

## Object Pascal Style Guide - by Charles Calvert

***Abstract:** This article documents a standard style for formatting Delphi code. It is based on the conventions developed by the Delphi team.*

### Object Pascal Style Guide

This article documents a standard style for formatting Delphi code. It is based on the conventions developed by the Delphi team.

We take it for granted that many well established shops will have conventions different than those specified here. As a result, we strongly recommend using a tool that can convert your code into Borland style before submitting it to Borland, Project JEDI, or any other public source repository. We don't want to force you to change your conventions, but we insist that all code that ships with Borland products follows these conventions. We strongly encourage you to follow these conventions when submitting code into any form of public repository.

Object Pascal is a beautifully designed language. One of its great virtues is its readability. These standards are designed to enhance that readability of Object Pascal code. When developers follow the simple conventions laid out in this guide, they will be promoting standards that benefit all Delphi developers by using a uniform style that is easy to read. Efforts to enforce these standards will increase the value of a developer's source code, particularly during maintenance and debugging cycles.

It goes without saying that these are conventions based primarily on matters of taste. Though we believe in, and admire the style promoted in these pages, we support them not necessarily because we believe they are right and others are wrong, but because we believe in the efficacy of having a standard which most developers follow. The human mind adapts to standards, and finds ways to quickly recognize familiar patterns, thereby assimilating meaning quickly and effortlessly. It is the desire to create a standard that will make reading code as simple as possible for the largest number of people that is behind this effort. If at first our guidelines seem strange to you, we ask you to try them for awhile, and then we are sure you will grow used to them over time. Or, if you prefer, keep your code in your own format, and run it through a program that follows our guidelines before submitting it to Borland or to a public repository.

Some text editors, such as Visual SlickEdit can help you format your code according to a particular style. Readers who are aware of other tools that provide this same service should write me at that address provided at the end of this section.

One free formatter developed by Egbert van Nes is available at the following URL:  
<http://www.slm.wau.nl/wkao/delforex.html>.

A commercial option is CrackerJax for Delphi:  
<http://www.kineticsoftware.com/html/products.html>.

Before closing this introduction, I want to reiterate that on the Borland web site, and on the CDs that we ship with our product, these standards are the law. We want to present our code in a unified and easy to read style, and enforcing the rules in this guide is the simplest way to achieve that end.

Do not post this specification on other web sites. Instead, simply link to this version of the document.

We accept feedback in the form of corrections or suggestions. Send your communications to [Charlie Calvert](#).

---

## Contents

- 1.0 [Introduction](#)
  - 1.1 [Background](#)
  - 1.2 [Acknowledgments](#)

- 2.0 Source Files
  - 2.1 Source-File Naming
  - 2.2 Source-File Organization
    - 2.2.1 Copyright/ID block comment
    - 2.2.2 unit declaration
    - 2.2.3 uses declarations
    - 2.2.4 class/interface declarations
- 3.0 Naming Conventions
  - 3.1 Unit Naming
  - 3.2 Class/Interface Naming
  - 3.3 Field Naming
  - 3.4 Method Naming
  - 3.5 Local Variable Naming
  - 3.6 Reserved Words
  - 3.7 Type Declarations
- 4.0 White Space Usage
  - 4.1 Blank Lines
  - 4.2 Blank Spaces
    - 4.2.1 A single blank space (not tab) should be used:
    - 4.2.2 Blanks should *not* be used:
  - 4.3 Indentation
  - 4.4 Continuation Lines
- 5.0 Comments
  - 5.1 Block Comments
  - 5.2 Single-Line Comments
- 6.0 Classes
  - 6.1 Class Body Organization
  - 6.2 Method Declarations
  - 6.3 Data Store Declarations
- 7.0 Interfaces
  - 7.1 Interface Body Organization
- 8.0 Statements
  - 8.1 Simple Statements
    - 8.1.1 Assignment and expression statements
    - 8.1.2 Local variable declarations
    - 8.1.3 Array declarations
  - 8.2 Compound Statements
    - 8.2.3 if statement
    - 8.2.4 for statement
    - 8.2.5 while statement
    - 8.2.6 repeat until statement
    - 8.2.7 case statement
    - 8.2.8 try statement

## 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, I do not lay that rule out in this style guide. This document is meant to define the proper course of action in places where the language gives you a choice. I usually remain mute on matters that can only be handled one way.

