Using ActiveX Controls with Delphi

CHAPTER

7

IN THIS CHAPTER

Delphi gives you the great advantage of easily integrating industry-standard ActiveX controls (formerly known as *OCX* or *OLE controls*) into your applications. Unlike Delphi's own custom components, ActiveX controls are designed to be independent of any particular development tool. This means that you can count on many vendors to provide a variety of ActiveX solutions that open up a world of features and functionality.

ActiveX control support in 32-bit Delphi works similarly to the way VBX support works in 16-bit Delphi 1. You select an option to add new ActiveX controls from Delphi's IDE main menu or package editor, and Delphi builds an Object Pascal wrapper for the ActiveX control—which is then compiled into a package and added to the Delphi Component Palette. Once there, the ActiveX control seamlessly merges into the Component Palette along with your other VCL and ActiveX components. From that point, you're just a click and a drop away from adding an ActiveX control to any of your applications. This chapter discusses integrating ActiveX controls into Delphi, using an ActiveX control in your application, and shipping ActiveX-equipped applications.

NOTE

Delphi 1 was the last version of Delphi to support VBX (Visual Basic Extension) controls. If you have a Delphi 1 project that relies on one or more VBX controls, check with the VBX vendors to see whether they supply a comparable ActiveX solution for use in your 32-bit Delphi applications.

What Is an ActiveX Control?

ActiveX controls are custom controls for 16-bit and 32-bit Windows applications that take advantage of the COM-based OLE and ActiveX technologies. Unlike VBX controls, which are designed for 16-bit Visual Basic (and therefore share Visual Basic's limitations), ActiveX controls were designed from the ground up with application independence in mind. Roughly speaking, you can think of ActiveX controls as a merging of the easy-to-use VBX technology with the open ActiveX standard. For the purposes of this chapter, you can think of OLE and ActiveX as the same thing. If you're looking for greater distinction between these terms, take a look at Chapter 23, "COM and ActiveX."

Under the skin, an ActiveX control is really an ActiveX server that, in one package, can provide all the power of ActiveX —including all OLE functions and services, visual editing, drag and drop, and OLE Automation. Like all ActiveX servers, ActiveX controls are registered in the System Registry. ActiveX controls can be developed using a variety of products, including Delphi, Borland C++Builder, Visual C++, and Visual Basic.

Microsoft is actively promoting ActiveX controls as the choice medium for application-independent custom controls; Microsoft has stated that VBX technology will not be directly supported in the Win32 operating systems and beyond. For this reason, developers should look to ActiveX controls rather than VBX controls when developing 32-bit applications.

NOTE

For a more complete description of ActiveX control technology, see Chapter 25, "Creating ActiveX Controls."

Deciding When To Use an ActiveX Control

There are typically two reasons why you would use an ActiveX control rather than a native Delphi component. The first reason is that no Delphi component is available that fits your particular need. Because the ActiveX control market is larger than that for VCL controls, you're likely to find a greater variety of fully featured "industrial strength" controls, such as word processors, World Wide Web browsers, and spreadsheets, as ActiveX controls. The second reason you would use an ActiveX control instead of a native Delphi control is if you develop in multiple programming languages and you want to leverage your expertise in some particular control or controls across the multiple development platforms.

Although ActiveX controls integrate seamlessly into the Delphi IDE, keep in mind some inherent disadvantages to using ActiveX controls in your applications. The most obvious issue is that, although Delphi components are built directly into an application executable, ActiveX controls typically require one or more additional runtime files that must be deployed with an executable. Another issue is that ActiveX controls communicate with applications through the COM layer, whereas Delphi components communicate directly with applications and other components. This means that a well-written Delphi component typically performs better than a well-written ActiveX control. A more subtle disadvantage of ActiveX controls is that they're a "lowest common denominator" solution, so they won't exploit all the capabilities of the development tool in which they're used.

Adding an ActiveX Control to the Component Palette

The first step in using a particular ActiveX control in your Delphi application is adding that control to the Component Palette in the Delphi IDE. This places an icon for the ActiveX control on the Component Palette among your other Delphi and ActiveX controls. After you add a particular ActiveX control to the Component Palette, you can drop it on any form and use it as you would any other Delphi control.

To add an ActiveX control to the Component Palette, follow these steps:

1. Choose Component, Import ActiveX Control from the main menu. The Import ActiveX dialog box appears (see Figure 7.1).

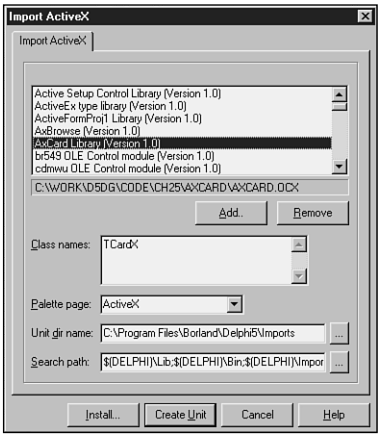

FIGURE 7.1
The Import ActiveX dialog box.

2. The Import ActiveX dialog box is divided into two parts: the top portion contains a list box of registered ActiveX controls and provides Add and Remove buttons that enable you to register and unregister controls. The bottom portion of the dialog box allows you to specify parameters for creating a Delphi component and unit that encapsulates the control.
3. If the name of the ActiveX control you want to use is listed in the top portion of the dialog box, proceed to step 4. Otherwise, click the Add button to register a new control with the system. Clicking the Add button invokes the Register OLE Control dialog box (see Figure 7.2). Select the name of the OCX or DLL file that represents the ActiveX control you want to add to the system and click the Open button. This registers the selected ActiveX control with the System Registry and dismisses the Register OLE Control dialog box.
4. In the upper portion of the Import ActiveX dialog box, select the name of the ActiveX control you want to add to the Component Palette. The lower portion of the dialog box contains edit controls for unit directory name, palette page, and search path as well as a memo control that lists the classes contained within the OCX file. The pathname shown in the Unit Dir Name edit box is the pathname of the Delphi wrapper component created to interface with the ActiveX control. The filename defaults to the same name as the

OCX file (with a `.pas` extension); the path defaults to the `\Delphi5\Imports` subdirectory. Although the default is fine to use, you can edit the directory path to your liking.

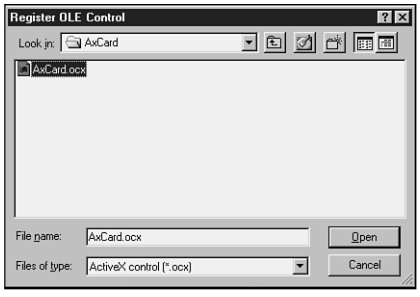

FIGURE 7.2
The Register OLE Control dialog box.

5. The Palette Page edit control in the Import ActiveX dialog box contains the name of the page on the Component Palette where you want this control to reside. The default is the ActiveX page. You can choose another existing page; alternatively, if you make up a new name, a corresponding page is created on the Component Palette.
6. The Class Names memo control in the Import ActiveX dialog box contains the names of the new objects created in this control. You should normally leave these names set to the default unless you have a specific reason for doing otherwise. For example, such a reason would be if the default class name conflicts with another component already installed in the IDE.
7. At this point, you can click either the Install or the Create Unit button in the Import ActiveX dialog box. The Create Unit button will generate the source code for the unit for the ActiveX control component wrapper. The Install button will generate the wrapper code and then invoke the Install dialog box, which allows you to choose a package into which you may install the component (see Figure 7.3).

