NG SerializationPack Guide

USER MANUAL

© 2018 LMD Innovative
LMD Innovative

This page is intentionally left blank.
Remove this text from the manual
template if you want it completely blank.

© 2018 LMD Innovative

This page is intentionally left blank.
Remove this text from the manual
template if you want it completely blank.

Overview

1 Overview

LMD NG SerializationPack is a part of Next Generation (NG) package suite. All these packages are based on new IDE and language features of latest Delphi IDE versions.

NG SerializationPack provides the ability to serialize/de-serialize Delphi objects into various storage formats. This allows to write data handling applications more easily and in more object oriented way. Common use cases of serialization engine are:

- Saving/loading application options.
- Sending business objects via net between client and server.
- Saving/loading CAD-like application documents, since they are usually represented at run-time as a tree-like object model.
- ect.

In the current version the following formats are supported:
- XML via Delphi standard XML access, e.g. `IXmlDocument`, `IXmlNode`.
- JSON via Delphi standard JSON access, e.g. `TJSONValue`.
- Binary stream via standard Delphi `TStream`.
- XML via third party libraries (OmnyXML and NativeXML).
- JSON via third party libraries (SuperObject).
- Other formats, such as storing directly into Windows Registry, ect. are considered for next versions.

Serialization engine is based on the new Delphi RTTI feature. Basically, values of most of Delphi data types, such as numbers, strings, classes, records or arrays, can be serialized. Special attributes can be used to annotate types or fields/properties to specify or adjust various aspects of serialization process.

Features

Following is a short feature list of NG SerializationPack package:

- The ability of handling most of Delphi types. Engine does not favor to classes, any type, starting from primitive types, such as `Integer` or **`string`**, continuing to more complex types, such as records or arrays, can be serialized/de-serialized.
- Support for serializing class/record fields or properties.
- Full inheritance support.
- The ability of adjusting serialization process using provided attributes, such as `SerializableAttribute`, `TransientAttribute`, `AliasAttribute`, ect.
- The ability of writing custom converters.
- Support of fill-read mode via `FillReadAttribute`. This mode allows to de-serialize owned by parent object sub-objects without re-creating them; that is, instead of usual action sequence, which is to create new sub-object instance, read its properties and assign this created instance to parent object's property, fill-read mode use the following action sequence: read parent object's property, and use read sub-object value to read its properties. This way parent object's property can be even read-only.

© 2018 LMD Innovative

Introduction

2 Introduction

Serialization process is controlled by special serializer/de-serializer objects. Such objects are implemented for every output format:

- XML serialization is performed by `TXmlSerializer` and `TXmlDeserializer` object,
- JSON serialization is performed by `TJsonSerializer` and `TJsonDeserializer` object,
- Binary serialization is performed by `TBinarySerializer` and `TBinaryDeserializer` objects.

These helper objects should be created (and later destroyed) by the user. Serializer objects descends from common `TSerializer` base class, which declare common serialization methods and properties. As well, de-serializer objects descends from `TDeserializer` common base class.

The library is designed so, that interfaces of these common base classes are sufficient for most usage, and its recommended to be used as a parameter types of various custom `MySerialize`/`MyDeserialize` methods. However, its important to note, that format specific serializer objects has some additional, non-common, stuff. First of all, it constructors, which takes different arguments for different formats. Then, some additional methods/properties, such as `FlushBuffer` for binary serializer or `RootValue` for XML serializer are also had to be used to achieve format specific needs; however they are designed to be usable at the same routine level, where serializer objects are created/destroyed.

Following is the very simple example, which shows how to serialize and de-serialize `TPoint` value using XML:

Serialize code:

```
var
  s: TXmlSerializer;
  p: TPoint;
begin
  p.X := 7;
  p.Y := 9;

  s := TXmlSerializer.Create(MyXmlDoc.Node);
  try
    s['MyPoint'].Value<TPoint>(p);
    MyXmlDoc.SaveToFile('...');
  finally
    s.Free;
  end;
end;
```

Resulting XML:

```
<MyPoint>
  <X>7</X>
  <Y>9</Y>
</MyPoint>
```

De-serialize back code:

```
var
  d: TXmlDeserializer;
  p: TPoint;
begin
  d := TXmlDeserializer.Create(MyXmlDoc.Node);
  try
```


