Leveraging TypeScript declarations in Pas2JS

Michaël Van Canneyt

22/02/2022

Abstract

Pas2js contains a tool to convert TypeScript declaration modules to a pascal unit with external class definitions. This can be used to create import units for many Javascript libraries. In this article, we show how to use this tool.

1 Introduction

To say that there are a lot of free Javascript libraries or frameworks out there is an understatement. Normally, any Javascript class or function can be used in Pas2JS: By falling back on assembler blocks, any Javascript function can be called. The transpiler will happily insert any Javascript in your final transpiled code. But if the transpiler has external declarations for the Javascript classes or functions, the transpiler can and will check your code against the definitions it has.

Plain Javascript has a major drawaback: it is not typesafe. To remedy this, people at Microsoft created TypeScript: a type system for Javascript. It is a superset of Javascript, which is transpiled to Javascript. (One of the authors of TypeScript was also one of the creators of Delphi)

This type system is made popular by Angular and other large Javascript frameworks. People writing TypeScript code face the same problem as Pas2JS users: how to make use of the many Javascript libraries, and still write Typesafe code?

The answer to this problem are declaration modules (files with extension .d.ts): these modules do not implement any functionality. They just describe the API offered by an external Javascript library. The TypeScript compiler reads this declaration and uses it to validate the TypeScript code that makes use of the Javascript library: It serves exactly the same purpose as a pas2js unit with external classes.

Many plain Javascript libraries offer such a TypeScript declaration module in their distribution. But there are also a lot of libraries that do not offer such a declaration module.

But because there are a lot of TypeScript programmers, there is an ongoing effort to describe these javascript libraries: the DefinitelyTyped repository on Github.

It is available at:

`https://github.com/DefinitelyTyped/DefinitelyTyped/`

It contains many tens of thousands of declaration modules. TypeScript programmers that wish to use a javascript library can just check out this repository and use the declaration module of the package they wish to use in their project.

Pas2JS could use a similar repository of import units. Indeed, ideally, the TypeScript declaration modules can just be re-used so all the hard work of all these TypeScript would benefit the pas2js users as well.

Fortunately, this is possible to a certain extent: The upcoming version of pas2js comes with a tool that converts a TypeScript declaration module to a pascal unit with external definitions: `dtstopas`.

Better yet, an online service exists which makes this possible today.

Last but not least, the tool and the webservice have been integrated in the Lazarus IDE. You can create an import unit directly in your project from within the Lazarus IDE: Simply use the `File-New` menu item.

We'll discuss each of these possibilities in turn.

2 dts2pas

The `dts2pas` tool is a small command-line tool which will transform a *.d.ts file to a pascal unit. Running it without options (or option `-h`) gives the following output:

```
Usage: dts2pas [options]
Where options is one or mote of:
-a --alias=ALIAS      Define type aliases (option can be speficied multiple times
                      where ALIAS is one of
                      a comma-separated list of Alias=TypeName values
                      a @FILE : list is read from FILENAME, one line per alias
-h --help             Display this help text
-i --input=FILENAME   Parse .d.ts file FILENAME
-l --link=FILENAME    add {$linklib FILENAME} statement. (option can be specified
-o --output=FILENAME  Output unit in file FILENAME
-s --setting=SETTINGS Set options. SETTINGS is a comma-separated list of the foll
                      coRaw
                      coGenericArrays
                      coUseNativeTypeAliases
                      coLocalArgumentTypes
                      coUntypedTuples
                      coDynamicTuples
                      coExternalConst
                      coExpandUnionTypeArgs
                      coaddOptionsToheader
                      coInterfaceAsClass (*)
                      coSkipImportStatements
                      Names marked with (*) are set in the default.
-u --unit=NAME        Set output unitname
-w --web              Add web unit to uses, define type aliases for web unit
-x --extra-units=UNITLIST   Add units (comma-separated list of unit names) to use
                      This option can be specified multiple times.
```

