Nullable types in Delphi Native

with records, methods, operator overloading & helpers

Jeroen Pluimers
better office benelux
jpluimers@better-office.com

Nullable types in Delphi

- Example showing the combination of some "recent" technology in Delphi
  - Operator overloading
  - Helpers
- Code is at bo.codexplex.com

Why nullable types?

- Relational databases have known NULL for years:
  - Bill Karwin (interbase):
    A NULL in SQL is considered an absence of a value, not a value itself.
    The mantra you should learn is "NULL is a state, not a value."
    If it were a value, you could use it in expressions.
    But a NULL combined in most expressions yields another NULL.
- There is not yet a real alternative in Delphi Native
  - Variants have strange behaviour
  - TField instances are not value types
  - TField instances are hard to make calculations with

How to create nullable types?

- Some knowledge is needed:
  - Value versus reference types
  - Operator overloading
  - Helpers
  - Properties
  - TypeInfo

What can nullable types do?

- Make calculations
  - much easier
  - function like they work in SQL
- Getting Data from/to your database in first cass Delphi types
- Be properties in classes and components

How to create nullable types?

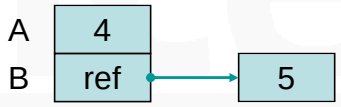
- Some knowledge is needed:
  - Value versus reference types
  - Operator overloading
  - Helpers
  - Properties
  - TypeInfo

Value & reference types

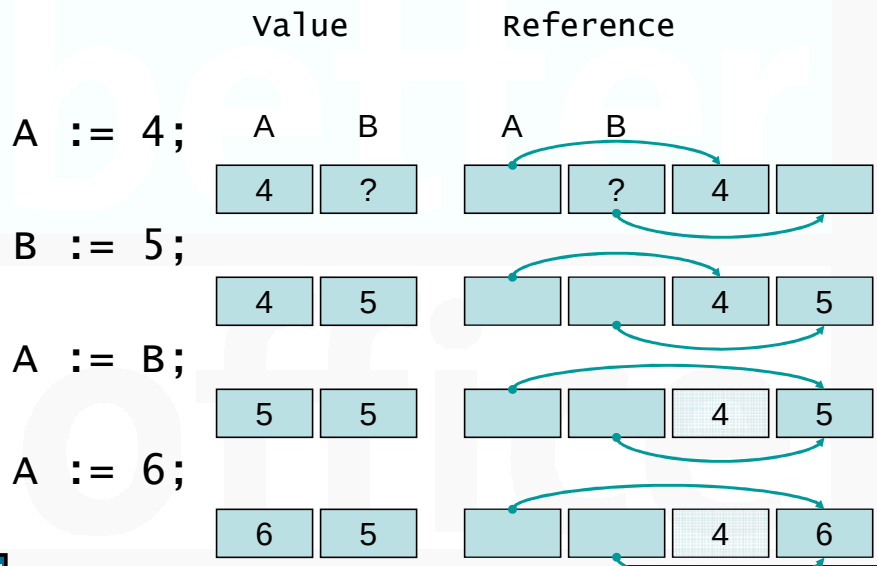
- Value types
  - Live on the stack
  - copy-on-assignment
  - Examples
    - Simple types
    - Records
    - Strings (behaviour)
- Reference types
  - Live on the heap
  - copy-reference-on-assignment
  - Examples
    - Objects
    - Interfaces
    - Pointers
    - Strings (storage)

Value & reference types

```
var
  A: Integer;  // value
  B: ^Integer; // reference
begin
  A := 4;
  New(B); // initialize reference
  B^:= 5;
end;
```



Value & reference types



Value & reference types

- Some facts:
  - Most built in Delphi operators function on simple types
    - Except :=, = and <> which work on all types
  - Simple types are value types
  - Records are value types too
- So:
  - Use records as fundament for nullable types

```
type
  TNullableInteger = record
    Value: Integer;
    IsNull: Boolean;
  end;
```

How to create nullable types?

