Cycle Consulting, Inc.

Delphi Coding Standards

Introduction

The goal of this document is to establish a standard coding style for use on all projects at Cycle Consulting, Inc. The intent is to make it so that every programmer on a team and within Cycle Consulting, Inc., can understand the code being written by other programmers. This is accomplished by making the code more readable through consistency and uniformity.

This document does not cover *user interface standards*. This is a separate but equally important topic. Enough third-party books and Microsoft documentation cover such guidelines that we decided not to replicate this information but rather to refer you to the Microsoft Developers Network and other sources where that information may be available.

General Source Code Formatting Rules

Indentation

Indenting will be three (3) spaces per level. You should not save tab characters to source your files. The reason for this is because tab characters are expanded to different widths with different user settings and by different source management utilities (print, archive, version control, etc.).

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

Margins

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

Formatting

Formatting of the source code in a project is critical to the readability and maintainability of the source code. The key formatting points for readable source code are the liberal use of white space (blank spaces and blank lines) and alignment of like items in lines of source code. Therefore, all source code in a project must be formatted using a block style. For example, when multiple local variables are declared, the names should be aligned along with the colon (:) characters, the data type names, and the documentation. Below is an example:

```
var
   oStore  : TDataStore;   // Referece to the data storage
   aLayout : TLayoutArray; // Structure array for the data storage
```

Assignments to variables should be handled similarly where the assignment operators (:=) are aligned. Below is an example:

```
aLayout[ 00, 00 ] := 'String Data';
aLayout[ 00, 01 ] := 'S';
aLayout[ 01, 00 ] := 'Integer Data';
```

```
aLayout[ 01, 01 ] := 'I';
aLayout[ 02, 00 ] := 'Float Data';
aLayout[ 02, 01 ] := 'F';
aLayout[ 03, 00 ] := 'Currency Data';
aLayout[ 03, 01 ] := '$';
aLayout[ 04, 00 ] := 'Data Data';
aLayout[ 04, 01 ] := 'D';
aLayout[ 05, 00 ] := 'Boolean Data';
aLayout[ 05, 01 ] := 'B';
aLayout[ 06, 00 ] := 'SubItem String 1';
aLayout[ 06, 01 ] := 'S';
aLayout[ 07, 00 ] := 'SubItem String 1';
aLayout[ 07, 01 ] := 'S';
aLayout[ 08, 00 ] := 'SubItem String 1';
aLayout[ 08, 01 ] := 'S';
aLayout[ 09, 00 ] := 'SubItem String 1';
aLayout[ 09, 01 ] := 'S';
```

Along with the alignment of lines of code in a block style, the source code should make liberal use of blank lines. This allows the source code to contain documentation without the comments running into subsequent lines as well as making the source code easier to read. Below is an example:

```
// If the current layout item does NOT contain the word MEMO...
if ( Pos( 'MEMO', sTest ) = 00 ) then begin

   // Add a new sub-item and return its position in the list.
   nNode := self.AddNode( aStruct[ i, 00 ] );

   // Recursively call this method to add
   // the items to the new node.
   self.SetStructure( aStruct, nItem + 01 )

end

else begin

   // Add a new memo item to the list.
   nNode := self.AddMemo( aStruct[ i, 00 ], '', '' );

   // Clear any items from the contained stringlist.
   TMemoStoreItem( self.Items[ nNode ] ).Value.Clear;

end;
```

In addition to the use of blank lines, all operators (i.e. +, -, *, /, div, =, :=, :, etc.) excluding commas and semi-colons should always be preceded and succeeded by a blank space.

Comments

Delphi provides three types of comment characters that may be used to write source code documentation. The first is the double forward slash "//" which can be used for single line comments. The second is the open curly brace, close curly brace pair "{}", which can be used for multiple line comments. Third, the developer can use the parenthesis asterisk, asterisk parenthesis pair "(**)" for multiple line comments as well.

At Cycle Consulting, Inc., we will use the double forward slash "//" for single line comments and the open curly brace, close curly brace pair "{}" for multiple line comments. The parenthesis asterisk, asterisk

parenthesis pair "(**)" will be reserved for temporarily removing code or commenting out a block of code during development.

Documentation

Each paragraph or logical block in the source code must be preceded by a multi-line comment describing the structure and function of the block of source code. Typically, these paragraphs begin with a logical construct (if..then..else, while, for, repeat..until, try..except, try..finally, etc.), and conclude with the keyword, "end". These paragraphs can also contain sub-paragraphs, which maintain this same requirement. Below is an example of logical block documentation:

```
// If the new key code is in the list of fields....
// Since the user has a campaign conflict, we must
// bring up the edit for the new key code on the
// posibility that the new key code is a duiplicate.
if ( nNewKey <> -1 ) then begin

   // Put the possible duplicate value in the
   // edit so that the user can change it.
   oForm.edKey.Text := UpperCase( TLoadField( oRecord[ nNewKey ] ).Data );

   // Enable the new key code edit.
   oForm.edKey.Enabled := true;

end;
```

