# Delphi/ Object Pascal Style Guide

By Gabriel Forner

Table of Contents

[Delphi/ Object Pascal Style Guide 1](#_Toc445468931)

[1.0 Introduction 1](#_Toc445468932)

[1.1 Acknowledgments 1](#_Toc445468933)

[2.0 Source Files 1](#_Toc445468934)

[2.1 Source-File Naming 1](#_Toc445468935)

[2.2 Source-File Organization 1](#_Toc445468936)

[2.2.1 Header block comment 1](#_Toc445468937)

[2.2.2 Unit declaration 1](#_Toc445468938)

[2.2.3 uses declarations 1](#_Toc445468939)

[2.2.4 class/interface declarations 1](#_Toc445468940)

[3.0 Naming Conventions 1](#_Toc445468941)

[3.1 Unit Naming 1](#_Toc445468942)

[3.2 Class/Interface Naming 1](#_Toc445468943)

[3.3 Field Naming 1](#_Toc445468944)

[3.4 Method Naming 1](#_Toc445468945)

[3.5 Local Variable Naming 1](#_Toc445468946)

[3.6 Reserved Words 1](#_Toc445468947)

[3.7 Primitive Data Types 1](#_Toc445468948)

[3.8 Type Declarations 1](#_Toc445468949)

[3.9 Interface Declarations 1](#_Toc445468950)

[3.10 UI Element Naming 1](#_Toc445468951)

[4.0 White Space Usage 1](#_Toc445468952)

[4.1 Blank Lines 1](#_Toc445468953)

[4.2 Blank Spaces 1](#_Toc445468954)

[4.2.2 Blanks should *not* be used: 1](#_Toc445468955)

[4.2.3 Blank spaces “may” be used: 1](#_Toc445468956)

[4.3 Indentation 1](#_Toc445468957)

[4.4 Continuation Lines 1](#_Toc445468958)

[5.0 Comments 1](#_Toc445468959)

[5.1 Block Comments 1](#_Toc445468960)

[5.2 Single-Line Comments 1](#_Toc445468961)

[6.0 Classes 1](#_Toc445468962)

[6.1 Class Body Organization 1](#_Toc445468963)

[6.1.1 Access levels 1](#_Toc445468964)

[6.1.2 Constructor declarations 1](#_Toc445468965)

[6.2 Method Declarations 1](#_Toc445468966)

[7.0 Interfaces 1](#_Toc445468967)

[7.1 Interface Body Organization 1](#_Toc445468968)

[8.0 Statements 1](#_Toc445468969)

[8.1 Simple Statements 1](#_Toc445468970)

[8.2 Compound Statements 1](#_Toc445468971)

[8.1.1 Assignment and expression statements 1](#_Toc445468972)

[8.1.2 Local variable declarations 1](#_Toc445468973)

[8.1.3 Array declarations 1](#_Toc445468974)

[8.2.3 if statement 1](#_Toc445468975)

[8.2.4 for statement 1](#_Toc445468976)

[8.2.5 while statement 1](#_Toc445468977)

[8.2.6 repeat until statement 1](#_Toc445468978)

[8.2.7 case statement 1](#_Toc445468979)

[8.2.8 try statement 1](#_Toc445468980)

# 1.0 Introduction

This document is not an attempt to define a grammar for the Object Pascal language. For instance, it is illegal to place a semicolon before an else statement; the compiler simply won't let you do it. As a result, this style guide does not lay out that rule. This document is meant to define the proper course of action in places where the language gives you a choice.

## 1.1 Acknowledgments

The majority of this document was defined internally by Borland/Inprise in 1998 and made freely available. There have been changes and updates to the original document that include code style specific to IQMS. The original document provided seems to no longer be available online.

The format of this document and some of its language is based on work done to define a style standard for the Java language. Java has had no influence on the rules for formatting Object Pascal source, but documents found on the Sun web site formed the basis for this document. In particular the style and format of this document were heavily influenced by "A Coding Style Guide for Java WorkShop and Java Studio Programming" by Achut Reddy. That document can be found at the following URL: <http://www.sun.com/workshop/java/wp-coding>

The Delphi team also contributed heavily to the generation of the original document that this document is primarily based on.