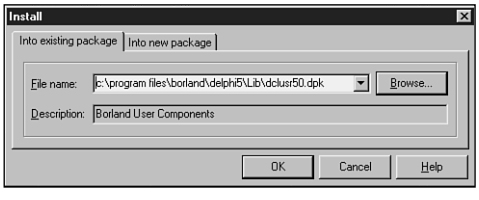

FIGURE 7.3
The Install dialog box.

8. In the Install dialog box, you can choose to add the control to an existing package or create a new package that will be installed to the Component Palette. Click OK in this dialog box, and the component will be installed to the palette.

Now your ActiveX control is on the Component Palette and ready to roll.

The Delphi Component Wrapper

Now is a good time to look into the Object Pascal wrapper created to encapsulate the ActiveX control. Doing so can help shed some light on how Delphi's ActiveX support works so that you can understand the capabilities and limitations inherent in ActiveX controls. Listing 7.1 shows the `Card_TLB.pas` unit generated by Delphi; this unit encapsulates the `AxCard.ocx` ActiveX control.

NOTE

`AxCard.ocx` is an ActiveX control developed in Chapter 25, "Creating ActiveX Controls."

LISTING 7.1 The Delphi Component Wrapper Unit for `AxCard.ocx`.

```
unit AxCard_TLB;

// ************************************************************************ //
// WARNING
// -------
// The types declared in this file were generated from data read from a
// Type Library. If this type library is explicitly or indirectly (via
// another type library referring to this type library) re-imported, or
// the 'Refresh' command of the Type Library Editor activated while
// editing the Type Library, the contents of this file will be
// regenerated and all manual modifications will be lost.
// ************************************************************************ //

// PASTLWTR : $Revision:   1.88  $
// File generated on 8/24/99 9:24:19 AM from Type Library described below


// ************************************************************************ //
// NOTE:
// Items guarded by $IFDEF_LIVE_SERVER_AT_DESIGN_TIME are used by
// properties which return objects that may need to be explicitly created
// via a function call prior to any access via the property. These items
```

```
// have been disabled in order to prevent accidental use from within the
// object inspector. You may enable them by defining
// LIVE_SERVER_AT_DESIGN_TIME or by selectively removing them from the
// $IFDEF blocks. However, such items must still be programmatically
// created via a method of the appropriate CoClass before they can be
// used.
// ************************************************************************ //
// Type Lib: C:\work\d5dg\code\Ch25\AxCard\AxCard.tlb (1)
// IID\LCID: {7B33D940-0A2C-11D2-AE5C-04640BC10000}\0
// Helpfile:
// DepndLst:
//   (1) v2.0 stdole, (C:\WINDOWS\SYSTEM\STDOLE2.TLB)
//   (2) v4.0 StdVCL, (C:\WINDOWS\SYSTEM\STDVCL40.DLL)
// ************************************************************************ //
{$TYPEDADDRESS OFF} // Unit must be compiled without type-checked
                    // pointers.
interface

uses Windows, ActiveX, Classes, Graphics, OleServer, OleCtrls, StdVCL;

// *********************************************************************//
// GUIDS declared in the TypeLibrary. Following prefixes are used:
//   Type Libraries     : LIBID_xxxx
//   CoClasses          : CLASS_xxxx
//   DISPInterfaces     : DIID_xxxx
//   Non-DISP interfaces: IID_xxxx
// *********************************************************************//
const
  // TypeLibrary Major and minor versions
  AxCardMajorVersion = 1;
  AxCardMinorVersion = 0;

  LIBID_AxCard: TGUID = '{7B33D940-0A2C-11D2-AE5C-04640BC10000}';

  IID_ICardX: TGUID = '{7B33D941-0A2C-11D2-AE5C-04640BC10000}';
  DIID_ICardXEvents: TGUID = '{7B33D943-0A2C-11D2-AE5C-04640BC10000}';
  CLASS_CardX: TGUID = '{7B33D945-0A2C-11D2-AE5C-04640BC10000}';

// *********************************************************************//
// Declaration of Enumerations defined in Type Library
// *********************************************************************//
// Constants for enum TxDragMode
type
  TxDragMode = TOleEnum;
const
  dmManual = $00000000;
  dmAutomatic = $00000001;
```

continues

LISTING 7.1 Continued

```
// Constants for enum TxCardSuit
type
  TxCardSuit = TOleEnum;
const
  csClub = $00000000;
  csDiamond = $00000001;
  csHeart = $00000002;
  csSpade = $00000003;

// Constants for enum TxCardValue
type
  TxCardValue = TOleEnum;
const
  cvAce = $00000000;
  cvTwo = $00000001;
  cvThree = $00000002;
  cvFour = $00000003;
  cvFive = $00000004;
  cvSix = $00000005;
  cvSeven = $00000006;
  cvEight = $00000007;
  cvNine = $00000008;
  cvTen = $00000009;
  cvJack = $0000000A;
  cvQueen = $0000000B;
  cvKing = $0000000C;

// Constants for enum TxMouseButton
type
  TxMouseButton = TOleEnum;
const
  mbLeft = $00000000;
  mbRight = $00000001;
  mbMiddle = $00000002;

// Constants for enum TxAlignment
type
  TxAlignment = TOleEnum;
const
  taLeftJustify = $00000000;
  taRightJustify = $00000001;
  taCenter = $00000002;

// Constants for enum TxBiDiMode
type
```

```
  TxBiDiMode = TOleEnum;
const
  bdLeftToRight = $00000000;
  bdRightToLeft = $00000001;
  bdRightToLeftNoAlign = $00000002;
  bdRightToLeftReadingOnly = $00000003;

type

// *********************************************************************//
// Forward declaration of types defined in TypeLibrary
// *********************************************************************//
  ICardX = interface;
  ICardXDisp = dispinterface;
  ICardXEvents = dispinterface;

// *********************************************************************//
// Declaration of CoClasses defined in Type Library
// (NOTE: Here we map each CoClass to its Default Interface)
// *********************************************************************//
  CardX = ICardX;


// *********************************************************************//
// Interface: ICardX
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {7B33D941-0A2C-11D2-AE5C-04640BC10000}
// *********************************************************************//
  ICardX = interface(IDispatch)
    ['{7B33D941-0A2C-11D2-AE5C-04640BC10000}']
    function  Get_BackColor: OLE_COLOR; safecall;
    procedure Set_BackColor(Value: OLE_COLOR); safecall;
    function  Get_Color: OLE_COLOR; safecall;
    procedure Set_Color(Value: OLE_COLOR); safecall;
    function  Get_DragCursor: Smallint; safecall;
    procedure Set_DragCursor(Value: Smallint); safecall;
    function  Get_DragMode: TxDragMode; safecall;
    procedure Set_DragMode(Value: TxDragMode); safecall;
    function  Get_FaceUp: WordBool; safecall;
    procedure Set_FaceUp(Value: WordBool); safecall;
    function  Get_ParentColor: WordBool; safecall;
    procedure Set_ParentColor(Value: WordBool); safecall;
    function  Get_Suit: TxCardSuit; safecall;
    procedure Set_Suit(Value: TxCardSuit); safecall;
    function  Get_Value: TxCardValue; safecall;
    procedure Set_Value(Value: TxCardValue); safecall;
```

