

Contents

- 1 Introduction, 1**
- 2 General Formatting Rules, 2**
 - Margins, 2
 - Indentation, 2
 - Spaces, 2
 - Parenthesis, 3
- 3 Statement Formatting, 4**
 - Routines, 4
 - begin..end Pair, 5
 - try..finally and try..except Pairs, 5
 - if Statements, 6
 - case Statements, 6
 - repeat and while Statements, 6
- 4 Naming Rules, 8**
 - Universal Naming Rules, 8
 - Data Types, 9
 - Variables, 9
 - Pointers, 10
 - Fields and Properties, 10
 - Forms, 10
 - Components, 11
 - Units, 11
 - Routines, 11
- 5 Programming Practices, 13**
 - Variables, 13
 - Control Constructs, 13
 - with Statements, 14
 - Exception Handling, 14
 - Class Definitions, 15
 - Length of Routines, 16
 - Comments, 16
- 6 Units, Forms and Components, 17**
 - Unit Sections, 17
 - Form Units, 18
 - Component Units, 19

APPENDIX

A Naming Component Instances, 21

Dialog Box Components, 24

1

Introduction

This document describes coding standards for Delphi programming in Borland's Object Pascal language. With a few minor exceptions, this document follows the standards established by Borland International, as documented in "Borland Delphi 4 Developer's Guide" (SAMS Publishing, 1998).

The purpose of this document is to present a method by which development teams can enforce a consistent style to the coding that they do, so that every programmer on a team can understand the code being written by others. This is accomplished by making the code more readable by use of consistency.

In this document, the term "routine" refers to Pascal procedures, functions and methods.

2

General Formatting Rules

Margins

Margins will be set to 80 characters. In general, source shall not exceed this margin with the exception to finish a word, but this guideline is somewhat flexible. Wherever possible, statements that extend beyond one line should be wrapped after a comma or an operator. When a statement is wrapped, it should be indented two characters from the original statement line.

Indentation

Indenting will be two spaces per level. Tab characters should not be used in source files, because they are expanded to different widths with different users settings and by different source management utilities. Use of tab characters can be disabled 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).

Spaces

Spaces shall not be used between parentheses or square brackets and the code they enclose, as in:

```
Format('%d inches', [Width]);
```

Spaces shall be used after (but not before) commas, and before and after colons, as in:

```
var
  I, J : integer;
```

Spaces shall be used before and after logical operators.

```
if (A < B) and (C >= D) then...
```

Spaces shall be used after, but not before, the assignment operator, as in:

```
IsMine:= true;
```

When declaring record types, spaces may be used to align the type declarations for each element in a distinct column.

```
type
  tEmployee = record
    Name      : string;
    Age       : integer;
    Rate      : double;
  end;
```

Parenthesis

Parentheses should only be used where required to achieve the intended meaning in source code. The following examples illustrate incorrect and correct usage:

```
if (I = 42) then           // extraneous parentheses
if (I = 42) or (J = 42) then // parentheses required
```

3

Statement Formatting

Routines

The use of `exit` to exit a routine prematurely, is discouraged. This technique should only be used if the alternatives would result in less readable code.

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

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

Input lists shall exist before output lists in left to right order.

Parameters should be placed in order of increasing specificity. example:

```
SomeProc(APlanet, AContinent, ACountry, AState, ACity).
```

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.

Use of constant parameters is recommended when:

- Parameters of record, array, `ShortString`, or interface type are unmodified by a routine. This ensures that the compiler will generate code to pass these unmodified parameters in the most efficient manner.
- Parameters of other types are unmodified by a routine. Although this will have no effect on efficiency, it provides more information about parameter use.

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.)

begin..end Pair

`begin..end` pairs are discouraged when their sole purpose is to enclose a single statement.

