Delphi Coding Standards Document

General Source Code Formatting Rules

Indentation

Indenting will be two spaces per level. Do not save tab characters to source files. The reason for this is because tab characters are expanded to different widths with different users settings and by different source management utilities (print, archive, version control, etc.).

You can disable saving tab characters by turning off the "Use tab character" and "Optimal fill" check boxes on the Editor page of the Environment Options dialog (accessed via Tools | Environment).

Margins

Margins will be set to 80 characters. In general, source shall not exceed this margin with the exception to finish a word, but this guideline is somewhat flexible. Wherever possible, statements that extend beyond one line should be wrapped after a comma or an operator. When a statement is wrapped, it should be indented so that logically grouped segments are on the same level of indentation.

Comments

For comments, either // or { } pairs can be used.

The alternative notation of (* *) shall be reserved for temporarily removing code ("commenting out") during development.

Conditional Defines

Conditional defines shall be created with curly braces - "{", "}" - and with the conditional command in uppercase.

Each conditional define is named again in the closing block to enhance readability of the code.

They shall be indented in the same manner as blocks - for example

```
if ... then
begin
  {$IFDEF VER90}
    raise Exception.CreateRes(SError);
  {$ELSE}
    raise Exception.Create(SError);
  {$ENDIF VER90}
end;
```

Begin..End Pair

The begin statement appears on its own line. For example, the following first line is incorrect; the second line is correct:

```
for I := 0 to 10 do begin // Incorrect, begin on same line as for

for I := 0 to 10 do  // Correct, begin appears on a separate line
begin
```

```
if some statement = ... then
begin
  ...
end
else
begin
  SomeOtherStatement;
end;
```

The end statement always appears on its own line, an optional comment can be added after the end identifying the conditional statement:-

```
End; // if some statement = ...
```

In lengthy blocks of code the comment should always be specified.

The corresponding end statement is always indented to match its begin part.

Object Pascal

Parentheses

There shall never be white space between an open parenthesis and the next character. Likewise, there shall never be white space between a closed parenthesis and the previous character. The following example illustrates incorrect and correct spacing with regard to parentheses:

```
CallProc( AParameter ); // incorrect
CallProc(AParameter);   // correct
```

Procedures and Functions (Routines)

Naming / Formatting

Routine names shall always begin with a capital letter and be camel-capped for readability. The following is an example of an incorrectly formatted procedure name:

```
procedure thisisapoorlyformattedroutinename;
```

This is an example of an appropriately capitalized routine name:

```
procedure ThisIsMuchMoreReadableRoutineName;
```

Routines shall be given names meaningful to their content. Routines that cause an action to occur will be prefixed with the action verb, for example:

```
procedure FormatHardDrive;
```

Routines that set values of input parameters shall be prefixed with the word set - for example,

```
procedure SetUserName;
```

Routines that retrieve a value shall be prefixed with the word get - for example,

```

```

```
function GetUserName: string;
```

Formal Parameters

Formatting

Where possible, formal parameters of the same type shall be combined into one statement:

```
procedure Foo(Param1, Param2, Param3: Integer; Param4: string);
```

Naming

All formal parameter names will be meaningful to their purpose and typically will be based off the name of the identifier that was passed to the routine. When appropriate, parameter names will be prefixed with the character A - for example,

```
procedure SomeProc(AUserName: string; AUserAge: integer);
```

The "A" prefix is a convention to disambiguate when the parameter name is the same as a property or field name in the class.

Constant Parameters

When parameters of record, array, ShortString, or interface type are unmodified by a routine, the formal parameters for that routine shall mark the parameter as const. This ensures that the compiler will generate code to pass these unmodified parameters in the most efficient manner.

Parameters of other types may optionally be marked as const if they are unmodified by a routine. Although this will have no effect on efficiency, it provides more information about parameter use to the caller of the routine.