continues

LISTING 7.1 Continued

```
    function  Get_DoubleBuffered: WordBool; safecall;
    procedure Set_DoubleBuffered(Value: WordBool); safecall;
    procedure FlipChildren(AllLevels: WordBool); safecall;
    function  DrawTextBiDiModeFlags(Flags: Integer): Integer; safecall;
    function  DrawTextBiDiModeFlagsReadingOnly: Integer; safecall;
    function  Get_Enabled: WordBool; safecall;
    procedure Set_Enabled(Value: WordBool); safecall;
    function  GetControlsAlignment: TxAlignment; safecall;
    procedure InitiateAction; safecall;
    function  IsRightToLeft: WordBool; safecall;
    function  UseRightToLeftAlignment: WordBool; safecall;
    function  UseRightToLeftReading: WordBool; safecall;
    function  UseRightToLeftScrollBar: WordBool; safecall;
    function  Get_BiDiMode: TxBiDiMode; safecall;
    procedure Set_BiDiMode(Value: TxBiDiMode); safecall;
    function  Get_Visible: WordBool; safecall;
    procedure Set_Visible(Value: WordBool); safecall;
    function  Get_Cursor: Smallint; safecall;
    procedure Set_Cursor(Value: Smallint); safecall;
    function  ClassNameIs(const Name: WideString): WordBool; safecall;
    procedure AboutBox; safecall;
    property BackColor: OLE_COLOR read Get_BackColor write Set_BackColor;
    property Color: OLE_COLOR read Get_Color write Set_Color;
    property DragCursor: Smallint read Get_DragCursor write
      Set_DragCursor;
    property DragMode: TxDragMode read Get_DragMode write Set_DragMode;
    property FaceUp: WordBool read Get_FaceUp write Set_FaceUp;
    property ParentColor: WordBool read Get_ParentColor write
      Set_ParentColor;
    property Suit: TxCardSuit read Get_Suit write Set_Suit;
    property Value: TxCardValue read Get_Value write Set_Value;
    property DoubleBuffered: WordBool read Get_DoubleBuffered write
      Set_DoubleBuffered;
    property Enabled: WordBool read Get_Enabled write Set_Enabled;
    property BiDiMode: TxBiDiMode read Get_BiDiMode write Set_BiDiMode;
    property Visible: WordBool read Get_Visible write Set_Visible;
    property Cursor: Smallint read Get_Cursor write Set_Cursor;
  end;

// ******************************************************************//
// DispIntf:  ICardXDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {7B33D941-0A2C-11D2-AE5C-04640BC10000}
// ******************************************************************//
  ICardXDisp = dispinterface
```

```
    ['{7B33D941-0A2C-11D2-AE5C-04640BC10000}']
    property BackColor: OLE_COLOR dispid 1;
    property Color: OLE_COLOR dispid -501;
    property DragCursor: Smallint dispid 2;
    property DragMode: TxDragMode dispid 3;
    property FaceUp: WordBool dispid 4;
    property ParentColor: WordBool dispid 5;
    property Suit: TxCardSuit dispid 6;
    property Value: TxCardValue dispid 7;
    property DoubleBuffered: WordBool dispid 10;
    procedure FlipChildren(AllLevels: WordBool); dispid 11;
    function  DrawTextBiDiModeFlags(Flags: Integer): Integer; dispid 14;
    function  DrawTextBiDiModeFlagsReadingOnly: Integer; dispid 15;
    property Enabled: WordBool dispid -514;
    function  GetControlsAlignment: TxAlignment; dispid 16;
    procedure InitiateAction; dispid 18;
    function  IsRightToLeft: WordBool; dispid 19;
    function  UseRightToLeftAlignment: WordBool; dispid 24;
    function  UseRightToLeftReading: WordBool; dispid 25;
    function  UseRightToLeftScrollBar: WordBool; dispid 26;
    property BiDiMode: TxBiDiMode dispid 27;
    property Visible: WordBool dispid 28;
    property Cursor: Smallint dispid 29;
    function  ClassNameIs(const Name: WideString): WordBool; dispid 33;
    procedure AboutBox; dispid -552;
  end;

// ******************************************************************//
// DispIntf:  ICardXEvents
// Flags:     (4096) Dispatchable
// GUID:      {7B33D943-0A2C-11D2-AE5C-04640BC10000}
// ******************************************************************//
  ICardXEvents = dispinterface
    ['{7B33D943-0A2C-11D2-AE5C-04640BC10000}']
    procedure OnClick; dispid 1;
    procedure OnDblClick; dispid 2;
    procedure OnKeyPress(var Key: Smallint); dispid 7;
  end;


// ******************************************************************//
// OLE Control Proxy class declaration
// Control Name     : TCardX
// Help String      : CardX Control
// Default Interface: ICardX
// Def. Intf. DISP? : No
```

continues

LISTING 7.1 Continued

```
// Event   Interface: ICardXEvents
// TypeFlags        : (34) CanCreate Control
// ********************************************************************//
  TCardXOnKeyPress = procedure(Sender: TObject; var Key: Smallint) of
    object;

  TCardX = class(TOleControl)
  private
    FOnClick: TNotifyEvent;
    FOnDblClick: TNotifyEvent;
    FOnKeyPress: TCardXOnKeyPress;
    FIntf: ICardX;
    function  GetControlInterface: ICardX;
  protected
    procedure CreateControl;
    procedure InitControlData; override;
  public
    procedure FlipChildren(AllLevels: WordBool);
    function  DrawTextBiDiModeFlags(Flags: Integer): Integer;
    function  DrawTextBiDiModeFlagsReadingOnly: Integer;
    function  GetControlsAlignment: TxAlignment;
    procedure InitiateAction;
    function  IsRightToLeft: WordBool;
    function  UseRightToLeftAlignment: WordBool;
    function  UseRightToLeftReading: WordBool;
    function  UseRightToLeftScrollBar: WordBool;
    function  ClassNameIs(const Name: WideString): WordBool;
    procedure AboutBox;
    property  ControlInterface: ICardX read GetControlInterface;
    property  DefaultInterface: ICardX read GetControlInterface;
    property DoubleBuffered: WordBool index 10 read GetWordBoolProp write
      SetWordBoolProp;
    property Enabled: WordBool index -514 read GetWordBoolProp write
      SetWordBoolProp;
    property BiDiMode: TOleEnum index 27 read GetTOleEnumProp write
      SetTOleEnumProp;
    property Visible: WordBool index 28 read GetWordBoolProp write
      SetWordBoolProp;
  published
    property  TabStop;
    property  Align;
    property  ParentShowHint;
    property  PopupMenu;
    property  ShowHint;
    property  TabOrder;
```

```
    property  OnDragDrop;
    property  OnDragOver;
    property  OnEndDrag;
    property  OnEnter;
    property  OnExit;
    property  OnStartDrag;
    property BackColor: TColor index 1 read GetTColorProp write
      SetTColorProp stored False;
    property Color: TColor index -501 read GetTColorProp write
      SetTColorProp stored False;
    property DragCursor: Smallint index 2 read GetSmallintProp write
      SetSmallintProp stored False;
    property DragMode: TOleEnum index 3 read GetTOleEnumProp write
      SetTOleEnumProp stored False;
    property FaceUp: WordBool index 4 read GetWordBoolProp write
      SetWordBoolProp stored False;
    property ParentColor: WordBool index 5 read GetWordBoolProp write
      SetWordBoolProp stored False;
    property Suit: TOleEnum index 6 read GetTOleEnumProp write
      SetTOleEnumProp stored False;
    property Value: TOleEnum index 7 read GetTOleEnumProp write
      SetTOleEnumProp stored False;
    property Cursor: Smallint index 29 read GetSmallintProp write
      SetSmallintProp stored False;
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
    property OnDblClick: TNotifyEvent read FOnDblClick write FOnDblClick;
    property OnKeyPress: TCardXOnKeyPress read FOnKeyPress write
      FOnKeyPress;
  end;

procedure Register;

implementation

uses ComObj;

procedure TCardX.InitControlData;
const
  CEventDispIDs: array [0..2] of DWORD = (
    $00000001, $00000002, $00000007);
  CControlData: TControlData2 = (
    ClassID: '{7B33D945-0A2C-11D2-AE5C-04640BC10000}';
    EventIID: '{7B33D943-0A2C-11D2-AE5C-04640BC10000}';
    EventCount: 3;
    EventDispIDs: @CEventDispIDs;
    LicenseKey: nil (*HR:$00000000*);
```