`begin..end` pairs will be formatted as shown in the examples below:

```
function GetName : string;
begin
end;

for I:= 0 to 10 do begin
end;

while I < 10 do begin
end;

if B then begin
end
else begin
end;

with Rec do begin
end;
```

try..finally and try..except Pairs

`try..finally` and `try..except` pairs will be formatted as shown in the examples below:

```
try
  { some code }
finally
  { clean up }
end;

try
  { some code }
except
  { exception handler }
end;
```

if Statements

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

Where possible, use `case` statements rather than chained `if` statements.

Do not nest `if` statements more than five levels deep. Instead, create a clearer approach to the code.

If multiple conditions are being tested in an `if` statement, conditions should be arranged from left to right in order of least to most computation intensive. This allows 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...
```

case Statements

The individual cases in a `case` statement should be ordered by the case constant either numerically or alphabetically.

The action 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 routine.

The use of the optional `else` clause of a `case` statement should be used only for legitimate defaults or to detect errors.

Indentation in case statements should follow the example below:

```
case I of
  1  : DoOne;
  2  : DoTwo;
  else DoElse;
end;
```

repeat and while Statements

All initialization code for a `repeat` or `while` loop should occur directly before entering the loop, and should not be separated by other non-related statements. Any ending housekeeping shall be done immediately following the loop.

`for` statements should be used in place of `repeat` or `while` statements when the loop will execute a known number of times.

The use of the `break` procedure to exit loops is discouraged, unless using a flag would adversely affect readability.

4

Naming Rules

Universal Naming Rules

All names shall be descriptive of the purpose of the item being named.

Normally, all names will use the singular, except for array declarations, which will use the plural.

Excessive abbreviating of names is to be avoided. A long name is preferable to one whose meaning is not immediately obvious.

Names will be consistent throughout a project's code files, and between code files and any associated documentation.

Names will avoid collisions with reserved words, key words, and other commonly used words (such as the property names of standard Delphi components).

Names shall always begin with a capital letter and be camel-capped for readability, as in

```
procedure thisisapoorlyformattedroutinename; // incorrect
procedure ThisIsMuchMoreReadableRoutineName; // correct
```

except for the following:

- Reserved words, key words and pre-defined type identifiers shall be completely lowercase.
- Win32 API types are generally completely uppercase. The convention used in the Windows.pas or other API unit should be followed.
- It is permitted to have user-defined constant names in uppercase, with the underscore character used between words.

```

var
  MyString      : string; // reserved word
  WindowHandle : HWND;   // Win32 API type

```

Data Types

Type names must include the prefix `T`.

```

type
  TUserID = longint;

```

Enumerated Types

In enumerated types, the listed identifiers must contain a lowercase two to three character prefix that relates it to the original enumerated type name.

```

type
  TSongType = (stRock, stClassical, stCountry, stAlternative,
              stHeavyMetal, stRB);
var
  SongType = TSongType;

```

Variables

Variable instances of a user-defined type will be given the same name as the type, without the `T` prefix, unless there is a reason to give the variable a more specific name.

Loop control variables are generally given a single character name such as `I`, `J`, or `K`. More meaningful names should be used wherever this will aid comprehension.

Booleans

Boolean variable names must make the meaning of `True` and `False` unambiguous. Names must have positive rather than negative connotation, to avoid confusion with double negatives. A prefix such as `Is` or `Has` should be added where this will make the boolean nature of a variable more obvious.

```

if not Invisible then ... // incorrect
if IsVisible then ...    // correct

```

Variants and OleVariants

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, which 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 (like a string var), but `OleVariant` converts all strings to OLE Strings (`WideChar` strings) and are not reference counted: they are always copied.

Pointers

If a pointer is declared, it must be prefixed with the character `P`. Pointers to user-defined types must be declared immediately prior to the type declaration.

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

Fields and Properties

In class definitions, field names must include the prefix `F`.

Properties that serve as accessors to private fields will be named the same as the fields they represent, without the `F` prefix. Property names shall be nouns, not verbs.

Forms

All forms (and their types) will have the default names assigned by Delphi replaced by a descriptive name, including the prefix `f`. Form instances will be named the same as their corresponding types without the `T` prefix.

