Extended RTTI in pas2js

Michaël Van Canneyt

April 21, 2025

Abstract

Extended RTTI exists since a long time in Delphi. Since some time, it also exists in FPC, and now it has also arrived in Pas2JS.

1 The history of RTTI

Since the first version of Delphi, RTTI exists: this is the mechanism provided by Delphi to get information about published fields, properties and methods in a class. The whole concept of streaming as implemented in the classes unit is built on this: A component publishes some properties. The list of properties can be queried by the object inspector and displayed, and the values of the properties can be retrieved or set. When you are done designing, the properties are written to the form file, and at runtime the form is recreated based on what was written in the form file. Free Pascal also introduced these concepts, and thus enabled the construction of Lazarus' LCL and the IDE.

For a long time, this was sufficient: the VCL and LCL work like this still today.

At some point, the authors of Delphi decided that having information about published properties, fields and methods was not enough, and they introduced Extended RTTI: From that point on, information of all fields, properties and methods in a class could be retrieved, regardless of their visibility. Because adding all this information to the binary increases the binary size, generation of this RTTI was put under control of the programmer, using the RTTI directive: It allows the programmer to specifiy exactly for which visibilities the RTTI must be generated. This can be specified for methods, fields and properties.

The default is:

```
{$RTTI INHERIT
    METHODS([vcPublic, vcPublished])
    FIELDS([vcPrivate..vcPublished])
    PROPERTIES([vcPublic, vcPublished])
}
```

This means that by default, RTTI information is available for public and published methods and properties, and for all fields regardless of their visibility.

So from one day to the next, a lot more RTTI was made available to the programmer.

The original published RTTI was limited to simple values and classes: it was not possible to publish e.g. arrays or records. Instead, the array was simulated by the `TCollectionItem` and `TCollection` classes, and the record was simulated by the `TPersistent` class. The programmer had to make sure that the `TPersistent` and

TCollectionItem were properly behaved, i.e. he had to implement a method to copy one TPersistent to another (the Assign method).

With the introduction of Extended RTTI, RTTI information for all types was supported, including records and arrays. The original API for retrieving the information (in the TypInfo unit) was in need of an update to support all this new information. The System.Rtti unit was created to handle all this new RTTI information. The API was modernized, instead of a procedural API now a set of objects exist that allow you to retrieve the RTTI and manipulate all types. Additionally, the Invoke function was created, to allow the programmer to call any method based on its RTTI.

Because all types were supported by Extended RTTI, a new data structure to handle a value of any possible type was needed: The 'Variant' record would fit this purpose, and so the TValue type was introduced: The TValue type allows you to store and manipulate a value of any type in Pascal.

Additionally, it became possible to introduce attributes: These are annotations to the code, which can be retrieved at run-time using RTTI.

All these new features offered new possibilities: It now became possible to introduce a ORM (Object Relational Mapping) and handle RPC (Remote Procedure Call) mechanisms with much less effort.

Introducing all these new possibilities in Free Pascal has taken a long time and progressed in several stages. The support for attributes was added quite quickly after Delphi introduced them, but only for published properties and classes. At the same time, the beginnings of the rtti unit were made. Over time the rtti unit was extended, and recently the FPC compiler was adapted to generate Extended RTTI.

For backwards compatibility the Extended RTTI in Free Pascal is not generated by default for TObject, only when you compile the RTL and packages with a special define, will the extended RTTI be activated for TObject and hence all objects. This special define (ENABLE_DELPHI_RTTI) is activated if you compile the so-called unicode RTL, which was introduced for maximum compatibility with recent versions of Delphi.