- Some knowledge is needed:
  - Value versus reference types
  - Operator overloading
  - Helpers
  - Properties
  - TypeInfo

Operator overloading

- Add your own "behaviour" to operators
  - Works only for records
    - In Delphi Native: not for classes!
  - An operator and the operand(s) are being implemented worden by a "class operator"; this is a kind of class method with name and argumen(s)

- Example:
  - Multiplication `X : = A * B;`
  - Operator: *
  - Name: Multiply
  - Operands: 2 -> two parameters

```
type
  TMyRecord = record
    class operator Multiply(A, B: TMyRecord): TMyRecord;
  end;
```

Operator overloading

operator	#	usage	name	category	*
and	2	R := A and B;	BitwiseAnd	bit	
not	1	R := not A;	//BitwiseNot	bit	inexisting
or	2	R := A or B;	BitwiseOr	bit	
xor	2	R := A xor B;	BitwiseXor	bit	
() cast	1	R := TValue(A);	Explicit	conversie	
:=	1	R := A;	Implicit	conversie	

Operator overloading

operator	#	usage	name	category	*
round	1	R := Round(A);	Round	function	
trunc	1	R := Trunc(A);	Trunc	function	
and	2	R := A and B;	LogicalAnd	logical	
not	1	R := not A;	LogicalNot	logical	
or	2	R := A or B;	LogicalOr	logical	
xor	2	R := A xor B;	LogicalXor	logical	

Operator overloading

operator	#	usage	name	category	*
+	1	R := A + B;	Add	binary	
/	2	R := A / B;	Divide	binary	
div	2	R := A div B;	IntDivide	binary	
mod	2	R := A mod B;	Modulus	binary	
*	2	R := A * B;	Multiply	binary	
-	2	R := A - B;	Subtract	binary	

Operator overloading

operator	#	usage	name	category	*
shl	2	R := A shl B;	LeftShift	binary	confusing
shr	2	R := A shr B;	RightShift	binary	confusing
-	1	R := -A;	Negative	unary	
+	1	R := +A;	Positive	unary	
dec	1	Dec(A);	Dec	self	
inc	1	Inc(A);	Inc	self	

Operator overloading

operator	#	usage	name	category	*
=	2	R := A = B;	Equal	comparison	
>	2	R := A > B;	GreaterThan	comparison	
>=	2	R := A >= B;	GreaterThanOrEqual	comparison	
<	2	R := A < B;	LessThan	comparison	
<=	2	R := A <= B;	LessThanOrEqual	comparison	
<>	2	R := A <> B;	NotEqual	comparison	

Operator overloading

- Documentation is not correct!
  - http://docwiki.embarcadero.com/RADStudio/en/Operator_Overloading_(Delphi)
  - Not only Win32, also x64
  - BitwiseNot does not exist (use LogicalNot)
  - At least 1 operand must be of the same type as your record data type
  - Result type may be any type
    - Watch the result type of comparison operators! Should be BOOLEAN
      - D2009 doc failure:
        Win32 works only for records; .NET for classes and records

Operator overloading

- Tips:
  - Some operators should be overloaded pair-wise

    = and <> shl and shr

    < and >= > and <= dec and inc

    + and - / and * div and mod
  - Prefer Explicit over Implicit operators
    - Beware of the built-in type coercion (implicit operators)
    - e.g Byte to Integer; Integer to Double; Variants from/to anything!

Operator overloading

```
type
  TNullableInteger = record
  strict private
    //1 Trick to force RTTI for a record (as per Barry Kelly)
    FForceRTTI: string;
    FIsFilled: Boolean;
    FValue: Integer;
    function GetIsNull: Boolean;
    procedure SetIsFilled(const Value: Boolean);
    procedure SetIsNull(const Value: Boolean);
    procedure SetValue(const Value: Integer);
  public
    procedure Clear;
    class function Compare(a, b: TNullableInteger): Integer; static;
    class function Null: TNullableInteger; static;
    class function Parse(const Value: string): TNullableInteger; static;
    function ToString: string;
    class operator Add(const a, b: TNullableInteger): TNullableInteger;
    //... Meer operatoren ...
    property IsFilled: Boolean read FIsFilled write SetIsFilled;
    property IsNull: Boolean read GetIsNull write SetIsNull;
    property Value: Integer read FValue write SetValue;
  end;
```

Operator overloading

```
class operator TNullableInteger.Add
  (const a, b: TNullableInteger): TNullableInteger;
begin
  if a.IsFilled and b.IsFilled then
    Result.Value := a.Value + b.Value
  else // at least 1 is NULL, so return NULL
    Result.Clear();
  end;
end;

class operator TNullableCurrency.Add
  (const A, B: TNullableCurrency): TNullableCurrency;
begin
  if A.IsFilled or B.IsFilled then
    Result.Value := A.Value + B.Value
  else // both are NULL, so return NULL
    Result.Clear();
end;
```