Structure

Since the entire software industry is moving to a more component oriented architecture to promote code reuse, the projects at Cycle Consulting, Inc., should always be designed with this trend in mind. For instance, the forms in a project should have very short event handlers that are often only one line of code. The event handler should delegate all of the "business" functionality of the application to non-event handling methods of the class or to an instance of a more business related class. Below is an example of an even handler delegating its "business" functionality to a non-event handling method:

```
procedure TfrmMain.miSelectSourceClick(Sender: TObject);
begin

   // Select the source publication that will provide the data that the
   // user will analyze to create a new mailing campaign.  The source
   // tables contain all of the data pertaining to the publication
   // including previous and current mailing campaign data.
   self.SelectSource;

end;


{
   VOID SelectSource()

   Method to enable the application to specify the source data that the
   user will perform analysis on to determine the viability of a new direct
   mail marketing campaign.

   Modifications: 02/08/99 - Created.

}
procedure TfrmMain.SelectSource;
```

```
begin

   ...

end;
```

Notice how the click event handler for the menu simply delegates all of the "business" functionality to another method of the form. This enables greater flexibility in maintenance and enhancement of the business logic of the form without altering code that is explicitly tied to the user interface. It also offers the possibility of moving the business logic to a separate class or component in order to implement an n-tier distributed architecture using the Component Object Model (COM) or the Common Object Request Broker Architecture (CORBA).

Conditional Defines

Conditional defines shall be created with curly braces - "{", "}" - and with the conditional command in uppercase. Each conditional define is named again in the closing block to enhance readability of the code. They shall be indented in the same manner as blocks - for example:

```
if ... then begin

   {$IFDEF VER90}

      raise Exception.CreateRes( sError );

   {$ELSE}

      raise Exception.Create( sError );

   {$ENDIF VER90}

end;
```

Begin..End Pair

The “begin” keyword must appear on the same line as the statement. For example, the following first line is correct; the second line is incorrect:

```
for i := 0 to 10 do begin // Correct, begin on same line as for

end;

for i := 0 to 10 do       // Incorrect, begin appears on a separate line
begin

end;
```

The “end” keyword always appears on its own line and at the end of the construct that was started with the “begin” keyword. The indentation of the “end” keyword should match the indentation of the statement that started the construct. Below is an example of an “if” statement using the correct style:

```
if some statement = ... then begin // Correct, begin on same line as if

   SomeStatement;

end

else begin                         // Correct, begin on same line as else

   SomeOtherStatement;
```

```
end;
```

Below is an incorrect use of the “begin” and “end” statements:

```
if some statement = ... then // Incorrect, begin appears on a separate line
begin

   SomeStatement;

end

else                         // Incorrect, begin appears on a separate line

begin

   SomeOtherStatement;

end;
```

Object Pascal

Parenthesis

There should always be white space between an open parenthesis and the next character. Likewise, there should always be white space between a close parenthesis and the previous character. The following example illustrates correct and incorrect spacing with regard to parentheses:

```
CallProc( aParameter ); // Correct
CallProc(aParameter);   // Incorrect
```

Liberal use of parentheses in a statement is required. Parentheses should be used to reduce the ambiguity of the source code as well as to assist the compiler, where required, to achieve the intended meaning in source code. The following examples illustrate correct and incorrect usage:

```
if ( i = 42 ) then                    // Correct
if i = 42 then                        // Incorrect
if ( ( i = 42 ) or ( j = 42 ) ) then  // Correct
if ( i = 42 ) or ( j = 42 ) then      // Incorrect
```

Reserved Words and Key Words

Object Pascal language reserved words and key words shall always be completely lowercase. By default, the syntax highlighting feature of the IDE will already print these words in bold face. You shall not use uppercase or mixed case for any of these words.

Types

Capitalization Convention

Type names that are reserved words shall be completely lowercase. All other native type names (integer, char, Boolean, single, double, etc.) shall be completely lower case as well. Win32 API types are generally completely uppercase, and you should follow the convention for a particular type name shown in the Windows.pas or other API unit. For other type names including user defined types, the first letter shall be uppercase and the rest shall be camel-capped (mixed case) for clarity. Here are some examples:

```
var
   sMyString     : string;     // Reserved word
   hWindowHandle : HWND;       // Win32 API type
   i             : integer;    // Type identifier introduced in System unit
```

```
    oList         : TList;      // VCL defined type
    oDataStore    : TDataStore; // User defined type
```

Integer Types

Use of the fundamental integer types (ShortInt, SmallInt, LongInt, Int64, Byte, Word, and LongWord) is discouraged since they result in slower performance for the underlying CPU and operating system than the generic integer types (Integer and Cardinal). Therefore, a project should only use the fundamental types when the physical byte size of the integer variable is significant (such as when using other-language DLLs).