# 2.0 Source Files

Object Pascal source is divided up primarily into units and Delphi Project files, which both follow the same conventions. A Delphi Project file has a DPR extension. It is the main source file for a project. Any units used in the project shall have a PAS extension. Additional files, such as batch files, html files, or DLLs, may play a role in a project, but this paper only treats the formatting of DPR and PAS files.

## 2.1 Source-File Naming

Object Pascal supports long file names. If you are appending several words to create a single name, then it is best to use capital letters for each word in the name: MyFile.pas. This is known as Infix-Caps, or CamelCase. Extensions should be in lower case. For historical reasons, the Delphi source itself often confines itself to 8:3 naming patterns, but developers do not need to feel constrained by those limits.

If you are translating a C/C++ header file, then your Pascal header translation will usually have the same name as the file you are translating, except it should have a PAS extension. For instance, Windows.h would become Windows.pas. If the rules of Pascal grammar force you to combine multiple header files into a single unit, then use the name of the base unit into which you are folding the other files. For instance, if you fold WinBase.h into Windows.h, then call the resulting file Windows.pas.

## 2.2 Source-File Organization

Object Pascal units should contain the following elements in the following order:

* + Header block comment
  + Unit Name
  + Interface section
  + Implementation
  + A closing end and a period.

At least one blank line should separate each of these elements.

Additional elements can be structured in the order you find most appropriate, except that the top of the file should always list the header block first, the unit name second, then any conditional defines, compiler directives or include statements, then the uses clause:

// Library Unit Title

// Description: Shared use component library for

// all internal projects

unit MyButtons;

{$S-,W-,R-}

{$C PRELOAD}

interface

uses

Windows, Messages, Classes, Controls, Forms,

Graphics, StdCtrls;

It does not matter if you place a type section before a const section, or if you mix type and const sections up in any order you choose.

The implementation should list the word implementation first, then the uses clause, then any include statements or other directives:

implementation

uses

Consts,

SysUtils,

ActnList,

ImgList;

{$R BUTTONS.RES}

### 2.2.1 Header block comment

You may add a header block comment containing a general information about the unit. This information should include a general description followed by specific interesting details within the unit.

// Library Unit Title

// Description: This shared library does foo

//

// Be aware that “Specific interesting detail 1” acts in this way

### 2.2.2 Unit declaration

Every source file shall contain a unit declaration. The word unit is a reserved word, so it shall be in lower case. The name of the unit shall be in mixed upper and lowercase, and must be the same as the name used by the operating system's file system. Example:

unit MyUnit;

This unit would be called MyUnit.pas when an entry is placed in the file system.

### 2.2.3 uses declarations

Inside units, a uses declaration shall begin with the word uses, in lowercase. Add the names of the units, following the capitalization conventions used in the declaration found inside the units:

uses

Windows;

Each unit must be separated from its neighbor by a comma, and the last unit should have a semicolon after it. It is preferred to have a carriage return between each unit name declaration. This is done to facilitate the use of code merging engines found in version control systems.

uses

Windows,

Messages,

Classes,

Controls,

Forms,

Graphics,

StdCtrls;

### 2.2.4 class/interface declarations

A class declaration begins with two spaces, followed by an identifier prefaced by a capital T. Identifiers should begin with a capital letter, and should have capital letters for each embedded word (InfixCaps). Never use tab characters in your Object Pascal source. Example:

TMyClass

Follow the identifier with a space, then an equals sign, then the word class, all in lower case:

TMyClass = class

If you want to specify the ancestor for a class, add a parenthesis, the name of the ancestor class, and closing parenthesis:

TMyClass = class(TObject)

Scoping directives should be two spaces in from the margin, and declared in the order shown in this example:

TMyClass = class(TObject)

private

protected

public

published

end;

Data should always be declared only in the private section, and its identifier should be prefaced by an F. All type declarations should be four spaces in from the margin:

TMyClass = class(TObject)

private

FMyData: Integer;

function GetMyData: Integer;

procedure SetMyData(Value: Integer);

public

published

property MyData: Integer read GetMyData write SetMyData;

end;

[Interfaces](http://community.borland.com/article/0,1410,10280,00.html#7.0) follow the same rules as class declarations, except you should omit any scoping directives or private data, and should use the word interface rather than class.

# 3.0 Naming Conventions

Except for reserved words and directives, which are in all lowercase, all Pascal identifiers should use InfixCaps, which means the first letter should be a capital, and any embedded words in an identifier should be in caps, as well as any acronym that is embedded:

MyIdentifier

MyFTPClass

The major exception to this rule is in the case of header translations, which should always follow the conventions used in the header. For instance, write WM\_LBUTTONDOWN, not wm\_LButtonDown.