### 1.1 Background

The guidelines presented here are based on the public portions of the Delphi source. The Delphi source should follow these guidelines precisely. If you find cases where the source varies from these guidelines, then these guidelines, and not the errant source code, should be considered

your standard. Nevertheless, you should use the source as a supplement to these guidelines, at least so far as it can help you get a general feel for how your code should look.

## 1.2 Acknowledgments

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 this document, and indeed, it would not have been possible to create it without their help.

## 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 will 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 InfixCaps, or Camel Caps. Extensions should be in lower case. For historical reasons, the Delphi source itself often confines itself to 8:3 naming patterns, but developers no longer need feel constrained by those limits, even if turning in source that might be used by the Delphi team.

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

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

1. Copyright/ID block comment
2. Unit Name
3. Interface section
4. Implementation
5. 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 copyright first, the unit name second, then any conditional defines, compiler directives or include statements, then the uses clause:

```
{ ***** }
{
{     Borland Delphi Visual Component Library     }
{
{     Copyright (c) 1995,98 Inprise Corporation    }
{
{ ***** }
{ ***** }
```

```

unit Buttons;

{$S-,W-,R-}
{$C PRELOAD}

interface

uses
  Windows, Messages, Classes,
  Controls, Forms, Graphics,
  StdCtrls, ExtCtrls, CommCtrl;

```

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 Copyright/ID block comment

Every source file should start with a block comment containing version information and a standard copyright notice. The version information should be in the following format:

```

{*****}
{
  Widgets Galore
}
{
  Copyright (c) 1995,98 Your Company
}
{*****}

```

The copyright notice should contain at least the following line:

```

  Copyright (c) yearlist CopyrightHolder.

```

If you are a third party creating a file for use by Borland, you may add your name at the bottom of the copyright notice:

```

{*****}
{
  Borland Delphi Visual Component Library
  Copyright (c) 1995,99 Borland International
  Created by Project JEDI
}
{*****}

```

## 2.2.2 Unit declaration

Every source file should contain a unit declaration. The word unit is a reserved word, so it should be in lower case. The name of the unit should 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 should 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 MyUnit;
```

Each unit must be separated from its neighbor by a comma, and the last unit should have a semicolon after it:

```
uses  
  Windows, SysUtils, Classes, Graphics, Controls, Forms,  
  TypInfo;
```

It is correct to start the uses clause on the next line, as in the previous example, or you may start the list of units on the same line:

```
uses Windows, SysUtils, Classes, Graphics, Controls, Forms,  
  TypInfo;
```

You may format the list of units in your uses clause so that they wrap just shy of 80 characters, or so that one unit appears on each line.

### 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 GetData: Integer;
  procedure SetData(Value: Integer);
public
published
  property MyData: Integer read GetData write SetData;
end;

```

[Interfaces](#) 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`.

## 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](#)

### 3.2 Class/Interface Naming

Use `InfixCaps`, as described at the beginning of this section. Begin each type declaration with a capital T:

```
TMyType
```

See also the section on [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.

Delphi is created in California, so we discourage the use of notation, except where required in header translations:

CORRECT

```
FMyString: string;
```

INCORRECT

```
lpstrMyString: string;
```

The exception to the Hungarian notation rule is in enumerated types.

```
TBitBtnKind = (bkCustom, bkOK, bkCancel, bkHelp,  
    bkYes, bkNo, bkClose, bkAbort, bkRetry,  
    bkIgnore, bkAll);
```

In this case the letters bk are inserted before each element of this enumeration. bk stands for ButtonKind.

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 you omit the initial F, since this is not a Field of an object. (see [section 3.3](#)).

### 3.6 Reserved Words

Reserved words and directives should be all lowercase. This can be a bit confusing at times. For instance types such as Integer are just identifiers, and appear with a first cap. Strings, however, are declared with the reserved word string, which should be all lowercase.