LISTING 7.1 Continued

```
    Flags: $00000009;
    Version: 401);
begin
  ControlData := @CControlData;
  TControlData2(CControlData).FirstEventOfs :=
    Cardinal(@@FOnClick) - Cardinal(Self);
end;

procedure TCardX.CreateControl;

  procedure DoCreate;
  begin
    FIntf := IUnknown(OleObject) as ICardX;
  end;

begin
  if FIntf = nil then DoCreate;
end;

function TCardX.GetControlInterface: ICardX;
begin
  CreateControl;
  Result := FIntf;
end;

procedure TCardX.FlipChildren(AllLevels: WordBool);
begin
  DefaultInterface.FlipChildren(AllLevels);
end;

function  TCardX.DrawTextBiDiModeFlags(Flags: Integer): Integer;
begin
  Result := DefaultInterface.DrawTextBiDiModeFlags(Flags);
end;

function  TCardX.DrawTextBiDiModeFlagsReadingOnly: Integer;
begin
  Result := DefaultInterface.DrawTextBiDiModeFlagsReadingOnly;
end;

function  TCardX.GetControlsAlignment: TxAlignment;
begin
  Result := DefaultInterface.GetControlsAlignment;
end;
```

```
procedure TCardX.InitiateAction;
begin
  DefaultInterface.InitiateAction;
end;

function  TCardX.IsRightToLeft: WordBool;
begin
  Result := DefaultInterface.IsRightToLeft;
end;

function  TCardX.UseRightToLeftAlignment: WordBool;
begin
  Result := DefaultInterface.UseRightToLeftAlignment;
end;

function  TCardX.UseRightToLeftReading: WordBool;
begin
  Result := DefaultInterface.UseRightToLeftReading;
end;

function  TCardX.UseRightToLeftScrollBar: WordBool;
begin
  Result := DefaultInterface.UseRightToLeftScrollBar;
end;

function  TCardX.ClassNameIs(const Name: WideString): WordBool;
begin
  Result := DefaultInterface.ClassNameIs(Name);
end;

procedure TCardX.AboutBox;
begin
  DefaultInterface.AboutBox;
end;

procedure Register;
begin
  RegisterComponents('ActiveX',[TCardX]);
end;

end.
```


Now that you've seen the code generated by the type library editor, let's look a little deeper at the type library import mechanism.

Where Do Wrapper Files Come From?

The first thing you might notice is that the filename ends in `_TLB`. More subtly, you might have caught on to the fact that there are several references to "library" in the generated source file. Both of these are clues as to the origin of the wrapper file: the control's type library. An ActiveX control's type library is special information linked to the control as a resource that describes the different elements of an ActiveX control. In particular, type libraries contain information such as the interfaces supported by a control, the properties, methods, and events of a control, and the enumerated types used by the control. The first entry in the wrapper file is the GUID of the control's type library.

NOTE

Type libraries are used more generally for any type of Automation object. Chapter 23, "COM and ActiveX," contains more information on type libraries and their use.

Enumerations

Looking at the generated unit from the top down, immediately following the type library GUID, are the enumerated types used by the control. Notice that the enumerations are declared as simple constants rather than true enumerated types. This is done because type library enumerations, like those in the C language, do not need to start at zero, and the element ordinals do not need to be contiguous. Because this type of declaration isn't legal in Object Pascal, the enumerations must be declared as constants.

Control Interfaces

Next in the wrapper file, the control's primary interface is declared. Here, you'll find all the properties and methods of the ActiveX control. The properties are also redeclared in a `dispinterface`, thus allowing the control to be used as a dual interface. The events are declared separately next in a `dispinterface`. You definitely don't need to know about interfaces to use ActiveX controls in your applications. What's more, working with interfaces can be a complicated topic, so we won't go into too much detail right now. You'll find more information on interfaces in general in Chapter 23, "COM and ActiveX," and information on interfaces with ActiveX controls in Chapter 25, "Creating ActiveX Controls."

TOleControl Descendant

Next in the unit file comes the class definition for the control wrapper. By default, the name of the ActiveX control wrapper object is `TXX`, where *`X`* is the name of the control's `coclass` in the type library. This object, like all ActiveX control wrappers, descends from the `TOleControl`

class. `TOleControl` is a window handle–bearing component that descends from `TWinControl`. `TOleControl` encapsulates the complexities of mapping the functionality of ActiveX controls to Delphi components so that ActiveX controls work seamlessly from Delphi. `TOleControl` is an *abstract class*—meaning that you never want to create an instance of one but instead use it as a starting place for other classes.

The Methods

The first procedure shown is the `InitControlData()` procedure. This procedure is introduced in the `TOleControl` object and is overridden for all descendants. It sets up the unique OLE class and event identification numbers in addition to other control-specific information. In particular, this method makes the `TOleControl` aware of important ActiveX control details such as class IDs, control miscellaneous flags, and a license key if the control is licensed. This method is found in the `protected` part of the class definition because it's not useful to users of the class—it only has meaning internal to the class.

The `InitControlInterface()` method is overridden to initialize the private `FIntf` interface field with a pointer to the control's `ICardsX` interface.

The `CardX` ActiveX control exposes only one other method: `AboutBox()`. It's standard for ActiveX controls to contain a method called `AboutBox()` that invokes a custom About dialog box for the control. This method is called through the `vTable` interface using the `ControlInterface` property, which is read from the `FIntf` field.

NOTE

In addition to `vTable` calls using the `ControlInterface` property, `Control` methods can also be invoked via Automation using `TOleControl`'s `OleObject` property. As you'll learn in Chapter 23, "COM and ActiveX," it's usually more efficient to call methods via the `vTable` rather than through Automation.

