

## CHAPTER

# 16

# Windows Applications and Packaging

---

**case ►** “I love using Java,” you say to Lynn Greenbrier, your mentor at Event Handlers Incorporated. “But a few weeks ago, the marketing department came to me with a bunch of new programming requests from corporate clients.”

“Tell me more,” says Lynn.

“These clients don’t want to use the Internet to supply us with event information,” you say. “They all use Windows 95, 98, or NT, and they want a stand-alone application they can use for event planning. I wrote a Java program, but it doesn’t look like a real Windows application. I had to re-create many of the standard Windows features, and the program ran very slowly. So, while I love Java, I’m feeling a bit frustrated.”

Lynn Greenbrier smiles and says, “I’ve got just the thing to solve your problems. Visual J++ has tools you can use to access Windows operating system functionality. Those tools will let you use Java to write real Windows applications. To start, let me tell you about Windows Foundation Classes.”

## Previewing the Windows Party Planner Application

In this chapter, you will use Java to create a Windows application that is similar to the Party Planner Applet you created in Chapter 11. You will create a Windows-based Party Planner program so you can see the differences between programs created with the Abstract Windows Tool Kit (AWT) and programs created as Windows applications. Like the applet in Chapter 11, the Event Handlers Incorporated Windows application lets a user determine the price of an event based on several event choices. Users can select some options in any combination (serve only cocktails, serve only dinner, serve both cocktails and dinner, or serve nothing). For other options, such as the entrée to serve for dinner or the entertainment selection, only one choice is allowed. You need check boxes, lists, and radio buttons to accommodate these different types of selections. The Chap16PartyPlanner application incorporates several such devices, which you can use now. You will create a similar application in this chapter.

### To preview the Windows Party Planner application:

- 1 Using Windows Explorer, go the Chapter.16 folder on your Student Disk and run the **WindowsPartyPlanner.exe** file. After a few moments, you will see the Windows Party Planner window shown in Figure 16-1.



Figure 16-1: Windows Party Planner application

- 2** You can use the Windows Party Planner application to plan an imaginary event by choosing whether to serve cocktails or dinner (or both or neither). Use the program now and observe how the event's price changes as you make selections. If you choose to serve dinner, you can select one of three main courses. You also can select from a list of entertainment choices and party favors. The event's price changes as you make each selection.
- 3** Close the application by clicking the **Exit** button.

# SECTION A

## objectives

In this section you will learn:

- About Windows architecture and the Win32 API
- How to use J/Direct
- About native applications and architecturally neutral applications
- About Windows Foundation Classes (WFCs) for Java
- How to create Windows applications using WFCs
- About WFC forms
- About WFC controls

# J/Direct and Windows Foundation Classes for Java

## Windows Architecture and the Win32 API

In this chapter, you will learn how to use Visual J++ and the Java programming language to create Windows applications. Using the Java programming language to create Windows applications allows you to use your knowledge of Java to harness the features of Windows operating systems such as prebuilt dialog boxes, buttons, and other Windows elements. To create Windows applications with Java, you first need to understand how Windows operating systems are designed.

There are two types of Windows operating systems: 16-bit and 32-bit. The older Windows 3.1 operating system is **16-bit**. Current **32-bit** Windows operating systems are Windows NT, Windows 95, Windows 98, and Windows CE. The term **bit** refers to the width of a computer microprocessor's data bus. You can think of the data bus as the number of roads leading into a microprocessor. The wider the data bus, the more information can be sent simultaneously to the microprocessor.

.....  
**Visual J++ is a 32-bit application designed for 32-bit Windows operating systems.**  
.....

The actual speed with which information is processed also depends on several other factors, including the microprocessor architecture (80386, i486, Pentium, and so on) and the megahertz at which the microprocessor operates. The 80386 and i486 microprocessors have 32-bits wide data buses, whereas the Pentium family of microprocessors has 64-bits wide data buses. The ability to take advantage of data bus width depends on the operating system or application that is accessing the microprocessor. For example, Windows 3.1 can operate on 80386, i486, and Pentium



microprocessors, but can use only 16 bits of each data bus, even though the 80386 and i486 microprocessors are 32-bits wide and the Pentium chip is 64-bits wide.

All 32-bit Windows operating systems share a common application programming interface known as the Win32 API. An **application programming interface (API)** is a library of methods and attributes that allows programmers to access the features and functionality of an application or operating system. Windows operating systems are actually complex applications written in the C and C++ programming languages. The Win32 API allows you to access and use standard Windows GUI features such as the Open and Save dialog boxes, and controls such as buttons, scrollbars, and list boxes in new applications that you write. You have already worked with similar GUI components in the AWT.

Using the common functionality available in the Win32 API helps to maintain a consistent look for Windows-based applications. In the same manner that the AWT saves time when writing Java applications, the Win32 API minimizes development time for Windows applications, since you use preexisting components without having to create them from scratch. Win32 API functions are grouped in the following categories:

- Window Management
- Window Controls
- Shell Features
- Graphics Device Interface
- System Services
- International Features
- Network Services



.....  
**APIs represent a more advanced form of object-oriented programming. Just as methods and attributes of an individual object are exposed through inheritance or an interface, libraries of methods and attributes are exposed through an API.**  
 .....

## Using J/Direct to Access the Win32 API

Many programming languages, including Visual Basic and Visual C++, directly access the functionality in the Win32 API. The Java programming language, however, does not directly access Win32 API functions—such as standard Windows GUI features. Visual J++ has a special utility called J/Direct that allows you to access the Win32 API. **J/Direct** translates the Win32 API syntax into a format that can be used with Java. With J/Direct, you can place calls to the Win32 API directly into your code.

To use J/Direct to call Win32 API functions, you include the following:

- A Win32 API function method declaration
- Special comments that contain a directive to locate a dynamic-link library (DLL) for the Win32 API function

The code in Figure 16-2 calls the Win32 API `MessageBox` function, which displays a simple Windows dialog box. The dialog box generated by the program in Figure 16-2 is shown in Figure 16-3.

```

public class MessageBoxApp
{
    public static void main(String args[])
    {
        MessageBox(0, "This is a message box.",
            "Message Box Title", 0);
    }
    /**
     * @dll.import("USER32", auto)
     */
    public static native int MessageBox(int hWnd, String lpText,
        String lpCaption, int uType);
}

```

Figure 16-2: MessageBoxApp using J/Direct to call the Win32 MessageBox function



Figure 16-3: Output of the Win32 MessageBoxApp

The last statement in the code in Figure 16-2 is the MessageBox method declaration. The MessageBox method declaration includes the **native modifier**, which informs Java that the method is implemented in another programming language—in this case, the programming language is the Win32 API. Unlike most methods with which you have worked, the MessageBox method declaration ends in a semicolon, rather than being followed by curly brackets containing the method's statements. Since the method is implemented outside Java, you do not need to create the method body.



By convention, you add J/Direct calls to the end of a class file.