### 3.7 Type Declarations

All type declarations should begin with the letter T, and should follow the same capitalization specification laid out in the [beginning](#) of this section, or in the section on [class declarations](#).

## 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:

1. After the copyright block comment, package declaration, and import section.
2. Between class declarations.
3. 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:

1. Between a method name and its opening parenthesis.
2. Before or after a `.`(dot) operator.
3. Between a unary operator and its operand.
4. Between a cast and the expression being cast.
5. After an opening parenthesis or before a closing parenthesis.
6. After an opening square bracket `[` or before a closing square bracket `]`.
7. 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 ] ;
```

### 4.3 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 six 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 80 columns. Lines longer than 80 columns 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.

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;

// CORRECT

if ((X = Y) or (Y = X) or
  (Z = P) or (F = J) then
begin
  S := J;
end;
```

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 );
```

A continuation line should never start with a binary operator.[???] Avoid breaking a line where normally no white space appears, such as between a method name and its opening parenthesis, or between an array name and its opening square bracket. If you must break under these circumstances, then one viable place to begin is after the opening parenthesis that follows a method name. Never place a begin statement on the same line with any other code.

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 "???:" so that they can easily be found afterwards. Ideally, all temporary comments should have been removed by the time a program is ready to be shipped.

Example:

```
// ???: Change this to call Sort when it is fixed
List.MySort;
```

### 5.1 Block Comments

Object Pascal supports two types of block comments. The most commonly used block comment is a pair of curly braces: { }. The Delphi team prefers to keep comments of this type as spare and simple as possible. For instance, you should avoid using asterisks to create patterns or lines inside your comments. Instead, make use of white space to break your comments up, much as you would in a word processing document. The words in your comments should start on the same line as the first curly brace, as shown in this excerpt from DsgnIntf.pas:

```
{ TPropertyEditor

Edits a property of a component, or list of components,
selected into the Object Inspector. The property
editor is created based on the type of the
property being edited as determined by the types
registered by...

etc...

  GetXxxValue
    Gets the value of the first property in the
    Properties property. Calls the appropriate
    TProperty GetXxxValue method to retrieve the
    value.

  SetXxxValue Sets the value of all the properties
    in the Properties property. Calls the appropriate
```

```
TProperty SetXxxxValue methods to set the value. }
```

A block comment is always used for the copyright/ID comment at the beginning of each source file. It is also used to "comment out" several lines of code.

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;
```

A second kind of block comment contains two characters, a parenthesis and an asterisk: (\* \*). This is sometimes called starparen comments. These 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:

```
(* 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:

```
// 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.

## 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:

- Private declarations
- Protected declarations
- Public declarations
- Published declarations

There are *four* access levels for class members in Object Pascal: published, public, protected, and 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. 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 instead of private, and the properties used to access private data should be given protected status.

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

#### 6.1.8 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, or to arrange them in alphabetical order

within the public section.

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)
```

## Interfaces

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

```
InterfaceName = 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:

1. Interface method declarations
2. 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.

Here is a simple statement:

```
A := B;
```

Here is a compound, or structured, statement:

```
begin
  B := C;
  A := B;
end;
```

## 8.0.1 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);
```

## 8.0.1 Compound Statements

Compound Statements always end with a semicolon, unless they immediately precede an end statement, in which case the semicolon is optional.

```
begin
  MyStatement;
  MyNextStatement;
  MyLastStatement // semicolon optional
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, that is, they should start with a capital letter, and have capital letters for the beginning of each embedded word. Do not preface variable names with an F, as that convention is reserved for Fields in a class declaration:

```
var
  MyData: Integer;
  MyString: string;
```

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

```
var
  ArraySize, ArrayCount: 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;
```

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;
```

One that has fallen out of favor but deserves honorable mention:

```
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 := FIncr div FLineDiv;
      FinalIncr := FIncr mod FLineDiv;
      Count := FLineDiv;
    end;
  SB_PAGEUP, SB_PAGEDOWN:
    begin
      Incr := FPageIncr;
      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;
```