Name Collisions

When using two units that each contain a routine of the same name, the routine residing unit appearing last in the uses clause will be invoked if you call that routine. To avoid these uses-clause-dependent ambiguities, always prefix such method calls with the intended unit name-for example,

```
SysUtils.FindClose(SR);
```

or

```
Windows.FindClose(Handle);
```

Variables

Variable Naming and Formatting

Variables will be given names meaningful to their purpose.

Loop control variables are generally given a single character name such as I, J, or K. It is acceptable to use a more meaningful name as well such as UserIndex.

Boolean variable names must be descriptive enough so that their meanings of True and False values will be clear.

Local Variables

Local variables used within procedures follow the same usage and naming conventions for all other variables. Temporary variables will be named appropriately.

When necessary, initialization of local variables will occur immediately upon entry into the routine. Local AnsiString variables are automatically initialized to an empty string, local interface and dispinterface type variables are automatically initialized to nil, and local Variant and OleVariant type variables are automatically initialized to Unassigned.

Use of Global Variables

Use of global variables is strongly discouraged. However, they may be used when necessary. When this is the case, you are encouraged to keep global variables within the context where they are used. For example, a global variable may be global only within the scope of a single unit's implementation section.

Global data that is intended to be used by a number of units shall be moved into a common unit used by all.

Global data may be initialized with a value directly in the var section. Bear in mind that all global data is automatically zero-initialized, so do not initialize global variables to "empty" values such as 0, nil, '', Unassigned, and so on. One reason for this is because zero-initialized global data occupies no space in the exe file. Zero-initialized data is stored in a 'virtual' data segment that is allocated only in memory when the application starts up. Non-zero initialized global data occupies space in the exe file on disk.

Types

Capitalization Convention

Win32 API types are generally completely uppercase, and you should follow the convention for a particular type name shown in the Windows.pas or other API unit. For other variable names, the first letter shall be uppercase, and the rest shall be camel-capped for clarity. Here are some examples:

```
var
  WindowHandle: HWND; // Win32 API type
  I: Integer;         // type identifier introduced in System unit
```

Floating Point Types

Use of the Real type is discouraged because it exists only for backward compatibility with older Pascal code. Use Double for general purpose floating point needs. Also, Double is what the processor instructions and busses are optimized for and is an IEEE defined standard data format. Use Extended only when more range is required than that offered by Double. Extended is an Intel specified type and not supported on Java. Use Single only when the physical byte size of the floating point variable is significant (such as when using other-language DLLs).

Enumerated Types

Names for enumerated types must be meaningful to the purpose of the enumeration. The type name must be prefixed with the T character to annotate it as a type declaration. The identifier list of the enumerated type must contain a lowercase two to three character prefix that relates it to the original enumerated type name-for example,

```
TSongType = (stRock, stClassical, stCountry);
```

Variable instances of an enumerated type will be given the same name as the type without the T prefix (SongType) unless there is a reason to give the variable a more specific name such as FavoriteSongType1, FavoriteSongType2, and so on.

Variant and OleVariant

The use of the Variant and OleVariant types is discouraged in general, but these types are necessary for programming when data types are known only at runtime, such as is often the case in COM and database development. Use OleVariant for COM-based programming such as Automation and ActiveX controls, and use Variant for non-COM programming. The reason is that a Variant can store native Delphi strings efficiently (same as a string var), but OleVariant converts all strings to Ole Strings (WideChar strings) and are not reference counted-they are always copied.

Structured Types

Array Types

Names for array types must be meaningful to the purpose for the array. The type name must be prefixed with a T character. If a pointer to the array type is declared, it must be prefixed with the character P and declared immediately prior to the type declaration - for example,

```
type
  PCycleArray = ^TCycleArray;
  TCycleArray = array[1..100] of Integer;
```

When practical, variable instances of the array type will be given the same name as the type name without the T prefix.

Record Types

A record type shall be given a name meaningful to its purpose. The type declaration must be prefixed with the character T. If a pointer to the record type is declared, it must be prefixed with the character P and declared immediately prior to the type declaration.