From this output we can see the minimal operation options are:

```
dts2pas -i 7zip-min/index.d.ts -o 7zip.pp
```

This will run the declaration conversion on the file 7zip-min/index.d.ts and will write the resulting pascal file to 7zip.pp

This is what the declaration input file looks like:

```
export function unpack(pathToArchive: string,
```

```
                        whereToUnpack: string,
                        errorCallback: (err: any) => void): void;
export function unpack(pathToArchive: string,
                        errorCallback: (err: any) => void): void;
export function pack(pathToDirOrFile: string,
                     pathToArchive: string,
                     errorCallback: (err: any) => void): void;
export function list(pathToArchive: string,
                     callback: (err: any, result: Result[]) => void): void;
export function cmd(command: string[],
                    errorCallback: (err: any) => void): void;
export interface Result {
    name: string;
    date: string;
    time: string;
    attr: string;
    size: string;
    compressed: string;
}
```

And this is what the tool produces as output (lines have been formatted for better readability):

```
Unit _7zip;

{$MODE ObjFPC}
{$H+}
{$modeswitch externalclass}

interface

uses SysUtils, JS;

{$INTERFACES CORBA}
Type
  // Forward class definitions
  TResult = Class;

  Tunpack_errorCallback = Procedure (err : JSValue);
  // Ignoring duplicate type Tunpack_errorCallback (errorCallback)
  Tpack_errorCallback = Procedure (err : JSValue);
  Tlist_callback = Procedure (err : JSValue; result : array of TResult);
  Tcmd_errorCallback = Procedure (err : JSValue);
  TResult = class external name 'Object' (TJSObject)
      name : string;
      date : string;
      time : string;
      attr : string;
      size : string;
      compressed : string;
  end;


Procedure cmd(command : array of string;
```

```
              errorCallback : Tcmd_errorCallback);
              external name 'cmd';
Procedure list(pathToArchive : string;
              callback : Tlist_callback);
              external name 'list';
Procedure pack(pathToDirOrFile : string;
               pathToArchive : string;
               errorCallback : Tpack_errorCallback);
               external name 'pack';
Procedure unpack(pathToArchive : string;
                 whereToUnpack : string;
                 errorCallback : Tunpack_errorCallback);
                 external name 'unpack'; overload;
Procedure unpack(pathToArchive : string;
                 errorCallback : Tunpack_errorCallback);
                 external name 'unpack'; overload;

implementation
end.
```

Some things to note:

- types are prepended with `T`: Javascript is case sensitive, and often you will encounter variables with the same name as a type, but with different casing - a class name usually starts with a capital. To avoid name clashes, the tool prepends a `T` to type names.
- The tool correctly spots overloaded versions and marks them as such.
- The tool creates auxiliary types for complex function argument types.
- The special `any` type is replaced with `JSValue`.
- The JS unit is automatically used.

The resulting file can be compiled as-is:

```
> pas2js 7zip.pp
Info: 11458 lines in 6 files compiled, 0.3 secs
```

The `dts2pas` tool has several options, we'll explain them here (using the long version of each option):

alias This can be used to define type aliases. Aliases can be specified in 2 ways:

1. As a comma-separated list of `Name=Alias` pairs:

   `--alias=AType=MyType`

   This will replace every occurence of the `AType` in the declaration file with `MyType`
2. Using a `@` character, a filename to load a list of `Name=Alias` pairs (one per line):

   `--alias=@MyAliasFile.lst`

   This will read file `MyAliasFile.lst`. Each line of the file must contain a pair.

help Display a help text

input as seen, this is used to specify the input file to parse.

link with an argument `FILENAME` will insert a linklib statement:

```
{$linklib FILENAME}
```

When using the resulting unit, this will insert an import statement in the final Javascript :

```
import FILENAME from "FILENAME";
```

output with an argument `FILENAME` sets the output filename.

