


## Issue #2 - April 1st, 1995

By Robert Vivrette - CIS: 76416,1373

Well, it would appear that *The Unofficial Newsletter of Delphi Users* is a success! I received a number of very favorable responses from many of you, and no one said they hated it, so I guess you get another issue... If you missed the first issue, I suggest you obtain a copy from the **Delphi-IDE** section of the **Delphi** forum. The filename is **DN0315.ZIP**.

I am going to make a real effort to not duplicate information, or to discuss material I have mentioned in previous issues of the newsletter. What I am going to do is to link the topics in the current issues with articles in the previous issues. Whenever you see a graphic like this , it indicates a link to a previous issue of this newsletter. Clicking on such a graphic will move you to the linked issue (the one shown here is not linked, so don't wear out your mouse clicking on it!). Once there, you can use the **Back** command (Alt-B) to get you back to the current issue. The **History** button will also allow you to get back to the current issue. If you want to navigate around that previous issue, simply click the **Contents** button, and you will be taken to the front page of that issue. Starting with issue #2 I will include a banner on all pages so that you can immediately identify the issue you are viewing. In order for this linking scheme to work, all issues of the newsletter must be kept in the same directory.

Well, let's get to it... We have a lot of material to cover!

[Free Lear Jet - Click Here!](#)

[Books On The Way](#)

[Cooking Up Components](#)

[Making A Splash](#)

[When Things Go Wrong](#)

## The Unofficial Newsletter of Delphi Users - Issue #2 - April 1st, 1995



### Hah! You Fell For It!

I just wanted to make sure you didn't pass this by. Sorry for being so sneaky...

I really want to make this newsletter a cooperative effort. If its content is left up to just me, I will dry up pretty soon and blow away. Therefore, I would like everyone be a part of this newsletter. If you have something that you think might interest others, please pass it on to me. The kinds of things I am looking for are:

1. Delphi Tips and Tricks
2. Code samples; Simple Applications
3. Custom Components (or just ideas for components)
4. Bug Reports
5. Reviews of Component Libraries
6. Reviews of Books/Publications

Don't worry if you think that your contribution is too obvious or not interesting enough. I have corresponded with a number of people who just got Delphi, but have little or no experience in programming. Techniques or code samples that others might take for granted would really be appreciated by those trying to get up to speed with the language. The important things are that the material is clear, easy to understand, and useful.

## The Unofficial Newsletter of Delphi Users - Issue #2 - April 1st, 1995



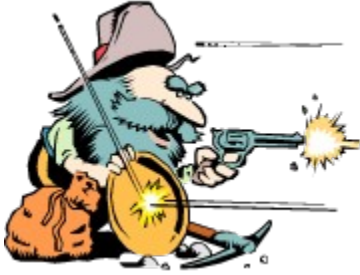
### Books On The Way

For the time being, many of us are patiently waiting for 3rd-party publications supporting Delphi. Each week (okay, sometimes more frequently than that), I go down to Stacy's (a large bookstore in San Francisco on Market Street) and annoy the nice people who work there with Delphi questions. I know that a lot of you have been doing the same, because they immediately know what I am talking about... (or, maybe they just recognize me...)

Anyway, at last count there is at least a dozen books that are "on their way". Add to these the half-dozen magazines and technical newsletters that are in the works, and I think you will see that we are going to have some decisions to make pretty soon.

I would like to hear from those of you who have noticed a new book or magazine arrival, or who have obtained such. Tell me what it covers or what the assumed user experience level the publication is aimed at. Or, better yet, write up a brief review of the book and send it to me. This will allow me to include your comments in the next issue of this newsletter, and therefore make it possible for everyone to make more informed purchasing decisions

## The Unofficial Newsletter of Delphi Users - Issue #2 - April 1st, 1995



### When Things Go Wrong...

Sometimes, things do go wrong. Fortunately the few problems/bugs I have found are easy to get around. I even had to dig a bit to get the few bug reports that are mentioned in this issue. I suppose that for this section, *"No News is Good News!"*

