Library support in Pas2JS

Michaël Van Canneyt

March 2022

Abstract

With version 2.2, Pas2JS introduces library support in the compiler. Libraries in Pas2JS translate to Javascript modules: independent blocks of Javascript code which must be explicitly imported in another block. In this article we show to use them.

1 Introduction

For the experienced pascal programmer, using libraries is not uncommon. Till recently, using libraries in Pas2JS was not possible. With release 2.2 (released on 22-02-2022) of Pas2JS, libraries can now also be used in Pas2JS.

For the pascal programmer, libraries - DLLs in windows - are independent programs which export certain functions and variables. In Javascript, a similar concept exists: Modules. Modules can import symbols from other modules, and can export symbols to other modules.

It is therefore natural to transpile a Pascal library to a Javascript module, and this is now what can be done with Pas2JS:

- Import symbols from a module.
- Create a module that exports functions and variables.

In difference with Delphi, no special precautions are needed for using strings or classes in a Pas2JS library: in particular, there is no need to enable a module to use shared memory.

2 Javascript modules

Javascript modules are nothing but Javascript files which export a number of symbols, but which otherwise do not share any code or namespaces. Especially the latter is important.

By default, if you link 2 Javascript scripts to a HTML page:

```
<script src="script1.js"></script>
<script src="script2.js"></script>
```

then the code `script1` has access to all symbols (variables, functions etc.) of `script2`, and vice versa. This means they can modify or even annihilate each others' working.

With Javascript modules, this is not the case. Take the following HTML snippet:

```
<script type="module" src="script1.js"></script>
<script type="module" src="script2.js"></script>
<script src="script3.js"></script>
```

Here `script1`, `script2` and `script3` have distinct namespaces. They do not interfere with each other: both `script1` and `script2` can have a variable `MyVar`, but each has a local copy of this variable. If `script2` writes to `MyVar`, it will only modify its own copy.

What is more, `script3` has no access to the symbols defined in `script1` and `script2`. Only modules can import symbols of other modules. Imagine `script1.js` has the following content

```
export const MyText1 = "Hello,";
export const MyText2 = " World!";
```

As you can see, it exports 2 constants, `MyText1` and `MyText2`.

This means `script2.js` can use these constants as follows:

```
import { MyText1, MyText2 } from  "./script1.js";

document.title = MyText1+MyText2;
```

When loading `script2`, the browser will also automatically load `script1.js`, there is no need to include it explicitly in the HTML file. The file `script1` must of course exist in the specified location.

In contrast with `script2`, `script3` can never access the symbols, because it is not a module itself. Only modules can import and export symbols.

It is of course possible to share some symbols between modules and non-modules by attaching them to a global symbol such as the `window`.

3 Importing libraries

To import symbols from a module (written in Pascal or not) 2 things are needed:

- a `linklib` directive:

  ```
  {$linklib ./my-file.js myfile}
  ```

  this will be transformed to the following Javascript

  ```
  import * as myfile from "./my-file.js";
  ```

  Javascript supports some more fine-grained import statements, but these are not yet supported in Pas2JS. The `myfile` name is optional, in which case the filename without path or extension will be used.

- an external declaration for each function or variable exported by the module (the declatation has been split over 2 lines for readability):

  ```
  Function MyFunction (S: String) : Integer;
     external name 'myfile.myFunction';

  var
    MyVar : String; external name 'myfile.myVar';
  ```

  Note that these external names contain the prefix 'myfile.' as part of their name: this is because all symbols of the module are available as `myfile.NNN`, due to the way the `import` statement is constructed from the `{$Linklib }` directive.

Note that the use of the `{$Linklib }` directive also requires the use of the `module` compiler target. More about this below.

4 Writing libraries

To write a library using Pas2JS, you can write a library just as you would in Delphi or Free Pascal, using the `library` keyword, instead of the default `program` keyword:

```
library htmlutils;

{$mode objfpc}

// Your exports Here
//  exports a, b, c;

begin
  // Your library initialization code here
end.
```

However, this is not enough. A new transpiler target was introduced: `module`.

The reason for introducing a new target is the following:

Depending on the target, the transpiler will include a Pas2JS `rtl.run();` statement at the end of the generated Javascript (or not). The output for the `nodejs` target includes such a statement, but the `browser` target does not - because, as a rule, the `rtl.run()` statement is included in the .html file: this will ensure that HTML tags and their ids have been processed by the browser before the program is run.