Except in header translations, do not use underscores to separate words. Class names should be nouns or noun phrases. Interface or class names depend on the salient purpose of the interface.

GOOD type names:

AddressForm, ArrayIndexOutOfBoundsException

BAD type names:

ManageLayout // verb phrase

delphi\_is\_new\_to\_me // underscores

## 3.1 Unit Naming

Use InfixCaps, as described at the beginning of this section. See also the section on [unit declarations](#_2.2.2_Unit_declaration).

Units that reference a top level UI design element, such as a TForm, TDataSource, or TFrame, should be named the same as the object with the addition of an underscore prefix. Since Delphi does not allow a form and a unit to be named the same, the prefix is added to avoid errors. For example, the unit containing a form named “MyMainForm” would be named “\_MyMainForm.pas.” This allows a quick scan of the project to identify the units that are forms and to maintain the same name for both form and unit.

## 3.2 Class/Interface Naming

Use InfixCaps, as described at the beginning of this section. Begin each type declaration with a capital T. Interface declarations should start with a capital I.

TMyType // type name

IMyInterface // interface name

See also the section on [class/interface declarations.](#_2.2.4_class/interface_declarations)

## 3.3 Field Naming

Use InfixCaps, as described at the beginning of this section. Begin each type declaration with a capital F, and declare all data types in the private section, using properties or getters and setters to provide public access. For example, use the name GetSomething to name a function returning an internal field value and use SetSomething to name a procedure setting that value.

Do not use all caps for const declarations except where required in header translations.

CORRECT

FMyString: string;

INCORRECT

lpstrMyString: string;

Legacy version of Delphi code contained prefixing for enumerated types. This is no longer the standard and has been deprecated. The new accepted standard for Delphi enumerated types is to use the dot reference (.). As in the examples below, using the type prefix and the enum value is new modern convention.

TBitBtnKind.Ok;

TBitBtnKind.Close;

When thinking about naming conventions, consider that one-character field names should be avoided except for temporary and looping variables.

Avoid variable l ("el") because it is hard to distinguish it from 1 ("one") on some printers and displays.

## 3.4 Method Naming

Method names should use the InfixCaps style. Start with a capital letter, and capitalize the first letter of any subsequent word in the name, as well as any letters that are part of an acronym. All other characters in the name are lower case. Do not use underscores to separate words. Note that this is identical to the naming convention for non-constant fields; however it should always be easy to distinguish the two from context. Method names should be imperative verbs or verb phrases.

Examples:

// GOOD method names:

ShowStatus, DrawCircle, AddLayoutComponent

// BAD method names:

MouseButton // noun phrase; doesn't describe function

drawCircle // starts with lower-case letter

add\_layout\_component // underscores

// The function of this method is unclear. Does

// it start the server running (better: StartServer),

// or test whether or not it is running

// (better: IsServerRunning)?

ServerRunning // verb phrase, but not imperative

A method to get or set some property of the class should be called GetProperty or SetProperty respectively, where Property is the name of the property.

Examples:

GetHeight, SetHeight

A method to test some boolean property of the class should be called IsVisible, where Visible is the name of the property.

Examples:

IsResizable, IsVisible

## 3.5 Local Variable Naming

Local variables follow the same naming rules as field names, except instead of using the initial F we use a capital A. However, simple names such as a looping variable “i” may also be used.

// CORRECT

procedure TMyForm.ProcedureOne(AParam: Integer);

var

ADisplayName: string; // this is our accepted format

i: Integer; // this is allowed

## 3.6 Reserved Words

Reserved words and directives should be all lowercase. This includes terms such as being, end, var, procedure, etc. A full list of Delphi reserved words is available on the Embarcadero website. <http://docwiki.embarcadero.com/RADStudio/Seattle/en/Fundamental_Syntactic_Elements#Reserved_Words>

## 3.7 Primitive Data Types

Primitive data types should be declared using InfixCaps capitalization. This would include types such as Integer, Boolean, Float, etc. The type declaration “string” is not a primitive data type, it is in fact a reserved word and should be declared by following the format described in the [Reserved Words](#_3.6_Reserved_Words) section.