[Problem with CurrEdit in last issue](#)

[Linking Lockup Revisited](#)

[Return Value of the \*\*ExtractFileExt\*\* Function](#)

[Minimizing Your Children](#)

## The Unofficial Newsletter of Delphi Users - Issue #2 - April 1st, 1995

### Problem With CurrEdit

Some of you may have noticed a rather sporadic problem with my Currency Edit field included in the last issue . Under some circumstances, the field would look like it has no text in it and would not accept the input focus. I tracked this down to a problem with the **AutoSize** property in edit boxes. If the box is not big enough vertically for the text to display, then Windows just doesn't let you see it at all. When I unchecked **AutoSize**, the problem went away which leads me to believe that **AutoSize** can sometimes resize the field too small for the field data to display at runtime (oddly, it seems to look OK at design time...)

I have also not ruled out the possibility that the problem is with the font, since it works correctly with about 99% of the typefaces I have.

## Making A Splash

*Contributed by Boris Bosnjak*

A splash screen is a little addition that can make a big impression on the users of your prized application. The following is a summary of steps required:

1. Create a Splash Form, customizing it for special display
2. Modify the Project Source to display the Splash Form before anything else
3. Add code to the Main Form to hide the Splash Form
4. Additional notes.

## The Unofficial Newsletter of Delphi Users - Issue #2 - April 1st, 1995

### Step 1: Create a splash form

Create a small form as you would normally. I recommend that you first add a panel with the Align property set to **alClient**. Make sure the panel has a bevel of 2 or 3 and has a border style of **bsSingle**. This gives the splash form an attractive beveled border. Then you can include a nifty graphic, the title, version, and copyright information, and anything else you care to add.

Set the following form properties:

BorderIcons	<code>[]</code>
BorderStyle	<code>bsNone</code>
Caption	<code>&lt;blank&gt;</code>
FormStyle	<code>bsStayOnTop</code>
Name	<code>SplashForm</code>
Position	<code>poScreenCenter</code>
Visible	<code>False</code>

Note that the **bsStayOnTop** causes the form to be the topmost window on the Windows desktop. The benefit is that your main form, when created, will appear behind the splash form during its initialization procedures. However, it also means that you have to take this into account in the case of exception error dialog boxes that pop up while the splash form is visible. In your exception handling code you'll need to hide the splash form in the **Finally** part of your **Try** block before the exception dialog appears.

Under the **Options** menu, select **Project**. Move the SplashForm from the auto-created column to the other column. This allows you to manually control the creation of the splash form.

### Step 3: Modify the Project Source Code

Under the View menu, select 'Project Source Code'. Insert the necessary lines of code after the 'Begin' and before any of the 'Application...' code. It should look like the sample below:

```
Begin  
  { Display the splash form }  
  SplashForm := TSplashForm.Create(NIL);  
  SplashForm.Show;  
  SplashForm.Repaint;  
  { continue application startup }  
  Application.Title := 'My Application';  
  Application.CreateForm(TMainForm, MainForm);
```

The NIL parameter is necessary here since there is no window yet that can act as a parent for the SplashForm. Specifying NIL makes the SplashForm a child of the Windows desktop window.

The **Show** method displays the form modelessly so that application execution can continue right away.

The **Repaint** method is necessary to force a complete display of the form immediately after creation. My experimentation proved that leaving out this line resulted in the display of only the outline of the splash form while the application continued initializing. Later, when the application is idle, the form would finish displaying itself. **Repaint** does this right away.



## The Unofficial Newsletter of Delphi Users - Issue #2 - April 1st, 1995

### Step 3: Add code to hide the Splash Form

Here you have several options. In order of complexity, you can

1. Have the main form hide the Splash Form just before it becomes visible,
2. Have the Splash Form hide itself after a few seconds, or
3. Have the Main Form hide the Splash Form after all initialization is finished and the app is ready to accept input from the user.

For this option, add the line

```
SplashForm.Free;
```

to the **OnShow** event of the main form. This event occurs just before the main form becomes visible. The drawback of this approach is that, due to the paint time required for the main form, there may be a period where nothing is on the screen.

