Rediscovering the Delphi Language

Exploiting language features for fun and profit

Oslo Delphi User Group
Asbjørn Heid
2017.11.08

About me

- Started programming QBasic at age 12
- Been a hobby ever since
- Used Delphi 3 to Delphi 2006 almost exclusively, hobby only
- Moved to the dark side in 2006
  - C++ and Python
  - no Delphi
- Got a Delphi job in 2012, started rediscovering the language
- Like having fun with programming languages, and like a challenge

Had no help in beginning, self-taught using F1 and a couple of examples.

New language features

- Nested types
- Anonymous methods
- Class-like features for records: methods, visibility (public/private)
- Generics
- Operator overloading
- for..in
- Attributes
- Unicode
- Type helpers

Nice list of features and when they were introduced:
https://stackoverflow.com/a/8460108

Language features introduced since Delphi 2005.

New language features

- Nested types
- Anonymous methods
- Class-like features for records: methods, visibility (public/private)
- Generics
- Operator overloading
- ~~for..in~~
- ~~Attributes~~
- ~~Unicode~~
- ~~Type helpers~~

Nice list of features and when they were introduced:
https://stackoverflow.com/a/8460108

Will not focus on all topics.

Started exploring the new language features.

Was missing concepts and abilities from C++ and Python.

Tried to explore ways to bring those concepts to Delphi.

I'm not an expert, some of the things covered in the talk may be know to some of you, but hopefully interesting to at least a few of you.

Not claiming any new discoveries here either.

Nested types

Nested types

- Ability to declare types inside classes/records
- Useful for implementation-detail types
- Can be used for namespaces:
  - A record with only type and class method declarations

```
type
  TRec = record
  type
    TDetail = class
      ...
    end;
  public
    ...
  end;
```


Use as namespace is useful for generics.

Use public/private as separator between nested type section and regular fields/methods, due to compiler issues.

Anonymous methods

Anonymous methods

- Enables inline function/procedure declaration
- Captures variables by reference

```
var
  i: integer;
  p: TProc; // type TProc = reference to procedure;
begin
  i := 123;

  // variable i will be captured by reference
  p := procedure() begin WriteLn('i = ', i); end;

  i := 42;
  p();
end.
```

```
i = 42
```

Asbjørn Heid 8

Also known as closures.

Two parts:
- a method reference
- the anonymous method declaration

Method references can reference free functions, regular methods and anonymous methods.

Function parameters are captured as if they were regular local variables.

Anonymous methods, details

- Implemented as interfaced objects with method named `Invoke`
- Captured variables are stored in object implementing the anonymous method

```
type TIntegerFunc = reference to function(): integer;

function NewCounter(const Start, Step: integer): TIntegerFunc;
var
  i: integer;
begin
  i := Start;
  result :=
    function(): integer
    begin
      result := i;
      i := i + Step;
    end;
end;
```


Anonymous methods are reference counted, which can cause lifetime management issues if it captures reference counted resources.

In the example, i is captured and stored as a member of the object implementing the anonymous method.

Anonymous methods, details cont.

- Captured local variables can be used to maintain state

```
var
  cnt: TIntegerFunc;
  i: integer;
begin
  cnt := NewCounter(100, 3);

  for i := 0 to 3 do
  begin
    WriteLn('i=', i, ' => ', cnt());
  end;
end.
```

```
i=0 => 100
i=1 => 103
i=2 => 106
i=3 => 109
```

Anonymous methods, exploits

- Can exploit implementation:
  - Method reference types can be used instead of an interface in class definitions
  - Interfaces can inherit from method reference types
  - Can overload `Invoke` method!
- Relies on implementation details, may break in future

Anonymous methods, exploits cont.

```
type
  TIntegerFunc = reference to function(): integer;

  TIntegerFuncImpl = class(TInterfacedObject, TIntegerFunc)
    function Invoke(): integer;
  end;

function TIntegerFuncImpl.Invoke: integer;
begin
  result := 42;
end;

var f: TIntegerFunc;
begin
  f := TIntegerFuncImpl.Create();

  WriteLn( f() );
end.
```

42

Defining an interface with an Invoke method with the correct signature allows one to assign an instance of that interface to a method reference variable.