Dynamic-link libraries contain the methods and other components in the Win32 API, and are files with an extension of .dll. When you use a J/Direct call in a Java program, the program locates the specific DLL containing the object you are calling by using a directive. A **directive** tells the compiler in which DLL a method is located. Immediately above the MessageBox method declaration in Figure 16-2 are comments containing the directive for the MessageBox function. (You first learned about comments in Chapter 2.) As you know, the compiler ignores text placed within comments. Unlike other types of text that are placed within comments, directives are not ignored by the compiler. You usually place directives immediately above a J/Direct call. There are three types of directives: @dll.import for declaring functions, @dll.struct for declaring structures, and @dll.structmap for declaring fixed-size

Strings and arrays embedded in structures. The `@dll.import` directive in Figure 16-2 instructs the compiler to look for the `MessageBox` function in `USER32.DLL`.



Structures are used in the C/C++ programming languages and are similar to classes.

The `MessageBox` function is available in two versions: ANSI or Unicode. The auto modifier in the `@dll.import` directive instructs the compiler to use the optimal version of the `MessageBox` function, ANSI or Unicode, depending on the version of Windows being used.

If you know the correct syntax for J/Direct calls and directives that you want to include in your Java application, then you can type them directly into a Java file. However, the syntax for J/Direct calls and directives is unique for the specific API function being called. An easier method of including J/Direct calls in your code is to use the Visual J++ J/Direct Call Builder dialog box. The J/Direct Call Builder dialog box inserts the appropriate syntax and directives for the J/Direct calls you want to use in your program. Figure 16-4 displays an example of the J/Direct Call Builder dialog box.

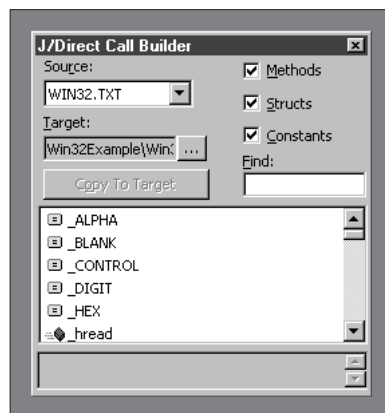


Figure 16-4: J/Direct Call Builder dialog box

The J/Direct Call Builder dialog box contains several options to assist you in locating a J/Direct call. The default option in the `Source` drop-down list box is `WIN32.TXT`, which lists methods, structures, and constants contained in standard Win32 API DLLs. You can click the `Methods` check box, `Structs` check box, or `Constants` check box to filter the visible elements. The `Target` text box displays the class to which the selected J/Direct call will be added. By default, a class named `Win32` will be added to your project to contain J/Direct calls. Using a separate class to contain J/Direct calls is good practice, particularly if you have multiple classes in your project, all of which may need to access J/Direct calls.

Next you will create a Java console application project and use the J/Direct Call Builder to add J/Direct calls to the program. In the following exercise, do not worry about the syntax for the arguments in the `MessageBox` and `MessageBeep` Win32 API calls. Win32 API programming is a complex topic and beyond the scope of this text. The only purpose of the exercise is to show how a Java

application can use Win32 API calls with J/Direct. Later in this section, you will learn an easier method for working with J/Direct calls in Visual J++.

#### To create a Java console application with J/Direct calls:

- 1 If necessary, start Visual J++, then create a new console application project named **Win32Example**. Save the **Win32Example** project folder in the Chapter.16 folder on your Student Disk.
- 2 Rename the default Class1.java file as **Win32Example.java**, then open the file in the Text Editor window.
- 3 Replace the Class1 class name in the `public class Class1` line with **Win32Example**.
- 4 Point to **Other Windows** on the **View** menu, then select **J/Direct Call Builder**.
- 5 Although placing J/Direct calls in a separate class is good programming practice, for this exercise you will add the J/Direct calls to the Win32Example class. Click the **ellipsis (...)** next to the Target text box. The Select class dialog box displays, as shown in Figure 16-5.

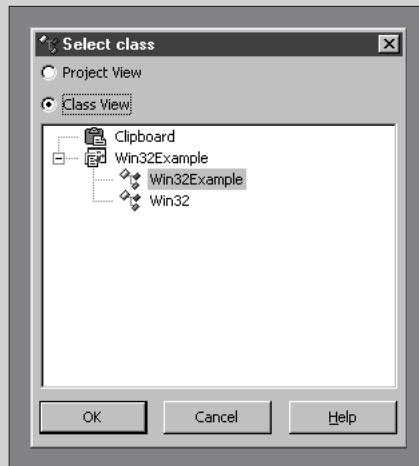



Figure 16-5: Select class dialog box

- 6 In the Select class dialog box, click the **Win32Example** class, and then select the **OK** button. The Select class dialog box closes, and the J/Direct Call Builder dialog box redisplayes with the Win32Example class visible in the Target text box.

help

The text in the Target text box will read "Win32Example\Win32Example." The text to the left of the backward slash (\) represents the name of the project and the text to the right of the backward slash represents the class. In this case, both the project and class are named "Win32Example."

- 7 Place your cursor in the **Find** text box and begin typing **MessageBox**. The list of J/Direct calls scrolls to match the typed text. When you see **MessageBox** in the list, click it, then click the **Copy To Target** button. The **MessageBox** J/Direct call and directive are added to the bottom of the class file.
- 8 Repeat Step 7 to add the **MessageBeep** call to the **Win32Example** class.
- 9 Close the J/Direct Call Builder dialog box by clicking the **Close** button .
- 10 In the **main()** method, replace the `// TODO: Add initialization code here` comment with the following code, which creates two **Strings** and adds the **Win32API** **MessageBox()** and **MessageBeep()** functions:

```
String messageBoxTitle = "Event Planners";  
String messageBoxText = "Thank you for planning with us!";  
MessageBox(0, messageBoxText, messageBoxTitle, 0);  
MessageBeep(0);
```

- 11 In the **Win32Example** properties window, deselect the **Launch as a console application** check box to run the **Win32Example** file using **WJVIEW**. You are selecting **WJVIEW** since the application will not require a separate console application window, as is created with **JVIEW**.
- 12 Build and save the project, then execute the program by selecting **Start** from the **Debug** menu. A dialog box appears as shown in Figure 16-6.

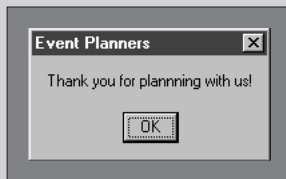


Figure 16-6: Output of the **Win32Example** program

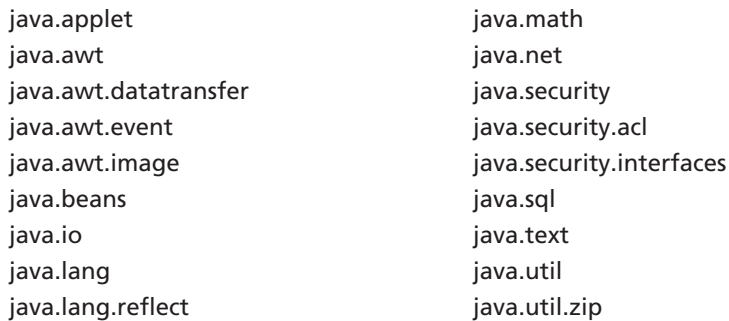
- 13 Click the **OK** button. You should hear an audible beep after the dialog box closes.