Operator overloading

```
class function TNullableInteger.Compare(a, b: TNullableInteger): Integer;
begin
  if a.IsFilled then
  begin
    if b.IsFilled then
    begin // a.IsFilled = true; b.IsFilled = true
      if a.Value > b.Value then
        Result := 1
      else
        if a.Value < b.Value then
          Result := -1
        else
          Result := 0;
    end
    else
    begin // a.IsFilled = true; b.IsFilled = false
      Result := 1; // a is greater because it is filled
    end
  end
  else
  begin
    if b.IsFilled then
    begin // a.IsFilled = false; b.IsFilled = true
      Result := -1;
    end
    else // a.IsFilled = false; b.IsFilled = false
    begin
      Result := 0;
    end
  end;
end;
```

Operator overloading

```
procedure TNullableInteger.Clear;
begin
  FIsFilled := False;
  FValue := 0;
end;

function TNullableInteger.GetIsNull: Boolean;
begin
  Result := not IsFilled;
end;

class function TNullableInteger.Null: TNullableInteger;
begin
  Result.Clear;
end;

procedure TNullableInteger.SetIsFilled(const Value: Boolean);
begin
  FIsFilled := Value;
  if not IsFilled then
    Clear();
end;

procedure TNullableInteger.SetIsNull(const Value: Boolean);
begin
  IsFilled := not Value;
end;

procedure TNullableInteger.SetValue(const Value: Integer);
begin
  FValue := Value;
  FIsFilled := True;
end;
```

Operator overloading

```
function RelativeDioptre(
  const MetersDeltaHA: TNullableDouble;
  const OriginalDioptre: TNullableDouble
): TNullableDouble;
begin
  Result := OriginalDioptre /
    (1 + MetersDeltaHA * OriginalDioptre);
end;
```

- This also shows why operators in Delphi Native are <u>not</u> possible for classes
  - Memory leak with intermediate results
  - .NET solves this with a garbage collector

How to create nullable types?

- Some knowledge is needed:
  - Value versus reference types
  - Operator overloading
  - Helpers
  - Properties
  - TypeInfo

Helpers

- Introduced in Delphi to support .NET
  - The .NET class hierarchy
    differs from the Native VCL class hierarchy
    In the .NET framework, VCL methods and properties were different or missing
- Helpers can make extensions at function level
  - Yes: methods and properties
  - No: instance data
- They also work in Delphi Native:
  - Class helpers since Delphi 2005
  - Record helpers since Delphi 2006
  - "Simple" helpers since Delphi XE3

Helpers

```
type
  TFormatSettingsHelper = record helper for TFormatSettings
  protected
    class function GetLocaleID: Integer; static;
    class procedure SetLocaleID(const Value: Integer); static;
  public
    class function GetDefaultFormatSettings: TFormatSettings; static;
    class function GetDayNameIndex(const WeekDay: Integer): Integer;
   static;
    class property LocaleID: Integer
      read GetLocaleID write SetLocaleID;
  end;
```

Helpers

```
var
  FCurrentLocaleId: Integer = -1;

class function TFormatSettingsHelper.GetDefaultFormatSettings: TFormatSettings;
begin
  GetLocaleFormatSettings(LocaleId, Result); // pas eventueel Result aan
end;

class function TFormatSettingsHelper.GetDayNameIndex(const WeekDay: Integer): Integer;
begin
  if WeekDay in [DayMonday..DaySaturday] then
    Result := 1 + WeekDay
  else
    Result := 1;
end;

class function TFormatSettingsHelper.GetLocaleID: Integer;
begin
  if FCurrentLocaleId = -1 then
    FCurrentLocaleId := GetThreadLocale;
  Result := FCurrentLocaleId;
end;

class procedure TFormatSettingsHelper.SetLocaleID(const Value: Integer);
begin
  FCurrentLocaleId := Value;
  SetThreadLocale(Value);
end;
```