Floating Point Types

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

Enumerated Types

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

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

Variant and OleVariant

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

Structured Types

Array Types

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

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

Record Types

A record type shall be given a name meaningful to its purpose. The type declaration must be prefixed with the character capital "T". If a pointer to the record type is declared, it must be prefixed with the character "P" and declared immediately prior to the type declaration. The type declaration for each element must be aligned in a column to the right, indented one level - for example:

```
type
```

```
PEmployee = ^TEmployee;
TEmployee = record
   sName : string;
   nRate : double;
end;
```

Variables

Variable Naming and Formatting

All variables should be given descriptive names meaningful to their purpose that are prefixed with a character or set of characters that denotes its data type. The prefix for the variable name should always be in lower case. The names should be in mixed case with the first letter after the prefix in uppercase followed by lower case letters. If the name of the variable is made up of more than one distinct word, then the first letter of each word should be capitalized followed by lower case letters. The underscore character should not be used in variable names. Below are the accepted variable prefixes:

Data Type	**Prefix Character**	**Example**
Integer	n	nCounter
Char	c	cLetter
Boolean	b	bContinue
Enumerated	e	eCards
Subrange	sub	subRange
Floating Point	n	nPercent
String	s	sLastName
Array	a	aData
Record	r	rHeader
Set	set	setStatus
Class (Instance of TClass)	cl	clDataStore
Object (Instance of TObject)	o	oDataStore
Pointer	ptr	ptrAccelerators
Variant	v	vExcel
Procedural	proc	procSetScope
File	f	fOutput

Note the fact that some data types share the same descriptive prefix character. For instance, all numeric types are prefixed with the letter “n”. Loop control variables, however, are generally given a single character name such as i, j, or k. It is acceptable to use a more meaningful name as well such as nUserIndex. Boolean variable names must be descriptive enough so that their meanings of “True” and “False” values will be clear.

Declaring Variables

When declaring variables, there shall be no multiple declarations for one type on a single line. Each variable is always assigned a specific type on its own line - for example:

```
var
   nLength : integer; // Correct
   nCount  : integer;

var
   nLength, nCount : integer; // Incorrect
```

It is acceptable to prefix each variable declaration with the var keyword - for example:

```
var nLength : integer;
var nCount  : integer;
```

Documentation

A description of the purpose and/or use of the variable should follow each variable declaration. The description should use the double slash "//" comment marker. Below is an example:

```
var
   nLevel : integer; // Security level for the current user
   sName  : string;  // Name of the current user
```

Local Variables

Local variables used within procedures follow the same usage and naming conventions for all other variables. Temporary variables will be named appropriately. When necessary, initialization of local variables will occur immediately upon entry into the routine. Local AnsiString variables are automatically initialized to an empty string, local interface and dispinterface type variables are automatically initialized to nil, and local Variant and OleVariant type variables are automatically initialized to Unassigned.

Use of Global Variables

Use of global variables is discouraged, however, they may be used when necessary. When this is the case, you are encouraged to keep global variables within the context where they are used. For example, a global variable may be global only within the scope of a single unit's implementation section. Global data that is intended to be used by a number of units shall be moved into a common unit used by all units in the project. Global data may be initialized with a value directly in the var section. Bear in mind that all global data is automatically zero-initialized, so do not initialize global variables to "empty" values such as 0, nil, '', Unassigned, and so on. One reason for this is because zero-initialized global data occupies no space in the exe file. Zero-initialized data is stored in a 'virtual' data segment that is allocated only in memory when the application starts up. Non-zero initialized global data occupies space in the exe file on disk. To explicitly document the assumption that global variables are zero-initialized, a comment to make this clear should be added - for example:

```
var
   nScope : integer { = 00 }; // Scope indicator for the application
```

Constants

Constant Naming and Formatting

The naming convention for constants is very similar to that for variables with the exception that the prefix should describe the function or usage of a set of constants. For example, if a set of constants designates the type of messages that a message broadcasting system can send, the prefix could be "bm" for "broadcast message". Below is an example of these constants:

```
const
   bmRefresh    = 00; // Refresh the result set
   bmRefreshAll = 01; // Refresh the result set
   bmRead       = 02; // Read the data from the result set
   bmWrite      = 03; // Write the data to the result set
   bmWriteRead  = 04; // Write and read the data for the result set
   bmOpen       = 05; // Open the data set
   bmClose      = 06; // Close the data set
   bmCloseOpen  = 07; // Close and then open the result set
   bmSpecial    = 08; // Special type of message
```

As noted in the section on documenting variables, a comment describing the function and usage of the constant should follow each constant declaration.

Typed Constants

Typed constants serve a very useful purpose in that they function as pre-initialized variables that have a global lifetime. Since they are not really constants in any sense of the word, they should follow the naming and formatting rules for variables instead of constants.

Procedures and Functions (Routines)

Naming / Formatting

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

```
procedure thisisapoorlyformattedroutinename;
```

This is an example of an appropriately capitalized routine name:

```
procedure ThisIsMuchMoreReadableRoutineName;
```

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

```
procedure FormatHardDrive;
```

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

```
procedure SetUserName;
```

Routines that retrieve a value shall be prefixed with the word “Get” - for example,

```
function GetUserName : string;
```

Formal Parameters

Formatting

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

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

Naming

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

```
procedure SomeProc( sUserName : string; nUserAge : integer );
```

Ordering of Parameters

Formal parameter ordering emphasizes taking advantage of register calling conventions. Most frequently used (by the caller) parameters shall be in the first parameter slots. Less frequently used parameters shall be listed after that in left to right order. Input lists shall exist before output lists in left to right order. Place most generic parameters before most specific parameters in left to right order. For example:

```
procedure Stuff( sPlanet, sContinent, sCountry, sState, sCity : string );
```

Exceptions to the ordering rule are possible, such as in the case of event handlers, when a parameter named Sender of type TObject is often passed as the first parameter.