Anonymous methods, exploits cont.

- Can overload Invoke method(!!!)
- Example adapted from blog post by Stefan Glienke

http://delphisorcery.blogspot.com/2015/06/anonymous-method-overloading.html

```
type
  TIntegerFunc = reference to function(): integer;

  IIntegerObj = interface(TIntegerFunc) // inherit from method reference type
    procedure Invoke(const Value: integer); overload;
  end;

  TIntegerObjImpl = class(TInterfacedObject, IIntegerObj)
    function Invoke(): integer; overload; // from TIntegerFunc
    procedure Invoke(const Value: integer); overload; // from IIntegerObj
  end;
```


The overloaded Invoke in IIntegerObj has similar definition to what one would expect TIntegerProc to have, i.e. TIntegerProc = reference to procedure(const Value: integer);

Anonymous methods, exploits cont.

```
function TIntegerObjImpl.Invoke: integer;
begin
  result := 42;
end;

procedure TIntegerObjImpl.Invoke(const Value: integer);
begin
  WriteLn(Value);
end;

var
  io: IIntegerObj;
begin
  io := TIntegerObjImpl.Create();
  io(123);
  WriteLn( io );
end.
```

```
123
42
```

Class-like features for records

Class-like features for records

- Records can have methods, class methods and properties
- Records supports (strict) private and public visibility

```
type
  TRec = record
  strict private
    FData: string;
  public
    class function Create(const Data: string): TRec; static; // don’t use constructor

    procedure AppendData(const NewData: string);

    property Data: string read FData;
  end;
```


Why not use a class? Records have value semantics and has operator overloading support.

I discourage use of constructor as it can be called on an "instance" but behaves as a function returning a newly constructed record, unlike objects.

Class-like features for records, cont.

- Records have value semantics, and no polymorphism
- Delegate to interface instance to get safe* reference semantics
- Delegate to class reference instance to get polymorphism
- Hide instances from users by using private section

*: reference counted

Delegate to private interface instance to get reference semantics. Overloading implicit cast operator, so one can assign nil, to get a record behaving like an interface, I.e. reference semantics.

Unlike for example C#, Delphi supports virtual class methods. This can be used to implement polymorphism while maintaining value semantics.

Class-like features for records, polymorphism

- Delegate to class reference instance to get polymorphism

```
type
  TImplBase = class;
  TImpl = class of TImplBase;

  TRec = record
  strict private
    FValue: integer;
    FImpl: TImpl;
  public
    class function Create(const Value: integer; const Impl: TImpl): TRec; static;

    procedure Print;
  end;
```

Class-like features for records, polymorphism

- Use abstract class methods in TImplBase to introduce polymorphism

```
type
  TImplBase = class
  public
    class function GetString(const Value: integer): string; virtual; abstract;
  end;

  TDing = class(TImplBase)
  public
    class function GetString(const Value: integer): string; override;
  end;

function NewDingRec(const Value: integer): TRec;
begin
  result := TRec.Create(Value, TDing);
end;
```


NewDingRec is convenience constructor

Class-like features for records, polymorphism

```
class function TRec.Create(const Value: integer; const Impl: TImpl): TRec;
begin
  result.FValue := Value;
  result.FImpl := Impl;
end;

procedure TRec.Print;
begin
  WriteLn( FImpl.GetString(FValue) );
end;
```

Class-like features for records, polymorphism

```
procedure TRec.Print;
begin
  WriteLn( FImpl.GetString(FValue) );
end;

class function TDing.GetString(const Value: integer): string;
var
  i: integer;
begin
  result := '';
  for i := 0 to Value-1 do result := result.Trim + ' ding';
end;
```