Helpers

```
var
  MyFormatSettings: TFormatSettings;
  MondayDayNameIndex: Integer;
  MondayLongDayName: string;
begin
  MyFormatSettings := TFormatSettings.GetDefaultFormatSettings();
  MondayDayNameIndex := MyFormatSettings.GetDayNameIndex(DayMonday);
  TFormatSettings.LocaleID := $0013; // nl     Nederlands
  SundayShortDayName := MyFormatSettings.LongDayNames[MondayDayNameIndex];
  TFormatSettings.LocaleID := $0413; // nl-nl  Nederlands - Nederland
  SundayShortDayName := MyFormatSettings.LongDayNames[MondayDayNameIndex];
  TFormatSettings.LocaleID := $0813; // nl-be  Nederlands - België
  SundayShortDayName := MyFormatSettings.LongDayNames[MondayDayNameIndex];
  TFormatSettings.LocaleID := $0409; // en-us  English - United States
  SundayShortDayName := MyFormatSettings.LongDayNames[MondayDayNameIndex];
  TFormatSettings.LocaleID := $0462; // fr-nl  Frysk - Nederlân
  SundayShortDayName := MyFormatSettings.LongDayNames[MondayDayNameIndex];
end;
```

nl	nl-nl	nl-be	en-us	fr-nl
maandag,	maandag,	maandag,	Monday,	Moandei

Helpers

- Helpers (class or record):
  - function as long as the helper is visible to the user
- So:
  - Helper in the same unit,
  - or helper in a unit in the uses list

TField Helpers

```
type
  TFloatFieldHelper = class helper for TFloatField // or for TField with AsDouble
  private
    function GetAsNullableDouble: TNullableDouble;
    procedure SetAsNullableDouble(const Value: TNullableDouble);
  public
    property AsNullableDouble: TNullableDouble
      read GetAsNullableDouble write SetAsNullableDouble;
  end;

function TFloatFieldHelper.GetAsNullableDouble: TNullableDouble;
begin
  if Self.IsNull then
    Result.Clear()
  else
    Result.Value := Self.Value;
end;

procedure TFloatFieldHelper.SetAsNullableDouble(const Value: TNullableDouble);
begin
  if Value.IsNull then
    Clear()
  else
    Self.Value := Value.Value;
end;
```

TField Helpers

```
function TDMEPFittingSet.GetFittingSetInfo(const aRefId: TRefId): REPFittingSetInfo;
var
  aItem: REPFittingSetInfo;
begin
  if aRefId = NullRefId then
    raise EDMEPFittingSet.Create('Empty RefId in GetInfo');
  try
    sqlqGetInfo.ParamByName('REFID').AsString := aRefId;
    sqlqGetInfo.Open;
    try
      if sqlqGetInfo.BOF and sqlqGetInfo.EOF then
        raise EDMEPFittingSet.CreateFmt('FittingSet not found in GetInfo (%s)', [aRefId]);
      aItem.RefId := sqlqGetInfoREFID.AsNullableString;
      // ...
      aItem.sDiam := sqlqGetInfoDIAMETER.AsNullableDouble;
      // ...
      aItem.sAxis := sqlqGetInfoAXIS.AsNullableInteger;
      // ...
    finally
      sqlqGetInfo.Close;
    end;
  except
    on E: Exception do
      raise EDMEPFittingSet.Create(E, daRead);
  end;
  Result := aItem;
end;
```

How to create nullable types?

- Some knowledge is needed:
  - Value versus reference types
  - Operator overloading
  - Helpers
  - Properties
  - TypeInfo

Properties

- Properties can be any type
- The object inspector shows only
  - Published properties that are of
    - simple types
    - class types (TPersistent is easiest to use)
- To get a nullable in the object inspector you have to create a TPersistent wrapper
  - TNullableWrapper types…

Properties

```
type
  TNullableIntegerWrapper = class(TPersistent)
  strict private
    FNullableValue: TNullableInteger;
    FOnChange: TNotifyEvent;
  strict protected
    procedure Changed; dynamic;
  public
    function GetIsNull: Boolean; virtual;
    function GetValue: Integer; virtual;
    procedure SetIsNull(const Value: Boolean); virtual;
    procedure SetValue(const Value: Integer); virtual;
    constructor Create(AValue: TNullableInteger);
    procedure Assign(Source: TPersistent); override;
    function GetNullableValue: TNullableInteger; virtual;
    procedure SetNullableValue(const Value: TNullableInteger); virtual;
    property NullableValue: TNullableInteger
      read GetNullableValue write SetNullableValue;
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
  published
    property IsNull: Boolean read GetIsNull write SetIsNull;
    property Value: Integer read GetValue write SetValue;
  end;
```