Each element within the record should have a common prefix based on the record name, e.g. em for employee, th for transaction header, etc…

The type declaration for each element may optionally be aligned in a column to the right - for example,

```
type
  PEmployee = ^TEmployee;
  TEmployee = record
    emId   : String[10];
    emName : String[30];
  end;
```

Statements

if Statements

The most likely case to execute in an if/then/else statement shall be placed in the then clause, with less likely cases residing in the else clause(s).

Try to avoid chaining if statements and use case statements instead if at all possible.

Do not nest if statements more than five levels deep. Create a clearer approach to the code.

If multiple conditions are being tested in an if statement, conditions should be arrange from left to right in order of least to most computation intensive. This enables your code to take advantage of short-circuit Boolean evaluation logic built into the compiler. For example, if Condition1 is faster than Condition2 and Condition2 is faster than Condition3, then the if statement should be constructed as follows:

```
if Condition1 and Condition2 and Condition3 then
```

When multiple conditions are tested it, sometimes is advisable to have each condition on a line of its own. This is particularly important in those cases, where one or more conditional statements are long. If this style is chosen, the conditions are indented, so that they align to each other - for example

```
if Condition1 and
   Condition2 and
   Condition3 then
```

Reading top-to-bottom usually is easier than reading left-to-right, especially when dealing with long, complex constructs, operators (and/or/…) should always be at the end of the lines.

When a part of an if statement extends beyond a single line, a begin/end pair shall be used to group these lines. This rule shall also apply when only a comment line is present or when a single statement is spread over multiple lines.

The else clause shall always be aligned with the corresponding if clause.

Case Statements

General Topics

The individual cases in a case statement should be ordered by the case constant either numerically or alphabetically. If you use a user-defined type, order the individual statements according to the order of the declaration of the type.

In some situations it may be advisable to order the case statements to match their importance or frequency of hit.

The actions statements of each case should be kept simple and generally not exceed four to five lines of code. If the actions are more complex, the code should be placed in a separate procedure or function. Local procedures and functions are well-suited for this.

The use of the else clause of a case statement should be used only for legitimate defaults. It should always be used to detect errors and document assumptions, for instance by raising an exception in the else clause.

All separate parts of the case statement have to be indented as shown below. All condition statements shall be written in begin..end blocks. The else clause aligns with the case statement - for example:

```
case Condition of

  condition1 : begin
                 ...
               end;

  condition2 : begin
                 ...
               end;

else { case }
  ...
end;
```

The else clause of the case statement shall have a comment indicating that it belongs to the case statement.

Formatting

case statements follow the same formatting rules as other constructs in regards to indentation and naming conventions.

while Statements

The use of the Exit procedure to exit a while loop is discouraged; when possible, you should exit the loop using only the loop condition.

All initialization code for a while loop should occur directly before entering the while loop and should not be separated by other non-related statements.

Any ending housekeeping shall be done immediately following the loop.

with Statements

General Topics

The with statement should be used sparingly and with considerable caution. Avoid overuse of with statements and beware of using multiple objects, records, and so on in the with statement. For example:

```
with Record1, Record2 do
```

These things can confuse the programmer and can easily lead to difficult-to-detect bugs.

Please note that With statements can improve performance when used with COM Objects as each reference to the COM Object needs to be looked up, in the example below the first section of code does 1 lookup of the oTransaction object whereas the second section does 2 lookups, 1 per line:-

```
with oTransaction do
begin
  thOurRef := 'SIN000001';
  thAcCode := 'ABAP01';
end; // with oTransaction

oTransaction.thOurRef := 'SIN000001';
oTransaction.thAcCode := 'ABAP01';
```

Formatting

with statements follow the same formatting rules in regard to naming conventions and indentation as described in this document.

Structured Exception Handling

General Topics

Exception handling should be used abundantly for both error correction and resource protection. This means that in all cases where resources are allocated, a try..finally must be used to ensure proper deallocation of the resource. The exception to this is cases where resources are allocated / freed in the initialization / finalization of a unit or the constructor / destructor of an object.