© 2018 LMD Innovative

```
    p := d.Value<TPoint>;
  finally
    d.Free;
  end;
end;
```

What can be serialized

The following types are supported for serialization:

- All numeric types, including integer types as well as floating point types.
- String types, such as `AnsiString`, `WideString`, `UnicodeString`, ect.
- Char types, such as `AnsiChar`, `WideChar`, `Char`, ect.
- Records: all public fields and properties are serialized by default; however, this can be overridden, using `SerializableAttribute` and `TransientAttribute`.
- Classes: all public and published fields and properties are serialized by default; however, this can be overridden, using `SerializableAttribute` and `TransientAttribute`. Serialization includes all inherited fields and properties; however, inherited field and properties can be excluded from serialization using `NoInheritedAttribute`. Serializable classes should have public parameter-less constructor; it is used by serialization engine to create new object instance while de-serializing object values.
- Enumerations, sets.
- Arrays: both static and dynamic arrays are supported. Multi-dimensional arrays are also supported, and treated as arrays of arrays.

Following is an example of overriding default serializable status of fields:

```
type
  TMyObject = class
  protected
    [Serializable]
    s: string;  // Will be serialized.
  public
    X: Integer;
   [Transient]
    Y: Integer; // Will not be serialized.
  end;
```

If only some special fields or properties need to be serialized, the type can be marked with TransientAttribute as a whole, which will mark all public and published properties to be transient by default:

```
type
  [Transient]
  TMyObject2 = class
  public
    X: Integer; // Will not be serialized.
    Y: Integer; //
    [Serializable]
    S: string;  // Will be serialized.
  end;
```

Here, it should be noted, that NG-Serialization engine will only process fields and properties, which are visible via Delphi RTTI. Delphi, does not generate RTTI for all fields or properties; actually, to use protected or private ones, the user should use {$RTTI ...} Delphi directive to force Delphi compiler to include required RTTI.

© 2018 LMD Innovative

Aliasing

Sometimes names of types, fields, properties or even array elements should be changed to improve readability of human readable output formats, such as XML or JSON. This can be accomplished using `AliasAttribute` (`ElemAliasAttribute` for array elements).

Example:

```
type
  [ElemAlias('Number')]
  TMyArray = array of Integer;

  TMyObject = class
  public
    [Alias('Numbers')]
    property Arr: TMyArray;
  end;
```

Resulting XML:

```
<MyObject>
  <Numbers>
    <Number>3</Number>
    <Number>5</Number>
    <Number>9</Number>
  </Numbers>
</MyObject>
```

Internal data model

As seen from the example above, the main method to use is the `Value` method. Note, that it is a template method and the serialized/de-serialized type should be specified. In current section other useful serializer/de-serializer methods will be described.

Despite the fact, that the package is made multi-format, it has common internal data model, which is primarily grabbed from JSON. So, at the core level, serialization engine works with the following data abstractions:

- Value - can be a simple typed value, for example, Integer or String, or a complex value such as array or object.
- Object - a complex value, which is a set of properties. Each property has a name and value. Serialization engine converts Delphi class instances and records to internal objects.
- Array - a complex value, which is a set of elements - an ordered sequence of values, which (unlike properties) has no names. Serialization engine converts Delphi arrays and collections (via custom converters) to internal arrays.

Its important to note that the value itself has no name. Actually, the only place where a name is associated with a value - are object properties. This implies the following very important idea: Root level values are **unnamed** in the internal data model. And since the library allows to store more than one root-level value with a single serializer object (calling Value method several times), such values are treated **sequentially**; and, actually, should be read back in the same order. Its important to understand sequential internal nature of the library.

© 2018 LMD Innovative

However, some formats, such as XML format, requires the name (tag) to be provided for every stored value; thus, even root-level values or array elements should have a name. The library API include routines to provide such additional names, however these names are just ignored in more compatible formats such as binary stream or JSON format. The example of such API is `RootValue` indexed property of `TXmlSerializer` class; in the example above it used implicitly in expression `s['MyPoint']`, because its a default Delphi property.

JSON format also conflicts slightly with internal data model. In fact, JSON allows only single root level value; more than one value cannot be represented with valid JSON string. To overcome this issue, JSON serializer's constructor takes an address of (array) buffer, into which all root level values are stored. Its up to the user, of what to do later with those values. However, if a single JOSN value capable to be converted into valid JSON string is still needed as a result of serialization, the user should wrap several values into JSON array or object manually, e.g. calling `BeginArray`/`EndArray` or `BeginObject`/`EndObject`/`Prop` methods.