## 3.8 Type Declarations

All type declarations shall begin with the letter T, and should follow the same capitalization specification laid out in the [beginning](#_Naming_Conventions) of this section, or in the section on [class and interface declarations.](#_2.2.4_class/interface_declarations)

## 3.9 Interface Declarations

All type interface shall begin with the letter I, and should follow the same capitalization specification laid out in the [beginning](#_Naming_Conventions) of this section, or in the section on [class and interface declarations.](#_2.2.4_class/interface_declarations)

## 3.10 UI Element Naming

Just as with variable names, UI elements are not named by using a prefix. Instead, UI elements should be named using a suffix. This process will allow you to create elements that are grouped together based on purposes. For example, when designing a form if you need to create several UI elements that reference an invoice you may find a list that resembles the following: InvoiceQuery, InvoiceDatasource, InvoiceGrid, InvoiceCompanyEdit, InvoiceDateEdit. This type of naming convention will allow all of your invoice (purpose) related UI elements to be grouped alphabetically within the object inspector.

# 4.0 White Space Usage

## 4.1 Blank Lines

Blank lines can improve readability by grouping sections of the code that are logically related. A blank line should also be used in the following places:

* + After the copyright block comment, package declaration, and import section.
  + Between class declarations.
  + Between method declarations.

## 4.2 Blank Spaces

Object Pascal is a very clean, easy to read language. In general, you don't need to add a lot of spaces in your code to break up lines. The next few sections give you some guidelines to follow when placing spaces in your code.

### 4.2.2 Blanks should not be used:

* + Between a method name and its opening parenthesis.
  + Before or after a .(dot) operator.
  + Between a unary operator and its operand.
  + Between a cast and the expression being cast.
  + After an opening parenthesis or before a closing parenthesis.
  + After an opening square bracket [ or before a closing square bracket ].
  + Before a semicolon.

Examples of correct usage:

function TMyClass.MyFunc(var Value: Integer);

MyPointer := @MyRecord;

MyClass := TMyClass(MyPointer);

MyInteger := MyIntegerArray[5];

Examples of incorrect usage:

function TMyClass.MyFunc( var Value: Integer ) ;

MyPointer := @ MyRecord;

MyClass := TMyClass ( MyPointer ) ;

MyInteger := MyIntegerArray [ 5 ] ;

### 4.2.3 Blank spaces “may” be used:

Blank spaces may be used in order to format a group of code into a column style display. Many times this is done with declarations and string concatenations. This is by no means required and is left to programmer discretion.

Property declaration example:

property ANewProperty : Integer read FFirstProperty;

property AnotherProperty: string read GetAnotherProperty write Set AnotherProperty;

property MoreProperties : TList read FMoreProperty write SetMoreProperties;

Variable declaration example:

var

ANewProperty : Integer;

AAnotherProperty: string;

AMoreProperties : TList;

String concatenation example:

MyQuery.Sql.Add(‘SELECT \* ‘ +

‘FROM MyTable ‘+

‘WHERE ID =:MyId’);

Creating a column display using a procedure signature

procedure Foo

(AParam1: Real;

AParam2: Real;

AParam3: Real);

## 4.3 Indentation

You should always indent two spaces for all indentation levels. In other words, the first level of indentation is two spaces, the second level four spaces, the third level 6 spaces, etc. Never use tab characters.

There are few exceptions. The reserved words unit, users, type, interface, implementation, initialization and finalization should always be flush with the margin. The final end statement at the end of a unit should be flush with the margin. In the project file, the word program, and the main begin and end block should all be flush with the margin. The code inside the begin..end block, should be indented at least two spaces.

## 4.4 Continuation Lines

Lines should be limited to about 80 columns. Lines longer than this should be broken into one or more continuation lines, as needed. All the continuation lines should be aligned and indented from the first line of the statement, and indented two characters. Always place begin statements on their own line. Since we allow declarations to be formatted into columns, the arguments of a method may also be organized as columns for increased legibility.