## Native Applications versus Architecturally Neutral Applications

The programs you have created in previous chapters have been true Java applications or applets. Recall from Chapter 1 that Java programs are architecturally neutral. You can use the Java programming language to write a program that will run on any platform for which there is a Java Virtual Machine (VM). The **Win32Example** program, however, can run only on a 32-bit Windows platform. Adding J/Direct calls to the **Win32Example** program removes its architectural neutrality—it is no longer a true Java program since it can run only on 32-bit Windows platforms. While you are giving up architectural neutrality, you are gaining the ability to use your Java programming skills to create Windows applications.

Throughout this book, you have imported Java packages into the applications and applets you created. As you learned in Chapter 4, packages are related groups of classes and class members that are grouped together into a single library. Examples of the packages you have used include the `java.applet` package for working with applets, and the `java.awt` (Abstract Windows Toolkit) package, which contains commonly used components such as labels, menus, and buttons. Many Java packages contain classes that are available only if you explicitly import a package into your program. For example, to include the `java.awt` package in a program, you must include the statement `import java.awt.*` before the declaration of a class header.

The packages that comprise the Java programming language are considered to be part of the Java API, very similar to the way internal Windows functions and features are part of the Win32 API. If you create a program using only the standard Java packages that are part of the original Java API developed by Sun Microsystems, then the program will run on any platform for which there is a Java VM. Packages that are part of the Java API are listed in Figure 16-7.



<code>java.applet</code>	<code>java.math</code>
<code>java.awt</code>	<code>java.net</code>
<code>java.awt.datatransfer</code>	<code>java.security</code>
<code>java.awt.event</code>	<code>java.security.acl</code>
<code>java.awt.image</code>	<code>java.security.interfaces</code>
<code>java.beans</code>	<code>java.sql</code>
<code>java.io</code>	<code>java.text</code>
<code>java.lang</code>	<code>java.util</code>
<code>java.lang.reflect</code>	<code>java.util.zip</code>

**Figure 16-7:** Java API Packages

In contrast to Java API packages, the Java VM cannot use components that exist within an individual operating system. You must rely on the internal capabilities of the Java programming language and the packages listed in Figure 16-7. For example, in a true Java application, to create a dialog box that opens files, you must use the components in the `java.awt` package to build the dialog box. The final result may resemble a Windows Open dialog box, but it will not be a *true* Windows dialog box. A true Windows Open dialog box is part of the Win32 API.

True Java applications do not run as fast as applications that are native to a specific operating system. This difference in speeds is a result of the Java VM. While the Java VM allows Java applications to be architecturally neutral, it adds an additional layer that an application must pass through to run. The flowchart in Figure 16-8 shows the additional Java VM layer through which an application must pass to work with a particular operating system. Figure 16-9 shows the direct access to an operating system that a native application has.



Figure 16-8: Java VM program execution

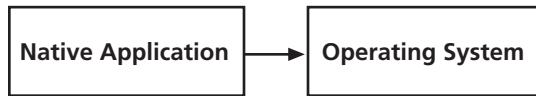


Figure 16-9: Native program execution

J/Direct allows you to include native Win32 API calls within a Java application. These J/Direct calls bypass the Java VM and directly access the Windows operating system, similar to the way native programs access an operating system. Since they can run only on 32-bit Windows platforms, programs that use J/Direct are no longer true Java programs. The trade-off is that J/Direct programs execute faster since they bypass the additional Java VM layer.

## Windows Foundation Classes for Java

Creating Windows applications requires a thorough knowledge of Win32 API programming. Although J/Direct helps you access the Win32 API features, using it is complex and requires familiarity with the Win32 API. To make the task of creating Windows applications with Visual J++ easier, Microsoft created Windows Foundation Classes for Java. **Windows Foundation Classes (WFCs)** are packages of J/Direct calls that are translated into the Java programming language. You still access the same functions that you access with J/Direct calls, but the complicated J/Direct calls and directives are unnecessary. Instead, you import a WFC package into a class and work with the package's components, similar to how you work with Java API packages. WFCs allow you to access the Win32 API directly and eliminate the need to understand Win32 API programming. The WFC packages are listed in Figure 16-10.



For experienced C++ programmers, WFCs are comparable to Microsoft Foundation Classes (MFCs), which provide a C++ language API that accesses the Windows native C-language API.

Package	Description
com.ms.wfc.app	Contains classes for application and Thread control and access to the Clipboard and Registry.
com.ms.wfc.core	Provides core WFC functionality; this package should be included in every WFC program.

Figure 16-10: WFC packages

Package	Description
com.ms.wfc.data	Contains data access classes using the ActiveX Data Objects (ADO) object model. Also contains classes that use ADO to access databases.
com.ms.wfc.html	Contains Java classes for accessing Dynamic HTML (DHTML).
com.ms.wfc.io	Provides classes for working with file input and output.
com.ms.wfc.ui	Contains user interface classes for controls such as buttons and labels.
com.ms.wfc.util	Contains miscellaneous utility classes.

Figure 16-10: WFC packages (continued)

Java programs using WFCs must include the following:

- The `import com.ms.wfc.core.*;` statement
- Import statements for other packages you want to use in your program
- Classes that are extended from parent classes in the WFC packages
- Calls to the specific methods and functions within the imported WFC packages

Figure 16-11 shows a modified version of the `MessageBoxApp` program from Figure 16-2 that uses WFCs instead of a `J/Direct` call to call the Win32 API `MessageBox` function. The program in Figure 16-11 creates the same dialog box as the program in Figure 16-2, but does not require the `J/Direct` call and directive. Instead, it imports the required `com.ms.wfc.core` package, which contains the `MessageBox` class, and uses the `show()` method of the `MessageBox` class to display the dialog box.

```
import com.ms.wfc.core.*;
import com.ms.wfc.ui.*;
class MessageBoxApp
{
    public static void main(String args[])
    {
        MessageBox.show("This is a message box",
                        "Message Box Title", MessageBox.OK);
    }
}
```

Figure 16-11: `MessageBoxApp` calling the Win32 `MessageBox` function with WFCs

The `MessageBox` class contains three constructors: the text to be displayed in the `MessageBox`, the dialog box title, and the dialog box style. The `MessageBox.OK` field determines the style of the dialog box; it displays a simple dialog box containing an OK button. You can create a `MessageBox` without the title and style constructors by typing `MessageBox.show("Text String");`. The title bar of a `MessageBox`

created without the title constructor is blank. The style of a `MessageBox` created without the style constructor defaults to the `MessageBox.OK` field. The `show()` method is the only method available in the `MessageBox` class.