Class-like features for records, polymorphism

```
procedure TRec.Print;
begin
  WriteLn( FImpl.GetString(FValue) );
end;

class function TDing.GetString(const Value: integer): string;
var
  i: integer;
begin
  result := '';
  for i := 0 to Value-1 do result := result.Trim + ' ding';
end;

var
  r: TRec;
begin
  r := NewDingRec(3);
  r.Print;
end.
```

```
ding ding ding
```

Generics

Generics

- A generic type is a type that has one or more type parameters

```
type TProc<T> = reference to procedure(Arg: T);
```

- An instantiation of a generic type with specific type arguments is called a parameterized type

```
type TIntegerProc = TProc<integer>;
```

- The parameterized type is created by replacing all instances of the type parameters with the respective type arguments.

Generics, cont.

- Each parameterized type exists as a separate type (class vars, RTTI)
- Can parameterize type and methods independently
- Cannot be used with free functions
  - Use a record as namespace, with generic class methods instead of free functions
- Type parameter can be restricted, to some degree
  - The `record` restriction also accepts basic types like integer and enums
  - Use `IInterface` or `IInvokable` to restrict type to interfaces

Generics, cont.

- Delphi has compile-time generics
- Worst of .Net generics and C++ template worlds
  - Can't use methods etc. beyond what's available for the (un)restricted type
  - Can't generate parameterized types at run-time
- Limited type parameter restriction (constraints)
- Very limited type inference
- But a lot better than no generics!

Generics, arrays

- Use TArray<T> instead of customs dynamical arrays

```
// these are assignment incompatible
type
  TIntArray1 = array of integer;
  TIntArray2 = array of integer;

// these are assignment compatible
type
  TGenericIntArray1 = TArray<integer>;
  TGenericIntArray2 = TArray<integer>;
```

Generics, compiler intrinsics

- TypeInfo() and TypeKind() are now compiler intrinsics
- Can be used to select optimized code paths for generic types
- Evaluated at compile-time so
  - no branch overhead
  - dead code eliminated
- Can be used for «manual» tag dispatching

- Default(), returns compiler default value for a given type

Generics, tag dispatching

Generics, tag dispatching

```
type
  TPingTag = record end;
  TDingTag = record end;

  TRec = record
    class procedure Ping<TTag>(); static;
  end;

class procedure TRec.Ping<TTag>;
begin
  if ( TypeInfo(TTag) = TypeInfo(TPingTag) ) then
    WriteLn('ping')
  else if ( TypeInfo(TTag) = TypeInfo(TDingTag) ) then
    WriteLn(‘ding')
  else
    raise EProgrammerNotFound.Create('Unknown tag type');
end;
```


One limitation is that tag errors cannot be caught at compile-time.

Generics, tag dispatching cont.

```
try
  TRec.Ping<TPingTag>();
except
  on E: Exception do Writeln(E.ClassName, ': ', E.Message);
end;
```

Generics, tag dispatching cont.

```
try
  TRec.Ping<TPingTag>();
except
  on E: Exception do Writeln(E.ClassName, ': ', E.Message);
end;
```

```
Generics1.dpr.21: begin
004D0ADC 55               push ebp
004D0ADD 8BEC             mov ebp,esp
Generics1.dpr.23: WriteLn('ping')
004D0ADF A19C874D00       mov eax,[$004d879c]
004D0AE4 BA080B4D00       mov edx,$004d0b08
004D0AE9 E8D279F3FF       call @Write0UString
004D0AEE E8CD7AF3FF       call @WriteLn
004D0AF3 E8A864F3FF       call @_IOTest
Generics1.dpr.28: end;
004D0AF8 5D               pop ebp
004D0AF9 C3               ret
```


Assembly output from TRec.Ping body.