Examples:

// CORRECT

function CreateWindowEx( dwExStyle: DWORD;

lpClassName: PChar; lpWindowName: PChar;

dwStyle: DWORD; X, Y, nWidth, nHeight: Integer;

hWndParent: HWND; hMenu: HMENU; hInstance: HINST;

lpParam: Pointer): HWND; stdcall;

// ALLOWED – COLUMN FORMAT

function CreateWindowEx( dwExStyle: DWORD;

lpClassName: PChar;

lpWindowName: PChar;

dwStyle: DWORD;

X, Y, nWidth, nHeight: Integer;

hWndParent: HWND;

hMenu: HMENU;

hInstance: HINST;

lpParam: Pointer): HWND; stdcall;

Never wrap a line between a parameter and its type, unless it is a comma separated list, then wrap at least before the last parameter so the type name follows to the next line. The colon for all variable declarations contains no whitespace between it and the variable. There should be a single space following the colon before the type name;

// CORRECT

procedure Foo(Param1: Integer; Param2: Integer);

// INCORRECT

procedure Foo( Param :Integer; Param2:Integer );

Breaking within an if condition is allowed and may occur in multiple places in order to make the statement more legible/lucid

// CORRECT

if ((X = Y) or (Y = X) or

(Z = P) or (F = J) then

begin

S := J;

end;

// Allowed variants for legibility

if ( (X = Y) or (Y = X) )

or

( (Z = P) or (F = J) ) then

begin

S := J;

end;

More Examples:

// INCORRECT

while (LongExpression1 or LongExpression2) do begin

// DoSomething

// DoSomethingElse;

end;

// CORRECT

while (LongExpression1 or LongExpression2) do

begin

// DoSomething

// DoSomethingElse;

end;

// CORRECT

if (LongExpression1) or

(LongExpression2) or

(LongExpression3) then

# 5.0 Comments

The Object Pascal language supports two kinds of comments: block, and single-line comments. Some general guidelines for comment usage include:

* + It is helpful to place comments near the top of unit to explain its purpose.
  + It is helpful to place comments before a class declaration.
  + It is helpful to place comments before some method declarations.
  + Avoid making obvious comments:

i := i + 1; // Add one to i

* + Remember that misleading comments are worse than no comments at all.
  + Avoid putting any information into comments that is likely to become out of date.
  + Avoid enclosing comments in boxes drawn with asterisks or other special typography.
  + Temporary comments that are expected to be changed or removed later should be marked with the special tag "TODO:" so that they can easily be found afterwards. Delphi includes a view that allows you to see all the “TODO” comments in a convenient list. Ideally, all temporary comments should have been removed by the time a program is ready to be shipped.

Example:

//TODO: Change this to call Sort when it is fixed

List.MySort;

## 5.1 Block Comments

Object Pascal supports two types of block comments. The first type of block comment uses curly braces: { }. A second kind of block comment contains two characters, a parenthesis and an asterisk: (\* \*). This is sometimes called starparen comments. Both types of block comments are generally useful only during code development, as their primary benefit is that they allow nesting of comments, as long as the nest level is less than 2. Object Pascal doesn't support nesting comments of the same type within each other, so really there is only one level of comment nesting: curly inside of starparen, and starparen inside of curly. As long as you don't nest them, any other standard Pascal comments between comments of this type will be ignored. As a result, you can use this syntax to comment out a large chunk of code that is full of mixed code and comments:

Modern commenting conventions have been adopted due to the availability of code editors that allow short cut key commenting/uncommenting. The line comment of a double slash “//” is now used for block comments. Commenting characters such as the braces “{ }” and the starparen “(\* \*)” are now used primarily while developing and debugging.

Block comments used to describe a method should appear before the method declaration.

Example:

// CORRECT

// TMyObject.MyMethod

// This routine allows you to execute code.

procedure TMyObject.MyMethod;

begin

end;

// INCORRECT

procedure TMyObject.MyMethod;

// TMyObject.MyMethod

// This routine allows you to execute code.

begin

end;

Using block the starparen for debugging and developing allows you to temporarily comment out large sections of code as show here:

(\* procedure TForm1.Button1Click(Sender: TObject);

begin

DoThis; // Start the process

DoThat; // Continue iteration

// We need a way to report errors here, perhaps using

// a try finally block ???

CallMoreCode; // Finalize the process

end; \*)

In this example, the entire Button1Click method is commented out, including any of the subcomments found between the procedure's begin..end pair.

## 5.2 Single-Line Comments

A single-line comment consists of the characters // followed by text. Include a single space between the // and the comment itself. Place single line comments at the same indentation level as the code that follows it. You can group single-line comments to form a larger comment.

