# RTTI for beginners

Michaël Van Canneyt

December 27, 2021

**Abstract**

RTTI is a part of Delphi since version 1 of Delphi. What is it, what does it do and what can you do with it ?A gentle introduction.

## 1 Introduction

RTTI is a very important part of the Object Pascal language. It has been around as long as Delphi is around, because Delphi as we know it today would not be able to exist without RTTI: the IDE's Form designer is built on top of RTTI.

Since the first version of Delphi, RTTI has been expanded to include much more information than in the original version. Meanwhile it can be used to e.g. call arbitrary methods (e.g. in the SOAP implementation), arbitrary attributes can be created, and the way it is accessed has evolved (the RTTI unit).

Other languages also offer RTTI equivalents: Java and C# have introspection and reflection, which corresponds to RTTI in Delphi. In Javascript, the feature is more or less built-in in the language.

## 2 The need for RTTI

Formally, RTTI stands for Run-Time Type Information. As the name indicates, it is information about the used types in a Delphi program, which is available at run-time. Here Run-Time is meant as opposed to Compile-Time.

Since Object Pascal is a strongly typed language, it is only natural that while compiling, the compiler has all information available about a property of a class.

This means that, given the declaration

```
TMyClass = class
private
  FMyProperty: Integer;
  procedure SetM(AValue: Integer);
Public
  Property MyProperty : Integer Read FMyProperty Write SetM;
end;

procedure TMyClass.SetM(AValue: Integer);
begin
  if FMyProperty=AValue then Exit;
  FMyProperty:=AValue;
```

```
end;
```

the compiler can do a lot of things when it encounters code as:

```
  M:=MyClass.Create;
  M.MyProperty:=3;
```

it will know that

1. M indeed has a property `MyProperty`
2. An integer value can be assigned to MyProperty.
3. it must call SetM to actually set the value.

However, at run-time, the compiler is not available to check such things.

Yet, when a Delphi program is run, the definition of a form is read from the form file and used to construct the form. This is thanks to the presence of RTTI: The presence of RTTI makes it possible to create arbitrary objects and assign values to the available properties.

When looking at a form file (to see it, hit Alt-F12). Things like the following can be seen:

```
object BSet: TButton
  Left = 8
  Top = 80
  Width = 90
  Height = 25
  Caption = 'Set property'
  Default = True
  TabOrder = 2
  OnClick = BSetClick
end
```

(When done, hit Alt-F12 again to see the actual form)

At run-time, when the form must be created, the above information is read, and the form is constructed.

This is achieved using RTTI: when the above information is read (using the streaming mechanism), things like

```
  Height = 25
```

are translated to the equivalent of Pascal statements like.

```
  BSet:=TButton.Create(Self)
  BSet.Height:=25;
```

The compiler is not available at runtime to evaluate the above instruction. Instead, the information contained in the RTTI in the binary is queried to know how to create the button, and how to apply 25 to the `Height` property of the button.

In essence, RTTI is a large table containing all the classes used in a program, and for each class, a table with all the published fields, properties and methods of the class, and how to access them. The information in these tables is then consulted to create the button, and set its height to 25.

## 3 Making use of basic RTTI

If constructing forms at run-time was the only use for RTTI, then the makers of Delphi need not have bothered: it would have been perfectly possible to let the IDE create a (private) method which was simply full of pascal code like:

```
BSet:=TButton.Create(Self)
BSet.Height:=25;
BSet.Left:=8;
BSet.Top:=80;
BSet.Width:=90;
BSet.Height:=25;
BSet.Caption:='Set property';
```

This method could be called by the constructor, and this would construct the form. Indeed, this would be faster than the current method which involves parsing a form file, looking up names in RTTI etc. In fact, IDE plugins exist that do exactly this (e.g. GExperts contains such a plugin).

Then why use RTTI anyway ? The reason is in the additional flexibility this offers. If the form was constructed using compiled code, there would be no possibility to change the definition, once the binary is shipped to the users: the definition is compiled in the binary.