Overloading and tag dispatching

Overloading and tag dispatching

- Can use similar tag dispatching with method overloading
- Selects implementation based on tag

```
type
  Functional = record
  type
    TBindArg1 = record end;
    TBindArg2 = record end;
  public
    class function Bind<T1, T2, R>(const F: TFunc<T1, T2, R>;
      const BindArg1: TBindArg1; const Arg2: T2): TFunc<T1, R>; overload; static;
  end;

function _1: Functional.TBindArg1; begin end;
function _2: Functional.TBindArg2; begin end;
```


Use nested type declaration as namespace for bind arg types, to avoid name clashes for other things which may want to use the same name for tag dispatching.

Overloading and tag dispatching, cont.

```
class function Functional.Bind<T1, T2, R>(const F: TFunc<T1, T2, R>;
  const BindArg1: TBindArg1; const Arg2: T2): TFunc<T1, R>;
begin
  result :=
    function(Arg1: T1): R
    begin
      result := F(Arg1, Arg2);
    end;
end;
```

Asbjørn Heid 35

So there's two orthogonal concepts going on here: the function generators, and the tag dispatching. The function generators is just used here to highlight the tag dispatching technique.

Bind<> is just a generator of convenience functions, which takes a function an fixes (binds) one or more arguments, and/or reorders the arguments.

A lot of Bind<> variants are needed in order to be able to bind the various possible argument fixing combinations. We can use operator overloading to implement them all with the same name. The tag dispatching is used to select the right one in a nice way.

In this slide we see a variant of Bind<> which fixes (binds) the second argument to function F, a function taking two parameters of type T1 and T2 respectively, and returning a value of type R.

By binding the second argument, it turns F into a function which takes a single argument of type T1 and returns a value of type R.

Overloading and tag dispatching, cont.

```
class function Functional.Bind<T1, T2, R>(const F: TFunc<T1, T2, R>;
  const BindArg1: TBindArg1; const Arg2: T2): TFunc<T1, R>;
begin
  result :=
    function(Arg1: T1): R
    begin
      result := F(Arg1, Arg2);
    end;
end;

var
  f: TFunc<boolean, string>;
begin
  // function BoolToStr(B: Boolean; UseBoolStrs: Boolean): string;
  f := Functional.Bind<boolean, boolean, string>(BoolToStr, _1, True);

  WriteLn( f(False) );
end.
```

```
false
```


Bind True as the UseBoolStrs argument of BoolToStr, so we don’t have to pass it every time.

Overloading and tag dispatching, cont.

- Multiple overloads allows for tag dispatching

```
type
  Functional = record
    class function Bind<T1, T2, R>(const F: TFunc<T1, T2, R>;
      const BindArg1: TBindArg1; const Arg2: T2): TFunc<T1, R>; overload; static;

    class function Bind<T1, T2, R>(const F: TFunc<T1, T2, R>;
      const Arg1: T1; const BindArg2: TBindArg2): TFunc<T2, R>; overload; static;

    class function Bind<T1, T2, R>(const F: TFunc<T1, T2, R>;
      const Arg1: T1; const Arg2: T2): TFunc<R>; overload; static;
  end;
```


Other overloads are also possible of course, like switching argument order etc. The tag dispatching allows the user to select the right overload in an intuitive manner.

Operator overloading

Operator overloading

- Allows one to use standard operators (+, *, xor etc) with custom types
- Limitation: at least one of the parameters *or* the return value must be of the custom type

```
type
  TRec = record
    class operator Add(const Rec: TRec; const s: string): TRec;

    class operator Implicit(const v: integer): TRec;
  end;

var
  r: TRec;
begin
  r := 42;
  r := r + 'foo';
end.
```

Operator overloading, cont.