The Properties

You might have noticed that the `TCardX` class has two distinct groups of properties. One group does not have read and write values specified. These are standard Delphi component properties and events inherited from the `TWinControl` and `TComponent` ancestor classes. The other group of properties all have an index as well as get and set methods specified. This group of properties includes the ActiveX control properties being encapsulated by the `TOleControl`.

The specialized get and set methods for the encapsulated properties provide the magic that bridges the gap between the ActiveX control properties and the Object Pascal component properties. Notice the read and write methods that get and set properties for every specific type

(such as `GetBoolProp()`, `SetBoolProp()`, `GetStringProp()`, `SetStringProp()`, and so on). Although there are get and set methods for each property type, they all operate similarly. In fact, the following code shows generic get and set methods for `TOleControl` that would work given a property of type *`X`*:

```
function TOleControl.GetXProp(Index: Integer): X;
var
  Temp: TVarData;
begin
  GetProperty(Index, Temp);
  Result := Temp.VX;
end;

procedure TOleControl.SetXProp(Index: Integer; Value: X);
var
  Temp: TVarData;
begin
  Temp.VType := varX;
  Temp.VX := Value;
  SetProperty(Index, Temp);
end;
```

In this code, the index of the property (as indicated by the `index` directive in the properties of the `TCardsCtrl` component) is implicitly passed to the procedures. The variable of type *`X`* is packaged into a `TVarData` (a record that represents a Variant) data record, and those parameters are passed to the `GetProperty()` or `SetProperty()` method of `TOleControl`. Each property of the ActiveX control has a unique index that acts as an identifier. Using this index and the `TVarData` variable `Temp`, `GetProperty()` and `SetProperty()` use OLE Automation to get and set the property values inside the ActiveX control.

If you've worked with other development packages before, you'll appreciate that Delphi provides easy access not only to the ActiveX control's own properties but also to normal `TWinControl` properties and methods. This enables you to use an ActiveX control like other handle-bearing controls in Delphi and makes it possible for you to use object-oriented principles to override the behavior of an ActiveX control by creating customized descendants of ActiveX controls in the Delphi environment.

Using ActiveX Controls in Your Applications

After you link your ActiveX control wrapper into the component library, you've actually fought most of the battle. After an ActiveX control has been placed on the Component Palette, its usage is much the same as that of a regular Delphi component. Figure 7.4 shows the Delphi environment with a `TCardX` focused in the Form Designer. Notice the `TCardX` properties listed in the Object Inspector.

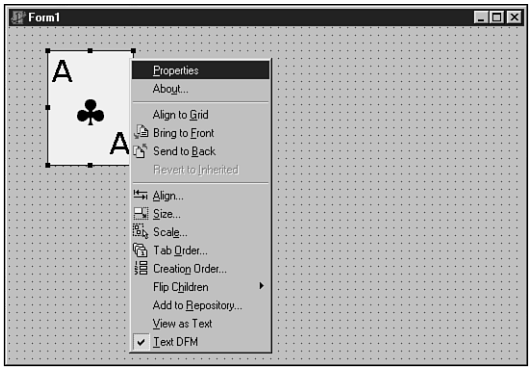

FIGURE 7.4
Working with an ActiveX control in Delphi.

In addition to an ActiveX control's properties being set with the Object Inspector, some ActiveX controls also provide a Properties dialog box that's revealed by selecting the Properties option from the context menu in the Delphi Form Designer. The context menu, also shown in Figure 7.4, is revealed by right-clicking over a particular control. The Properties dialog box actually lives within the ActiveX control; its look, feel, and contents are determined entirely by the control designer. Figure 7.5 shows the Properties dialog box for the `TCardX` ActiveX control.

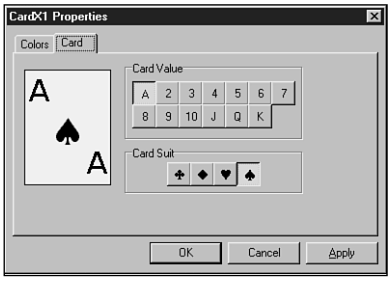

FIGURE 7.5
The `TCardX` *Properties dialog box.*

As you can imagine, this particular control comes equipped with properties that enable you to specify card suit, value, color and picture for card back as well as the standard properties that refer to position, tab order, and so on. The card in Figure 7.4 has its `Value` property set to `1 (Ace)` and its `Suit` property set to `3 (Spades)`.

Shipping ActiveX Control–Equipped Applications

When you're ready to ship your ActiveX control–equipped application, there are some deployment issues to bear in mind as you prepare to send your ActiveX control and associated files to your customers:

- You must ship the OCX or DLL file that contains the ActiveX controls you're using in your application. OCX files, being DLLs, are not linked into your application's executable. Additionally, before the user can use your application, the ActiveX control must be registered in that user's System Registry. ActiveX control registration is discussed in the following section.
- Some ActiveX controls require one or more external DLLs or other files to operate. Check the documentation for your third-party ActiveX controls to determine whether any additional files must be deployed with your ActiveX control. See Chapter 25, "Creating ActiveX Controls," for information on what additional files might need to be deployed along with your Delphi-written controls.
- Many ActiveX controls come with a license file that's required if you want to use the control at design time. This file comes from the ActiveX control vendor, and it prevents your end users from designing applications with ActiveX controls you ship with your applications. You should not ship these LIC files with your application unless you intend for users of your application to use the licensed controls in a development tool and you have the appropriate license for such redistribution.

ActiveX Control Registration

Before an ActiveX control can be used on any system (including those of customers or clients who run your applications), it must be registered with the System Registry. Most commonly, this is accomplished using the `RegSvr32.exe` application, which comes with most versions of Windows. Alternatively, you can use the `TRegSvr.exe` command-line registration utility found in the Delphi `bin` directory. Occasionally, you might want to register the control more transparently to give your application an integrated feel. Luckily, it's not difficult to integrate ActiveX control registration (and unregistration) into your application. Inprise provides the source code for the `TRegSvr` utility as a sample application, and it provides an excellent demonstration for how to register ActiveX servers and type libraries.

BlackJack: An OCX Application Example

The best way to demonstrate how to use an ActiveX control in an application is to show you how to write a useful application that incorporates an ActiveX control. This example uses the `TCardX` ActiveX control; what better way to demonstrate a card control than to make a blackjack game? For the sake of argument, assume that all programmers are high rollers and don't

need to be told the rules of the game (didn't know this book was a comedy, did you?). This way, you can concentrate on the programming job at hand.

As you can imagine, most of the code for this application deals with the logic of the game of blackjack. All the code is provided in the listings later in this chapter; right now, the discussion is narrowed to the individual portions of code that deal directly with managing and manipulating the ActiveX controls. The name of this project is `BJ`; to give you an idea of where the code comes from, Figure 7.6 shows a game of DDG BlackJack in progress.