For this option, you add a Timer component to the SplashForm, leaving it enabled and the timer interval set to as many seconds you feel appropriate (e.g. 3000 for 3 seconds). In the **OnTimer** event, add the following lines:

```
Timer1.Enable := false;  
Close;
```

The first line prevents any more timer events from occurring. The second line closes the splash form. The drawback here is similar to that of option #1, except it's even more out of touch with the state of the Main Form. On slower and faster PC's, the main form may become visible later or sooner than on your development PC. This approach leaves the two forms' visibility uncoordinated.

This option is my preferred approach, especially in database applications. I like to present a splash form, and then the main form behind it. With the splash form still visible, I continue application startup (connecting to the database, opening tables, performing other startup initializations) accompanied by status line messages (e.g. "Connecting to database...", "Retrieving settings...", etc.). Then, when everything is in place, I remove the splash form and allow the user to begin interacting with my application.

This approach requires several steps to implement. First, add a timer component to the Main Form, setting its **Enabled** property to false and the **Interval** property to a value between 250 and 500 (1/4 to 1/2 second). Second, in the **OnShow** event, add the line

```
Timer1.Enabled := True;
```

This activates the timer which will send its first event just after the Main Form has been displayed.

Third, in the **OnTimer** event, add the line

```
Timer1.Enabled := False;
```

to turn off the timer to prevent further events. Follow the line with any initialization calls you'd like to make. These will execute with both the SplashForm and the Main Form visible.

Lastly, at the end of the **OnTimer** event add the line

```
SplashForm.Free;
```

to remove the Splash Form from the screen.

## Additional notes:

In quick startup applications, you can augment the splash form with code in its **OnCreate** event to set a timestamp variable and subsequent code in the **OnQueryClose** event to loop repeatedly until the current time is a given number of seconds past the initial timestamp. This ensures that the splash form stays up a minimum time on the screen, regardless of how long main form takes to display. The Splash Form is still closed from the main form, but this is delayed by **OnQueryClose** if the minimum display time has not yet elapsed.

For example:

```
Repeat  
    WaitMessage  
Until Now > StartTime + 3*(1/24/60/60);  
CanClose := True;
```

This ensures the splash form is visible for at least 3 seconds.

*For questions and comments, contact Boris Bosnjak on Compuserve at 71564,733 or on the Internet at [boris.bosnjak@canrem.com](mailto:boris.bosnjak@canrem.com)*

## Linking Lockup Revisited

In the last issue there was a discussion of a lockup situation while linking . As I look back on that section, my explanation really had very little to do with the problem of lockups. The points discussed were still valid though completely mislabeled.

Since the time of that issue, I have had some time to examine the original Linking Lockup issue in more depth. As you can see in the previous article, the problem was that a program being developed by someone I work with would occasionally get into a "state" where every compile would produce a lockup at the "linking..." stage. When this occurred, his only recourse was to reboot the machine. When he came back into Delphi after rebooting, he discovered that when he picked "compile" again, Delphi realized that the code had already been compiled and linked and would immediately run, skipping the compiling process. This indicated to us that the compile/link process was functioning correctly and that it was something after this that was causing the lockup. The only things culprits we could think of was the auto-saving of the program files, desktop settings, and symbol tables. Sure enough... As soon as we changed the **Desktop Contents** radio group under **Options | Environment** to indicate **Desktop Only**, the problem went away. Apparently something in his code confused the routine for saving the Symbol Tables. As soon as I get a definitive answer on what it was that caused the problem, I will let you know in a future article.

## Return Value of the ExtractFileExt Function

The documentation of the **ExtractFileExt** Function is a little misleading. It indicates that it *returns a string containing the three-character extension*. In actuality it is returning a string that includes the "dot" separator, making the return-value up to 4 characters long.

## Zoom Panel Component

The ZoomPanel is a funky little component. To be honest with you, I would be surprised if anyone found a real use for it. However, I have included it here in the newsletter because it does illustrate a few component design concepts that might be new to some of you.