Constant Parameters

When parameters of record, array, string, or interface type are unmodified by a routine, the formal parameters for that routine shall mark the parameter as const. This ensures that the compiler will generate code to pass these unmodified parameters in the most efficient manner. Parameters of other types may optionally be marked as const if they are unmodified by a routine. Although this will have no effect on efficiency, it provides more information about parameter use to the caller of the routine.

Name Collisions

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

```
SysUtils.FindClose( rFindFile );
```

or

```
Windows.FindClose( hFindFile );
```

Documentation

Use of an informational header is required for all procedures, functions, methods, and so on. A proper routine header must contain the following information:

```
{
   OBJECT  GetFirstItem( aIndex, nItem )

   ARRAY   aIndex - array of tokens to use to find the target item

   NUMERIC nItem  - position of the token to use for the search

   Method to return a reference to an item in the list.  It searches the
   structure recursively to find the specified item.  If the item is found
   the Found property is set so that the calling routine will know that the
   search was successful.

   Modifications: 02/08/99 - Created.
                  02/16/99 - Changed the name from GetItem() to
                             GetFirstItem().  Added test for the length of
                             the  search index array to suspend the search
                             if there are no tokens left to search for.
                             Added support for the Found property.
}
```

The first line of the header contains the prototype of the routine, as it will be called from another routine. Following the prototype, the header should contain the documentation of each parameter in the parameter list. Next, it should contain a paragraph describing the structure and function of the routine. Finally, it should contain a modification list describing each change made to the routine ordered by the date the changes were made. The comments for the modifications can include the name or initials of the author or authors of the modifications as well.

Statements

The if Statement

The most likely case to execute in an if/then/else statement shall be placed in the then clause, with less likely cases residing in the else clause(s). Try to avoid chaining if statements and use case statements instead if at all possible. Do not nest if statements more than five levels deep. Create a clearer approach to the code. Liberal use of parentheses in an if statement is strongly recommended. If multiple conditions are being tested in an if statement, conditions should be arrange from left to right in order of least to most computation intensive. This enables your code to take advantage of short-circuit Boolean evaluation logic built into the compiler. For example, if Condition1 is faster than Condition2 and Condition2 is faster than Condition3, then the if statement should be constructed as follows:

```
if ( ( Condition1 ) and ( Condition2 ) and ( Condition3 ) ) then begin

end;
```

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

```
if ( ( Condition1 ) and
     ( Condition2 ) and
     ( Condition3 ) ) then begin

end;
```

Reading top-to-bottom usually is easier than reading left-to-right, especially when dealing with long, complex constructs. When a part of an if statement extends beyond a single line, a begin/end pair shall be used to group these lines. This rule shall also apply when only a comment line is present or when a single statement is spread over multiple lines. The else clause shall always be aligned with the corresponding if clause.

Do not test Boolean variables in the condition against a value of "true" or "false". This is unnecessary since the if/then/else statement requires a Boolean value, which is already in the Boolean variable. Creating a boolean expression using a boolean variable and a boolean literal ("true" or "false") is invalid and will cause the compiler to generate less efficient code - for example:

```
var
   bFlag : boolean; // Boolean flag for test

begin

   // Assign a value to the flag.
   bFlag := true;

   // If the flag is true...
   if ( bFlag = true ) then begin // Incorrect - unnecessary expression

      ...

   end;

   // If the flag is true...
   if ( bFlag ) then begin        // Correct - boolean variable test

      ...

   end;

   // Assign a value to the flag.
   bFlag := false;

   // If the flag is false...
   if ( bFlag = false ) then begin // Incorrect - unnecessary expression

      ...

   end;

   // If the flag is true...
   if ( not bFlag ) then begin     // Correct - boolean variable test

      ...

   end;
```

```
end;
```

This rule applies to all statements or logical constructs that test a Boolean value or expression (i.e. while, repeat..until, etc.). This will enable the compiler to generate the most efficient code as well as making the source code of the project easier to analyze and enhance especially when a particular Boolean expression is complex.

The case Statement

General Topics

The individual cases in a case statement should be ordered by the case constant either numerically or alphabetically. If you use a user-defined type, order the individual statements according to the order of the declaration of the type. In some situations it may be advisable to order the case statements to match their importance or frequency of hit. The actions statements of each case should be kept simple and generally not exceed four to five lines of code. If the actions are more complex, the code should be placed in a separate procedure or function. Local procedures and functions are well suited for this.

The else clause of a case statement should be used only for legitimate defaults. It should always be used to detect errors and document assumptions, for instance by raising an exception in the else clause. All separate parts of the case statement have to be indented. All condition statements shall be written in begin..end blocks. The else clause aligns with the case statement - for example:

```
case ( Condition ) of

   condition : begin
         ...
   end;

else // case
    ...
end;
```

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

Formatting

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

The while Statement

The use of the Exit procedure to exit a while loop is discouraged; when possible, you should exit the loop using only the loop condition. All initialization code for a while loop should occur directly before entering the while loop and should not be separated by other non-related statements. Any ending housekeeping shall be done immediately following the loop.