Properties

```
procedure TNullableIntegerWrapper.Assign
  (Source: TPersistent);
var
  NewNullableValue: TNullableIntegerWrapper;
begin
  if Source is TNullableIntegerWrapper then
  begin
    NewNullableValue := TNullableIntegerWrapper(Source);
    Self.NullableValue := NewNullableValue.NullableValue;
    Exit;
  end;
  if Source = nil then
  begin
    Self.IsNull := True;
    Exit;
  end;
  inherited Assign(Source);
end;
```

Properties

```
procedure TNullableIntegerWrapper.Changed;
begin
  if Assigned(FOnChange) then
    FOnChange(Self); // belangrijk voor de component: die moet op OnChanged reageren
end;

procedure TNullableIntegerWrapper.SetIsNull(const Value: Boolean);
var
  NewNullableValue: TNullableInteger;
begin
  if Self.IsNull <> Value then
  begin
    NewNullableValue := Self.NullableValue;
    NewNullableValue.IsNull := Value; // zodat we via Changed() kunnen lopen
    Self.NullableValue := NewNullableValue;
  end;
end;

procedure TNullableIntegerWrapper.SetNullableValue(const Value: TNullableInteger);
begin
  if Self.NullableValue <> Value then
  begin
    Self.FNullableValue := Value;
    Changed();
  end;
end;
```

Properties

```
constructor TCustomNullableIntegerStaticText.Create(aOwner: TComponent);
begin
  inherited;
  FValueEditor := TNullableIntegerWrapper.Create(TNullableInteger.Null());
  ValueEditor := FValueEditor;
  // De Object Inspector wijzigt alleen de ValueEditor sub-properties
  // ValueEditorChanged wijzigt dan de onderliggende Value propery
  ValueEditor.OnChange := ValueEditorChanged;
  Value := 984; // http://www.stetson.edu/~efriedma/numbers.html
end;

destructor TCustomNullableIntegerStaticText.Destroy;
begin
  FValueEditor.Free;
  FValueEditor := nil;
  inherited;
end;

procedure TCustomNullableIntegerStaticText.CalculateText;
begin
  if Assigned(Self) then
    TControlUtils.SetCaption(Self, Value);
end;
```

Properties

```
function TCustomNullableIntegerStaticText.GetIsNull: Boolean;
begin
  if Assigned(Self.ValueEditor) then
    Result := Self.ValueEditor.IsNull
  else
    Result := False;
end;

function TCustomNullableIntegerStaticText.GetValue: TNullableInteger;
begin
  Result := FValue;
end;

function TCustomNullableIntegerStaticText.GetValueEditor:
   TNullableIntegerWrapper;
begin
  if Assigned(Self.FValueEditor) then
    Self.FValueEditor.NullableValue := TNullableInteger.Parse(Text);
  Result := Self.FValueEditor;
end;
```

Properties

```
procedure TCustomNullableIntegerStaticText.SetIsNull(const Value: Boolean);
begin
  if Assigned(Self.ValueEditor) then
    Self.ValueEditor.IsNull := Value;
end;

procedure TCustomNullableIntegerStaticText.SetValue(const NewValue: TNullableInteger);
begin
  FValue := NewValue;
  CalculateText(); // reflecteer Value naar Text/Caption
end;

procedure TCustomNullableIntegerStaticText.SetValueEditor(const NewValue: TNullableIntegerWrapper);
begin
  if Assigned(Self.FValueEditor) then
    Self.FValueEditor.Assign(NewValue)
  else
    Text := '';
end;

procedure TCustomNullableIntegerStaticText.ValueEditorChanged(Sender: TObject);
var
  NullableIntegerWrapper: TNullableIntegerWrapper;
begin
  if Assigned(Sender) then
  begin
    if Sender is TNullableIntegerWrapper then
    begin // reflecteer ValueEditor naar Value
      NullableIntegerWrapper := TNullableIntegerWrapper(Sender);
      Self.Value := NullableIntegerWrapper.NullableValue;
    end;
  end;
end;
```

How to create nullable types?