As has been already noted, any single value can be serialized using `Value<>` method. This case includes all possible values, even complex values such as objects or arrays. However there is an additional API which allows to serialize object or array like data without using (declaring) corresponding Delphi types. This API designed to be used primarily in the following two cases:

- In high level routines, to format data manually; usually the API is used directly in the procedure which creates/destroys serialization object.
- In custom converters, which are classes descended from `TCustomConverter` base class.

So, to serialize object like data, `BeginObject`/`EndObject` and `Prop` additional serializer's methods can be used. The following example, shows how to serialize point data manually, without using `TPoint` Delphi type:

Serialize code:

```
var
  s: TXmlSerializer;
begin
  s := TXmlSerializer.Create(MyXmlDoc.Node);
  try
    s['MyPoint'].BeginObject('', False);
    s.Prop('X').Value<Integer>(7);
    s.Prop('Y').Value<Integer>(9);
    s.EndObject;
  finally
    s.Free;
  end;
end;
```

Resulting XML:

```
<MyPoint>
  <X>7</X>
  <Y>9</Y>
</MyPoint>
```

Note, how `RootValue` property, which is again used implicitly in `s['MyPoint']` expression, is used with the `BeginObject` method in a single code line; as well, `Prop` method is used with `Value` method in a single code line also. This is possible because `RootValue` and `Prop` always returns `Self` object which is serializer, which makes the resulting code more readable.

© 2018 LMD Innovative

So, to serialize array like data, `BeginArray`/`EndArray` additional serializer's methods can be used in a manner similar to manual object data serialization. The following example, shows how to serialize some integers as an array, without using real array Delphi type:

Serialize code:

```
var
  s: TXmlSerializer;
begin
  s := TXmlSerializer.Create(MyXmlDoc.Node);
  try
    s['MyArray'].BeginArray('Item');
    s.Value<Integer>(3);
    s.Value<Integer>(7);
    s.Value<Integer>(9);
    s.EndArray;
  finally
    s.Free;
  end;
end;
```

Resulting XML:

```
<MyArray>
  <Item>3</Item>
  <Item>7</Item>
  <Item>9</Item>
</MyArray>
```

The rule with these additional API is that `BeginObject`/`BeginArray` methods can be used anywhere the `Value` method is used; more precisely, its valid to use them to serialize property values, thus - after a `Prop` method call, or, while serializing array elements. So, following this way its possible to have object or array property values, as well, as array of objects or array of arrays.

© 2018 LMD Innovative

Inheritance

3 Inheritance

NG-Serialization library supports Object Pascal inheritance. This means the following: if a field (or property) of some class type `TAncestor` hold an object value of some another class type `TDescendant`, where `TDescendant` is a descendant of `TAncestor`, then the field value will be serialized/de-serialized correctly; that is all serializable properties declared in `TDescendant` class type will be written in the output medium, and, as well, the value object of class type `TDescendant` will be created during de-serialization, and all its serialized properties will be read back. For example, consider the following type declarations, and `myobj` object variable initialization code:

```
type
  TAncestor = class
  public
    X: Integer;
  end;

  TDescendant = class(TAncestor)
  public
    Y: Integer;
  end;

  TMyObject = class
  public
    O: TAncestor; // The type of the field is TAncestor...
  end;

var
  myobj: TMyObject;
begin
  myobj   := TMyObject.Create;
  myobj.o := TDescendant.Create; // ...But, the assigned value
                                 //    is of TDescendant type.
  //...
end;
```

Serializing `myobj` using XML serializer will result in the following XML:

```
<MyObject>
  <O Class="TDescendant">
    <X>0</X>
    <Y>0</Y>
  </O>
</MyObject>
```

There are three things to note here:

- First, is a `Class` attribute, which stores property value class name; this value is required to be stored, because it used while de-serialization process; generally, the class of object value is stored only when its different from the corresponding field or property type class.
- Second, class name, stored as a `Class` attribute value can be adjusted using `AliasAttibute`, making XML more readable.
- Third, as has been described above, `Y` field, which is a subclass field, is also stored in XML.

Classes Registration

This section will discuss, how meta-data is used by serialization engine and, why, sometimes, an explicit classes registration is required.