A single-line comment or comment group should always be preceded by a blank line, unless it is the first line in a block. If the comment applies to a group of several statements, then the comment or comment group should also be followed by a blank line. If it applies only to the next statement (which may be a compound statement), then do not follow it with a blank line.

Example:

Statement;

🡨 blank line here 🡪

// Open the database

Table1.Open;

Single-line comments can also follow the code they reference. These comments, sometimes referred to as trailing comments, appear on the same line as the code they describe. They should have at least one space-character separating them from the code they reference. If more than one trailing comment appears in a block of code, they should all be aligned to the same column.

Example:

if (not IsVisible) then

Exit; // nothing to do

Inc(StrLength); // reserve space for null terminator

Avoid commenting every line of executable code with a trailing comment. It is usually best to limit the comments inside the begin..end pair of a method or function to a bare minimum. Longer comments can appear in a block comment before the method or function declaration.

# 6.0 Classes

## 6.1 Class Body Organization

The body of a class declaration should be organized in the following order:

* + Field declarations
  + Method declarations
  + Property declarations

The fields, properties and methods in your class should be arranged alphabetically by name.

### 6.1.1 Access levels

Except for code inserted by the IDE, the scoping directives for a class should be declared in the following order:

* + Strict Private
  + Private declarations
  + Strict Protected
  + Protected declarations
  + Public declarations
  + Published declarations

There are six access levels for class members in Object Pascal: published, public, protected, strict protected, private, and strict private -- in order of decreasing accessibility. By default, the access level is published. In general, a member should be given the lowest access level which is appropriate for the member. For example, a member which is only accessed by classes in the same unit should be set to private access. And members that are only accessed within the same class should be *strict private*. Also, declaring a lower access level will often give the compiler increased opportunities for optimization. On the other hand, use of private makes it difficult to extend the class by sub-classing. If there is reason to believe the class might be sub-classed in the future, then members that might be needed by sub-classes should be declared protected or strict protected instead of private, and the properties used to access private data should be given protected or strict protected status, as appropriate.

You should never allow public access to data. Data should always be declared in the private sections, and any public access should be via getter and setter methods, or properties.

### 6.1.2 Constructor declarations

Methods should be arranged alphabetically. It is correct either to place your constructors and destructors at the head of this list in the public section, followed by any class methods, then event handlers, and the remainder of the list can be in alphabetical order.

If there is more than one constructor, and if you choose to give them all the same name, then sort them lexically by formal parameter list, with constructors having more parameters always coming after those with fewer parameters. This implies that a constructor with no arguments (if it exists) is always the first one. For greatest compatibility with C++Builder, try to make the parameter lists of your constructors unique. C++ cannot call constructors by name, so the only way to distinguish between multiple constructors is by parameter list.

## 6.2 Method Declarations

If possible, a method declaration should appear on one line.

Examples:

// Broken line is aligned two spaces in from left.

procedure ImageUpdate(Image img, infoflags: Integer,

x: Integer, y: Integer, w: Integer, h: Integer)

# 7.0 Interfaces

Interfaces are declared in a manner that runs parallel to the declaration for classes:

IInterfaceName = interface([Inherited Interface])

InterfaceBody

end;

An interface declaration should be indented two spaces. The body of the interface is indented by the standard indentation of four spaces. The closing end statement should also be indented two characters. There should be a semi-colon following the closing end statement.

There are no fields in an interface declaration. Properties, however, are allowed.

All interface methods are inherently public and abstract; do not explicitly include these keywords in the declaration of an interface method.

Except as otherwise noted, interface declarations follow the same style guidelines as classes.

## 7.1 Interface Body Organization

The body of an interface declaration should be organized in the following order:

* + Interface method declarations
  + Interface property declarations

The declaration styles of interface properties and methods are identical to the styles for class properties and methods.

# 8.0 Statements

Statements are one or more lines of code followed by a semicolon. Simple statements have one semicolon, while compound statements have more than one semicolon and therefore consist of multiple simple statements.

\*\*\* Length of compound statements should be kept to a minimum. For example, if there is a method or an if statement that contains a large amount of lines, this should probably be broken down into method calls and the code moved into those corresponding methods.