But since the form definition is stored in text in the resources, it is possible to change the text, or even load 2 form definitions, one after the other. This is in fact what is done if a form is localized: the original definition is loaded using RTTI, and then it is altered using a localized definition of the form; (not only the texts can be changed, but also control positions etc.)

There exist many components that use the RTTI to save and store form positions (and in fact, any property of any component) at runtime. The JVCL's `TJvFormPlacement` class, for example. The Inspex package from Raize software also makes use of the RTTI to allow you to alter the properties of classes at runtime.

There are 2 units in the RTL that offer access to the RTTI. The original TypeInfo unit (which is procedural and very low-level), and the more modern, object-oriented RTTI unit which uses classes and generics. Both offer the same information, and which one to use is largely a matter of preference.

## 4 A simple example

To demonstrate the use of RTTI, we'll make a small program that allows to enter a property name, and a value. The program will then attempt to set a form property with the given name, using the provided value. There are 2 edit boxes: one for the name, one for the value. A button `Set property` can be pushed to actually set the property.

The first version of the form uses the typinfo unit. The `OnClick` handler of the button looks as follows:

```
procedure TMainForm.BSetClick(Sender: TObject);

begin
  SetPropValue(Self,EName.Text,EValue.Text);
end;
```

This is of course very simple. The `SetPropValue` function is simplicity itself.

```
procedure SetPropValue(Instance: TObject;
                       const PropName: string;
                       const Value: Variant); overload;
```

It takes 3 arguments. An instance of a class, a property name and the value (as a variant). It will look up the name of the property, check the type, and attempts to convert the variant to the appropriate type for the property. If all checks out, the property is set.

Using the RTTI unit, a little more work is needed. This unit lacks the ability to convert a variant to the appropriate type, so this must be done manually.

Whenever classes of the RTTI unit are used, a context `TRTTIContext` record must be used. This record contains the necessary methods to query the RTTI. For our example, the `GetType` method must be used to retrieve the type information of the form. This is an object of type `TRTTIType`. That object can then be used to get the type information of the property whose name was entered: for this, the `GetProperty` method of `TRTTIType` must be used. It will result in a object of type `TRTTIProperty`

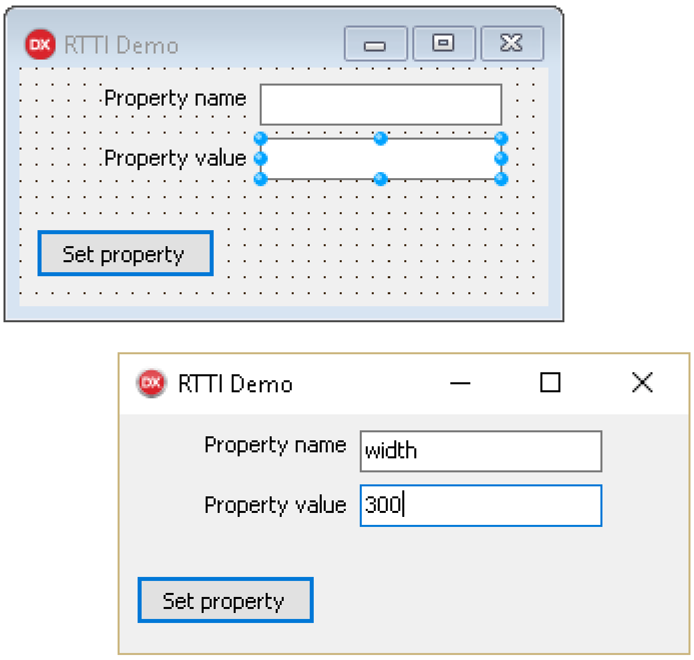
Once the property information is available, the actual property can be set:

```
procedure TMainForm.BSetClick(Sender: TObject);

Var
  Ctx : TRTTIContext;
  RT : TRTTIType;
  P : TRttiProperty;
  AValue : String;

begin
  AValue:=EValue.Text;
  Ctx:=TRTTIContext.Create;
  try
   RT:=Ctx.GetType(ClassInfo);
   P:=rt.GetProperty(EName.Text);
   if P.IsWritable then
     case P.PropertyType.TypeKind of
       tkInteger : P.SetValue(Self,StrToInt(AValue));
       tkEnumeration : P.SetValue(Self,StrToInt(AValue));
       tkUString : P.SetValue(Self,AValue);
     end;
  finally
    Ctx.free;
  end;
end;
```