.....  
 The `MessageBox` Fields topic in Visual J++ on-line help lists other style fields that can be used with the `MessageBox` class.  
 .....

Next you will modify the `Win32Example` program so that it uses both a WFC package and `J/Dirct` call.

**To modify the `Win32Example` program so that it uses both a WFC package and a `J/Dirct` call:**

- 1** Return to the `Win32Example.java` file in the Text Editor window and save the file as **`Win32Example2.java`**.
- 2** Replace the `Win32Example` class name in the `public class Win32Example` header with **`Win32Example2`**.
- 3** Replace the default comments above the class header with the following import statements:  

```
import com.ms.wfc.core.*;
import com.ms.wfc.ui.*;
import java.awt.*;
```
- 4** Delete the comments containing the directive and `J/Dirct` `MessageBox` call located at the bottom of the class file.
- 5** Replace the statement that reads `MessageBox(0, messageBoxText, messageBoxTitle, 0);` with **`MessageBox.show(messageBoxText, messageBoxTitle, MessageBox.OK);`**.
- 6** Set the project properties to run the **`Win32Example2`** file instead of the `Win32Example` file. Rebuild and save the project, then run the program by selecting **Start** from the **Debug** menu. The WFC package generates the `MessageBox`. A `J/Dirct` call creates the audible beep that sounds when you click the **OK** button.

Programs created in Visual J++ can include any combination of Java API packages, WFC packages, and `J/Dirct` calls. Remember that whenever you call the Win32 API using WFC packages and `J/Dirct` calls, your application can run only on 32-bit Windows platforms. If you know your application will run only in Windows, using the Win32 API is an appropriate and useful programming technique. However, if you plan to distribute your program widely and intend for it to run on multiple platforms as a true Java application or applet, you should avoid using the Win32 API. In any case, the Win32 API is not a replacement for the core Java programming language—it is only a supplement that allows Java programs to take advantage of Windows operating systems.

## Using the Visual J++ Application Wizard

You can create Windows applications by including J/Direct calls, importing a WFC package into a class file, and adding the appropriate classes and methods from the package to your code. Windows applications, however, usually include a form, which is a standard element of most Windows applications. A form contains a title bar, Minimize button, Maximize button, Close button, and control menu. You use forms to display information and receive input from the user. You could manually add a form to a project, then add the appropriate code to manipulate the form, but doing so can be a time-consuming process. In addition, Windows applications with forms require some special code to function properly. An easier way to create a form-based Windows application is to use the Application Wizard. The **Application Wizard** is a tool in Visual J++ that walks you through the process of creating a Windows application project.



Forms were first discussed in Chapter 1. You will learn more about forms in Section B of this chapter.

When you run the Application Wizard, you are presented with several screens from which you select options for your Windows application project. The first screen you see is the Introduction screen, which allows you to load a profile containing settings from a previous Application Wizard session. Figure 16-12 displays an example of the Introduction screen.

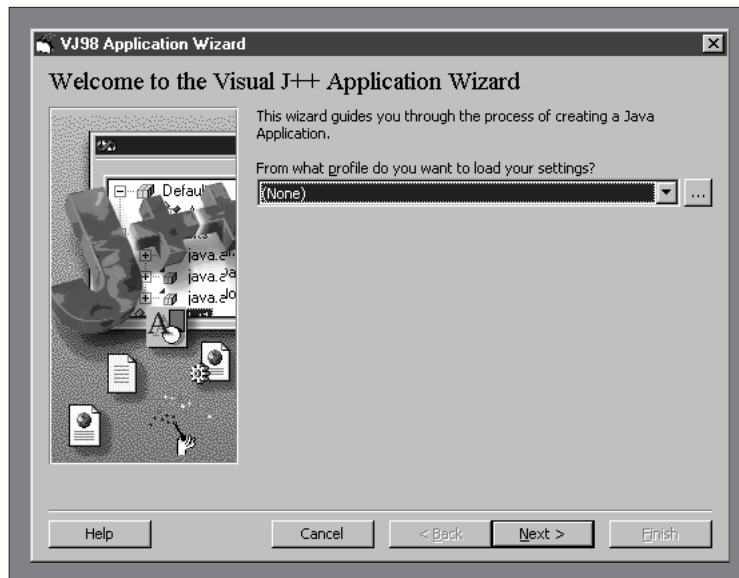


Figure 16-12: Introduction screen of the Application Wizard

The next screen in the Application Wizard is the Application Features screen which contains four elements that can be added to a program: Menu, Edit, Tool Bar, and Status Bar. The Menu option adds a predefined menu to the form. The Edit option fills the form with an editable text box, similar to a page in a word-processing program. The Tool Bar option adds predefined toolbar buttons to the form. The Status Bar option adds a status bar to the bottom of the form. Figure 16-13 displays an example of the Application Features screen.



Figure 16-13: Application Features screen of the Application Wizard



.....

If you are using Visual J++ Professional or Enterprise edition, the second screen in the Application Wizard is the Application Type screen. The Application Type screen allows you to choose Form Based Application or Form Based Application with Data. The Form Based Application with Data option creates a Windows-based application that reads data from an external database file.

.....

Following the Application Features screen in the Application Wizard is the Commenting Style screen. The Commenting Style screen determines the types of comments the Application Wizard will generate for your program. The three available options are JavaDoc comments, TODO comments, and Sample Functionality comments. Figure 16-14 displays an example of the Commenting Style screen.

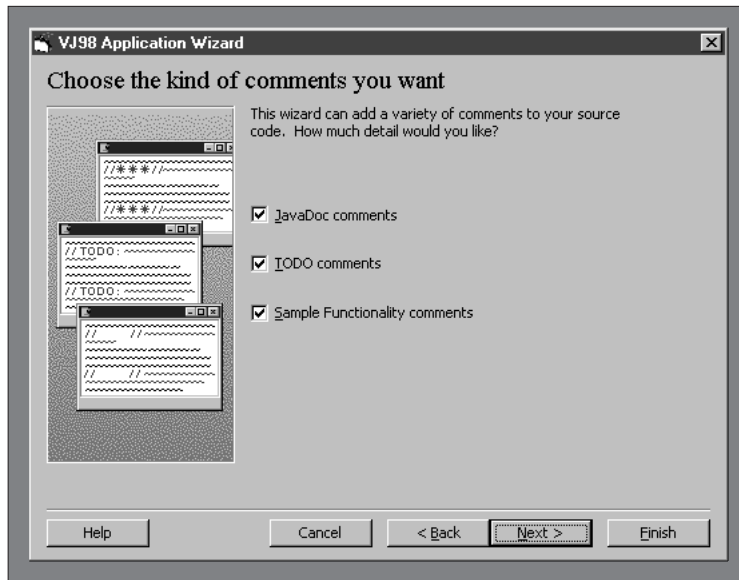


Figure 16-14: Commenting Style screen of the Application Wizard

The last screen in the Application Wizard is the Summary screen, which saves the settings for the current Application Wizard session and allows you to view and save a report of the current settings. Figure 16-15 displays an example of the Summary screen.