In a nutshell, the ZoomPanel is normal panel that can zoom open or closed at variable speeds and smoothness. It introduces three new properties to the Panel component:

**ZoomSpeed** controls the pacing of the zoom steps. The range for this value can be from 1 (slowest) to 1000 (top speed). This property controls the **Interval** setting of the timer contained in this component. When the ZoomPanel needs to open or close, it activates the timer with a interval equal to  $1000 \div \text{ZoomSpeed}$ . This gives the timer a operating range of .001 of a second to a full second.

**ZoomSteps** controls the number of steps that the ZoomPanel will take to get from one state (open/closed) to the other. Typically, you would want this to be in the range of around 5 to 20. Higher numbers will result in a smoother, but slower zoom effect.

**ZoomStyle** controls how the panel will zoom from open to close. Currently there are 5 states. One allows zooming to/from the center of the panel, and the other four allow "wipes" from left to right, right to left, top to bottom and bottom to top.

There are some very interesting behaviors of ZoomPanel. First, it will Zoom in design mode, allowing the user to see the chosen zoom effect before the program even runs. Go look at the last line in **SetZoomStyle** and note how you can achieve some effects in design mode, and others in run mode. The styles available can easily be extended by adding a new case item in **ZoomThePanel** and **SetOpenClosedRects**.

ZoomPanel also illustrates how a component can use a timer. I plan on using this technique for a few component ideas I have. Keep in mind however, that each ZoomPanel has its own timer, which is a bit of a no-no in windows programming, as there are limited timers to use. Rumor has it this will not be a problem under Windows 95 however.

Take care when using the Align property. Setting it to alClient will completely defeat the zoom effect, as each redraw will be overridden to snug the panel up against it's parent. The other Align options can produce similar bad effects depending on the ZoomStyle you have chosen.

Also a bit of a problem with ZoomPanel is the screen redraws. Because the panel is redrawing itself each zoom step, it will also redraw any children it possesses as well. Therefore, if you are using a ZoomPanel to obscure buttons, labels, checkboxes, or whatever, you will notice that it can get quite jittery as it redraws. It would take a fairly different approach to the component to avoid this issue.

[Source Code for ZoomPanel](#)