- Some knowledge is needed:
  - Value versus reference types
  - Operator overloading
  - Helpers
  - Properties
  - TypeInfo

TypeInfo

- The Object Inspector requires TypeInfo
  - Records do not TypeInfo, unless it is managed because it (recursively) has at least one field that is managed:
    - string,
    - interface,
    - method reference,
    - dynamic array,
    - a record that itself is managed
  - Being managed is required Initialize/Finalize handling
  - Managed record TypeInfo is very limited
- So the object inspector will not support records soon
  - TypeInfo for records will likely be extended in the future
  - Maybe NullableWrappers won't be needed any more

TypeInfo

- Without TypeInfo this does not compile:

```
type
  TNoTypeInfoRecord  = record
    X: Integer;
    Y: Double;
  end;

procedure TLogic.Go;
var
  NoTypeInfoRecordTypeInfo: PTypeInfo;
  TypeInfoRecordTypeInfo: PTypeInfo;
begin
  NoTypeInfoRecordTypeInfo :=
    TypeInfo(TNoTypeInfoRecord);
end;
```

  – [DCC Error] TypeInfoConsoleProject.dpr(39): E2134 Type 'TNoTypeInfoRecord' has no type info

TypeInfo

- But this compiles:

```
type
  TTypeInfoRecord  = record
    X: Integer;
    Y: Double;
    S: string;
  end;

procedure TLogic.Go;
var
  TypeInfoRecordTypeInfo:
  PTypeInfo;
begin
  TypeInfoRecordTypeInfo :=
  TypeInfo(TTypeInfoRecord);
  Logger.Log(
    'TypeInfoRecordTypeInfo',
    TypeInfoRecordTypeInfo);
end;
```

TypeInfoRecordTypeInfo:
  TypeInfo for type TTypeInfoRecord
    TypeInfo.Kind: tkRecord
    RecordFieldTable.X: 25714
    RecordFieldTable.Size: 24
    RecordFieldTable.Count: 1
    RecordFieldTable[0] Offset
      00000010:
      TypeInfo for type string
        TypeInfo.Kind: tkUString

TypeInfo

- A published record property might be possible in the object inspector with a lot of low level work,
  but
  - Would be very Delphi version specific
  - A lot of work
  - Hard to get stable

Compiler bugs

- There are and were compiler bugs like this:
  - http://qc.codegear.com/wc/qcmain.aspx?d=30131
  - The cause is that expressions can return records and classes, and that the compiler has a complex graph to go through in order to resolve them
    - Operatoren add an extra level of complexity
  - Since Delphi 2007 most of these bugs have been solved
- Solutions for a less complex graph:
  - usage of temporary varibles
  - Implement a propery through a field in stead of through a Getter/Setter methods
    - It is the reason both IsFilled (read from field) and IsNull (with getter method) are part of the nullable types

What can nullable types do?

- Make calculations
  - much easier
  - function like they work in SQL
- Getting Data from/to your database in first cass Delphi types
- Be properties in classes and components

- All are reliably possible from Delphi 2007 (parts from Delphi 2005 and 2006)
- Delphi 2009 possibly can do parts with generics
  - Allen Bauer has created a TNullable<T> that suppors the (in)equality operators = en <>
    - http://blogs.codegear.com/abauer/2008/09/18/38869
  - Using that as a base, it might be possible to create generic versions of other operators

Q & A

Jeroen Pluimers
better office benelux
jpluimers@better-office.com
If you have questions after the session,
please mail me

Ideas for further reading

- NotNull
  - http://neude.net/2008/08/the-opposite-of-nullable-types/
- System.pas
  - procedure _FinalizeRecord(p: Pointer; typeInfo: Pointer);
  - procedure _InitializeRecord(p: Pointer; typeInfo: Pointer);
- StringList als een ValueType:
  - http://cc.codegear.com/Item/25670
- Auto pointers in Delphi:
  - http://barrkel.blogspot.com/2008/09/smart-pointers-in-delphi.html
  - http://66.102.9.104/translate_c?hl=en&sl=zh-CN&tl=en&u=http://www.cnblogs.com/felixYeou/archive/2008/08/27/1277250.html&usg=ALkJrhj_IqVBH4Yj61WinwNk48IpEpfjGw
  - http://translate.google.com/translate?u=http%3A%2F%2Fwww.cnblogs.com%2FfelixYeou%2Farchive%2F2008%2F09%2F06%2F1285806.html&hl=en&ie=UTF-8&sl=zh-CN&tl=en
- Delphi Generics introductie:
  - http://www.felix-colibri.com/papers/oop_components/delphi_generics_tutorial/delphi_generics_tutorial.html
  - http://hallvards.blogspot.com/2007/08/highlander2-beta-generics-in-delphi-for.html