Use of try..finally

Where possible, each allocation will be matched with a try..finally construct:

```
SomeClass1 := TSomeClass.Create
SomeClass2 := TSomeClass.Create;
try
  { do some code }
finally
  SomeClass1.Free;
  SomeClass2.Free;
end;
```

However, if other operations are performed between the Create's then separate try..finally blocks should be used:

```
SomeClass1 := TSomeClass.Create
Try

  { do stuff }

  SomeClass2 := TSomeClass.Create;
  try
    { do some code }
  finally
    SomeClass2.Free;
  end;
finally
  SomeClass1.Free;
end;
```

Use of try..except

Use try..except only when you want to perform some task when an exception is raised. In general, you should not use try..except to simply show an error message on the screen because that will be done automatically in the context of an application by the Application object. If you want to invoke the default exception handling after you have performed some task in the except clause, use raise to re-raise the exception to the next handler.

Use of try..except..else

The use of the else clause with try..except is discouraged because it will block all exceptions, even those for which you may not be prepared.

Classes

Naming / Formatting

Type names for classes will be meaningful to the purpose of the class. The type name must have the T prefix to annotate it as a type definition-for example,

```
type
  TCustomer = class(TObject)
```

Instance names for classes will generally match the type name of the class without the T prefix - for example,

```
var
  Customer: TCustomer;
```

Fields

Naming / Formatting

Class field names follow the same naming conventions as variable identifiers except that they are prefixed with the F annotation to signify they are field names.

Visibility

All fields should be private. Fields that are accessible outside the class scope must be made accessible through the use of a property.

Declaration

Each field shall be declared with a separate type on a separate line - for example

```
TNewClass = class(TObject)
private
  FField1: Integer;
  FField2: Integer;
end;
```

Methods

Naming / Formatting

Method names follow the same naming conventions as described for procedures and functions in this document.

Use of Static Methods

Use static methods when you do not intend for a method to be overridden by descendant classes.

Use of virtual Methods

Use virtual methods when you intend for a method to be overridden by descendant classes

Use of Abstract Methods

Do not use abstract methods on classes of which instances will be created. Use abstract only on base classes that will never be created.

Property Access Methods

All access methods must appear in the private or protected sections of the class definition.

Property access methods naming conventions follow the same rules as for procedures and functions. The read accessor method (reader method) must be prefixed with the word Get. The write accessor method (writer method) must be prefixed with the word Set. The parameter for the writer method will have the name Value, and its type will be that of the property it represents - for example,

```
TSomeClass = class(TObject)
private
  FSomeField: Integer;
protected
  function GetSomeField: Integer;
  procedure SetSomeField(Value: Integer);
public
  property SomeField: Integer read GetSomeField write SetSomeField;
end;
```

Properties

Naming / Formatting

Properties that serve as accessors to private fields will be named the same as the fields they represent without the F annotator.

Property names shall be nouns, not verbs. Properties represent data, methods represent actions.

Array property names shall be plural. Normal property names shall be singular.

Files

Form Files

A form file will be given a name descriptive of the form's purpose postfixed with ‘F’, e.g. the About Form will have a filename of AboutF.pas. The Main Form will have the filename MainF.pas.

Data Module Files

A data module will be given a name that is descriptive of the datamodule's purpose. The name will be postfixed with the two characters DM. For example, the Customers data module will have a filename of CustomersDM.pas/dfm.

Unit Files

Unit Name

Unit files will be given descriptive names. For example, a unit containing utility functions for customers might be called CustUtils.Pas.

It is suggested that units containing class definitions be prefixed with ‘o’, e.g. a unit containing a Customer class could be called oCustomer.Pas.

Uses Clauses

The uses clause in the interface section will only contain units required by code in the interface section. Remove any extraneous unit names that might have been automatically inserted by Delphi.

The uses clause of the implementation section will only contain units required by code in the implementation section. Remove any extraneous unit names.

Interface Section

The interface section will contain declarations for only those types, variables, procedure / function forward declarations, and so on that are to be accessible by external units. Otherwise, these declarations will go into the implementation section.

Implementation Section

The implementation section shall contain any declarations for types, variables, procedures / functions and so on that are private to the containing unit.