## Source For ZoomPanel

Copy this code to the clipboard using the **Edit | Copy** command and then paste it into a new unit window in Delphi.

```
Unit Zoom;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls, Menus;

type
  TZoomStyles = (zsCenter, zsWipeRight, zsWipeLeft, zsWipeDown, zsWipeUp);
  TZoomStates = (stClosed, stOpen);
  TZoomPanel = class(TCustomPanel)
  private
    FTimer      : TTimer;
    FZoomSpeed  : Integer;
    FZoomSteps  : Integer;
    FZoomStyle  : TZoomStyles;
    FZoomState  : TZoomStates;
    OpenRect    : TRect;
    ClosedRect  : TRect;
    {Left, top, width & height when open}
    OL,OT,OW,OH : Integer;
    {Left, top, width & height when closed}
    CL,CT,CW,CH : Integer;
    CurrStep    : Integer;
    ZoomDir     : Integer;
    procedure SetZoomSpeed(Value: Integer);
    procedure SetZoomSteps(Value: Integer);
    procedure SetZoomStyle(Value: TZoomStyles);
  protected
    procedure TimerTick(Sender: TObject);
    procedure ZoomThePanel;
    procedure RestartTimer;
    procedure CreateParams(var Params: TCreateParams); override;
    procedure SetOpenClosedRects(L,T,W,H: Integer);
    procedure OpenOrClose;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure OpenIt;
    procedure CloseIt;
  published
    property Align;
    property Alignment;
    property BevelInner;
    property BevelOuter;
    property BorderStyle;
    property BorderWidth;
    property Caption;
    property Color;
    property Ctl3D;
    property DragCursor;
    property DragMode;
    property Enabled;
    property Font;
    property Locked;
    property ParentColor;
    property ParentCtl3D;
```

```

    property ParentFont;
    property ParentShowHint;
    property PopupMenu;
    property ShowHint;
    property TabOrder;
    property TabStop;
    property Visible;
    property ZoomSpeed : Integer read FZoomSpeed write SetZoomSpeed default 1000;
    property ZoomSteps : Integer read FZoomSteps write SetZoomSteps default 5;
    property ZoomStyle : TZoomStyles read FZoomStyle write SetZoomStyle default
zsCenter;
    property OnClick;
    property OnDblClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnEnter;
    property OnExit;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
    property OnResize;
end;

procedure Register;

implementation

procedure Register;
begin
    RegisterComponents('Additional', [TZoomPanel]);
end;

constructor TZoomPanel.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);
    {ZoomSpeed controls the Speed at which the panel steps through its
zooming. The actual millisecond delay used is 1000 divided by ZoomSpeed.
Therefore setting ZoomSpeed to a higher number will result in less
of a delay between steps and hence a faster zoom.}
    FZoomSpeed := 1000;
    {This defines the total number of steps that the panel will take to get
to its final size and location}
    FZoomSteps := 5;
    CurrStep := -1;
    {ZoomStyle defines the effect used when zooming.}
    FZoomStyle := zsCenter;
    {Set the panel as Open}
    FZoomState := stOpen;
    {First, we need to create a timer that will control the pacing of the
panel's Zoom effect.}
    FTimer := TTimer.Create(Self);
    {Set the procedure that will handle the timer. Every time the timer expires
it will transfer control to this procedure. The procedure moves the panel a
single step and then the timer starts over.}
    FTimer.OnTimer := TimerTick;
    {Set the timer as initially off. CreateParams turns it on}
    FTimer.Enabled := False;
    {Set the rate at which the panel opens or closes}
    FTimer.Interval := 1000 div ZoomSpeed;
end;

destructor TZoomPanel.Destroy;

```

```

begin
  {When the panel is destroyed, kill the timer}
  FTimer.Free;
  inherited Destroy;
end;

procedure TZoomPanel.TimerTick(Sender: TObject);
begin
  {Go and do a single zoom step}
  ZoomThePanel;
end;

procedure TZoomPanel.SetZoomSpeed(Value: Integer);
begin
  {ZoomSpeed must be at least 1 (slowest) and no greater than 1000 (top speed)}
  if FZoomSpeed <> Value then
    begin
      if Value > 1000 then FZoomSpeed := 1000
      else
        if Value < 1 then FZoomSpeed := 1
        else FZoomSpeed := Value;
        {Set the rate at which the panel opens or closes}
        If FTimer <> Nil then FTimer.Interval := 1000 div ZoomSpeed;
    end;
end;

procedure TZoomPanel.SetZoomSteps(Value: Integer);
begin
  {ZoomSteps must be at least 1 and no greater than 100}
  if FZoomSteps <> Value then
    if Value < 1 then FZoomSteps := 1
    else
      if Value > 100 then FZoomSteps := 100
      else FZoomSteps := Value;
end;

procedure TZoomPanel.SetZoomStyle(Value: TZoomStyles);
begin
  if FZoomStyle <> Value then
    begin
      FZoomStyle := Value;
      {Go set the sizes of the open and closed states of the panel}
      SetOpenClosedRects(Left,Top,Width,Height);
      {Do a Zoom in design mode so we can see the effect}
      If (csDesigning in ComponentState) and (CurrStep=-1) then OpenOrClose;
    end;
end;

procedure TZoomPanel.SetZoomState(Value: TZoomStates);
begin
  if FZoomState <> Value then FZoomState := Value;
end;

procedure TZoomPanel.ZoomThePanel;
var
  OW,OH,CW,HW : Integer;
begin
  If FZoomState = stOpen then Inc(CurrStep) else Dec(CurrStep);
  OW := OpenRect.Right-OpenRect.Left; {open width}
  OH := OpenRect.Bottom-OpenRect.Top; {open height}
  CW := Trunc(OW/ZoomSteps*CurrStep); {current width}
  CH := Trunc(OH/ZoomSteps*CurrStep); {current height}
  Case ZoomStyle of