© 2018 LMD Innovative

For each serialized/de-serialized type the engine initializes and stores in memory additional information. This information contains read via RTTI field and property set, serialization related attributes, ect. In general, this information dramatically improves run-time performance of the engine. Type related meta-data is initialized lazily and stored in a global dictionary. For following discussion imagine a very simple serialization case:

```
s.Value<TPoint>(p);
```

In the code above, a point, which is a record of `TPoint` type is serialized. In this a reference to `TPoint` type info is explicitly provided to Value template method, so the engine will be able to initialize related meta-data on-the-fly. Moreover, the type info of all point fields, which are in this case two `Integer` fields, are also available from the parent `TPoint` type info. So, serializing a point this way **do not** require to register types explicitly.

The same way meta-data can be initialized during de-serialization:

```
p := d.Value<TPoint>;
```

As seen from the code, a reference to `TPoint` type is provided here too. So, in most cases, including serializing `TMyObject` object with `TDescendant` sub-object from the first example, do not require explicit types registration.

However, de-serialization of the produced in the first example XML **can fail**. Consider the following de-serialization code:

```
myobj := d.Value<TMyObject>;
```

In this code `TMyObject` type info is provided to the engine, however, this type info do not contain a reference to `TDescendant` type info, because the field `O` is, actually, of `TAncestor` type only. So, in this case its required to register `TDescendant` class explicitly. `TMetadata` class should be used to achieve this:

```
TMetadata.RegisterClass(TDescendant);
```

To register several classes at once `TMetadata.RegisterClasses` method can be used instead. Usually, such explicit registration is performed at application startup, for example, in `initialization` clause of some application unit.

There is another one use case, where explicit class registration is required. Consider the code:

```
var
  obj: TObject;

obj := d.Value<TObject>;
if obj is TMyObject then
  ; //...
```

This code de-serializes an object of `TMyObject` class, providing just `TObject` as a template argument for `Value` method. Sometimes, code like this is useful, e.g. where no a-priory knowledge about de-serializing object type exists. Here again, all possible object classes should be registered explicitly.

Default Value Class

© 2018 LMD Innovative

Recall the XML, produced by the first example:

```
<MyObject>
  <O Class="TDescendant">
   ...
  </O>
</MyObject>
```

As seen from XML text, `TDescendant` value object class is stored in XML as `Class` attribute. Since in some human readable formats, like XML or JSON this affects readability, engine stores the name of the class only when necessary. The same can be said about other formats, such as binary stream; however, the main goal here is to reduce output stream size.

The default rule is to store class name, if value run-time class is different from field or property class type. However, this rule can be adjusted using `DefClassAttribute`, which allows to specify default class, different from the field or property class type:

```
type
  TAncestor = class
  public
    X: Integer;
  end;

  TDescendant = class(TAncestor)
  public
    Y: Integer;
  end;

  TMyObject = class
  public
    [DefClass(TDescendant)]
    O: TAncestor; // The type of the field is TAncestor...
  end;
```

The same rule is applied to root-level object values. For example, consider the code:

```
s.Value<TMyObject>(obj);
```

This code will store `obj` class name, if the object run-time type, which is `obj.ClassType`, is not equal to provided (as template argument) `TMyObject` class.

Suppressing Inherited Fields and Properties

Sometimes it is useful to descend serialization related classes from some other standard or third-party classes; for example, VCL classes such as `TPersistent` or `TComponent`, can be used as ancestors. Usually, these classes contains huge amount of properties, which, actually, are not intended to be serialized. Moreover, in such cases serialization engine can raise exceptions, because not all that properties even can be serialized.

In such situations, all ancestor fields and properties can be suppressed applying `NoInheritedAttribute` to the type:

```
type
  [NoInherited]
  TMyObject = class(TComponent)
  public
```


© 2018 LMD Innovative

```
    X: Integer;
    S: string;
  end;
```

Inheritance and Custom Converters

Inheritance is **not** automatically supported with custom converters. So, for example, if `TAncestor` class is serialized using custom converter, the implementation of the converter should be aware of all possible values, including values of `TDescendant` type (if required, of course); and moreover, the converter should be able to distinguish these values based on de-serializing information.

Generally, this was a design decision, mainly motivated by the fact, that custom converter can convert an object value to any type of internal value; for example - to array or just number.