```
type
  TfMain = class(TForm)
  private
    { private declarations }
  public
    { public declarations }
  end;

var
  fMain : TfMain;
```

Components

All components will have the default name assigned by Delphi replaced by a descriptive name, unless no reference is made to the component in code (e.g. some `TLabel` components).

The names of standard Delphi components will include a lowercase prefix to designate their type, as listed in Appendix A.

User-defined components will have a 3-character lowercase prefix identifying the author, copyright owner or other entity.

```
type
  TgesClock = class(TComponent);
```

Units

Program units, whether or not they are associated with a Delphi form, will have the prefix `u` added to their names and their filenames.

The unit file for a form will be given the same name as its corresponding form file. This is the Delphi default.

Unit names will be unique across all packages used by a project.

Routines

Routine names shall be unique throughout a project.

When it is unavoidable to have two units containing routines with the same name, calls to either routine will be prefixed with the unit name, to avoid dependence on the order of units in the `uses` clause.

```
  SysUtils.FindClose(SR);
  or
  Windows.FindClose(Handle);
```

Routines that cause an action to occur will be prefixed with the action verb, for example:

```
  procedure FormatHardDrive;
```

Routines that set values shall be prefixed with the word `Set`, for example:

```
procedure SetUserName (const Name : string);
```

Routines that retrieve a value shall be prefixed with the word *Get*, for example:

```
function GetUserName : string;
```

Formal Parameters

Formal parameter names will typically be based on the name of the identifier that will be passed to the routine. When the natural name of a parameter collides with another name inside the routine, the parameter's name will have the prefix "A" added:

```
procedure SetUserInfo (AName, ACity : string);
begin
  with UserInfo do begin
    Name:= AName;
    City:= ACity;
  end;
end;
```

5

Programming Practices

Variables

Variables should have as small a scope as possible.

Global Variables

Use of global variables is discouraged. When they are necessary, they should be kept within the context where they are used. For example, a global variable may be global only within the scope of the a single unit's implementation section.

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

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

Floating Point Variables

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. Use `extended` only when more range is required than that offered by `double`. Use `single` only when the physical byte size of the floating point variable is significant (such as when using other-language DLLs).

Control Constructs

Code must be written using structured programming discipline. Routines should have one entry and exit point, and use of `goto`, `break` and `exit` should be avoided. However, these conventions should not be used at the expense of readability.

with Statements

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, as in:

```
with Record1, Record2 do...           // not recommended
```

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

Exception Handling

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` construct 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.

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

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

A safer approach to the above allocation would be:

```
SomeClass1:= tSomeClass.Create
try
  SomeClass2:= tSomeClass.Create;
  try
    { do some code }
  finally
```

```

        SomeClass2.Free;
    end;
finally
    SomeClass1.Free;
end;

```

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.

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

Class Definitions

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

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

Use virtual methods when you intend for a method to be overridden by descendant classes. Dynamic methods should only be used on classes to which 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.

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 procedures and functions. Thus, the read accessor method (reader method) must be prefixed with the word `Get`. The write accessor method (writer method) must be prefixed with the word `Set`. The parameter for the writer method will have the name `Value`, and its type will be that of the property it represents.

```

tSomeClass = class(tObject)
    private
        FSomeField : integer;
    protected
        function GetSomeField : integer;

```