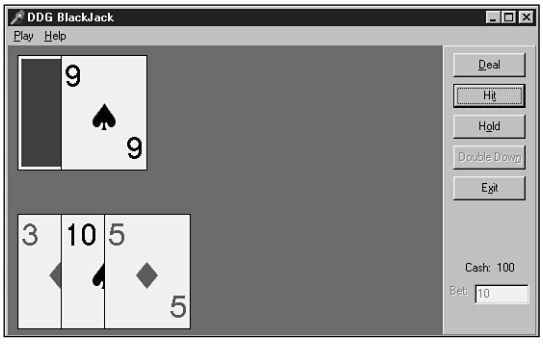

FIGURE 7.6
Playing DDG BlackJack.

The Card Deck

Before writing the game itself, you must first write an object that encapsulates a deck of playing cards. Unlike a real card deck (in which cards are picked from the top of a scrambled deck), this card deck object contains an unscrambled card deck and uses pseudorandom numbers to pick a random card from the unscrambled deck. This is possible because each card has a notion of whether it has been used. This greatly simplifies the shuffle procedure, too, because all the object has to do is set each of the cards to unused. The code for the `PlayCard.pas` unit, which contains the `TCardDeck` object, is shown in Listing 7.2.

LISTING 7.2 The `PlayCard.pas` Unit

```
unit PlayCard;

interface

uses SysUtils, Cards;

type
```

continues

LISTING 7.2 Continued

```
  ECardError = class(Exception);  // generic card exception

  TPlayingCard = record           // represents one card
    Face: TCardValue;             // card face value
    Suit: TCardSuit;              // card suit value
  end;

  { an array of 52 cards representing one deck }
  TCardArray = array[1..52] of TPlayingCard;

  { Object which represents a deck of 52 UNIQUE cards. }
  { This is a scrambled deck of 52 cards, and the      }
  { object keeps track of how far throughout the deck  }
  { the user has picked. }
  TCardDeck = class
  private
    FCardArray: TCardArray;
    FTop: integer;
    procedure InitCards;
    function GetCount: integer;
  public
    property Count: integer read GetCount;
    constructor Create; virtual;
    procedure Shuffle;
    function Draw: TPlayingCard;
  end;

{ GetCardValue returns the numeric value of any card }
function GetCardValue(C: TPlayingCard): Integer;

implementation

function GetCardValue(C: TPlayingCard): Integer;
{ returns a card's numeric value }
begin
  Result := Ord(C.Face) + 1;
  if Result > 10 then Result := 10;
end;

procedure TCardDeck.InitCards;
{ initializes the deck by assigning a unique value/suit combination }
{ to each card. }
var
  i: integer;
```

```
  AFace: TCardValue;
  ASuit: TCardSuit;
begin
  AFace := cvAce;                        // start with ace
  ASuit := csClub;                       // start with clubs
  for i := 1 to 52 do                    // for each card in deck...
  begin
    FCardArray[i].Face := AFace;         // assign face
    FCardArray[i].Suit := ASuit;         // assign suit
    if (i mod 4 = 0) and (i <> 52) then  // every four cards...
      inc(AFace);                        // increment the face
    if ASuit <> High(TCardSuit) then     // always increment the suit
      inc(ASuit)
    else
      ASuit := Low(TCardSuit);
  end;
end;

constructor TCardDeck.Create;
{ constructor for TCardDeck object. }
begin
  inherited Create;
  InitCards;
  Shuffle;
end;

function TCardDeck.GetCount: integer;
{ Returns a count of unused cards }
begin
  Result := 52 - FTop;
end;

procedure TCardDeck.Shuffle;
{ Re-mixes cards and sets top card to 0. }
var
  i: integer;
  RandCard: TPlayingCard;
  RandNum: integer;
begin
  for i := 1 to 52 do
  begin
    RandNum := Random(51) + 1;            // pick random number
    RandCard := FCardArray[RandNum];      // swap next card with
    FCardArray[RandNum] := FCardArray[i]; // random card in deck
    FCardArray[i] := RandCard;
  end;
```

continues

LISTING 7.2 Continued

```
  FTop := 0;
end;

function TCardDeck.Draw: TPlayingCard;
{ Picks the next card from the deck. }
begin
  inc(FTop);
  if FTop = 53 then
    raise ECardError.Create('Deck is empty');
  Result := FCardArray[FTop];
end;

initialization
  Randomize;                    // must seed random number generator
end.
```

The Game

The DDG BlackJack game's interaction with the `TCardX` object occurs primarily in three procedures. One procedure, called `Hit()`, is called when the player chooses to accept another card. Another procedure, `DealerHit()`, is called when the dealer wants another card. Finally, the `FreeCards()` procedure is called to dispose of all the cards on the screen to prepare for another hand to be dealt.

The `Hit()` procedure works with the `TCardX` ActiveX control uniquely in a couple ways. First, it creates all the controls dynamically rather than using controls dropped from the Component Palette. Also, it never uses an instance variable of type `TCardX`—instead, it takes advantage of a `with..do` construct to create and use the object in one step. The following code shows the `Hit()` procedure:

```
procedure TMainForm.Hit;
{ Player hit }
begin
  CurCard := CardDeck.Draw;              // draw card
  with TCardX.Create(Self) do            // create card OCX
  begin
    Left := NextPlayerPos;               // set position
    Top := PYPos;
    Suit := Ord(CurCard.Suit);           // set suit
    Value := Ord(CurCard.Face);          // set value
    Parent := Self;                      // assign parent
    Inc(NextPlayerPos, Width div 2);     // track position
    Update;                              // Display card
```

```
  end;
  DblBtn.Enabled := False;               // hit disables double down
  if CurCard.Face = cvAce then PAceFlag := True;  // set ace flag
  Inc(PlayerTotal, GetCardValue(CurCard));        // keep running total
  PlayLbl.Caption := IntToStr(PlayerTotal);        // cheat
  if PlayerTotal > 21 then                        // track bust
  begin
    ShowMessage('Busted!');
    ShowFirstCard;
    ShowWinner;
  end;
end;
```


In this procedure, a random card, called `CurCard`, is drawn from a `TCardDeck` object called `CardDeck`. A `TCardX` ActiveX control is then created, and property values are assigned. `NextPlayerPos` is a variable that keeps track of the position on the X-axis for the next card. `PYPos` is a constant that dictates the Y-axis position of the player hand. The `Suit` and `Value` properties are assigned values that correspond to the `Suit` and `Face` of `CurCard`. `MainForm` is assigned to be the `Parent` of the control, and the `NextPlayerPos` variable is incremented by half the width of a card. After all that, the `PlayerTotal` variable is incremented by the card value to keep track of the player's score.

The `DealerHit()` procedure works similarly to the `Hit()` procedure. The code from that procedure is shown here:

```
procedure TMainForm.DealerHit(CardVisible: Boolean);
{ Dealer takes a hit }
begin
  CurCard := CardDeck.Draw;              // dealer draws a card
  with TCardX.Create(Self) do            // create the ActiveX control
  begin
    Left := NextDealerPos;               // place card on form
    Top := DYPos;
    Suit := Ord(CurCard.Suit);           // assign suit
    FaceUp := CardVisible;
    Value := Ord(CurCard.Face);          // assign face
    Parent := Self;                      // assign parent for OCX
    Inc(NextDealerPos, Width div 2);     // track where to place next card
    Update;                              // Display card
  end;
  if CurCard.Face = cvAce then DAceFlag := True;   // set Ace flag
  Inc(DealerTotal, GetCardValue(CurCard));         // keep count
  DealLbl.Caption := IntToStr(DealerTotal);        // cheat
  if DealerTotal > 21 then                         // track dealer bust
    ShowMessage('Dealer Busted!');
end;
```

This method accepts a Boolean parameter called `CardVisible`, which indicates whether the card should be dealt face up. This is because blackjack rules dictate that the dealer's first card must remain face down until the player has chosen to hold or has busted. Observing this rule, the first call to `DealerHit()` will result in `False` being passed in `CardVisible`.