2 RTTI in Pas2JS

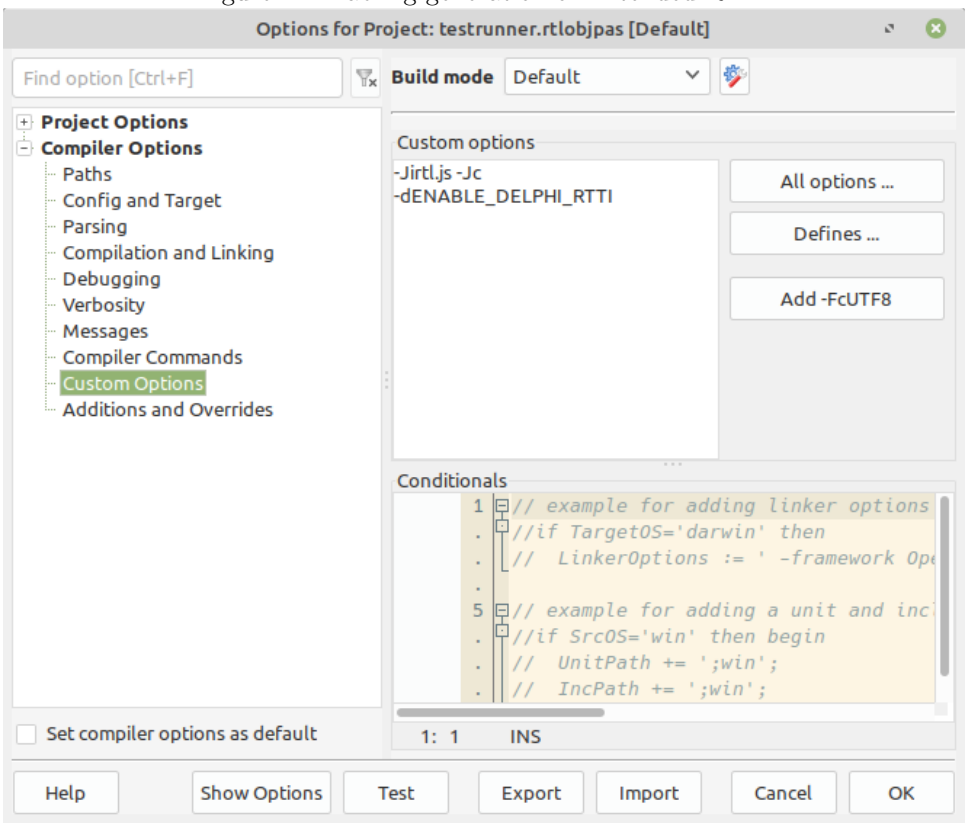
With FPC able to handle Extended RTTI, Pas2JS could not stay behind. Thanks to some sponsoring, the extended RTTI has been implemented in Pas2JS as well:

- The pas2js transpiler now recognizes the RTTI directive and adapts the RTTI generation accordingly.
- The RTTI unit has been enhanced to allow the use of the new RTTI.

The implementation in Pas2JS is not binary compatible with the implementation in Free Pascal or Delphi: In Delphi and FPC the RTTI is encoded in special data structures in the binary. Pas2js does not create binary code, it translates pascal to Javascript, and it makes more sense to encode the RTTI in lightweight Javascript objects.

Pas2js does not create pre-compiled units. This means that when you compile, it does a complete build. So, order to have Delphi-Compatible RTTI available in all objects, it is sufficient to define ENABLE_DELPHI_RTTI when compiling your project.

Figure 1: Enabling generation of Extended RTTI



On the command-line this means adding `-dENABLE_DELPHI_RTTI` to the command-line options:

```
pas2js -dENABLE_DELPHI_RTTI yourproject.pas
```

if you use the Lazarus IDE, then you can add the define to your project settings, as shown in figure 1 on page 3. After that, the use of the rtti is the same as in FPC or delphi.

To demonstrate this, we'll create a small demo program. So, we start a 'Web Browser Application' in Lazarus, tell it to create a HTML file and use the 'BrowserConsole' unit to write output to the console.

In the new project, we define the following class:

```
TIntegerObject = class(TObject)
Private
  FValue : Integer;
Public
  procedure WriteValue(const aValue : string);
  property Value : Integer Read FValue;
end;
```

The following routine will find the RTTI information for the 'FValue' private field, sets the value, and then writes the 'Value' property to verify that the correct value has been set.

```
procedure TestField;

var
  C : TRTTIContext;
  Info : TRttiType;
  LClassInfo : TRttiInstanceType absolute Info;
  Field : TRttiField;
  Obj : TIntegerObject;
  lValue : TValue;

begin
  Obj:=TIntegerObject.Create;
  C:=TRTTIContext.Create;
  Info:=C.GetType(TypeInfo(TIntegerObject));
  if not (Info is TRttiInstanceType) then
    Writeln('No class info available for TIntegerObject')
  else
    begin
    Field:=LClassInfo.GetField('FValue');
    If not assigned(Field) then
      Writeln('Class has no field named FValue!')
    else
      begin
      lValue:=TValue.FromOrdinal(TypeInfo(Integer),123);
      Field.SetValue(Obj,lValue);
      Writeln('New value is : ',Obj.Value,' (expected 123)');
      end;
    end;
end;
```