```
    procedure SetSomeField (Value : integer);  
public  
    property SomeField : integer read GetSomeField  
                                write SetSomeField;  
end;
```

Although not required, it is encouraged to use at a minimum a write access method for properties that represent a private field.

Length of Routines

As a goal, a routines should comprise no more than 25 lines of code. This allows the whole routine to be seen on the screen at one time.

No more than 50% of routines shall exceed 60 lines in length; no more than 5% shall exceed 120 lines in length; and none shall exceed 240 lines in length.

Comments

Code that follows the standards and practices in this document will be largely self-documenting, and will require only minimal commenting. Comments are required when the purpose and behavior of code would not be readily apparent to a programmer of normal skill level. Comments should not be used to state the obvious.

Code which is known to have problems or deficiencies should be marked with a comment beginning "***".

6

Units, Forms and Components

Large programs shall be divided into units, with each unit containing the definitions and routines associated with a custom component, a user-defined class, a program form, or an application capability. Custom components should always reside in units separate from other code.

Typically, it is good practice to have the event handlers associated with a form in the form's unit, but to have any significant computational code located in a separate unit. This provides a clean separation between user interface and calculations.

Unit Sections

Use of an informational header is encouraged for all unit files. Where applicable, header information should include:

- The author and owner of the code
- A copyright notice
- The purpose of the unit and how it works
- References to external sources used, such as algorithms
- Files used, and method of access (read, write, modify, append, etc.)
- Date of creation
- Change log

Interface Section

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

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.

Implementation Section

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

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

Initialization Section

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

Finalization Section

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

Form Units

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

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. The form variable will be removed from the unit and declared locally in the form instantiation function. For example, the following unit illustrates such a function for a GetUserData form.

```
unit UserDataFrm;

interface

uses

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

type
  TUserDataForm = class(TForm)
    eUserName : TEdit;
    eUserID   : TEdit;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

function GetUserData (var AUserName : string;
```

```

                                var AUserID : integer) : word;

implementation

{$R *.DFM}

function GetUserData (var AUserName : string;
                      var AUserID : integer) : word;
var
  UserDataForm : TUserDataForm;
begin
  UserDataForm:= TUserDataForm.Create(Application);
  try
    UserDataForm.Caption:= 'Getting User Data';
    Result:= UserDataForm.ShowModal;
    if Result = mrOK then begin
      AUserName:= UserDataForm.eUserName.Text;
      AUserID:= StrToInt(UserDataForm.eUserID.Text);
    end;
  finally
    UserDataForm.Free;
  end;
end;

end.

```

Component Units

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

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 a project.

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, and so on. Component registering shall be done only in the design packages; therefore the registration unit shall be contained in the design package and not in the runtime package.

It is suggested that registration units are named as: XxxReg.pas Where the Xxx shall be a 3-character prefix used to identify a company, person, or any other entity. For example, the registration unit for the components in the Delphi 4 Developer's Guide would be named DdgReg.pas.

Packages

Runtime 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 package. Registration units will be placed into a design package.

Packages will be named according to the following templates:

```
IIILibVV.pkg    design package
IIIStdVV.pkg    runtime package
```

where the characters III signify a 3-character prefix identifying the author, owner, or other entity. The characters VV signify 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 both design and runtime packages, the files will be named identically except for these 3 letters.

Appendix A

Naming Component Instances

Instances of Delphi components will have a lowercase prefix to designate their type, as listed in the table below. Frequently used components have optional shorter prefixes that can be used for conciseness.

Table 1 . Assignment of prefix codes to Delphi components

Component Type	Prefix	Option
Standard Tab		
TMainMenu	mm	
TPopupMenu	pm	
TMainMenuitem	mmi	
TPopupMenuitem	pmi	
TLabel	lbl	l (ell)
TEdit	edt	e
TMemo	mem	m
TButton	btn	b
TCheckBox	chb	x
TRadioButton	rbn	r
TListBox	lb	
TComboBox	cb	
TScrollBar	scb	
TGroupBox	gb	
TRadioGroup	rg	
TPanel	pnl	
TCommandList	cl	
Additional Tab		
TBitBtn	bbtn	bb
TSpeedButton	sb	
TMaskEdit	me	
TStringGrid	sg	
TDrawGrid	dg	
TImage	img	i
TShape	shp	

Table 1 (contd). Assignment of prefix codes to Delphi components

Component Type	Prefix	Option
TBevel	bvl	
TScrollBar	sbx	
TCheckListBox	clb	
TSplitter	spl	
TStaticText	stx	
TChart	cht	
Win32 Tab		
TTabControl	tbc	tc
TPageControl	pgc	pc
TImageList	il	
TRichEdit	re	
TTrackBar	tbr	
TProgressBar	prb	
TUpDown	ud	
THotKey	hk	
TAnimate	ani	
TDateTimePicker	ntp	
TTreeView	tv	
TListView	lv	
THeaderControl	hdr	
TStatusBar	stb	
TToolBar	tib	
TCoolBar	clb	
System Tab		
TTimer	tm	t
TPaintBox	pb	
TMediaPlayer	mp	
TOleContainer	olec	
TDDEClientConv	ddcc	
TDDEClientItem	ddci	
TDDEServerConv	ddsc	
TDDEServerItem	ddsi	
Internet Tab		
tClientSocket	csk	
tServerSocket	ssk	
tWebDispatcher	wbd	
tPageProducer	pp	
tQueryTableProducer	tp	
tDataSetTableProducer	dstp	
tNMDayTime	nmdt	
tNMEcho	nec	
tNMFinger	nf	
tNMFtp	nftp	
tNMHttp	nhttp	
tNMMsg	nMsg	
tNMMSGServ	nmsg	

Table 1 (contd). Assignment of prefix codes to Delphi components

Component Type	Prefix	Option
tNMNTP	nntp	
tNMPop3	npop	
tNMUUProcessor	nuup	
tNMSMTP	smtp	
tNMStrm	nst	
TNMStrmServ	nsts	
TNMTime	ntm	
TNMUdp	nudp	
TPowerSock	psk	
TNMGeneralServer	ngs	
THtml	html	
TNMUrl	url	
TSimpleMail	sml	
Data Access Tab		
TDataSource	ds	
TTable	tbl	
TQuery	qry	
TStoredProc	sp	
TDataBase	db	
TSession	ssn	
TBatchMove	bm	
TUpdateSQL	usql	
Data Controls Tab		
TDBGrid	dbg	
TDBNavigator	dbn	
TDBText	dbt	
TDBEdit	dbe	
TDBMemo	dbm	
TDBImage	dbi	
TDBListBox	dblb	
TDBComboBox	dbcb	
TDBCheckBox	dbch	
TDBRadioGroup	dbrg	
TDBLookupListBox	dbll	
TDBLookupComboBox	dblc	
TDBRichEdit	dbre	
TDBCtrlGrid	dbcg	
TDBChart	dbch	
QReport Tab		
TQuickReport	qr	
TQRSubDetail	qrzd	
TQRBand	qrb	
TQRChildBand	qrcb	
TQRGroup	qrg	
TQRLabel	qrl	
TQRText	qrt	

Table 1 (contd). Assignment of prefix codes to Delphi components

Component Type	Prefix	Option
TQRExpr	qre	
TQRSysData	qrs	
TQRMemo	qrm	
TQRRichText	qrrt	
TQRDBRichText	qrdr	
TQRShape	qrsh	
TQRImage	qri	
TQRDBMImage	qrdbi	
TQRCompositeReport	qrcr	
TQRPreview	qrp	
TQRChart	qrch	
Win31 Tab		
TDBLookupList	dbll	
TDBLookupCombo	dblc	
TTabSet	ts	
TOutline	ol	
TTabbedNoteBook	tnb	
TNoteBook	nb	
THeader	hdr	
TFileListBox	flb	
TDirectoryListBox	dlb	
TDriveComboBox	dcb	
TFilterComboBox	fcbl	
Samples Tab		
TGauge	gg	
TColorGrid	cg	
TSpinButton	spb	
TSpinEdit	spe	se
TDirectoryOutline	dol	
TCalendar	cal	
TIBEventAlerter	ibea	
ActiveX Tab		
TChartFX	cfx	
TVSSpell	vsp	
TF1Book	f1b	
TVTChart	vtc	
TGraph	grp	

Dialog Box Components

The dialog box components (on the Dialogs tab) 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:

Table 2 . Naming for Standard Dialog Components

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