setting with an argument `SETTINGS` sets various conversion options, they are discussed below.

unit with an argument `NAME` sets the output unitname to `NAME`. When not specified, it is deduced from the output filename. Names marked with (*) are set in the default.

web Adds the `web` unit to the uses clause and defines type aliases for all web unit classes: this unit is part of pas2js and contains definitions of all classes exposed by the browser.

extra-units with an argument `UNITLIST` will add the units in `UNITLIST` (a comma-separated list of unit names) to the uses clause. Some TypeScript modules depend on other modules using import statements: the `dts2pas` tool will not recursively translate these other modules, but if you have translated them already, this option can be used to add the converted unit names to the uses clause.

The `setting` argument accepts a comma-separated list of named flags that influence the conversion process and the generated code. When translating TypeScript to Pascal, sometimes choices must be made because some TypeScript structures to not translate one-on-one to pascal constructs. Many of these choices are controlled using the flags which you can specify in the settings option. Here is an overview:

coRaw This will not generate a unit header or implementation section. You can use this to generate an include file.

coGenericArrays Instead of using `array of Type` for array types, the converter tool will write arrays as `TArray<Type>`. There is no functional difference in pas2js between the 2 declarations.

coUseNativeTypeAliases This will translate some basic types such as `long` to `Integer`.

coLocalArgumentTypes If auxiliary types are generated for methods, these will be generated in a `Type` section within the class, for example:

```
type
  TSomeClass = Class
  Public
    Type
      TMyMethod_B_Array = Array of integer;
    Function MyMethod(B : TMyMethod_B_Array) : Integer;
 end;
```

The default behaviour is to generate a global type with the class name prepended:

```
type
  TSomeClass_MyMethod_B_Array = Array of integer;
  TSomeClass = Class
  Public
    Function MyMethod(B : TSomeClass_MyMethod_B_Array) : Integer;
 end;
```

coUntypedTuples A tuple in TypeScript is a fixed-length array of values. If the `dts2pas` tool can determine the type of the element, it will generate a typed array:

```
Type
   TSomeTuple = array[1..3] of string;
```

If this flag is set, the array element will be untyped (type `JSValue`):

```
Type
   TSomeTuple = array[1..3] of JSValue;
```

coDynamicTuples A tuple in TypeScript is a fixed-length array of values. The vardts2pas tool will declare the type with the same number of elements. However, javascript allows you to specify less elements than in the definition of the tuple. To accomodate for this, using this flag you can let the converter generate a dynamic array:

```
Type
   TSomeTuple = array of string;
```

coExternalConst A constant in a TypeScript declaration will be translated literally. For example:

```
const myConst = "Hello, World";
```

Is translated as:

```
const
  myConst = 'Hello, World';
```

This means the constant is duplicated in the pascal code. Using the flag `coExternalConst`, the constant is translated as a reference instead:

```
const
  myConst : String; external name 'myConst';
```

coExpandUnionTypeArgs A variable of union type in TypeScript can have one of the possible types in the union type. This cannot be expressed in Pascal, so the default behaviour is to replace this with the `JSValue` catch-all type:

```
function func (a : string | number) : int;
```

is translated to Pascal as

```
function func (a : jsvalue) : integer;
```

With the `coExpandUnionTypeArgs` switch, for function or method arguments with union type, the converter will create overloaded versions for each type. In the above example this means the following declarations are produced:

```
function func (a : string) : integer; overload;
function func (a : double) : integer; overload;
```

coaddOptionsToheader If this switch is present, the converter will insert a comment with the used conversion options to the unit header. If the unit needs to be regenerated, the options used to create the original are available.

coInterfaceAsClass TypeScript knows interface definitions. The standard behaviour of the dts2pas tool is to translate this to an interface definition. With this switch, the interface will be declared as a pascal class.

coSkipImportStatements Any import statements in a TypeScript module are written to the converted pasal file as comments. With this option, these comments are not generated.