The for Statement

For statements should be used in place of while statements when the code must execute for a known number of increments. In those cases, where stepping is needed, use a while statement that starts from the known end of the loop down to the start condition - for example:

```
nList := oList.Count - 01;

while ( nList => 00 ) do begin

   nList := nList - 02;

end;
```

The repeat Statement

Repeat statements are similar to while loops and should follow the same general guidelines.

The with Statement

General Topics

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

```
with ( rRecord1, rRecord2 ) do begin

end;
```

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

Formatting

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

Structured Exception Handling

General Topics

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

Use of try..finally

Where possible, each allocation will be matched with a try..finally construct. For example, the following code could lead to possible bugs:

```
oSomeClass1 := TSomeClass.Create;
oSomeClass2 := TSomeClass.Create;

try

   { do some code }

finally

   oSomeClass1.Free;
   oSomeClass2.Free;

 end;
```

A safer approach to the above allocation would be:

```
oSomeClass1 := TSomeClass.Create;
try

   oSomeClass2 := TSomeClass.Create;

   try

      { do some code }
```

```
    finally
       oSomeClass2.Free;
     end;
finally
    oSomeClass1.Free;
end;
```

Use of try..except

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

Use of try..except..else

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

Classes

Naming/Formatting

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

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

Instance names for classes will generally match the type name of the class without the capital "T" prefix. The variable for the class instance should be prefixed with the proper variable type prefix, which in this case is "o" for object - for example:

```
var
    oCustomer : TCustomer;
```

Note: See the section on User-defined Components for further information on naming components.

Instance Variables (Fields)

Naming /Formatting

Instance variable names follow the same naming conventions as variable identifiers.

Visibility

All instance variables should be private. An instance variable that is accessible outside the class scope will be made accessible through the use of a property.

Declaration

Each instance variable shall be declared with a separate type on a separate line - for example:

```
TNewClass = class( TObject )
```

```
   private

      nField1 : integer; // Instance variable for some stuff
      nField2 : integer; // Instance variable for some other stuff

end;
```

Usage

Instance variables referenced within methods of the class should always be prefixed with the identifier, "self" to avoid ambiguity between instance variables, local variables, and formal parameters. For example:

```
constructor TDataStore.Create( sStart, sFinish, sDelim : string );
begin

   // Execute the ancestor class constructor.
   inherited Create;

   // Initialize the instance variables.
   self.Start  := sStart;
   self.Finish := sFinish;
   self.sDelim := sDelim;
   self.bFound := false;

end;
```

Methods

Naming /Formatting

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

Use of Static Methods

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

Use of virtual/dynamic Methods

Use virtual methods when you intend for a method to be overridden by descendant classes. Dynamic should only be used on classes to which there will be there will be many descendants (direct or indirect). For example, a class containing one infrequently overridden method and 100 descendent classes should make that method dynamic to reduce the memory use by the 100 descendent classes. It is not guaranteed, though, that making a method dynamic instead of virtual will reduce the memory requirements. Additionally, the benefits from using dynamic in terms of resource consumption are so negligible that it is possible to say:

Always make methods virtual, and only under *exceptional* circumstances dynamic.

Use of Abstract Methods

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

Property Access Methods

All access methods must appear in the private or protected sections of the class definition. Property access methods naming conventions follow the same rules as for procedures and functions. The read access method (reader method) must be prefixed with the word Get. The write access method (writer method) must be prefixed with the word Set. The parameter for the writer method will have the name, "New", prefixed with the proper variable type prefix, and its type will be that of the property it represents - for example:

```
TSomeClass = class(TObject)

   private

      nSomeField : integer; // Some value used for something useful

   protected

      // Property access method to return the field value.
      function GetSomeField : integer;

      // Property assign method to alter the field value.
      procedure SetSomeField( nNew : integer );

  public

      // Access/Assign for the "some field" data.
      property SomeField : integer read GetSomeField write SetSomeField;

end;
```

Properties

Naming / Formatting

Properties that serve as access to private instance variables will be named the same as the instance variables they represent without the variable type prefix. Property names shall be nouns, not verbs. Properties represent data and methods represent actions. Array property names shall be plural. Normal property names shall be singular.

Use of Access Methods

Although not required, it is encouraged to use at a minimum a write access method for properties that represent a private field. This insures that assignments to the private data of a class are always done through a well-defined functional interface reducing the possibility of assignment errors. It also simplifies extending the assignment behavior for the private data of the class.

Documentation

Use of an informational header is required for all classes. A proper class header must contain the following information:

```
{
   TDataStoreItem is an abstract implementation of a dynamic data
   storage class.  This class should NOT be instantiated since it has no
   facility for storing any data.  This class is inherited by type
   specific data storage classes that are able to store data.  These
   type specific classes provide the data storage services to the
   TDataStore class.
```

```
    Modifications: 02/08/99 - Created.
 }
```

First, it contains a description of the design and use of the class. In addition it should contain a modification list describing each change made to the class ordered by the date the changes were made. The comments for the modifications can include the name or initials of the author or authors of the modifications as well.