To set the property, its type must be known: the RTTI unit does not offer functionality to automatically convert from a string to the property's actual type. The type of the property is available in the `P.PropertyType.TypeKind` property: this is an enumerated type which contains an element for all possible pascal types. The code above checks for a few common types, but in actual code, all possible type kinds should be checked (and an exception raised if it is not supported). The `SetPropValue` does all this automatically.

The result of this code can be seen in figure 1 on page 5.

Figure 1: Setting a property



## 5 Getting property lists

It is not very interesting to be able to set only properties of the form, so we'll first extend the program to fill a combobox with the available components in the form. Additionally, letting the user guess the names of available properties is also not very conventient, so we'll present the user with a list of properties of the selected component.

Showing a list of available components can be done without RTTI, using the methods of TComponent is sufficient (although RTTI could be used to get the design-time components of the form). We'll store a reference to the component in the Items.Objects property of the combobox.

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  FillCBComponents;
end;

procedure TMainForm.FillCBComponents;

Var
  C : TComponent;
  S : String;

begin
  S:=CBComponent.Text;
  With CBComponent,CBComponent.Items do
    begin
    beginUpdate;
    try
      Clear;
      Sorted:=False;
      AddObject(Name,Self);
      for C in self do
        AddObject(C.Name, C);
      Sorted:=True;
      CBComponent.ItemIndex:=CBComponent.Items.IndexOf(S);
    finally
      EndUpdate;
    end;
  end;
end;
```

The `BSet OnClick` handler is now refactored so it will first retrieve the selected component, and then calls the `SetProperty` routine, which simply contains our previous code, changed to accept 3 arguments: component, property name and value.

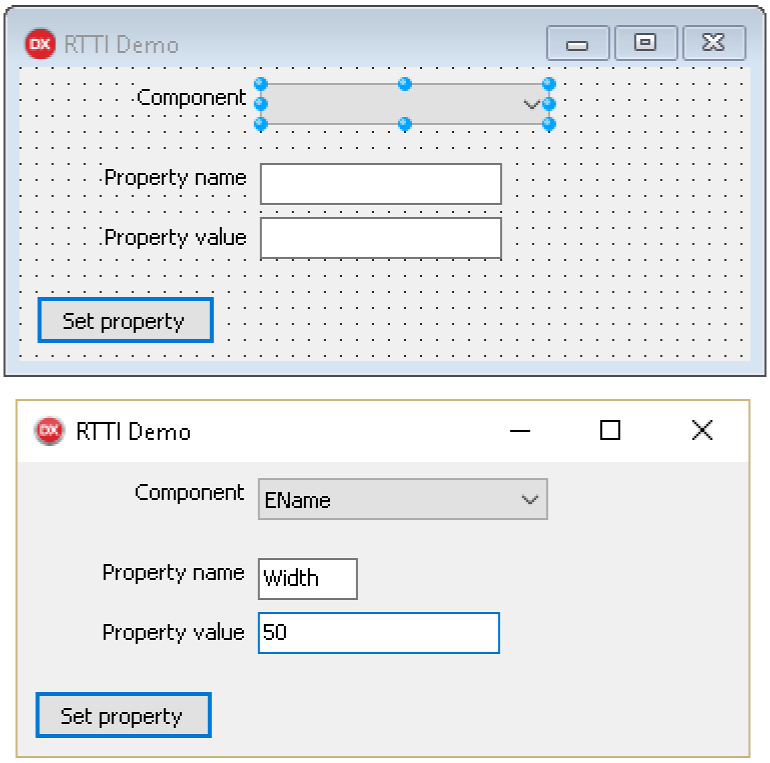
```
procedure TMainForm.BSetClick(Sender: TObject);

Var
  IDX : Integer;
  C : TComponent;

begin
  IDX:=CBCOmponent.ItemIndex;
  if (IDX=-1) then
    Raise Exception.Create('Select a component first');
  C:=CBCOmponent.Items.Objects[IDX] as TComponent;
  SetProperty(C,EName.Text,EValue.Text);
end;
```

The result looks as in figure 2 on page 7

Now for the more interesting part: Once the component is selected, we present to the user a list of published properties that the selected component has.