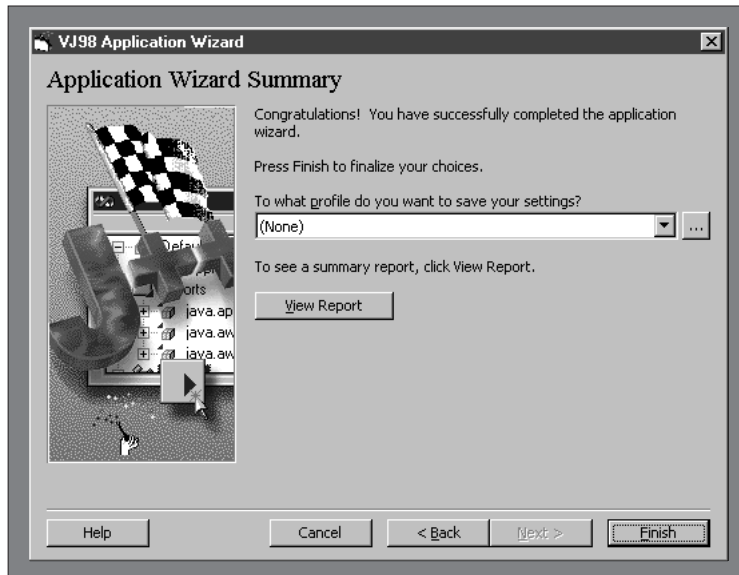


Figure 16-15: Summary screen of the Application Wizard



.....

If you are using Visual J++ Professional or Enterprise edition, the Packaging Options screen appears before the Summary screen. The Packaging Options screen allows you to choose the type of package that will be created when you build your application.

.....

Next you will use the Application Wizard to create a Windows application project.

#### To create a Windows application project using the Application Wizard:

- 1** Select **New Project** from the **File** menu. The New Project dialog box appears. Click the **Applications** folder on the **New** tab, then click **Application Wizard**. Replace the suggested project name in the Name text box with **WindowsPartyPlanner**, change the location of the project folder to the Chapter.16 folder on your Student Disk if necessary, and then click the **Open** button. The Introduction screen of the Application Wizard displays.
- 2** Since this is your first time using the Application Wizard, you should not have any stored profiles. Click the **Next** button to continue. The Application Features screen of the Application Wizard displays.
- 3** Deselect all four options in the Application Features screen: Menu, Edit, Tool Bar, and Status Bar. Click the **Next** button. The Commenting Style screen of the Application Wizard displays.

Any of the options in the Application Wizard can be removed or added once you create the project.

- 4** Deselect all three options in the Commenting Style screen: JavaDoc comments, TODO comments, and Sample Functionality comments. Click the **Next** button. The Summary screen of the Application Wizard displays.
- 5** Click the **Finish** button to create the project. The Application Wizard builds the new project and opens the WindowsPartyPlanner form. Your form should resemble Figure 16-16.



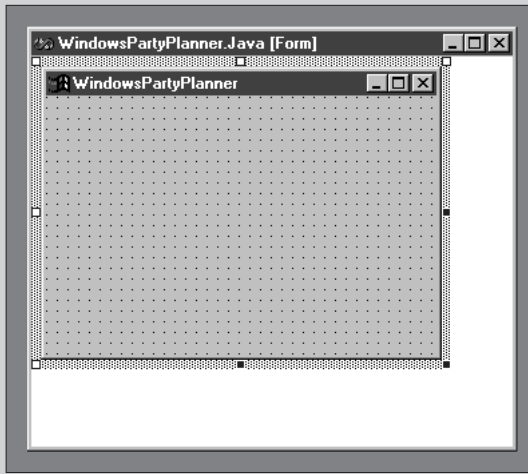


Figure 16-16: WindowsPartyPlanner form



.....

You can quickly create a form-based Windows application by selecting **Windows Application** from the **Applications** folder in the **New Project** dialog box. The skeleton of a form-based Windows application is created containing **JavaDoc** comments, **TODO** comments, and **Sample Functionality** comments.

.....

## WFC Forms

You create Java Windows applications using WFC form files. **WFC form files** are very similar to standard Java files: They have an extension of `.java`, they contain Java code, and you can edit them using the **Text Editor** window. However, you can also work with a WFC form in the **Forms Designer** window. You use the **Forms Designer window** to create and edit the visual aspects of a WFC form. Although you are using a graphical interface to create the visual aspects of the application, all of the visual elements in the **Forms Designer** window, including the form itself, are represented by Java code. For example, the form displayed in the **Forms Designer** window in Figure 16-17 is actually generated by the code displayed in the **Text Editor** window in Figure 16-18.

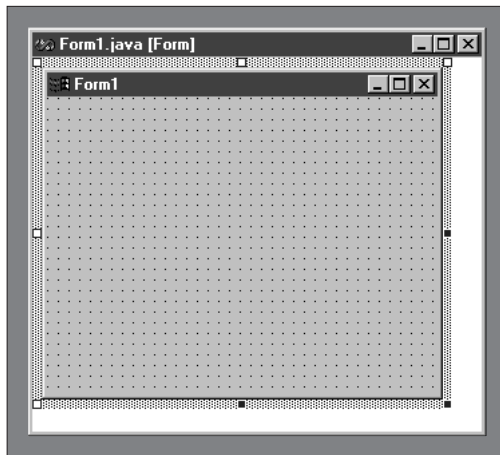


Figure 16-17: WFC form in the Forms Designer window

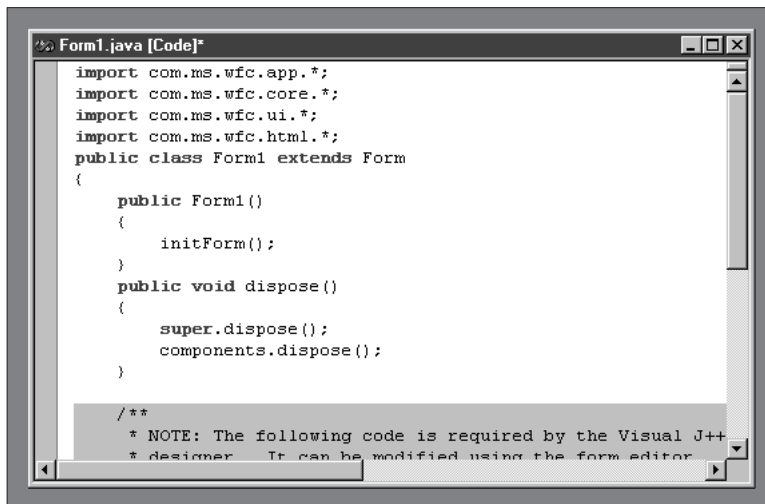


Figure 16-18: WFC form in the Text Editor window

### To display the code in a WFC application:

- 1 Activate the Forms Designer window.
- 2 Select **Code** from the **View** menu. The form's code displays in the Text Editor window.
- 3 Redisplay the Forms Designer window by selecting **Designer** from the **View** menu.