Since a library (or module) can be used both in node.js and in the browser, a new target has been created: `module`. This target will always include the `rtl.run()` statement.

The `{$Linklib }` directive also requires the use of the `module` target. No `import` statement will be generated, unless the target is set to `module`.

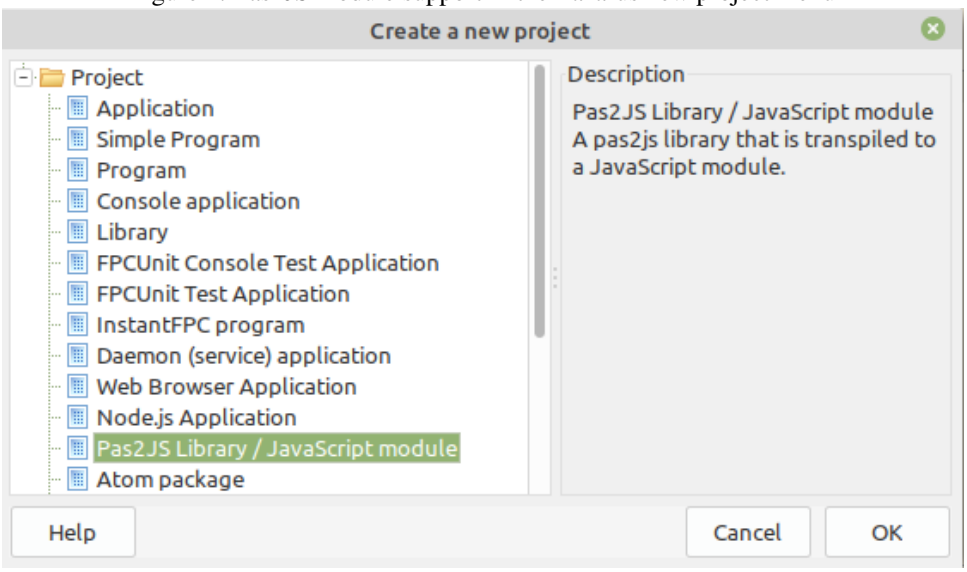
5 Creating Javascript modules using Pascal

So, how to use libraries and `{$Linklib }` directives to create Javascript modules? We will demonstrate this with an example.

We create a library that allows to clear the HTML page below a certain tag (identified by it's `id` attribute), and which allows to set the page title. This is quite simple:

```
library htmlutils;

uses web;

Var
  DefaultClearID : String;

Procedure SetPageTitle(aTitle : String);

begin
  Document.Title:=aTitle;
end;

Procedure ClearPage(aBelowID : String);

Var
```

Figure 1: Pas2JS module support in the Lazarus new project menu




```
  EL : TJSElement;

begin
  if (aBelowID='') then
    aBelowID:=DefaultClearID;
  if (aBelowID='') then
    el:=Document.body
  else
    el:=Document.getElementById(aBelowID);
  if Assigned(El) then
    El.innerHTML:='';
end;

exports
  DefaultClearID, SetPageTitle, ClearPage;

end.
```

To demonstrate the export of variables, we also export a variable `DefaultClearID`. The value of this variable is used to determine which HTML tag to clear. If it is not set, and no tag ID was specified in the call to `ClearPage`, the whole HTML body element is cleared.

This library can be compiled with the `-Tmodule` target:

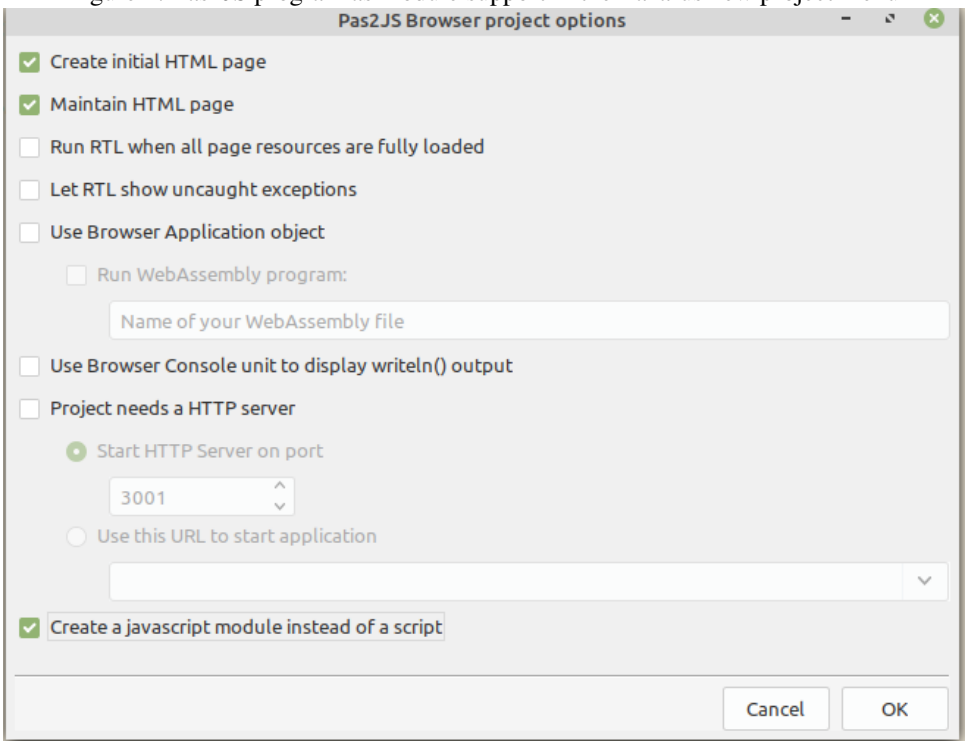
```
/home/michael/bin/Pas2JS -Tmodule -Jirtl.js -Jc htmlutils.pas
```

The lazarus IDE has support for creating a Pas2JS library in the `Project-New project` menu, which will set all necessary options, as can be seen in figure 1 on page 4.

As indicated earlier, to use a library (or module), we must use again a Javascript module: only javascript modules can use other modules. To create this module, we have 2 options:

- Create another library.
- Create a program.

Figure 2: Pas2JS program as module support in the Lazarus new project menu



It is clear why a library will work: the Javascript script will need to have the `module` type and must be compiled with the `module` target. However, a program will also work. From the Javascript point of view, there is no difference between a library and a program. From a pascal point of view, the only factor of importantce is whether you want to export symbols from your module. If you do, then you must create a library.

For demonstration purposes, we'll create a program, because the Lazarus IDE wizard then also creates a HTML page which we will need to show the functionality of our library.

In older versions of Lazarus the `TargetOS` of our program must manually be set to `module` in the compiler options. In the latest (trunk, hence not yet released) version, the `Project - New project` dialog already offers an option which does this for you, see figure 2 on page 5.

We start by creating all code that is needed to import the library:

```
program htmlutilsdemo;

{$mode objfpc}
{$linklib ./htmlutils.js utils}

uses
  Web;

Procedure SetPageTitle(aTitle : String);
    external name 'utils.SetPageTitle';
Procedure ClearPage(aBelowID : String);
    external name 'utils.ClearPage';
```

```
var
  DefaultClearID : string;
    external name 'utils.vars.DefaultClearID';
```

Note the `utils.vars.DefaultClearID`: the prefix `vars` is needed for all variables exported by a Pas2JS-created library.

To use these routines, we create a HTML page with 2 edits (IDs `edtTitle`, `edtBelowID`) and a checkbox (ID `cbUserDefaultClearID`) and 2 buttons (IDs `btnSetTitle` and `btnClear`). These edits can be used to specify a page title and an element ID, the `onclick` event handlers of the buttons will call our imported functions.

The element definitions are bound to the HTML tags in the `BindElements` function:

```
Var
  BtnSetTitle,BtnClear : TJSHTMLButtonElement;
  edtTitle,edtBelowID,cbUseDefaultClearID : TJSHTMLInputElement;

Procedure BindElements;

begin
  TJSElement(BtnSetTitle):=Document.getElementById('btnSetTitle');
  BtnSetTitle.OnClick:=@DoSetTitle;
  TJSElement(BtnClear):=Document.getElementById('btnClear');
  BtnClear.onclick:=@DoClear;
  TJSElement(edtTitle):=Document.getElementById('edtTitle');
  TJSElement(edtBelowID):=Document.getElementById('edtBelowID');
  TJSElement(cbUseDefaultClearID):=
    Document.getElementById('cbUseDefaultClearID');
end;
```

The `BindElements` function is called in the program startup code.