Files

Project Files

Project files will be given descriptive names. For example, the project, “A Total List Analysis System (ATLAS)” is given the project name: Atlas.dpr. A system information program will be given a name like SystemInformation.dpr. **Note: Use of long file names for every file in a project is highly encouraged.**

Form Files

All unit files that Delphi generates for a form should be given a name descriptive of the form's purpose that ends with the suffix, "FormUnit". For instance, if a form performs "Find and Replace" functionality, then the unit file should be named, "FindReplaceFormUnit.pas" and the form file should be named, "FindReplaceFormUnit.dfm". The Main Form will have the unit file name, “MainFormUnit.pas” and a form file name, “MainFormUnit.dfm”.

Data Module Files

A data module will be given a name that is descriptive of the data module's purpose. The name will end with the suffix, “DataModuleUnit”. For example, the Customers data module will have a form file name of “CustomersDataModuleUnit.dfm” and a unit file name of “CustomersDataModuleUnit.pas”.

Remote Data Module Files

A remote data module will be given a name that is descriptive of the remote data module's purpose. The name will end with the suffix, “RemoteDataModuleUnit”. For example, the Customers remote data module will have a form file name of “CustomersRemoteDataModuleUnit.dfm” and a unit file name of “CustomersRemoteDataModuleUnit.pas”.

Unit Files

Unit Name

If a unit is not based on a form, it should be given a descriptive name that ends in the suffix "Unit". For instance, if a unit contains a class that handles data storage, it should be named, "DataStoreUnit.pas" or "DataStoreClassUnit.pas".

Uses Clauses

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

Interface Section

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

Implementation Section

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

Initialization Section

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

Finalization Section

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

Form Units

A unit file for a form will be given the same name as its corresponding form file. For example, the About Form will have a unit file name of “AboutFormUnit.pas”. The Main Form will have the unit file name of “MainFormUnit.pas”.

Data Module Units

Unit files for data modules will be given the same names as their corresponding form files. For example the Customers data module unit will have a unit file name of “CustomersDataModuleUnit.pas”.

General Purpose Units

A general-purpose unit will be given a name meaningful to the unit's purpose. For example, a utilities unit will be given a name of “BugUtilitiesUnit.pas”. A unit containing global variables will be given the name of “CustomerGlobalsUnit.pas”. Keep in mind that unit names must be unique across all packages used by a project. Generic or common unit names are not recommended.

Component Units

Component units will be placed in a separate directory to distinguish them as units defining components or sets of components. They will never be placed in the same directory as the project. The unit name must be meaningful to its content. Note: See the section on User-defined Components for further information on component naming standards.

File Headers

Use of an informational file header is required for all source files, project files, units, and so on. A proper file header must contain the following information:

```
{
   DataStoreUnit contains the definition for a dynamic data storage
   mechanism defined in the TDataStore class.  This class inherits from
   the stock TList class and uses the TDataStoreItem subclasses to
   store the actual data.  It is also able to read data from and write
   data to a text file.  This text file must be in a specific format.
```

```
          Author: Mark D. Lincoln

       Copyright: (c) 1999 by Cycle Consulting, Inc.
                  All Rights Reserved.

   Modifications: 02/08/99 - Created.
                  02/16/99 – Converted type specific “Add” methods to
                             an overloaded Add() method for greater
                             flexibility.
}
```

The first item in the header is a paragraph containing the description of the structure and function of the file. The author item should contain the name of the original author of the file. The modification list should contain all of the changes that have been made to the file organized by the date that the change was made. The comments for the modifications can include the name or initials of the author or authors of the modifications as well.

Forms and Data Modules

Forms

Form Type Naming Standard

Form types will be given names descriptive of the form's purpose. The type definition will be prefixed with a capital “T” along with the component specifier “frm”. A descriptive name will follow the prefix. For example, the type name for the About Form will be:

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

The main form definition will be:

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

The customer entry form will have a name like:

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

Form Instance Naming Standard

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

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

Auto-creating Forms

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

Modal Form Instantiation Functions

All form units will contain a form instantiation function that will create, set up, show the form modally, and free the form. This function will return the modal result returned by the form. Parameters passed to this

function will follow the "parameter passing" standard specified in this document. This functionality is to be encapsulated in this way to facilitate code reuse and maintenance.