```
procedure TMainForm.CBComponentChange(Sender: TObject);

Var
  IDX : Integer;
  C : TComponent;
begin
  IDX:=CBCOmponent.ItemIndex;
  if (IDX=-1) then
    C:=Nil
  else
    C:=CBCOmponent.Items.Objects[IDX] as TComponent;
  ShowProperties(C);
end;

procedure TMainForm.ShowProperties(C : TComponent);

begin
  With CBName,Items do
```

Figure 2: Setting a property of a selected component



Component: EName

Property name: Width

Property value: 50

```
    begin
    try
      BeginUpdate;
      Clear;
      Sorted:=False;
      if Assigned(C) then
        GetPropertyNames(C,CBName.Items);
      Sorted:=True;
    finally
      EndUpdate;
    end;
    end;
end;
```

Using the typinfo unit, the routine to get the property names of the selected component, looks as this:

```
procedure TMainForm.GetPropertyNames(C : TComponent;AList : TStrings);

Var
  P : PPropList;
  I,N : Integer;

begin
  N:=TypInfo.GetPropList(C,P);
  try
    I:=0;
    While I<N do
      begin
      AList.Add(P^[I].Name);
      Inc(I);
    end;
  finally
    FreeMem(P);
   end;
end;
```

The `GetPropList` function of the `TypInfo` unit returns a list of all properties of an object (or a class). The result is a count of properties, and a pointer to an array of `PPropInfo` pointers (`P`). When finished with this array, it must be freed. The varPPropInfo points to a `TPropInfo` record. This is one of the central records in the TypInfo unit:

```
TPropInfo = packed record
  PropType: PPTypeInfo;
  GetProc: Pointer;
  SetProc: Pointer;
  StoredProc: Pointer;
  Index: Integer;

  Default: Integer;
  NameIndex: SmallInt;
  Name: TSymbolName;
end;
```

The `PropType` pointer points to the type information of the property's type. The `Name` is the name of the property, which is what interests us: this is used to add the name of the property to the list of property names. The other fields tell us how the property is accessed (getter,setter etc.), and whether has a default value.

Using the RTTI unit, the routine to get the property names is quite simple as well:

```
procedure TMainForm.GetPropertyNames(C : TComponent;AList : TStrings);

Var
  Ctx : TRTTIContext;
  P : TRttiProperty;

begin
  Ctx:=TRTTIContext.Create;
  try
    For P in ctx.GetType(C.ClassInfo).GetProperties do
     AList.Add(P.Name);
  finally
    Ctx.free;
  end;
end;
```

The CTX.GetType call is the same as used in the code to set a property name: it returns a `TRTTIType` descendent. The `GetProperties` method of the `TRTTIType` class returns an array of `TRttiProperty` instances: one for each property in the class. This array is traversed, and the names of each of the properties is added to the list.

The resulting code results in a form that looks as in figure 3 on page 10

## 6 Setting a known property on several classes

The code demonstrated till now was informative, but hardly practical. Let's turn to a more practical example.

To disable all controls of a form, their 'enabled' property can be set to false. this can be done with a simple loop:

```
Var
  I : Integer;
begin
  For I:=0 to ControlCount-1 do
    Controls[i].Enabled:=False;
end;
```

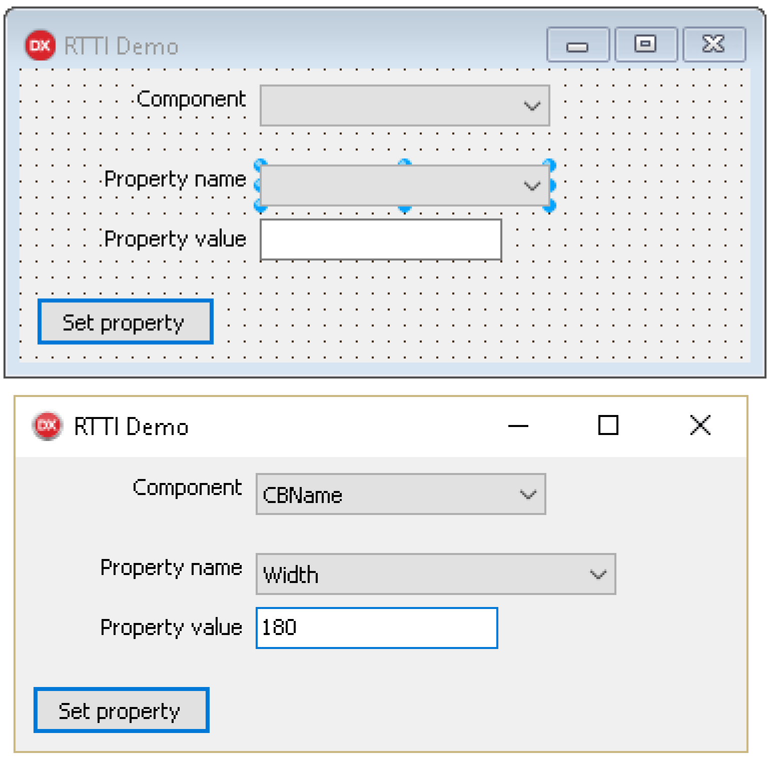
This can be done because all controls on the form descend from `TControl` and they all inherit the `Enabled` property.

But imagine another scenario: only Data-Aware controls must be disabled. Data-Aware controls have been created as descendants of regular controls which implement Data-related properties (`DataField` and`DataSource`). That means that there is no property at the TControl level which can be checked to see if the property is data-aware.

Without RTTI, the only method to achieve this would have been to check the type of the control:

```
Var
```

Figure 3: Selecting a property of a selected component



Component: CBName

Property name: Width

Property value: 180

```
  I : Integer;
  C : TControl;
begin
 For I:=0 to ControlCount-1 do
   begin
   C:=Controls[i];
   if (C is TDBText)
      or (C is TDBGrid)
      // check all other types
      or (C is TDBEdit) then
     C.Enabled:=False;
   end;
end;
```

It is clear that this is error-proof and slow.

With RTTI, we can do this a lot easier. When the Data-Aware controls were made, care was taken that they all use the same property names to link to a datasource and indicate a field (`DataField` and`DataSource`). Using this fact, we can just check whether the control has the `Datasource` property to know if it is data aware, and disable it. Using the typinfo unit, this is done as follows:

```
Var
  I : Integer;
  C : TControl;

begin
  For I:=0 to ControlCount-1 do
    begin
    C:=Controls[i];
    if isPublishedProp(C,'DataSource') then
      C.Enabled:=False;
    end;
end;
```

The `isPublishedProp` function of the `TypInfo` unit checks if a class (or class instance) has the indicated property.

To do this with the RTTI unit, the following code can be used:

```
Var
  I : Integer;
  C : TControl;
  Ctx : TRTTIContext;
  P : TRttiProperty;

begin
  Ctx:=TRTTIContext.Create;
  try
    For I:=0 to ControlCount-1 do
      begin
      C:=Controls[i];
      for P in Ctx.GetType(C.ClassInfo).GetDeclaredProperties do
        if SameText(P.Name,'DataSource') then
          begin
          C.Enabled:=False;
          Break;
          end;
     end;
  finally
    Ctx.free;
  end;
end;
```

## 7 Conclusion

The RTTI of Delphi is a powerful tool. For simple projects, it's mostly hidden in the form streaming code, but as projects become more advanced, it can be useful to know how RTTI functions and what can be done with it. In this article, a start has been made, showing some simple things like how to set an arbitrary property using RTTI, how to get a list of properties, and how to check if a class has a certain property. RTTI can do many more things, such as examine attributes or invoke arbitrary methods. These topics will be treated in a future article.