- Possible to use `var` parameters when overloading operators
- Means we can store address of parameter

```
type
  Reference<T> = record
  strict private
    type Ptr = ^T;
  strict private
    FPtr: Ptr;
  private
    function GetValue: T;
    procedure SetValue(const Value: T);
  public
    class operator Implicit(var v: T): Reference<T>;

    property Value: T read GetValue write SetValue;
  end;
```

Operator overloading, cont.

```
function Reference<T>.GetValue: T;
begin
  result := FPtr^;
end;

class operator Reference<T>.Implicit(var v: T): Reference<T>;
begin
  result.FPtr := @v;
end;

procedure Reference<T>.SetValue(const Value: T);
begin
  FPtr^ := Value;
end;
```

Operator overloading, cont.

```
function RunSql(const SqlStatement: string; const QueryResults: TArray<Reference<Variant>>): boolean;
begin
  // run actual query here
  QueryResults[0].Value := 42;
  QueryResults[1].Value := 'foo';
  result := True;
end;

var
  id, name: variant;
begin
  RunSql('select id, name from table', [id, name]);

  WriteLn(VarToStr(id));
  WriteLn(VarToStr(name));
end.
```

```
42
foo
```


Example based on a Google+ community user who wanted to be able to run a SQL query and return the data in an array of variants.

Note that the compiler uses type inference to discover the overload of the implicit conversion operator.

Operator overloading, cont.

```
type
  Channel<T> = record
  strict private
    FImpl: Channels.Detail.IChannel<T>;
  private
    property Impl: IChannel<T> read FImpl;
  public
    class function Create(): Channel<T>; static;

    // Assign nil to release implementation
    class operator Implicit(const Impl: Channels.Detail.IChannel<T>): Channel<T>;

    procedure Close;

    // Sending
    class operator LessThanOrEqual(const Chan: Channel<T>; const Value: T): boolean; overload;
    class operator LessThanOrEqual(const Chan: Channel<T>; const Values: TArray<T>): boolean; overload;
    // Receiving
    class operator LessThanOrEqual(var Value: T; const Chan: Channel<T>): boolean;
    class operator LessThan(var Value: T; const Chan: Channel<T>): boolean;
  end;
```


Channel example inspired by Go channels.

Also showcases record delegating to interface for reference semantics.

Operator overloading, cont.

```
var
  chan: Channel<integer>;
  doneEvent: Event;
begin
  chan := Channel<integer>.Create();

  TThread.CreateAnonymousThread(
    procedure
    var
      data: TArray<integer>;
    begin
      data := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

      chan <= data; // blocking send data over channel

      chan.Close;
    end
  ).Start;
```

Operator overloading, cont.

```
  doneEvent := Event.Create();

  TThread.CreateAnonymousThread(
    procedure
    var
      v: integer;
    begin
      while (v <= chan) do // blocking read from channel
      begin
        WriteLn(v);
      end;

      doneEvent.Signal;
    end
  ).Start;

  doneEvent.WaitFor(INFINITE);
end;
```


The overloaded less-or-equal operator returns false once the channel is closed, and writes the value from the channel into the left-hand operand.

Operator overloading, teaser

```
var
  input, output: TArray<double>;
  P10: Expr;
  sqr: Expr.Func1;
begin
  // initialize input, input values are in [-1, 1]
  SetLength(input, 20000000);
  for i := 0 to High(input) do input[i] := 2 * i / High(input) - 1;

  sqr := Func.Sqr;

  // Legendre polynomial P_n(x) for n = 10
  P10 := (1 / 256) *
    (((((46189*sqr(_1)) - 109395)*sqr(_1) + 90090)*sqr(_1) - 30030)*sqr(_1) + 3465)*sqr(_1) - 63;

  // computes output[i] := P10(input[i]) on the GPU
  output := Compute.Transform(input, P10);

end;
```

Asbjørn Heid 46

Builds an expression tree, stored in P10. The Transform() function transforms each element in the input array by the supplied expression.

It takes the expression, generates OpenCL code on-the-fly, executes it on the GPU and downloads the result.

The _1 placeholder is used to represent the current array element.

The end…

- Example code will be published at https://github.com/aheid/RediscoveringDelphiTalk