As always when working with Rtti, it starts by creating a Rtti context, which is used to keep the created Rtti info while using Rtti. It then proceeds by querying the RTTI info for the `TIntegerObject` class and it's `FValue` private field. If all info has been found, it constructs a `TValue` record holding an integer value of 123. The `SetValue` method of the `TRttiField` class can then be used to set the value.

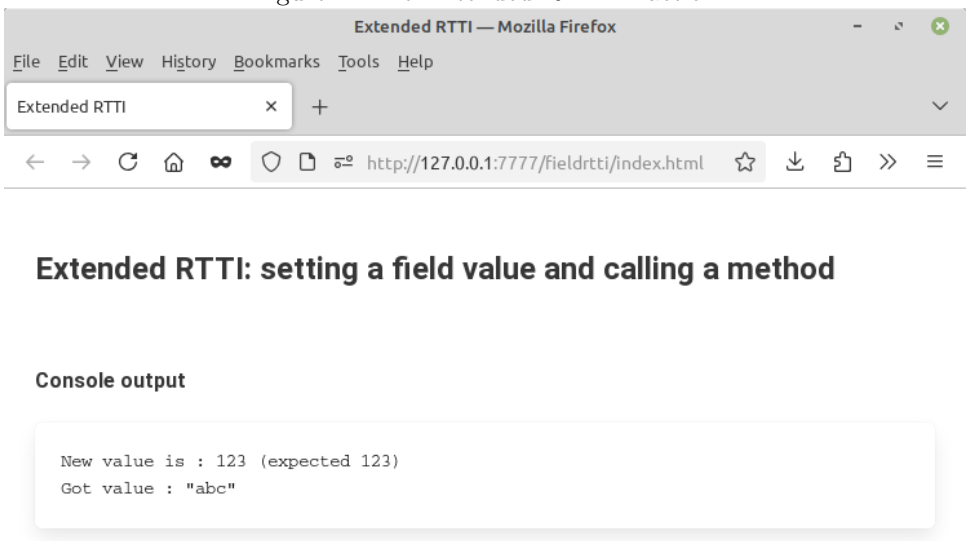
A second demo is to call a method using 'Invoke'. For this, we fetch the method's RTTI in the same way as was done for the field.

```
procedure TestMethod;

var
  C : TRTTIContext;
  Info : TRttiType;
  LClassInfo : TRttiInstanceType absolute Info;
  lMethod : TRttiMethod;
  Obj : TIntegerObject;
  lValue : TValue;

begin
  Obj:=TIntegerObject.Create;
  C:=TRTTIContext.Create;
  Info:=C.GetType(TypeInfo(TIntegerObject));
  if not (Info is TRttiInstanceType) then
    Writeln('No class info available for TIntegerObject')
```

Figure 2: The Extended RTTI in action




```
  else
    begin
    lMethod:=LClassInfo.GetMethod('WriteValue');
    If not assigned(lMethod) then
      Writeln('Class has no method named WriteValue!')
    else
      begin
      lValue:=TValue.specialize From<string>('abc');
      lMethod.Invoke(Obj,[lValue]);
      end;
    end;
end;
```

The `TValue` is constructed slightly differently (using ObjFPC's syntax for specializing a generic), but other than that the code is quite similar. Adding a main program source to call these procedures is simple, and with some additional HTML and CSS we can obtain an output as shown in figure 2 on page 5.

3 Conclusion

With the support for extended rtti, the Delphi compatibility of pas2js has been much improved. It should be noted that adding Extended RTTI increases the size of the generated javascript dramatically. For a desktop application this probably does not matter so much, but the size of the generated Javascript has a negative impact on HTML page loading times. So limiting the use of RTTI to the classes that actually need it may be a good idea.