3 The web-based service

On the Free Pascal server, a (cgi) web service exists that can be used to translate any file from the `DefinitelyTyped` repository to a pascal unit. The service is located at

```
https://www.freepascal.org/~michael/service/dts2pas.cgi
```

On the server, the `DefinitelyTyped` repository is checked out, and is updated daily. By specifying a file name (relative to the `types` directory in the repository), the service outputs the translated unit. Using the following URL

```
https://www.freepascal.org/~michael/service/dts2pas.cgi/
  convert/?file=7zip-min/index.d.ts&unit=7zip
```

(the line has been split for readability) you will get the same file as in the result above.

The following query variables are accepted, they have the same meaning as their command-line counterparts.

file the file to convert, relative to the `types` directory in the DefinitelyTyped repository

unit the unit name to use.

aliases Aliases to to use, using the same format as the command-line tool.

extraunits Extra units to add to the command-line tool.

prependlog Insert conversion log as comments in the source.

flagname=1 Switch on any of the flags mentioned earlier.

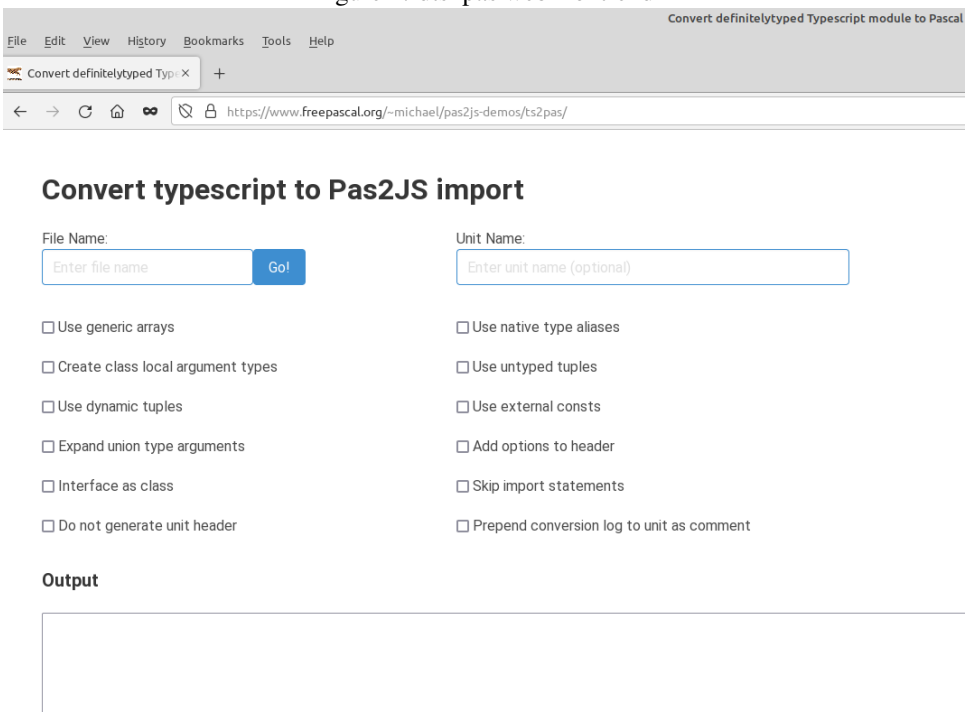
You can get a list of files available for conversion, one per line:

```
https://www.freepascal.org/~michael/service/dts2pas.cgi/list?raw=1
```

By leaving out the `raw=1` the output is a javascript array variable definition.

The latter option is used in a small web page, shown in figure 1 on page 8:

Figure 1: dts2pas web front-end



`https://www.freepascal.org/~michael/pas2js-demos/ts2pas/`

This page (obviously written in pas2js) is a simple front-end to the service. The service and front-end page will still be extended to provide more options, such as entering aliases or uploading a TypeScript file to convert.