© 2018 LMD Innovative

This page is intentionally left blank.
Remove this text from the manual
template if you want it completely blank.

Fill-Read Mode

4 Fill-Read Mode

In Delphi object oriented programming its common to have sub-objects owned by its parent objects; such sub-objects commonly created in the corresponding parent object's constructor and destroyed in parent object's destructor. Usually, they are not re-created during parent object's life time. In some cases, these sub-objects are exposed via public parent object's property.

There are a lot of examples of such sub-objects in Delphi VCL:

- Font property, which is an object of type TFont.
- Constraints property, which is also an object.
- All collections, such as Items, Lines, ect.

Following is a simple example of declarations containing sub-object property:

```
type
  TSubObject = class
  public
    X: Integer;
  end;

  TMyObject = class
  private
    FSubObject: TSubObject;
  public
    constructor Create;
    destructor Destroy; override;
    property SubObject: TSubObject read FSubObject;
  end;

constructor TMyObject.Create;
begin
  FSubObject := TSubObject.Create;
end;

destructor TMyObject.Destroy; override;
begin
  FSubObject.Free;
end;
```

The main problem with these sub-objects is that they cannot be handled by serialization engine as usually. Its incorrect to assign newly created sub-object instance during de-serialization. Moreover, the corresponding public property can be read-only at all. So, the example above will not be serializable.

To make the code above serializable, `FillReadAttribute` should be used. It specifies that serialization engine should use so-called fill-read mode. Serialization of fill-read properties works the same way as in normal mode: property value, which is sub-object reference is read and its properties are serialized. However, de-serialization in fill-read mode works differently:

- Property value, which is sub-object is read from the property.
- And this existing sub-object value is used to de-serialize sub-object properties.

`FillReadAttribute` can be applied to sub-object property like this:

```
type
  TMyObject = class
  private
```


© 2018 LMD Innovative

```
    FSubObject: TSubObject;
  public
    constructor Create;
    destructor Destroy; override;
    [FillRead]
    property SubObject: TSubObject read FSubObject;
  end;
```

Or, alternatively, it can be applied to sub-object class itself:

```
type
  [FillRead]
  TSubObject = class
  public
    X: Integer;
  end;
```

In this case fill-read mode will be used with all properties of `TSubObject` type. There are several things should be kept in mind, while applying `FillReadAttribute` to the whole sub-object type:

- First, this way fill-read mode can't be overridden back to normal mode in descendants; nor it can be overridden at the property level.
- Second, since serialization engine never attempt to replace an instance of sub-object with another one, it actually, never create such new instances. Thus, the parameter-less constructor is not required for serializable classes, which are marked with `FillReadAttribute`.

Fill-Read Mode with Non-Object Types and Root-Level Values

For working with values in fill-read more manually, special overload of the de-serializer's `Value` method can be used. This method is a procedure, which does not return any value, but instead take a single `var` parameter. Use it as follows:

```
obj := TSubObject.Create;
d.Value<TSubObject>(obj); // Read already created obj in fill-read mode.
```

Fill-read mode can be used with values of any type, not only object values. There are some restrictions, depending of type used. For example, it is fully correct to have a dynamic array property marked as fill-read. Since, dynamic array is a reference type in Delphi, the reference to array data will be read and filled with de-serializing elements. The restriction here is that array length can't be changed during de-serialization, because this will cause array data reallocation; so, if de-serializing array element count will differ from actual dynamic array length, the exception will be raised during de-serialization.

Fill-read mode can be used even with so-called value-types, such as numbers, strings, static arrays or records. However, the value here should be "L-value", that is it should be possible to take value address. Following there is a list of cases where it possible:

- Value is a class field.
- Value is field of L-value record.
- Value is an element of L-Value static array.
- Value is a dynamic array element.
- A reference to value is passed to de-serializer's `Value` method as a `var` parameter.

Note, that property values are not considered to be L-values.

© 2018 LMD Innovative

This page is intentionally left blank.
Remove this text from the manual
template if you want it completely blank.

