# JCL Delphi Language Style Guide

ma

# Table of Contents

# JCL Delphi Language Style Guide

*Version 1.2, Last modified on August 14, 2002 – Matthias Thoma*
*Version 1.1, Last modified on January 10, 2002 – Matthias Thoma*
*Version 1.0, Last modified on September 19, 2001 – Marcel van Brakel*

## Preample

This article documents a standard style for formatting Delphi code which is to be included in the JEDI Code Library (JCL). It is based on the conventions developed by the Delphi team at Borland. This document is a modified/annotated version of the article originally written by Charles Calvert (with his permission to do so). The original article can be obtained from the Borland Community site.

Delphi is a beautifully designed language. One of its great virtues is its readability. These standards are designed to enhance that readability of Delphi 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 a while, 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 Project JEDI, Borland or to a public repository.

As documented elsewhere, it's not necessary for you to format your code according to the rules of this document before submitting it to Project JEDI, although that would be great. Instead the reformatting is done by the JCL unit owners where necessary. We accept feedback in the form of corrections or suggestions. Send your communications to the JCL team. Note that with a few exceptions this document does not include coding guideliness, only formatting guideliness. A separate document for coding guideliness is in the making.

Do not post this specification on other web sites. Instead, simply link to either this version of the document or the original one on Charlie's website.

## Contents

# 1.0 Introduction

This document is not an attempt to define a grammar for the Delphi 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 Delphi 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.

Many of the modifications to this document we're at least partly based on feedback by Mike Lischke. Other people who contributed are Robert Marquardt and Matthias Thoma.

# 2.0 Source Files

Delphi 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

Delphi 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: JclMyFile.pas. This is known as InfixCaps, or Camel Caps. Extensions should be in lower case. All JEDI Code Library source−files must be prefixed with 'Jcl'. Also, since these files are to be ported to Linux be carefull that you use the same capitalization everywhere a

source–file is referenced (Linux filenames are, as opposed to Win32, case–sensitive).

## 2.2 Source–File Organization

For the JCL the following header is used. Replace JclGraphics with the appropriate unit name. The Last Modified date is kept up to date by the unit owner and should always match the last modified date in the filesystem.

```
{*****************************************************************************}
{                                                                           }
{ Project JEDI Code Library (JCL)                                           }
{                                                                           }
{ The contents of this file are subject to the Mozilla Public License Version }
{ 1.0 (the "License"); you may not use this file except in compliance with the }
{ License. You may obtain a copy of the License at http://www.mozilla.org/MPL/ }
{                                                                           }
{ Software distributed under the License is distributed on an "AS IS" basis, }
{ WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for }
{ the specific language governing rights and limitations under the License. }
{                                                                           }
{ The Original Code is JclGraphics.pas.                                     }
{                                                                           }
{ The Initial Developer of the Original Code is documented in the accompanying }
{ help file JCL.chm. Portions created by these individuals are Copyright (C) }
{ 2000 of these individuals.                                                }
{                                                                           }
{ Contains various graphics related classes and subroutines such as a Win32 }
{ regions encapsulatiion, a very fast TBitmap replacement and various        }
{ transformation and filtering routines.                                    }
{                                                                           }
{ Unit owner:    Wim de Cleen                                               }
{ Last modified: June 7, 2000                                               }
{                                                                           }
{*****************************************************************************}

unit Buttons;
```

Compiler directives are not directly included in the source files. Instead a global JCL.INC include file is used which defines all standard directives. Ex.

```
{$I JCL.INC}

interface
```

If overriding directives are needed they can be included below this include but this must be avoided. If overriding directives are needed these must be documented. You should strive to override directives only a local scope. For example, for a single procedure.

```
{$S-,W-,R-}
{$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 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.

### 2.2.2 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;
```

The uses clause is always started on the next line and units are written down one after another, wrapping at 80 columns. 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;
```

Furthermore, you should separate the standard Delphi units, JCL units and Platform dependent units. Finally, it is prefered to list to units in alpabetical order unless the order is important (this should never be the case but sometimes is, an example is the Windows unit which – by convention – should always be listed first). An example follows (comments shouldn't be included):

```
uses
  {$IFDEF WIN32}
  Windows, ActiveX,    // Windows units
  {$ENDIF}
  {$IFDEF LINUX}
  ..Linux specific units go here
  {$ENDIF}
  Math, SysUtils,      // Standard Delphi platform independent units
  JclBase, JclStrings; // JCL units
```

### 2.2.3 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 Delphi source. Example:

```
TMyClass
```

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

```
TMyClass = class (TOject)
```

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;
```

JCL classes always have their ancestor explicitly declared, even for TObject. The class keyword and ancestor should be separated with a space:

```
TJclMyClass = class (TObject)
...
```

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.

## 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)

It seems to be unavoidable but every now and then someone suggests using hungarian, or a similar, notation for identifier naming. Although good arguments can be provided in favor of hungarian notation, at least as many arguments can be given against it. The JCL will not use hungarian notation, ever! There, that's out of the way. Identifiers in the JCL should be named as the examples above, names which describe the purpose of the identifier not what type they happen to be of.

