting');
  Prefix := ParamStr(1);
  if Prefix = '' then
    Prefix := '>';
  WriteStrLn(StdErr, ProgName + ' - Using prefix: "' + Prefix + '"');
  Count := 0;
  while not EOF do
  begin
    Inc(Count);
    ReadLn(Line);
    WriteLn(Prefix, Line);
  end;
  WriteStrLn(StdErr, ProgName + ' - ' + IntToStr(Count) + ' lines written');
  WriteStrLn(StdErr, ProgName + ' - Finished');
end.
```