Custom Converters

5 Custom Converters

NG-Serialization library provides an ability to change the output representation of serializing value. This can be done writing custom converter, which is essentially a descendant of `TCustomConverter` base class. The converter writer should declare `TCustomConverter` base class descendant, and override and implement all its abstract methods:

- `GetReadMode` method implementation should just return a value indicating in which mode the converter works: `rmNormal` for normal mode or `rmFillRead` for fill-read mode.
- `Write` method implementation should serialize provided `V` value using provided `S` serializer object. Implementation should use public serializer's methods, such as `Value`, `BeginObject`/`Prop`/`EndObject` and `BeginArray`/`EndArray`, to serialize `V` value in a way it wants.
- Read method implementation should de-serialize value using provided D de-serializer object. Implementation should use public de-serializer's methods, such as `HasNext`, `Value`, `BeginObject`/`EndObject` and `BeginArray`/`EndArray`, to de-serialize value in a way it wants. If the converter works in normal mode it should treat `V` parameter as **output** parameter and assign de-serialized value to it. However, if the converter works in fill-read mode it should treat `V` parameter as **input** parameter and use provided `V` value to fill it with de-serializing information. Thus, `V` parameter should **never** be treated as bi-directional (in-out).

To associate custom converter class with the converted type `ConverterAttribute` should be used like this:

```
type
  TMyCollConverter = class(TCustomConverter)
  public
    function  GetReadMode: TReadMode; override;
    procedure Write(S: TSerializer; const V); override;
    procedure Read(D: TDeserializer; var V); override;
  end;

  [Converter(TMyCollConverter)]
  TMyCollection = class
    //...
  end;
```

Following is a list of potential use-cases where custom converters can be used:

- Collections. Since collections usually represented by objects, there are required to implement custom converter to serialize collection object as an array of element (items) instead of collection object itself with all its public properties.
- Variant like data, which can be implemented as a record or an object. Look for example below.
- Serializing references to business objects as the corresponding object's IDs.

Example of Serialization of Variant Like Data

Consider the following declaration:

```
type
  TMyKind = (mkNull, mkBool, mkInteger, mkString);

  TMyVariant = record
  private
    // Implementation is not shown.
  public
```


© 2018 LMD Innovative

```
    property Kind: TMyKind read GetKind write SetKind;
    property AsBool: Boolean read GetAsBool write SetAsBool;
    property AsInteger: Integer read GetAsInteger write SetAsInteger;
    property AsString: string read GetAsString write SetAsString;
  end;
```

As seen from declaration there are no need to serialize all `TMyVariant` properties, because only one of tree data properties is matter. So, the custom converter can be used to achieve more readable serialization result, like this:

```
<MyVariant>
  <Kind>Integer</Kind>
  <Value>7</Value>
</MyVariant>
```

Following is a converter code used to achieve above result:

```
type
  TMyVarConverter = class(TConverter)
  public
    function  GetReadMode: TReadMode; override;
    procedure Write(S: TSerializer; const V); override;
    procedure Read(D: TDeserializer; var V); override;
  end;

function TMyVarConverter.GetReadMode: TReadMode;
begin
  Result := rmNormal;
end;

procedure TMyVarConverter.Write(S: TSerializer; const V);
var
  mv: ^TMyVariant;
begin
  mv := @TMyVariant(V);

  S.BeginObject('MyVariant', False);
  S.Prop('Kind').Value<string>(KindToStr(mv.Kind));

  case mv.Kind of
    mkNull:    ; // Do nothing.
    mkBool:    S.Prop('Value').Value<Boolean>(mv.AsBool);
    mkInteger: S.Prop('Value').Value<Integer>(mv.AsInteger);
    mkString:  S.Prop('Value').Value<String>(mv.AsString);
  end;

  S.EndObject;
end;

procedure TMyVarConverter.Read(D: TDeserializer; var V);
var
  mv: TMyVariant;
  tp: string;
begin
  D.BeginObject(tp);

  mv.Kind := StrToKind(D.Prop('Kind').Value<string>);
  case mv.Kind of
    mkNull:    ; // Do nothing.
    mkBool:    mv.AsBool    := D.Prop('Value').Value<Boolean>;
```


© 2018 LMD Innovative

```
    mkInteger: mv.AsInteger := D.Prop('Value').Value<Integer>;
    mkString:  mv.AsString  := D.Prop('Value').Value<String>;
  end;

  D.EndObject;
  TMyVariant(V) := mv;
end;
```

After converter is written is should be associated with `TMyVariant` class using `ConverterAttribute`. Thus, `TMyVariant` declaration should be changed as follows:

```
type
  [Converter(TMyVarConverter)]
  TMyVariant = record
    ...
  end;
```


© 2018 LMD Innovative