### 3.1 Unit Naming

Use InfixCaps, as described at the beginning of this section. See also the section on unit declarations

As described earlier, all JCL units should have the "Jcl" prefix.

### 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.

All JCL classes are prefixed with 'TJcl' not just a capital T. Types which are used only internally don't have to include the 'Jcl' prefix although you should be carefull with the naming when they are declared in the interface section.

### 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 hungarian 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.

Looping variables are by convention named I (capital i) and J. Other commonly used single character identifier names are: S (string) and R (Result). Single letter variables/field names should always be capitals but other then the ones mentioned above you should avoid them and use more meaningful names.

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)
>> ServerRunning (verb phrase, but not imperative)

A note about the last example (ServerRunning): 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)?

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**

Delphi 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 6 spaces, etc. Never use tab characters.

There are few exceptions. The reserved words unit, uses, 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;


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;

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


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;

    while (LongExpression1 or LongExpression2) do
    begin
      // DoSomething
      // DoSomethingElse;
    end;

    if (LongExpression1) or
       (LongExpression2) or
       (LongExpression3) then
```

# 5.0 Comments

The Delphi 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. 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

Delphi 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;


        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:

```
    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 Delphi: 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:

```
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.

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, even if it is syntactically not required.

```
begin
  MyStatement;
  MyNextStatement;
  MyLastStatement;   // semicolon optional but required by this style guide
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.

You should only declare identifiers on a single line if they are logically related.

### 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:

```
if A < B then DoSomething;

if A < B then
  DoSomething;
```

In the JCL the first example is allowed but discouraged. Use it only in "obvious" situations such as "if ParameterIncorrect then Exit;"

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:

Except for the second one, all these variations are discouraged.

```
    // 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;

    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;
```

Avoid extraneous parentheses when formulating the conditional in an if statement. In other words, don't encapsulate the conditional statement in parenthesis if it's not syntactically required and doesn't provide additional readability. An obvious example:

```
    // CORRECT
    if I > 0 then
      DoSomething;

    // INCORRECT
```

```
      if (I > 0) then
        DoSomething;
```

**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;
```

If the body of the for loop consist of a single statement then both of the examples below are allowed. As with if statements, the first one is discouraged though.

```
      for I := 0 to 10 do DoSomething;

      for I := 0 to 10 do
        DoSomething;
```

**8.2.5 while statement**

Example:

```
      // INCORRECT
      while x < j  do begin
        DoSomething;
        DoSomethingElse;
      end;

      // CORRECT
      while x < j do
      begin
        DoSomething;
        DoSomethingElse;
      end;
```

The same as with for loops applies here. Both of the following examples are allowed but the first one is discouraged.

```
      while x < j do Something;

      while x < j do
        Something;
```

**8.2.6 repeat until statement**

Example:

```
      // CORRECT
      repeat
        x := j;
        j := UpdateValue;
      until j > 25;
```

**8.2.7 case statement**

Example:

```
      // Discouraged
      case Control.Align of
        alLeft, alNone: NewRange := Max(NewRange, Position);
        alRight: Inc(AlignMargin, Control.Width);
      end;

      // Discouraged
      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;
```

In the JCL only this last variation is correct. That is, each case label starts on a separate line with a two column indent relative to the case statement and the body of that particular case follows on the next line with an additional two column indent. If the body on a particular case consists of only a single statement then the begin...end pair can be omitted. In this case the statement is aligned using a two column indent relative to the case statement (identical to the begin reserved word in the example above).

**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;
```

# 9.0 JCL Additions

Below are some more additonal rules and conventions which should be followed for all JCL sourcecode files.

**9.1 Const, Var and Type**

The reserved words var, const and type always appear alone on a line. These example are correct:

```
type
  TMyType = Integer;

const
  MyConstant = 100;

var
  MyVar: Integer;
```

But these are not:

```
type TMyType = Integer;

const MyConstant = 100;

var MyVar: Integer;
```

Additionally a procedure should only have a single type, const and var section and, if possible, in that order. For example:

```
procedure SomeProcedure;
type
  TMyType = Integer;
const
  ArraySize = 100;
var
  MyArray: array [1..ArraySize] of TMyType;
begin
  ...
end;
```

**9.2 Conditional compilation**

All JCL units must include the JCL.INC file. This file defines a number of global directives. The include statement should be placed between the unit and interface keywords. As an aside, no one is allowed to modify the JCL.INC file without first consulting the JCL coordinators team (through jcl@delphi–jedi.org). When using any of the directives from this file, or others for that matter, you should always repeat the conditional in the ENDIF directive. For example:

```
{$IFDEF WIN32}
  // conditionally compiled code
{$ENDIF WIN32}
```

This may seem overkill if the conditionally compiled code only extends a few lines but is a tremendous visual aid when the code actually spans multiple pages of code (as is often the case for platform dependent code).