.....  
 You can also view the Code window by pressing F7 and the Designer window by pressing Shift+F7.  
 .....

As you learned in Chapter 4, the package that is implicitly (or automatically) imported into every standard Java program is named `java.lang`. The classes it contains are fundamental classes of Java. The package that contains the fundamental classes for WFC applications is `com.ms.wfc.core`. This class must be explicitly imported into every WFC application. In other words, you must use the `import com.ms.wfc.core;` statement at the beginning of a WFC class file.



.....  
 When you create a WFC program using the Application Wizard or the Windows Application option in the New Project dialog box, Visual J++ automatically includes the necessary import statements for WFC packages.  
 .....

Extending the Form class of the `com.ms.wfc.ui` package into a subclass creates the form you see in the Forms Designer window. The Form class in WFCs is similar to the Frame class in the AWT. Like the AWT Frame class, the WFC Form class extends the Component class. Specific components in WFCs, such as labels, buttons, and check boxes, are contained in the Control class. Since Forms are usually created to hold other components, the Form class also extends the Control class. Therefore, a Form is a Component as well as a Control.



.....  
 You will learn more about the Control class later in this section.  
 .....

When you extend the Frame class, you inherit several useful methods. Figure 16-19 lists the method header and purpose of several WFC Form class methods.

Method	Purpose
<code>void add( Control control )</code>	Adds a control to the form.
<code>void close()</code>	Closes the form.
<code>Control getActiveControl()</code>	Returns the control that has the focus.
<code>int getBorderStyle()</code>	Returns the form's border style in the form of an integer representing a border style constant.
<code>void setActiveControl( Control value )</code>	Sets the focus to a specified control.
<code>void setBorderStyle( int borderStyle )</code>	Sets the form's border style using an integer representing a border style constant.
<code>void setVisible( boolean value )</code>	Sets a form's visible property to true or false.


**Figure 16-19:** Useful methods of the WFC Form class

Standard Java console applications are executed using statements in a `main()` method, whereas Java applets rely on the `init()`, `start()`, `stop()`, `destroy()`,

and `paint()` methods. WFC applications also use a `main()` method, but they must be executed using the `run()` method of the `Application` class found in the `com.ms.wfc.app` package. The `Application.run()` method is placed in the `main()` method of a WFC class with a single argument containing the keyword `new` and the name of the WFC class. For example, if you have a WFC class named `sampleWFCClass`, then you must write the `main()` method as follows:

```
public static void main(String args[])
{
    Application.run(new sampleWFCClass());
}
```

When a WFC application is started, the `Application.run` method is executed, which runs any statements in the main class body, then calls the class constructor. The class constructor then calls the `initForm()` method. The **`initForm()`** method is used to set the properties of the form and initialize any components it contains. Any additional constructor code is placed after the call to the `initForm()` method. You cannot use the `initForm()` method to remove components or to reset properties; you use the method only for setting default values.

When you close a WFC application using the Close button  in the form's title bar, any statements in the `main()` method following the `initForm()` method are executed, then the `Application.exit` method of the `com.ms.wfc.app` package is called. You can also quit a WFC application by calling the `Application.exit` method from anywhere in the code. The `Form` class contains an automatically created method named `dispose()` which is called when the `Application.exit` method is executed to release any of the `Form`'s resources. The `dispose()` method is similar to the `destroy()` method that is called in an applet when a user closes the browser or `AppletViewer`. (You learned about the `destroy()` method in Chapter 7.)

WFC applications usually override the `dispose()` method to remove any components (such as buttons or labels) that the `Form` creates. For example, when you use the `Application Wizard` or the `Windows Application` option in the `New Project` dialog box to create a new WFC program, Visual J++ automatically creates a `Container` named `components` in the class body to contain the `Form`'s components. A `dispose()` method is also added as follows:

```
public void dispose()
{
    super.dispose();
    components.dispose();
}
```

The `super.dispose();` statement is added to call the base `dispose()` method to clean up the application's system-level resources. The `components.dispose();` statement then disposes of the components `Container`.

Any statements following the `Application.run()` method in the `main()` method are called after the `dispose()` method.

Figure 16-20 shows the life cycle of a WFC application.

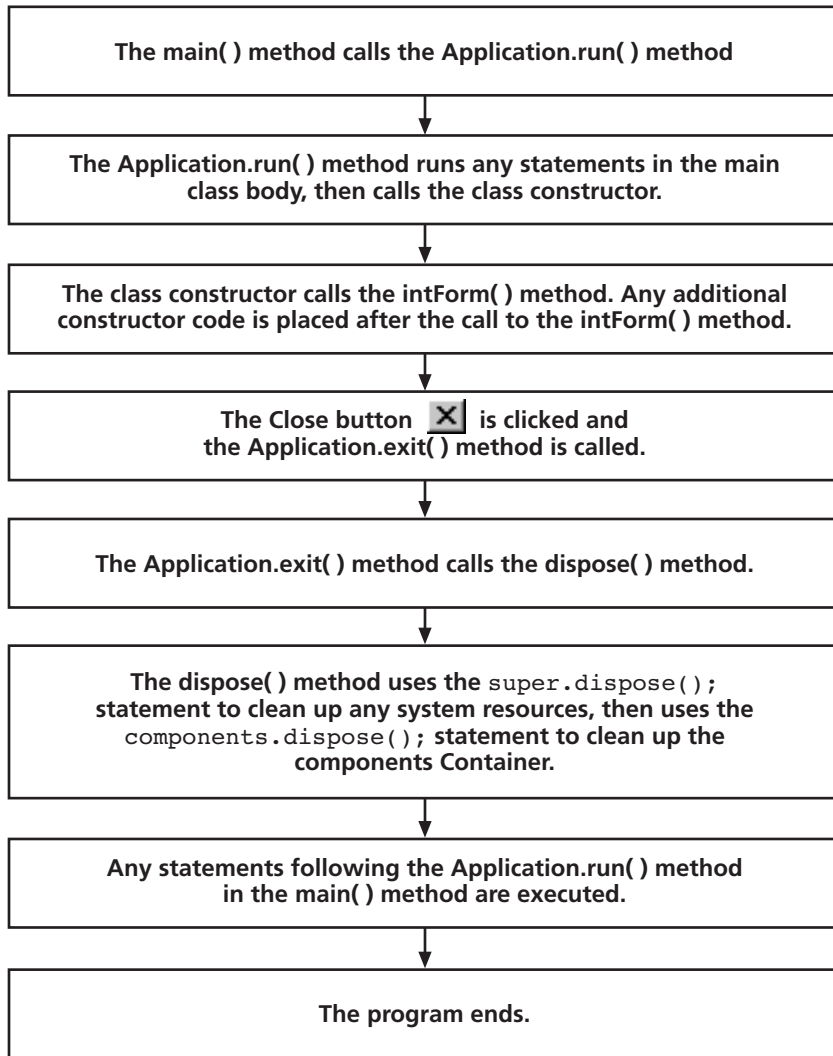



Figure 16-20: Life cycle of a WFC application

Next you will change properties for the WindowsPartyPlanner program and examine the code behind the Forms Designer window.