4 Integration in the Lazarus IDE

Both the web-based service as the command-line tool have been integrated in the Lazarus IDE: using the `File-New` menu, you can directly convert a TypeScript file and make the resulting pascal file part of your program, see figure 2 on page 9.

When clicked, a small wizard pops up which allows you to select a description file from disk, or you can opt to use the web-based service: enter the name of a module - a list of matching files will be presented as soon as you enter 2 characters: see figure 3 on page 9.

On the same tab, you can enter extra units, aliases and indicate that the web unit must be used - together with all known aliases: basically the same options as available in the web interface or command-line.

The second tab (figure 4 on page 10) of the wizard page allows you to specify the conversion settings (or flags).

When done, you can click OK, and the IDE will create a new unit, part of your pas2js project, containing the converted TypeScript declaration module. If all goes well, it is ready to use, as seen in figure 5 on page 10

Figure 2: The File-New entry to import a TypeScript descripton file

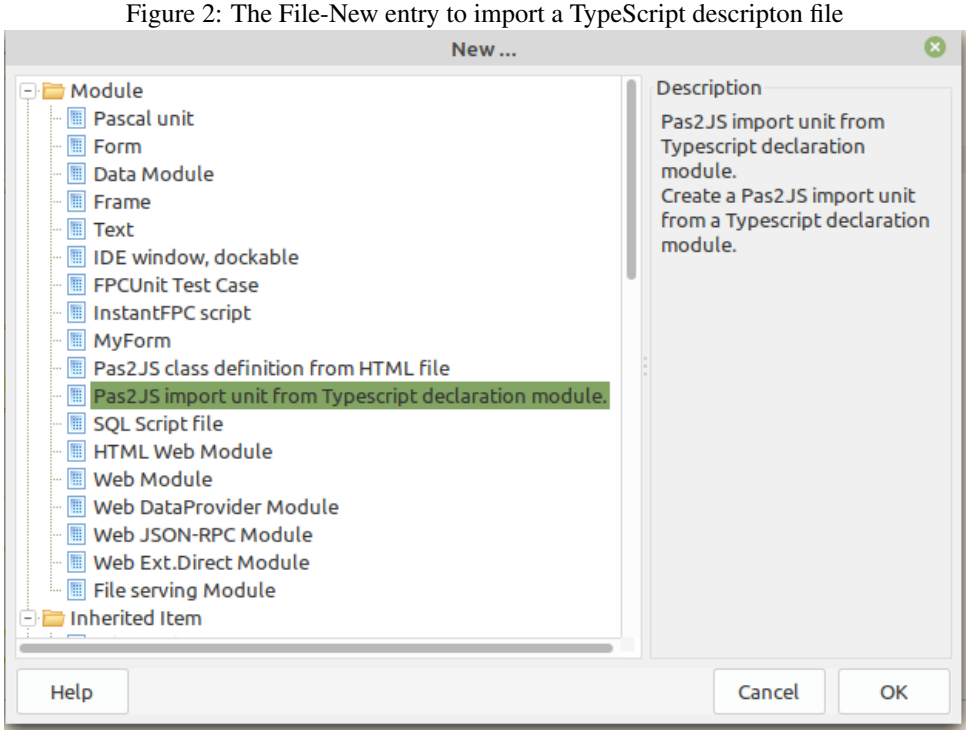


Figure 3: Selecting a TypeScript file or module

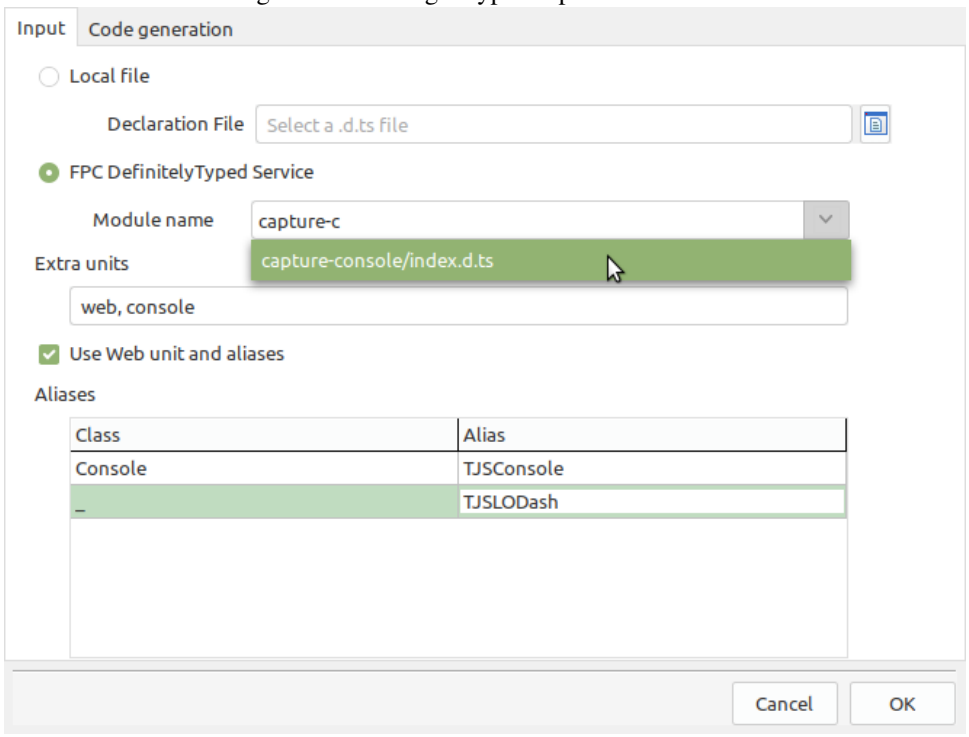


Figure 4: Setting the conversion flags

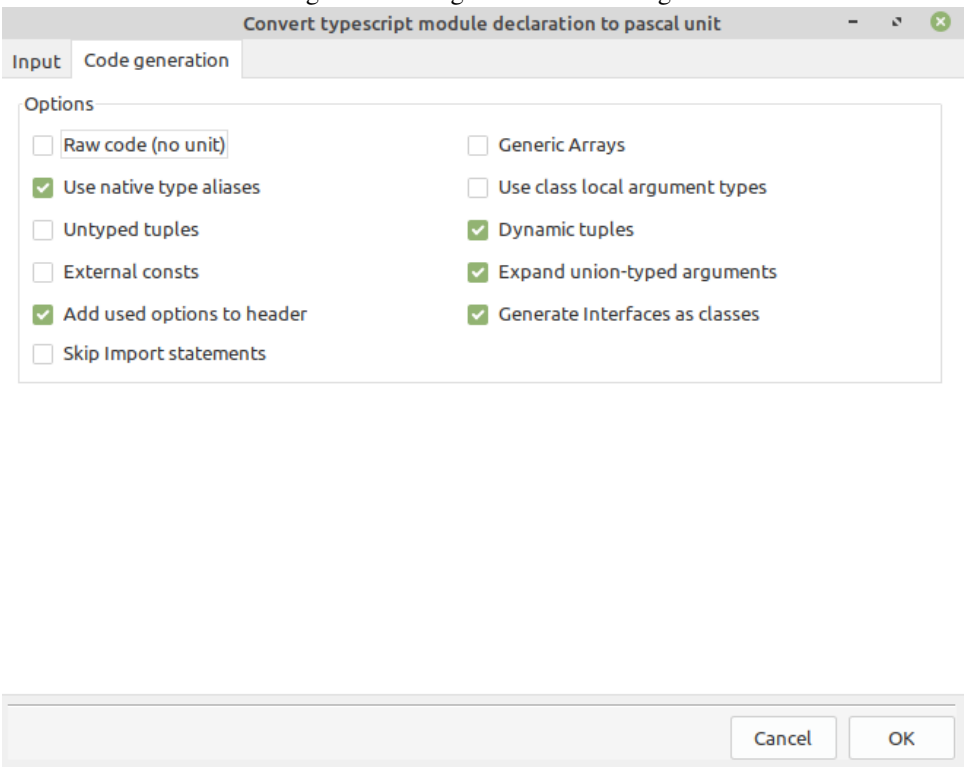
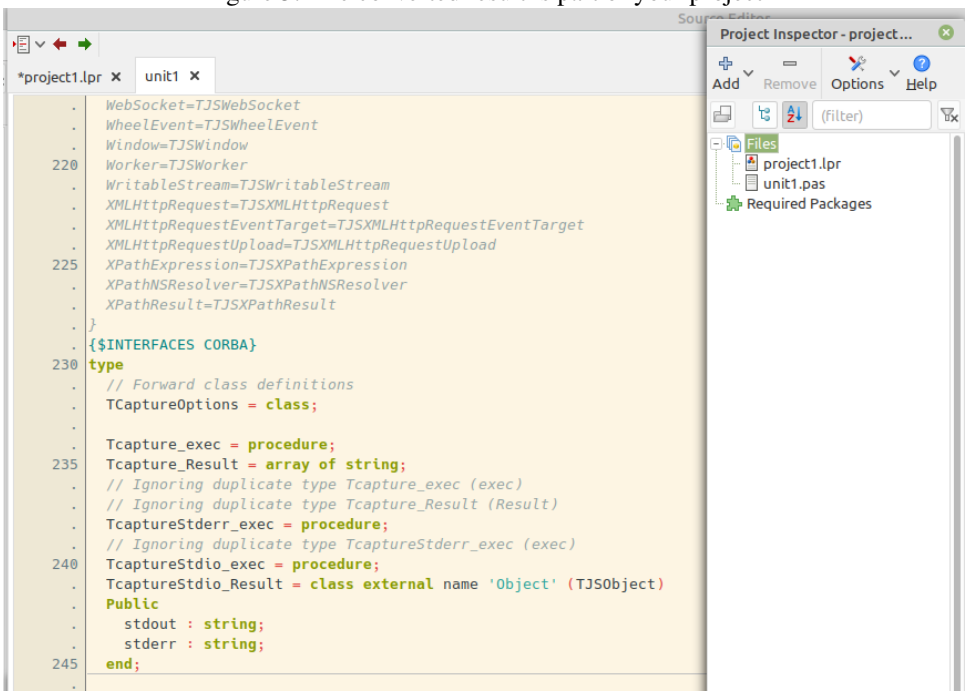


Figure 5: The converted result is part of your project



5 Conclusion

At the time of writing this article, the DefinitelyTyped contained well over 36.000 declaration files. Theoretically, these can now all be used in pas2js. You may wonder why the converted units are not made part of the pas2js repository. The answer to this question is twofold:

- The archive evolves continously: the pas2js units would be outdated almost daily.
- The conversion is not always perfect: sometimes some manual work is needed to fix the generated unit.

Javascript and Typescript have a lot of idioms which do not always translate well to Pascal. What is more, the declaration files are sometimes 'messy' – despite being more strict than Javascript, TypeScript still leaves a lot of room for interpretation and the translator sometimes simply cannot translate correctly what is being defined in TypeScript: Different people may have used different methods to describe the same Javascript interface and some descriptions may translate better to pascal than others.

Some declarations are simply outdated: TypeScript has evolved, but the declaration files have not been updated accordingly.

The Javascript/Typescript parser included in Free Pascal is not perfect either: it translates well over 99% of the files in definitelytyped, but not 100%.

Still, using the tool does most of the work for you. Even if some manual work is involved, the amount of work that you must still do will be negligable compared to writing the import units manually.