**9.3 Resource strings**

All resourcestrings should be of the format 'Rs'[Category][Name]. [Category] should be (an abbreviation of) the category in which the code resides, [Name] is a descriptive name for the string itself. For example, the TJclCriticalSectionEx CreateEx constructor raises an exception on initialization failure. The exception message is declared as a resourcestring with the name RsSynchInitCriticalSection.

All resourcestrings must be declared in the global JclResources.pas file which is included in each JCL unit. This is to ease translation. Literal strings should be avoided where possible (use a constant whenever you can).

**9.4 Exceptions**

Exceptions are prefixed with 'EJcl' instead of 'TJcl'. All JCL exceptions should be ultimately derived from EJclError which is declared in JclBase.

When raising an exception you should prefer the CreateRes(ResStringRec: PResStringRec) constructor for efficiency. Thus, an exception is raised like this:

```
raise EJclSomeException.CreateRes(@RsSomeResourceString);
```

**9.5 Categories and routine separation**

Typically each JCL unit is a single category. For example, JclSynch contains all kinds of synchronization classes and subroutines. Within a unit there is usually a further categorization, for example JclSynch has a number of 'Locked Integer Manipulation' routines which form a subcategory within this unit. In the interface section each subcategory is divided using two 80 column width lines in between which there is a one line description of the subcategory. For example:

```
function LockedAdd(var Target: Integer; Value: Integer): Integer;
function LockedCompareExchange(var Target: Integer; Exch, Comp: Integer): Integer;
```

In the implementation section this separation is identical except that the lines are composed using the equals character (=).

```
function LockedAdd(var Target: Integer; Value: Integer): Integer;
asm
  MOV     ECX, EAX
```

In the implementation section each routine or method is separated from its predecessor using a 80 column width line composed of minus characters (–).

```
    if (L > 0) and (Path[L]  PathSeparator) then Result := Path + PathSeparator;
  end;


  function PathAddExtension(const Path, Extension: string): string;
  begin
    Result := Path;
    if (Path  '') and (ExtractFileExt(Path) = '') and (Extension  '') then
    begin
      if Extension[1] = '.' then
        Result := Result + Extension
      else
        Result := Result + '.' + Extension;
    end;
  end;


  function PathAppend(const Path, Append: string): string;
  var
    PathLength: Integer;
    B1, B2: Boolean;
  begin
    if Append = '' then
      Result := Path
```

## 9.6 Assembler

Assembler is formatted like this:

```
        REP     MOVSW
        JMP     @@2
@@1:
        LEA     ESI, [ESI + 2 * ECX – 2]
        LEA     EDI, [EDI + 2 * ECX – 2]
```

That is, the opcode is indented 8 spaces and the operands are aligned on the 16th column. Labels should be indented with two spaces or aligned on the left side and be camel case. All opcodes and registers should be written fully in uppercase. Numeric labels are acceptable but a more descriptive name is preferred. General punctuation formatting still applies, e.g. a single space after each comma and a space on both sides of an operator (such as the addition operator +). Additionally, never put labels and commands on the same line and always prefix labels with the @ character to make the label scope local. Generically speaking, assembler should be avoided but if it is used it should be heavily commented.

## 9.7 Local routines

Local functions should be indented two spaces in their entirety and separated from the procedure declaration and begin statement by a single line. If the 'outer' procedure (SomeProcedure in the example) has local variables these should be declared before the local procedure, regardless of whether the local procedure needs access to them. However, local routines should be avoided. Whenever it seems reasonable to extract code as a local subroutine, think carefully whether the routine can be made more generic and extracted as a normal, global routine (of course should be moved to the appropriate unit as well). For example:

```
  procedure SomeProcedure;
  var
    I: Integer;
```

```
    procedure LocalProcedure;
    begin
      ...
    end;

begin
  ...
  LocalProcedure;
  ...
end;
```

## 9.8 Parameter declarations

When declaring the parameters list of a procedure, function or method, observe the following recommendations:

- Combine formal parameters of same type into one statement
- Usage of A in parameter names is discouraged unless it concerns a method of a class which has as a property which is named identical.

Although technically these are not formatting issues, I'd like to methion them here anyway:

- Ordering of parameters should be: input, input./output and output parameters and within that ordering: most used, least used. Parameters with default values are, as required by the Delphi language rules, always placed at the back of the list.
- Use of const for parameter types is recommended even if it's use does not increases the efficiency of the parameter passing. For example, a parameter of type Integer should be declared with the const modifier if that's the semantic meaning of the parameter in question.

## 9.9 Initialization of global variables

Global variables are, like class members, automatically initialized to 0. This has different meaning for different types. For example, and Integer is initialized to 0 while a pointer is initialized to nil. Because of this, if the global variable needs to be 0 initialized, which is often the case, this should not be done explicitly. Instead you should rely on Delphi to do this for you. This is for efficiency reasons btw because it influences how the variable ends up in the executable and later in memory. If you desire you can add a comment to indicate reliance on 0 initialization like so:

```
var
  MyGlobalVariable: Pointer{ = nil};
```