**To change properties for the WindowsPartyPlanner program and examine the code behind the Forms Designer window:**

- 1** Return to the Forms Designer window in the WindowsPartyPlanner program. If necessary, display the Properties window by selecting **Properties Window** from the **View** menu. Locate and click the text box for the text property in the Properties window. The text property for a form represents the text that appears in the form's title bar. Change the present value to **Event Handlers Incorporated**. The new text appears in the form's title bar.

.....  
**You first learned about the Properties window in Chapter 1.**  
.....

- 2** Locate and click the **backColor** property in the Properties window. The property in the Settings list for the backColor property should be a gray box followed by the word "Control". The Control color property represents a constant in the Color class. (The Color class is a member of the com.ms.wfc.ui package.) Click the drop-down arrow next to the Settings box. The list displays, containing a System tab and a Palette tab. The System tab contains additional constants available in the Color class. Click the **Palette** tab and select a light blue. The background of the form changes to the selected color.
- 3** Locate the **size** property in the Properties window and click the **Plus** box  to expand the property. Beneath the size property are the x and y properties. Change the x property to **350** and the y property to **300**.

.....  
**You can also resize a form with your mouse by dragging one of the sizing handles located at the form's corners and sides.**  
.....

- 4** Locate and click the **borderStyle** property in the Properties window. Click the drop-down arrow next to the Settings box and select **Fixed Dialog**. The Fixed Dialog border prevents users from resizing the form when the application is running.
- 5** Locate and click the **startPosition** property in the Properties window. The startPosition property determines where the form is initially placed on your screen. Change the default option of Windows Default Location to **Center Screen**.
- 6** Build the project as you would any Java program by selecting **Build** from the **Build** menu.
- 7** Select **Code** from the **View** menu to examine the form's code. The first lines of code in the class are the required WFC import statements. The class header extends the Form class. The first two methods in the class are the class constructor and the dispose() method. Following the dispose() method is a gray box containing code that includes the initForm() method.

 help

The code in the gray box is automatically generated by Visual J++ for the visual aspects of the form. Within the `initForm()` method, you should recognize several methods including the `setText()` method, which sets the form's title bar to "Event Handlers Incorporated," and the `setBorderStyle()` method, which sets the form's border style to Fixed Dialog. The last method in the class is the `main()` method, which contains a single statement, `Application.run(new WindowsPartyPlanner());`, to start the application.

If you view the code for a WFC class while its associated Forms Designer window is open, the code that is being controlled by the Forms Designer window is marked in gray. You cannot edit this code while the WFC file's Forms Designer window is open. To edit code that is automatically generated by Visual J++, close the WFC file's associated Forms Designer window. It is easier to allow Visual J++ to handle code generation for the `initForm()` method.


Next you will execute the `WindowsPartyPlanner` class.

**To execute the `WindowsPartyPlanner` class:**

- 1 Select **Start** from the **Debug** menu. The form appears in the center of your screen. The title bar should read "Event Handlers Incorporated" and the background color of the form should appear as the color you selected in the preceding steps.

 tip

You execute WFC applications with `JVIEW` or `WJVIEW`, just as you would execute standard Java applications. However, since the `WindowsPartyPlanner` program is a Windows application, running the program does not require a separate command line, as does a console application. Visual J++ automatically selects `WJVIEW` as the default program in the Properties dialog box for the `WindowsPartyPlanner` project; you don't need to set the program properties manually.

- 2 Close the form by clicking the **Close** button  in the title bar. The Close button automatically calls the `Application.exit()` method and the program ends.

So far, the `WindowsPartyPlanner` program consists of only a form with a title bar. Next you will use the Toolbox to add WFC controls to the form.

## WFC Controls

You already know from Chapter 11 that Java's creators packaged GUI components in the Abstract Windows Toolkit so you can adapt them for your purposes. You insert the import statement `import java.awt;` at the beginning of your Java program files so you can take advantage of the GUI components and their methods, which are

stored in the AWT package. Within the AWT package, components such as buttons, check boxes, and labels descend from the Component class. In WFC programming, components descend from the Control class of the com.ms.wfc.ui package.



**You have already used a child class of the Control class—the MessageBox class descends from Control. Every MessageBox “is a” Control.**

The WFC Control class descends from the WFC Component class. Like the AWT Component class, the WFC Component class contains several methods that you can use with any of its descendants, such as components that extend the WFC Control class. For example, the getChildOf() method is used to determine whether the current component is the child of a specified component. However, many of the methods contained in the AWT Component class, such as the setSize() and setVisible() methods, are not available in the WFC Component class. Instead, they are located in the WFC Control class.

Components in the AWT package are usually placed in containers. A container is a type of component that holds other components so you can treat a group of several components as a single entity. In the AWT, the Container class descends from the Component class. Child classes of the AWT Container class include the Panel class, Window class, and Frame class. These classes are containers as well as components. In standard Java programming, the Container class is a physical component that contains other programs. For example, in the AWT, the Frame class, which is a Container, contains other components such as Buttons and Labels.



**In standard Java packages, the Container and Component classes are part of the java.awt package. The Container and Component classes in WFCs are part of the com.ms.wfc.core package.**

In WFC programming, the Control class (which contains many of the same components as the AWT Component class) does not descend from the Container class. Instead, the Control class descends directly from the Component class. Therefore, components in the Control class are not containers as they are in the AWT Component class. The Container class in WFC is used to organize other components logically, rather than physically, as is the Container class in the AWT. For example, a Frame created with the AWT “is a” Container since it can contain other components such as Buttons and Labels. In WFCs, a Form (which is also a Control) is not a container, although components are placed on it. Instead, a new Container is declared in the body of a Form’s code to contain the Form’s components using the statement `Container components = new Container();`.



**When you use the Application Wizard or the Windows Application option in the New Project dialog box, the statement `Container components = new Container();` is automatically created for you and should not be modified.**

The Toolbox in the Forms Designer window contains various WFC controls that you can add to your form. Next you will use the Toolbox to add WFC controls


to the WindowsPartyPlanner program. In Section B, you will add the code that is necessary to make the WFC controls function.



You first learned about the Toolbox in Chapter 1.




### To use the Toolbox to add WFC controls to the WindowsPartyPlanner program:

- 1** Return to the WindowsPartyPlanner form in the Forms Designer window.
- 2** If necessary, select **Toolbox** from the **View** menu, and then click the **WFC Controls** button.
- 3** First you will add a Cocktails CheckBox and a Dinner CheckBox. To add the first CheckBox, click once on the **CheckBox** control  in the WFC Controls tab, and then click once on the form in the approximate location where you want to place the check box. (Refer to Figure 16-1 to see where to place each control for the WindowsPartyPlanner.)