The `FreeCards()` procedure is responsible for removing all the `TCardX` controls on the main form. Because the application doesn't keep an array or a bunch of variables of type `TCardX` around to manage the cards on the screen, this procedure iterates through the form's `Controls` array property looking for elements of type `TCardX`. When a control of that type is found, its `Free` method is called to remove it from memory. The trick here is to be sure to go *backward* through the array. If you don't go backward, you run the risk of changing the order of controls in the array while you're traversing the array, which can cause errors. The code for the `FreeCards` procedure is shown here:

```
procedure TMainForm.FreeCards;
{ frees all AX Ctl cards on the screen }
var
  i: integer;
begin
  for i := ControlCount - 1 downto 0 do  // go backward!
    if Controls[i] is TCardX then
      Controls[i].Free;
end;
```

That completes the explanation of the main portions of the code that manipulates the ActiveX controls. The complete listing for `Main.pas`, the main unit for this application, is shown in Listing 7.3.

LISTING 7.3 The `Main.pas` Unit for the `BJ` Project

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  OleCtrls, Card_TLB, Cards, PlayCard, StdCtrls, ExtCtrls, Menus;

type
  TMainForm = class(TForm)
    Panel1: TPanel;
    MainMenu1: TMainMenu;
    Play1: TMenuItem;
    Deal1: TMenuItem;
    Hit1: TMenuItem;
```

```
    Hold1: TMenuItem;
    DoubleDown1: TMenuItem;
    N1: TMenuItem;
    Close1: TMenuItem;
    Help1: TMenuItem;
    About1: TMenuItem;
    Panel2: TPanel;
    Label3: TLabel;
    CashLabel: TLabel;
    BetLabel: TLabel;
    HitBtn: TButton;
    DealBtn: TButton;
    HoldBtn: TButton;
    ExitBtn: TButton;
    BetEdit: TEdit;
    DblBtn: TButton;
    CheatPanel: TPanel;
    DealLbl: TLabel;
    PlayLbl: TLabel;
    Label4: TLabel;
    Label6: TLabel;
    Cheat1: TMenuItem;
    N2: TMenuItem;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure About1Click(Sender: TObject);
    procedure Cheat1Click(Sender: TObject);
    procedure ExitBtnClick(Sender: TObject);
    procedure DblBtnClick(Sender: TObject);
    procedure DealBtnClick(Sender: TObject);
    procedure HitBtnClick(Sender: TObject);
    procedure HoldBtnClick(Sender: TObject);
  private
    CardDeck: TCardDeck;
    CurCard: TPlayingCard;
    NextPlayerPos: integer;
    NextDealerPos: integer;
    PlayerTotal: integer;
    DealerTotal: integer;
    PAceFlag: Boolean;
    DAceFlag: Boolean;
    PBJFlag: Boolean;
    DBJFlag: Boolean;
    DDFlag: Boolean;
    Procedure Deal;
    procedure DealerHit(CardVisible: Boolean);
```

continues

LISTING 7.3 Continued

```
    procedure DoubleDown;
    procedure EnableMoves(Enable: Boolean);
    procedure FreeCards;
    procedure Hit;
    procedure Hold;
    procedure ShowFirstCard;
    procedure ShowWinner;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses AboutU;

const
  PYPos = 175;        // starting y pos for player cards
  DYPos = 10;         // ditto for dealer's cards

procedure TMainForm.FormCreate(Sender: TObject);
begin
  CardDeck := TCardDeck.Create;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  CardDeck.Free;
end;

procedure TMainForm.About1Click(Sender: TObject);
{ Creates and invokes about box }
begin
  with TAboutBox.Create(Self) do
    try
      ShowModal;
    finally
      Free;
    end;
end;

procedure TMainForm.Cheat1Click(Sender: TObject);
begin
```

```
  Cheat1.Checked := not Cheat1.Checked;
  CheatPanel.Visible := Cheat1.Checked;
end;

procedure TMainForm.ExitBtnClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.DblBtnClick(Sender: TObject);
begin
  DoubleDown;
end;

procedure TMainForm.DealBtnClick(Sender: TObject);
begin
  Deal;
end;

procedure TMainForm.HitBtnClick(Sender: TObject);
begin
  Hit;
end;

procedure TMainForm.HoldBtnClick(Sender: TObject);
begin
  Hold;
end;

procedure TMainForm.Deal;
{ Deals a new hand for dealer and player }
begin
  FreeCards;                        // remove any cards from screen
  BetEdit.Enabled := False;         // disable bet edit ctrl
  BetLabel.Enabled := False;        // disable bet label
  if CardDeck.Count < 11 then       // reshuffle deck if < 11 cards
  begin
    Panel1.Caption := 'Reshuffling and dealing...';
    CardDeck.Shuffle;
  end
  else
    Panel1.Caption := 'Dealing...';
  Panel1.Show;                      // show "dealing" panel
  Panel1.Update;                    // make sure it's visible
  NextPlayerPos := 10;              // set horiz position of cards
  NextDealerPos := 10;
```

continues

LISTING 7.3 Continued

```
  PlayerTotal := 0;                 // reset card totals
  DealerTotal := 0;
  PAceFlag := False;                // reset flags
  DAceFlag := False;
  PBJFlag := False;
  DBJFlag := False;
  DDFlag := False;
  Hit;                              // hit player
  DealerHit(False);                 // hit dealer
  Hit;                              // hit player
  DealerHit(True);                  // hit dealer
  Panel1.Hide;                      // hide panel
  if (PlayerTotal = 11) and PAceFlag then
    PBJFlag := True;                // check player blackjack
  if (DealerTotal = 11) and DAceFlag then
    DBJFlag := True;                // check dealer blackjack
  if PBJFlag or DBJFlag then        // if a blackjack occurred
  begin
    ShowFirstCard;                  // flip dealer's card
    ShowMessage('Blackjack!');
    ShowWinner;                     // determine winner
  end
  else
    EnableMoves(True);              // enable hit, hold double down
end;

procedure TMainForm.DealerHit(CardVisible: Boolean);
{ Dealer takes a hit }
begin
  CurCard := CardDeck.Draw;           // dealer draws a card
  with TCardX.Create(Self) do         // create the ActiveX control
  begin
    Left := NextDealerPos;            // place card on form
    Top := DYPos;
    Suit := Ord(CurCard.Suit);        // assign suit
    FaceUp := CardVisible;
    Value := Ord(CurCard.Face);       // assign face
    Parent := Self;                   // assign parent for AX Ctl
    Inc(NextDealerPos, Width div 2);  // track where to place next card
    Update;                           // show card
  end;
  if CurCard.Face = cvAce then DAceFlag := True;   // set Ace flag
  Inc(DealerTotal, GetCardValue(CurCard));         // keep count
  DealLbl.Caption := IntToStr(DealerTotal);        // cheat
  if DealerTotal > 21 then                         // track dealer bust
```