Here is a simple statement:

A := B;

Here is a compound, or structured, statement:

begin

B := C;

A := B;

end;

## 8.1 Simple Statements

A simple statement contains a single semicolon. If you need to wrap the statement, indent the second line two spaces in from the previous line:

MyValue :=

MyValue + (SomeVeryLongStatement / OtherLongStatement);

## 8.2 Compound Statements

Compound Statements always end with a semicolon. Object Pascal allows the statement that they immediately precedes the end statement to be optional. However, this style guide requires the use of a semicolon on the final statement.

begin

MyStatement;

MyNextStatement;

MyLastStatement; // semicolon optional by compiler.

end;

### 8.1.1 Assignment and expression statements

Each line should contain at most one statement. For example:

a := b + c; Inc(Count); // INCORRECT

a := b + c; // CORRECT

Inc(Count); // CORRECT

### 8.1.2 Local variable declarations

Local variables should have Camel Caps and should be preceded with a capital A. Do not preface variable names with an F, as that convention is reserved for Fields in a class declaration:

var

AMyData: Integer;

AMyString: string;

You may declare multiple identifiers of the same type on a single line:

var

AArraySize, AArrayCount: Integer;

This practice is discouraged in class declarations. There you should place each field on a separate line, along with its type.

### 8.1.3 Array declarations

There should always be a space before the opening bracket "[" and after the closing bracket.

type

TMyArray = array [0..100] of Char;

### 8.2.3 if statement

If statements should always appear on at least two lines.

Example:

// INCORRECT

if A < B then DoSomething;

// CORRECT

if A < B then

DoSomething;

In compound if statements, put each element separating statements on a new line:

Example:

// INCORRECT

if A < B then begin

DoSomething;

DoSomethingElse;

end else begin

DoThis;

DoThat;

end;

// CORRECT

if A < B then

begin

DoSomething;

DoSomethingElse;

end

else

begin

DoThis;

DoThat;

end;

When building an else-if chain, it is acceptable to and the following if statement to the previous else. Like this:

// CORRECT

if Condition then

begin

DoThis;

end

else if NextCondition then

begin

DoThat;

end

else if FinalCondition then

begin

DoSomethingElse;

end

else

begin

DoDefaultThing;

end; *// end if chain*

Here are a few more variations that are considered valid:

// CORRECT

if Condition then

begin

DoThis;

end

else

begin

DoThat;

end;

// CORRECT

if Condition then

begin

DoThis;

end

else

DoSomething;

// CORRECT

if Condition then

begin

DoThis;

end

else

DoSomething;

A couple of examples that have fallen out of favor but deserves honorable mention. These are no longer used:

if Condition then

DoThis

else DoThat;

if Condition then DoThis

else DoThat;

### 8.2.4 for statement

Example:

// INCORRECT

for i := 0 to 10 do begin

DoSomething;

DoSomethingElse;

end;

// CORRECT

for i := 0 to 10 do

begin

DoSomething;

DoSomethingElse;

end;

### 8.2.5 while statement

Example:

// INCORRECT

while x < j do begin

DoSomething;

DoSomethingElse;

end;

// CORRECT

while x < j do

begin

DoSomething;

DoSomethingElse;

end;

### 8.2.6 repeat until statement

Example:

// CORRECT

repeat

x := j;

j := UpdateValue;

until j > 25;

### 8.2.7 case statement

Example:

// CORRECT

case Control.Align of

alLeft, alNone: NewRange := Max(NewRange, Position);

alRight: Inc(AlignMargin, Control.Width);

end;

// CORRECT

case x of

csStart:

begin

j := UpdateValue;

end;

csBegin: x := j;

csTimeOut:

begin

j := x;

x := UpdateValue;

end;

end;

// CORRECT

case ScrollCode of

SB\_LINEUP, SB\_LINEDOWN:

begin

Incr := FIncrement div FLineDiv;

FinalIncr := FIncrement mod FLineDiv;

Count := FLineDiv;

end;

SB\_PAGEUP, SB\_PAGEDOWN:

begin

Incr := FPageIncrement;

FinalIncr := Incr mod FPageDiv;

Incr := Incr div FPageDiv;

Count := FPageDiv;

end;

else

Count := 0;

Incr := 0;

FinalIncr := 0;

end;

### 8.2.8 try statement

Example:

// Correct

try

try

EnumThreadWindows(CurrentThreadID, @Disable, 0);

Result := TaskWindowList;

except

EnableTaskWindows(TaskWindowList);

raise;

end;

finally

TaskWindowList := SaveWindowList;

TaskActiveWindow := SaveActiveWindow;

end;