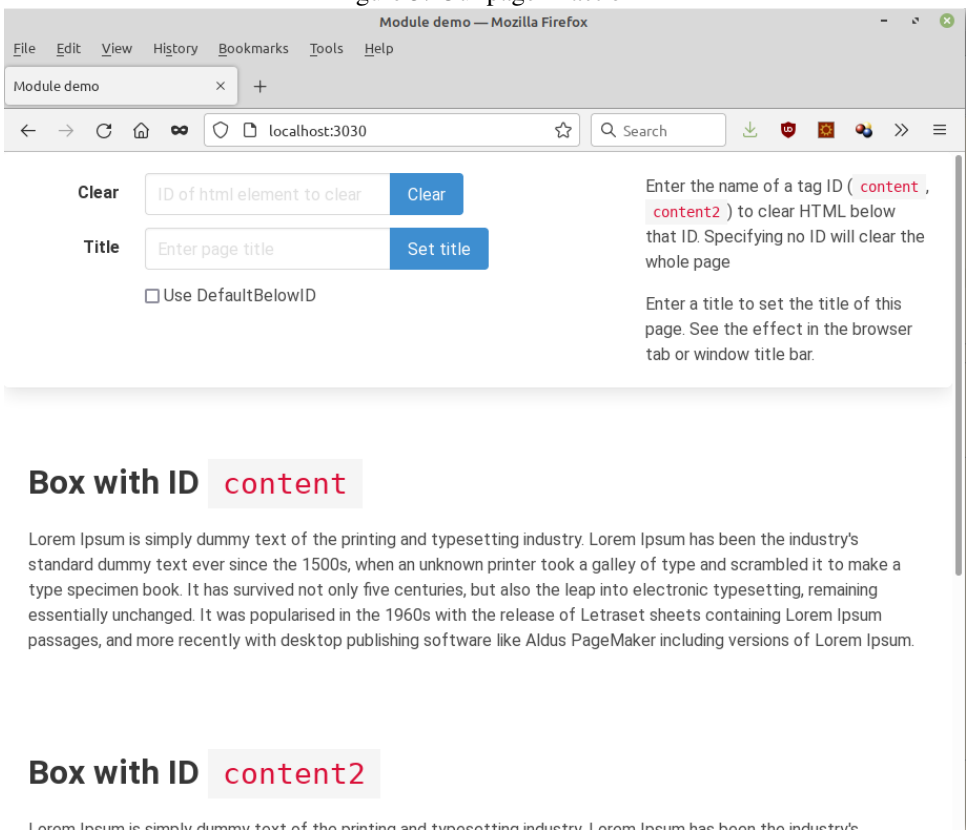
The `DoSetTitle` and `DoClear` methods are callbacks that will call our imported function:

```
function DoSetTitle(aEvent: TJSMouseEvent): boolean;
begin
  Result:=False;
  SetPageTitle(edtTitle.Value);
end;
```

The `DoClear` function is a little longer, since it must take into account the value of the `cbUseDefaultClearID` element:

```
function DoClear(aEvent: TJSMouseEvent): boolean;
begin
  Result:=False;
  if cbUseDefaultClearID.Checked then
    begin
    DefaultClearID:=edtBelowID.value;
    ClearPage('');
    end
  else
    begin
    DefaultClearID:='';
```

Figure 3: Our page in action




```
    ClearPage(edtBelowID.value);
    end;
end;
```

The HTML will not be presented here, except to show that the script tag must be modified, the type of the script must be set to `module`:

```
<script type="module" src="htmlutilsdemo.js"></script>
```

(again, in the latest development version of Lazarus, this is already done for you).

The resulting HTML page can be seen in see figure 3 on page 7, it is available online at

`https://www.freepascal.org/~michael/pas2js-demos/modules/htmlutils/`

6 Exporting classes

In the `exports` statement only variables and functions can be specified. Despite this restriction, it is possible to use classes which are exported from libraries.

The simplest way to do so is to create a function that creates an instance of a class. Alternatively, for global instances, you can declare a variable of type of the desired class.