```

```

    zsCenter      : with ClosedRect do
                        SetBounds(Left-CW div 2,Top-CH div 2,CW,CH);
    zsWipeRight   : with ClosedRect do
                        SetBounds(Left,Top,CW,OH);
    zsWipeLeft    : with ClosedRect do
                        SetBounds(Right-CW,Top,CW,OH);
    zsWipeUp      : with ClosedRect do
                        SetBounds(Left,Top-CH,OW,CH);
    zsWipeDown    : with ClosedRect do
                        SetBounds(Left,Top,OW,CH);
end;
{Disable the timer if we have done our last zoom step}
If (CurrStep >= ZoomSteps) or (CurrStep <= 0) Then
begin
    FTimer.Enabled := False;
    CurrStep := -1;
end;
end;

procedure TZoomPanel.SetOpenClosedRects(L,T,W,H: Integer);
begin
    {Save the size that the panel should be when open. I had to do this in
    the CreateParams method because the size of the control is not set until
    after the Create is complete - so that users could override the size.}
    OpenRect := Rect(L,T,L+W,T+H);
    {Save the size that the panel should be when closed.}
    Case ZoomStyle of
        {This is the default zooming behavior. It picks a spot in the exact
        center of the control as the center of the zoom effect. The panel
        will open from and close into this point.}
        zsCenter      : ClosedRect := Rect(L+W div 2,T+H div 2,L+W div 2,T+H div 2);
        {The panel will open like a "screen door" to the right}
        zsWipeRight   : ClosedRect := Rect(L,T,L,T+H);
        {The panel will open like a "screen door" to the left}
        zsWipeLeft    : ClosedRect := Rect(L+W,T,L+W,T+H);
        {The panel will open like a "window shade" going up}
        zsWipeUp      : ClosedRect := Rect(L,T+H,L+W,T+H);
        {The panel will open like a "window shade" going down}
        zsWipeDown    : ClosedRect := Rect(L,T,L+W,T);
    end;
end;

procedure TZoomPanel.CreateParams(var Params: TCreateParams);
begin
    inherited CreateParams(Params);
    {Go set the sizes of the open and closed states of the panel}
    with Params do SetOpenClosedRects(Left,Top,Width,Height);
    {Activate the timer to start the zoom open}
    OpenOrClose;
end;

Procedure TZoomPanel.OpenOrClose;
begin
    if (FZoomState = stOpen) then OpenIt else CloseIt;
end;

procedure TZoomPanel.OpenIt;
begin
    FZoomState := stOpen;
    RestartTimer;
end;

procedure TZoomPanel.CloseIt;

```

```
begin
  FZoomState := stClosed;
  RestartTimer;
end;

procedure TZoomPanel.RestartTimer;
begin
  If FZoomState = stOpen then CurrStep := 0 else CurrStep := ZoomSteps;
  {Turn the timer on}
  FTimer.Enabled := True;
end;

end.
```



## Cooking Up Components

"Cooking Up Components" is going to be a regular section in this newsletter. Each issue I will present at least one new component, either developed by myself, or contributed by one of you. It will include source-code that you can copy out of the newsletter and incorporate into your Delphi Component Library. In addition, I will endeavor to bring you articles on design and creation of components.

There is going to be one standing rule for all components distributed by means of this newsletter, whether of my own design, or by that of another person. This policy is that you may freely use any code/work presented here, but you may not sell the component on its own, or as part of a component library. You can use a component as part of the normal development and distribution of an application, but the author of the component reserves all rights to the component as "intellectual property". If you have any special requirements for component code presented here that lies outside of the "spirit" of this policy, I recommend you talk directly to the component author.

[Custom Component - ZoomPanel](#)

[Creating Objects On The Fly](#)

## Minimizing Your Children

No really... I like kids. But Delphi has one little quirk when dealing with them. When writing an MDI application, if you attempt to minimize one of the MDI child windows at design-time, you may get a GPF (General Protection Fault). Borland acknowledges this as a problem and says that even though it only occurs at design time, everything works normally at runtime. They say that you can choose to ignore the GPF and continue development as usual, or you can just choose not to minimize an MDI child at design time.