The form variable will be removed from the unit and declared locally in the form instantiation function. Note that this will require that the form be removed from the auto-create list in the Project Options dialog box. See Auto-Creating Forms in this document. For example, the following unit illustrates such a function for a GetUserData form:

```
unit UserDataFromUnit;

interface

uses
   Windows, Messages, SysUtils, Classes, Graphics,
   Controls, Forms, Dialogs, StdCtrls;

type

   TUserDataForm = class( Tform )

      edUserName : TEdit;
      edUserID   : TEdit;

   private
      { Private declarations }
   public
      { Public declarations }
   end;

   // Displays the form modally and returns the user information.
   function GetUserData( var sUserName : string;
                         var nUserID : integer ) : integer;

implementation

{$R *.DFM}

{
   NUMERIC GetUserData( sUserName, nUserID )

   STRING  sUserName – name of the current user

   NUMERIC nUserID   - identifier for the current user

   Function to return the user information to the caller in the variable
   parameters.  The return value of the function denotes whether the user
   pressed the okay push button.

   Modifications: 02/08/99 - Created.

}
function GetUserData( var sUserName : string;
                      var nUserID   : integer ) : integer;
var
   frmUserData : TUserDataForm; // Reference to the form

begin
```

```
    // Instantiate the form making the application its owner.
    frmUserData := TUserDataForm.Create( Application );

    try

       // Assign a new caption to the form.
       frmUserData.Caption := 'Getting User Data';

       // Display the form and return the button pressed by the user.
       result := frmUserData.ShowModal;

       // If the user pressed the okay push button…
       if ( result = mrOK ) then begin

          // Get the current user information from the form.
          sUserName := frmUserData.edUserName.Text;
          sUserID   := StrToInt( frmUserData.edUserID.Text );

       end;

    finally

       // Free the resources allocated for the form.
       frmUserData.Free;

    end;

end;

end.
```

Event Handling Procedures

Event handling procedures should be grouped by control and organized within each group in alphabetical order. The groups of event handlers should be ordered so that the form event handlers appear first in the implementation section of the unit followed by all of the other event handlers. For example, all of the form specific event handlers should be grouped together, followed by the menu item event handlers, the button event handlers, and possibly the edit control event handlers. Each group of event handlers should be separated by a dividing comment like the following:

```
{*********************************************************************}
{********************* Form Event Handling Methods *******************}
{*********************************************************************}

procedure TfrmFind.FormCreate(Sender: TObject);
begin
end;

{*********************************************************************}
{***************** Button Event Handling Methods *********************}
{*********************************************************************}

procedure TfrmFind.pbFindClick(Sender: TObject);
begin
end;
```

Data Modules

Data Module Naming Standard

A DataModule type will be given a name descriptive of the data module's purpose. The type definition will be prefixed with a capital “T” along with the component specifier “dm”. A descriptive name will follow the prefix. For example, the type name for the Customer data module would be something like:

```
TdmCustomer = class( TDataModule )
```

The Orders data module would have a name like:

```
TdmOrders = class( TDataModule )
```

Data Module Instance Naming Standard

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

Type Name	*Instance Name*
TdmCustomer	dmCustomer
TdmOrders	dmOrders

Packages

Use of Run Time versus Design Time Packages

Run time packages will contain only units/components required by other components in that package. Other, units containing property/component editors and other design only code shall be placed into a design time package. Registration units will be placed into a design time package.

File Naming Standards

Packages will be named according to the following templates:

```
"iiilibvv.pkg" - design time package
"iiistdvv.pkg" – run time package
```

where the characters "iii" signify a 3-character identifying prefix. This prefix may be used to identify the company, person or any other identifying entity. The characters, "vv", signify a version for the package corresponding to the Delphi version for which the package is intended.

Note that the package name contains either "lib" or "std" to signify it as a runtime or design time package. Where there are design time and run time packages, the files will be named similarly. For example, packages for the Cycle Consulting, Inc., System Utilities for Delphi 4.0 are named as:

```
SysLib40.pkg - design time package
SysStd40.pkg – run time package
```

Components

User-defined Components

Components shall be named similarly to classes as defined in the section entitled "Classes" with the exception that they are given an identifying prefix. This prefix may be used to identify the company, person or any other entity. For example, a clock component written for Cycle Consulting, Inc., would be defined as:

```
TcciClock = class( TComponent )
```
Note that the prefix is in lower case.

Component Units

Component units shall contain only one major component. A major component is any component that appears on the Component Palette. Any ancillary or supporting components or objects may also reside in the same unit for the major component.

Use of Registration Units

The registration procedure for components shall be removed from the component unit and placed in a separate unit. This registration unit shall be used to register any components, property editors, component editors, experts, etc.

Component registering shall be done only in design time packages, therefore the registration unit shall be contained in the design time package and not in the run time package. It is suggested that registration units be named as:

*Xxxx*Reg.pas

where the "Xxxx" shall be a prefix used to identify a company, person or any other entity. For example, the registration unit for the components constructed at Cycle Consulting, Inc., would be named "CciReg.pas".

Component Instance Naming Conventions

All components must be given descriptive names. **No components will be left with their default names assigned by Delphi.** Components will have a lowercase prefix to designate their type similar to the prefixes used for variables. The reasoning behind prefixing component names rather than post-fixing them is to make searching component names in the Object Inspector and Code Explorer easier by component type.

Component Prefixes

The following prefixes will be assigned to the standard components that ship with Delphi. Please add to this list for third-party components as they are added.