To demonstrate this, we'll rewrite our example to use a class called `THTMLUtils`:

```
library htmlutils;

uses
  web;

Type
   THTMLUtils = class(TObject)
   Public
     DefaultClearID : String;
     Procedure SetPageTitle(aTitle : String);
     Procedure ClearPage(aBelowID : String);
   end;

Procedure THTMLUtils.SetPageTitle(aTitle : String);

begin
  Document.Title:=aTitle;
end;

Procedure THTMLUtils.ClearPage(aBelowID : String);

Var
  EL : TJSElement;

begin
  if (aBelowID='') then
    aBelowID:=DefaultClearID;
  if (aBelowID='') then
    el:=Document.body
  else
    el:=Document.getElementById(aBelowID);
  if Assigned(El) then
    El.innerHTML:='';
end;
```

Since we cannot export a class directly from our module, in order for users of the library to be able to use the class, we must export a function that creates an instance of the class:

```
Function CreateUtils : THTMLUtils;

begin
  Result:=THTMLUtils.Create;
end;

exports
  CreateUtils;

end.
```

Obviously, if you need to specify options to your class' constructor you'll need to define these options in your function.

> *Note:* Due to a bug in the released Pas2JS compiler it is necessary to disable optimizations when compiling this library: in the `Custom options` part

of the compiler options dialog, the `-O-` option must be added. This bug has meenwhile been fixed.

To use this class, we must also rewrite our program. We start by defining the `THTMLUtils` class as an external class:

```
program htmlutilsdemo;

{$mode objfpc}
{$linklib ./htmlutils.js utils}
{$modeswitch externalclass}

uses
  JS, Web;

type
  THTMLUtils = class external name 'Object' (TJSObject)
  Public
    DefaultClearID : String;
    Procedure SetPageTitle(aTitle : String);
    Procedure ClearPage(aBelowID : String);
  end;

Function CreateUtils : THTMLUtils; external name 'utils.CreateUtils';
```

Note the use of the `{$moduleswitch externalclass}`, needed to be able to define external classes.

Now, to use this class, we must also rewrite our program a little. We define a variable of the class, which we use in our callbacks:

```
Var
  BtnSetTitle,BtnClear : TJSHTMLButtonElement;
  edtTitle,edtBelowID,cbUseDefaultClearID : TJSHTMLInputElement;
  UtilsObj : THTMLUtils;

function DoSetTitle(aEvent: TJSMouseEvent): boolean;
begin
  Result:=False;
  UtilsObj.SetPageTitle(edtTitle.Value);
end;

function DoClear(aEvent: TJSMouseEvent): boolean;
begin
  Result:=False;
  if cbUseDefaultClearID.Checked then
    begin
    UtilsObj.DefaultClearID:=edtBelowID.value;
    UtilsObj.ClearPage('');
    end
  else
    begin
    UtilsObj.DefaultClearID:='';
    UtilsObj.ClearPage(edtBelowID.value);
    end;
```

```
end;
```

We initialize the variable with the `CreateUtils` call exported from our library:

```
begin
  UtilsObj:=CreateUtils;
  BindElements;
end.
```

The resulting page works in exactly the same way as the original example, only now it uses a class.

You can test this at:

`https://www.freepascal.org/~michael/pas2js-demos/modules/classes/`

For this simple example, exporting a variable of the correct type is also sufficient. It requires only a few changes. In the library, the `CreateUtils` function can be replaced with an exported variable declaration:

```
var
  Utils : THTMLUtils;

exports
  Utils;

initialization
  Utils:=THTMLUtils.Create;
end.
```

The variable is initializd in the initialization section of the library.

To use this variable, only a small change is needed in our program. We remove the 'CreateUtils' function, and change the declaration of the `UtilsObj` variable:

```
var
  UtilsObj : THTMLUtils; external name 'utils.vars.Utils';
```

And of course the statement to assign the variable must be removed.

After these changes, again the example will function as the original example. You can convince yourself at the live demo:

`https://www.freepascal.org/~michael/pas2js-demos/modules/classusingvar/`

7 Conclusion

In this article we've shown one of the latest features of the Pas2JS transpiler: libraries and how to use them. We've also shown that libraries in Pas2JS are more powerful than libraries in native code: there is no need for special memory managers, and classes can be used as-is . There are some small glitches in the library support for classes: the optimization switch, and using overloads is possible but requires some tweaking of the external names. Despite this, the support for modules is sufficiently mature to be used in production.