```
    ShowMessage('Dealer Busted!');
end;

procedure TMainForm.DoubleDown;
{ Called to double down on dealt hand }
begin
  DDFlag := True;               // set double down flag to adjust bet
  Hit;                          // take one card
  Hold;                         // let dealer take his cards
end;

procedure TMainForm.EnableMoves(Enable: Boolean);
{ Enables/disables moves buttons/menu items }
begin
  HitBtn.Enabled := Enable;             // Hit button
  HoldBtn.Enabled := Enable;            // Hold button
  DblBtn.Enabled := Enable;             // Double down button
  Hit1.Enabled := Enable;               // Hit menu item
  Hold1.Enabled := Enable;              // Hold menu item
  DoubleDown1.Enabled := Enable;        // Double down menu item
end;

procedure TMainForm.FreeCards;
{ frees all AX Ctl cards on the screen }
var
  i: integer;
begin
  for i := ControlCount - 1 downto 0 do  // go backward!
    if Controls[i] is TCardX then
      Controls[i].Free;
end;

procedure TMainForm.Hit;
{ Player hit }
begin
  CurCard := CardDeck.Draw;          // draw card
  with TCardX.Create(Self) do        // create card AX Ctl
  begin
    Left := NextPlayerPos;           // set position
    Top := PYPos;
    Suit := Ord(CurCard.Suit);       // set suit
    Value := Ord(CurCard.Face);      // set value
    Parent := Self;                  // assign parent
    Inc(NextPlayerPos, Width div 2); // track position
    Update;                          // Display card
  end;
```

continues

LISTING 7.3 Continued

```
  DblBtn.Enabled := False;               // hit disables double down
  if CurCard.Face = cvAce then PAceFlag := True;  // set ace flag
  Inc(PlayerTotal, GetCardValue(CurCard));        // keep running total
  PlayLbl.Caption := IntToStr(PlayerTotal);       // cheat
  if PlayerTotal > 21 then                        // track bust
  begin
    ShowMessage('Busted!');
    ShowFirstCard;
    ShowWinner;
  end;
end;

procedure TMainForm.Hold;
{ Player holds. This procedure allows dealer to draw cards. }
begin
  EnableMoves(False);
  ShowFirstCard;                  // show dealer card
  if PlayerTotal <= 21 then       // if player hasn't busted...
  begin
    if DAceFlag then              // if dealer has an Ace...
    begin
      { Dealer must hit soft 17 }
      while (DealerTotal <= 7) or ((DealerTotal >= 11) and
        (DealerTotal < 17)) do
        DealerHit(True);
    end
    else
      // if no Ace, keep hitting until 17 is reached
      while DealerTotal < 17 do DealerHit(True);
  end;
  ShowWinner;                     // Determine winner
end;

procedure TMainForm.ShowFirstCard;
var
  i: integer;
begin
  // make sure all cards are face-up
  for i := 0 to ControlCount - 1 do
    if Controls[i] is TCardX then
    begin
      TCardX(Controls[i]).FaceUp := True;
      TCardX(Controls[i]).Update;
    end;
```

```
end;

procedure TMainForm.ShowWinner;
{ Determines winning hand }
var
  S: string;
begin
  if DAceFlag then                      // if dealer has an Ace...
  begin
    if DealerTotal + 10 <= 21 then      // figure best hand
      inc(DealerTotal, 10);
  end;
  if PACeFlag then                      // if player has an Ace...
  begin
    if PlayerTotal + 10 <= 21 then      // figure best hand
      inc(PlayerTotal, 10);
  end;
  if DealerTotal > 21 then              // set score to 0 if busted
    DealerTotal := 0;
  if PlayerTotal > 21 then
    PlayerTotal := 0;
  if PlayerTotal > DealerTotal then     // if player wins...
  begin
    S := 'You win!';
    if DDFlag then                      // pay 2:1 on double down
      CashLabel.Caption := IntToStr(StrToInt(CashLabel.Caption) +
        StrToInt(BetEdit.Text) * 2)
    else                                // pay 1:1 normally
      CashLabel.Caption := IntToStr(StrToInt(CashLabel.Caption) +
        StrToInt(BetEdit.Text));
    if PBJFlag then                     // pay 1.5:1 on blackjack
      CashLabel.Caption := IntToStr(StrToInt(CashLabel.Caption) +
        StrToInt(BetEdit.Text) div 2)
  end
  else if DealerTotal > PlayerTotal then    // if dealer wins...
  begin
    S := 'Dealer wins!';
    if DDFlag then                      // lose 2x on double down
      CashLabel.Caption := IntToStr(StrToInt(CashLabel.Caption) -
        StrToInt(BetEdit.Text) * 2)
    else                                // normal loss
      CashLabel.Caption := IntToStr(StrToInt(CashLabel.Caption) -
        StrToInt(BetEdit.Text));
  end
  else
    S := 'Push!';                       // push, no one wins
```

continues

LISTING 7.3 Continued

```
  if MessageDlg(S + #13#10'Do you want to play again with the same bet?',
    mtConfirmation, [mbYes, mbNo], 0) = mrYes then
    Deal;
  BetEdit.Enabled := True;             // allow bet to change
  BetLabel.Enabled := True;
end;

end.
```

Invoking an ActiveX Control Method

In Listing 7.3, you might have noticed that the main form contains a method that creates and displays an About dialog box. Figure 7.7 shows what this About dialog box looks like when invoked.

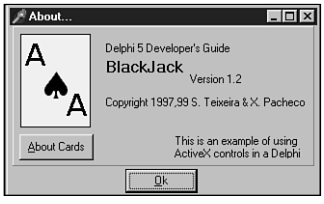

FIGURE 7.7
DDG BlackJack's About dialog box.

This About dialog box is special because it contains a button that, when selected, shows an About box for the `CardX` ActiveX control by calling its `AboutBox()` method. The About box for the `CardX` control is shown in Figure 7.8.

The code that accomplishes this task follows. Looking ahead, the same technique used to make `vTable` calls to OLE Automation servers in Chapter 23, "COM and ActiveX," is used here.

```
procedure TAboutBox.CardBtnClick(Sender: TObject);
begin
  Card.AboutBox;
end;
```

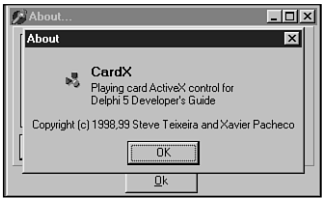

FIGURE 7.8
The `Cards` *ActiveX control About dialog box.*

Summary

After reading this chapter, you should understand all the important aspects of using ActiveX controls in the Delphi environment. You learned about integrating an ActiveX control into Delphi, how the Object Pascal ActiveX control wrapper works, how to deploy an ActiveX control–equipped application, how to register an ActiveX control, and how to incorporate ActiveX controls into an application. Because of their market presence, ActiveX controls often can offer a blast of instant productivity. However, because ActiveX controls have some disadvantages, also remember to look for native VCL components when shopping for controls.