Initialization Section

Do not place time-intensive code in the initialization section of a unit. This will cause the application to seem sluggish when first appearing.

Finalization Section

Ensure that you deallocate any items that you allocated in the Initialization section.

General Purpose Units

A general purpose unit will be given a name meaningful to the unit's purpose. For example, a utilities unit will be given a name of BugUtilities.pas. A unit containing global variables will be given the name of CustomerGlobals.pas.

Keep in mind that unit names must be unique across all packages used by a project. Generic or common unit names are not recommended.

Component Units

Component units will be placed in a separate directory to distinguish them as units defining components or sets of components. They will never be placed in the same directory as the project. The unit name must be meaningful to its content.

Forms and Data Modules

Forms

Form Type Naming Standard

Forms types will be given names descriptive of the form's purpose. The type definition will be prefixed with Tfrm. A descriptive name will follow the prefix. For example, the type name for the About Form will be

```
TfrmAbout = class(TForm)
```

The main form definition will be

```
TfrmMain = class(TForm)
```

The customer entry form will have a name like

```
TfrmCustomerEntry = class(TForm)
```

Form Instance Naming Standard

Form instances will be named the same as their corresponding types without the T prefix. For example, for the preceding form types, the instance names will be as follows:

Type Name	***Instance Name***
TfrmAbout	frmAbout
TfrmMain	frmMain
TfrmCustomerEntry	frmCustomerEntry

Auto-creating Forms

Only the main form will be auto-created unless there is good reason to do otherwise. All other forms must be removed from the auto-create list in the Project Options dialog box. See the following section for more information.

Coupling

Forms should be loosely coupled to promote maintenance and reusability. For example, a detail window should not be aware of how or where it was created and should be provided all the information it needs to function by the creating routine. This information should be specified as

parameters into the constructor or as properties on the form, it should not be done via global variables.

Modal Form Instantiation Functions

All form units will contain a form instantiation function that will create, set up, show the form modally, and free the form. This function will return the modal result returned by the form. Parameters passed to this function will follow the "parameter passing" standard specified in this document. This functionality is to be encapsulated in this way to facilitate code reuse and maintenance.

The form variable will be removed from the unit and declared locally in the form instantiation function. Note, that this will require that the form be removed from the auto-create list in the Project Options dialog box. See Auto-Creating Forms in this document.

For example, the following unit illustrates such a function for a GetUserData form.

```
  unit UserDataF;
  interface
  uses
    Windows, Messages, SysUtils, Classes, Graphics,
    Controls, Forms, Dialogs, StdCtrls;
  type
    TfrmUserData = class(TForm)
      edtUserName: TEdit;
      edtUserID: TEdit;
    private
      { Private declarations }
    public
      { Public declarations }
    end;

  function GetUserData(var aUserName: string; var aUserID: Integer):
Word;

  implementation

  {$R *.DFM}

  function GetUserData(var aUserName: string; var aUserID: Integer):
Word;
  var
    frmUserData: TfrmUserData;
  begin
    frmUserData := TfrmUserData.Create(Application);
    try
      frmUserData.Caption := 'Getting User Data';
      Result := frmUserData.ShowModal;
      if Result = mrOK then
      begin
        aUserName := frmUserData.edtUserName.Text;
        aUserID := StrToInt(frmUserData.edtUserID.Text);
```

```
      end;
    finally
      frmUserData.Free;
    end;
  end;

  end.
```

Data Modules

Data Module Naming Standard

As for forms except using the dm prefix.

Components

Component Instance Naming Conventions

All components that are referenced in the code or referenced from other components must be given descriptive names.

Components will have a logical lowercase prefix to designate their type. The reasoning behind prefixing component names rather than post-fixing them is to make searching component names in the Object Inspector and Code Explorer easier by component type.

e.g.

btn	Buttons
chk	Check Boxes
cmb	Combo Box
edt	Edit Boxes
lbl	Label
lst	List Boxes
lv	List Views
mem	Memo Controls
men	Main Menu
pan	Panels
pop	Popup Menu
rad	Radio Buttons
ts	TabSheet