## Creating Objects On The Fly

Several people I have talked with have expressed some confusion about how they can create objects at run-time. Typically, the issue is that they need some object, form or component created during program execution that is not easily laid-out at design time. An example of this might be the creation of a timer to control some unusual activity. Therefore, I am going to discuss the procedures involved with the help of information presented in Borland's Section 5 FAQ sheet.

Let's assume you want to create a button on a form. Granted, you could simply place it in design mode on the form, hide it, and then show it when you need it at runtime. However, sometimes, you might need to create something dynamically in response to a users actions, and you don't want to have forms or components created and lying around when they will only be used under rare circumstances. The sections in bold italics are from the FAQ sheet Borland has provided and my comments will be added in normal print.

1. ***Declare an instance variable of the component type that you wish to create {i.e. TButton}.*** ***Note: instance variables are used to point to an actual instance of an object. They are not objects themselves.*** This declaration would of course be done in a "Var" statement. Keep in mind the scope that you need for the object. If you are simply creating something and then disposing of it immediately, it is better to declare the instance of the object locally (like within a procedure or function for example). If the object will be used in other portions of the program, you will need to change where you define the instance of the object so that all segments of code can have the appropriate access to it. From what I have learned, most (if not all) class declarations of the pre-defined Delphi Objects are pointers to the object rather than the object itself. Therefore if you declare a variable of type TButton, your variable is actually a **pointer** to a TButton. Not an actual structure of all the TButton variables, methods, and procedures. Refer to line A of the example below.
2. ***Use the component's Create constructor to create an instance of the component and assign the instance to the instance variable declared in step 1. All components' Create constructors take a parameter - the component's owner. Except in special circumstances, you should always pass the form as the owner parameter of the component's Create constructor.*** This part seems to throw a few people. Refer to line B of the example below. From any previous experience with Pascal you might have had, you might be more inclined to do something like this: `TempButton.Create;` This is saying: "I am a TButton and now I need to create myself". Rather the appropriate style is to do as indicated on line B below which is saying. "Yo... Delphi... Go create an instance of a TButton for me and give me a pointer to it." As for the parameter used in the Create constructor, do make sure that you are using the form that will own this component. The reason for this is that Delphi needs to know how to destroy the component when it is done. This establishes a link to the owner so that when it is destroyed, the component is destroyed. Only use something else if you really understand the ramifications of the change.
3. ***Assign a parent to the component's Parent property (i.e. Form1, Panel1, etc). The Parent determines where the component will be displayed, and how the component's Top and Left coordinates are interpreted. To place a component in a groupbox, set the component's Parent property to the groupbox. For a component to be visible, it must have a parent to display itself within.*** Interestingly some examples of code I have seen indicate that you can switch around an object's parent. Visually, I think this would be interesting. I think I may fiddle with it a bit to see what happens.
4. ***Set any other properties that are necessary (i.e. Width, Height).***



5. **Finally, make the component appear on the form by setting the component's Visible property to True.**
6. **If you created the component with an owner, you don't need to do anything to free the component - it will be freed when the owner is destroyed. If you did not give the component an owner when you created it, you are responsible for making sure the component is freed when it is no longer needed.**
7. Something you may also wish to do that is not covered in the FAQ sheet (particularly for this example) is to point the button's OnClick method to a procedure that will handle a user clicking on the button. This would look something like this: `TempButton.OnClick := GoDoMyThing;`

The following demonstrates how to add a TButton component to the current form at run-time:

```
| var
A |   TempButton : TButton; { This is only a pointer to a TButton }
| begin
B |   TempButton := TButton.Create(Self); { Self refers to the form }
C |   TempButton.Parent := Self;        { Must assign the Parent }
D |   TempButton.Caption := 'Run-time';  { Assign properties now }
E |   TempButton.Visible := True;       { Show to button }
| end;
```