Standard Tab

Prefix	Component
mm	TMainMenu
pm	TPopupMenu
mmi	TMainMenuItem
pmi	TPopupMenuItem
lbl	TLabel
ed	TEdit
mem	TMemo
btn	TButton
chk	TCheckBox
rb	TRadioButton
lb	TListBox
cb	TComboBox
scb	TScrollBar
gb	TGroupBox
rg	TRadioGroup
pnl	TPanel

cl	TCommandList

Additional Tab

Prefix	Component
bbtn	TBitBtn
sb	TSpeedButton
me	TMaskEdit
sg	TStringGrid
dg	TDrawGrid
img	TImage
shp	TShape
bvl	TBevel
sbx	TScrollBox
clb	TCheckListbox
spl	TSplitter
stx	TStaticText
cht	TChart

Win32 Tab

Prefix	Component
tbc	TTabControl
pgc	TPageControl
il	TImageList
re	TRichEdit
tbr	TTrackBar
prb	TProgressBar
ud	TUpDown
hk	THotKey
ani	TAnimate
dtp	TDateTimePicker
tv	TTreeView
lv	TListView
hdr	THeaderControl
stb	TStatusBar
tlb	TToolBar
clb	TCoolBar

System Tab

Prefix	Component
tm	TTimer
pb	TPaintBox
mp	TMediaPlayer
olec	TOleContainer
ddcc	TDDEClientConv
ddci	TDDEClientItem
ddsc	TDDEServerConv
ddsi	TDDEServerItem

Internet Tab

Prefix	Component
csk	TClientSocket
ssk	TServerSocket

wbd	TWebDispatcher
pp	TPageProducer
tp	TQueryTableProducer
dstp	TDataSetTableProducer
nmdt	TNMDayTime
nec	TNMEcho
nf	TNMFinger
nftp	TNMFtp
nhttp	TNMHttp
nMsg	TNMMsg
nmsg	TNMMSGServ
nntp	TNMNNTP
npop	TNMPop3
nuup	TNMUUProcessor
smtp	TNMSMTP
nst	TNMStrm
nsts	TNMStrmServ
ntm	TNMTime
nudp	TNMUdp
psk	TPowerSock
ngs	TNMGeneralServer
html	THtml
url	TNMUrl
sml	TSimpleMail

Data Access Tab

Prefix	Component
ds	TDataSource
tbl	TTable
qry	TQuery
sp	TStoredProc
db	TDataBase
ssn	TSession
bm	TBatchMove
usql	TUpdateSQL

Data Controls Tab

Prefix	Component
dbg	TDBGrid
dbn	TDBNavigator
dbt	TDBText
dbe	TDBEdit
dbm	TDBMemo
dbi	TDBImage
dblb	TDBListBox
dbcb	TDBComboBox
dbch	TDBCheckBox
dbrg	TDBRadioGroup
dbll	TDBLookupListBox
dblc	TDBLookupComboBox
dbre	TDBRichEdit
dbcg	TDBCtrlGrid
dbch	TDBChart

Decision Cube Tab

Prefix	Component
dcb	TDecisionCube
dcq	TDecisionQuery
dcs	TDecisionSource
dcp	TDecisionPivot
dcg	TDecisionGrid
dcgr	TDecisionGraph

QReport Tab

Prefix	Component
qr	TQuickReport
qrsd	TQRSubDetail
qrb	TQRBand
qrcb	TQRChildBand
qrg	TQRGroup
qrl	TQRLabel
qrt	TQRText
qre	TQRExpr
qrs	TQRSysData
qrm	TQRMemo
qrrt	TQRRichText
qrdr	TQRDBRichText
qrsh	TQRShape
qri	TQRImage
qrdi	TQRDBMImage
qrcr	TQRCompositeReport
qrp	TQRPreview
qrch	TQRChart

Dialogs Tab

The dialog box components are really forms encapsulated by a component. Therefore, they will follow a convention similar to the form naming convention. The type definition is already defined by the component name. The instance name will be the same as the type instance without the numeric prefix, which is assigned by Delphi. Examples are as follows:

Type	Instance Name
TOpenDialog	OpenDialog
TSaveDialog	SaveDialog
TOpenPictureDialog	OpenPictureDialog
TSavePictureDialog	SavePictureDialog
TFontDialog	FontDialog
TColorDialog	ColorDialog
TPrintDialog	PrintDialog
TPrintSetupDialog	PrinterSetupDialog
TFindDialog	FindDialog
TReplaceDialog	ReplaceDialog

Win31 Tab

Prefix	Component
dbll	TDBLookupList
dblc	TDBLookupCombo
ts	TTabSet

ol	TOutline
tnb	TTabbedNoteBook
nb	TNoteBook
hdr	THeader
flb	TFileListBox
dlb	TDirectoryListBox
dcb	TDriveComboBox
fcb	TFilterComboBox

Samples Tab

Prefix	Component
gg	TGauge
cg	TColorGrid
spb	TSpinButton
spe	TSpinEdit
dol	TDirectoryOutline
cal	TCalendar
ibea	TIBEventAlerter

ActiveX Tab

Prefix	Component
cfx	TChartFX
vsp	TVSSpell
f1b	TF1Book
vtc	TVTChart
grp	TGraph

Midas Tab

Prefix	Component
prv	TProvider
cds	TClientDataSet
qcds	TQueryClientDataSet
dcom	TDCOMConnection
olee	TOleEnterpriseConnection
sck	TSocketConnection
rms	TRemoteServer
mid	TmidasConnection