To learn the function of each control on the WFC Controls tab of the Toolbox, hold your mouse over a control to display its ToolTip.



You can move any control after you have added it to a form by selecting the control, holding down the left mouse button, and dragging the control to the desired location.

- 4** After you add the CheckBox to the form, click the CheckBox to display its properties in the Properties window. Change the CheckBox's name property to **cocktailBox**, and its Text property to **Cocktails**. The name property is the name of the control, and the text property is the text that is displayed as the control's label or default value.
- 5** Add to the form a second CheckBox that you will name dinnerBox. Then change the CheckBox's name property to **dinnerBox**, and its text property to **Dinner**.
- 6** Use the ComboBox control  to add a ComboBox to the form directly beneath cocktailBox. Change the ComboBox's name property to **entertainmentChoice** and its text property to **No Entertainment**.



A WFC ComboBox is similar to the AWT Choice component, which you learned about in Chapter 11.

- 7** You can add more items to the ComboBox using the items property box. The items property box contains the items that will appear in the ComboBox at run time. Click once in the items property box for entertainmentChoice. Notice that the items property default setting is (Strings). Click the **ellipsis (...)** button that appears to the right of the items property default setting (Strings).


The String List Editor dialog box appears. Enter the following additional four items into the dialog box, then select the **OK** button:

**No Entertainment**

**Rock Band**

**Pianist**

**Clown**

- 8** Use the ListBox control  to add a ListBox to the form directly beneath the entertainmentChoice control. The selectionMode property of ListBox determines what type of selections users can make: None, One, Multi Simple, or Multi Extended. A selectionMode property of None sets the properties so that users cannot make a selection. A selectionMode property of One lets users select only one choice and allows them to select a different item if they change their mind. Multi Simple allows users to select multiple items. Multi Extended is similar to Multi Simple, but allows users to use their Shift, Ctrl, and arrow keys to make selections. Change the ListBox's name property to **partyFavorList**, and change the selectionMode to **One**.

**tip**

A WFC ListBox is similar to the AWT List component, which you learned about in Chapter 11.


- 9** Use the items property to open the String List Editor dialog box for partyFavorList, enter the following four items into the dialog box, then select the **OK** button:

**Hats**

**Streamers**

**Noise Makers**


**Balloons**

- 10** Beneath the dinnerBox control, use the GroupBox control  to add a GroupBox. To add a GroupBox, click once on the **GroupBox** icon in the WFC Controls tab. Then point your mouse at the approximate location where you want the upper-left corner of the GroupBox to appear on the form, click and hold your left mouse button, and drag until the GroupBox reaches the desired size. You can resize a GroupBox after creating it using one of the sizing handles located at its corners and sides. Change the GroupBox's name property to **dinnerGrp** and change its text property to **Entree**.

**tip**

A GroupBox is the equivalent of the AWT CheckboxGroup, which you learned about in Chapter 11.

- 11** Within dinnerGrp, add three RadioButtons named chickenButton, beefButton, and fishButton. Make sure that each RadioButton and its associated text is contained within the boundaries of dinnerGrp. If part of a RadioButton is outside the boundaries of a GroupBox, then it is not considered to be part of the group.




To add each **RadioButton**, click once on the **RadioButton** control  in the WFC Controls tab. Then click once on the form in the approximate location where you want to place the **RadioButton**.

- 12** Change the name property of the first **RadioButton** to **chickenButton** and its text property to **Chicken**. Change the name property of the second **RadioButton** to **beefButton** and its text property to **Beef**. Change the name property of the third **RadioButton** to **fishButton** and its text property to **Fish**.
- 13** Set **chickenButton** as the default **RadioButton** in the group by changing its checked property to **true**.

.....

**Only one **RadioButton** within a **GroupBox** can have its **check** property set to **true**. All other **RadioButtons** must be set to **false**.**

.....

- 14** Below the **partyFavor** list and the **dinnerGrp** **GroupBox**, use the **Label** control  to add a **Label** to the form. Click once on the **Label** control in the WFC Controls tab. Then click once on the form in the approximate location where you want to place the **Label**. Change the **Label**'s name property to **companyLabel** and its text property to **Event Handlers Incorporated**. Use the font property to change the **Label**'s font to **MS Sans Serif**, its size to 14 points, and its style to **bold**.
- 15** Add another **Label** to the form. Change the **Label**'s name property to **priceLabel** and its text property to **Event price estimate**. Use the font property to change the **Label**'s font to **MS Sans Serif**, its size to 14 points, and its style to **bold**.
- 16** To the right of the **priceLabel**, add a final **Label** to the form. Change the **Label**'s name property to **totalPriceLabel** and its text property to **200**. Set the **Label**'s font properties to **MS Sans Serif**, 14 points, and **bold**.
- 17** Add a **Button** to the bottom of the form. Click once on the **Button** control  in the WFC Controls tab. Then click once on the form in the approximate location where you want to place the **Button**. Change the **Button**'s name property to **buttonExit** and its text property to **Exit**.
- 18** Build and save the project, then run the program by selecting **Start** from the **Debug** menu. Your program should appear similar to Figure 16-1. You should be able to select all the options on the form. However, you still need to write code to generate the event price.
- 19** Close the form by clicking the **Close** button  in the title bar.

Remember that although Visual J++ automatically generates the code that builds the controls you added, you can write the code manually when the Forms Designer window is closed. The controls you just created require a fairly long segment of code. For example, you created a **GroupBox** containing three

CheckBoxes, and modified several properties for each control. The code required to create and set properties for the GroupBox and its CheckBoxes is as follows:

```
GroupBox dinnerGrp = new GroupBox();
RadioButton chickenButton = new RadioButton();
RadioButton beefButton = new RadioButton();
RadioButton fishButton = new RadioButton();

dinnerGrp.setLocation(new Point(184, 48));
dinnerGrp.setSize(new Point(136, 112));
dinnerGrp.setTabIndex(0);
dinnerGrp.setTabStop(false);
dinnerGrp.setText("Entree");

chickenButton.setLocation(new Point(16, 16));
chickenButton.setSize(new Point(100, 23));
chickenButton.setTabIndex(0);
chickenButton.setText("Chicken");
chickenButton.setChecked(true);

beefButton.setLocation(new Point(16, 40));
beefButton.setSize(new Point(100, 23));
beefButton.setTabIndex(1);
beefButton.setText("Beef");
beefButton.setChecked(false);

fishButton.setLocation(new Point(16, 64));
fishButton.setSize(new Point(100, 23));
fishButton.setTabIndex(2);
fishButton.setText("Fish");
fishButton.setChecked(false);

dinnerGrp.setNewControls(new Control[] {
    fishButton,
    beefButton,
    chickenButton});
```


You should understand the purpose of each of these statements, even though you can have Visual J++ generate them automatically.

In Section B, you will learn about events in WFC programs and write the code to finish the WindowsEventPlanner application.

## S U M M A R Y

- There are two types of Windows operating systems: 16-bit and 32-bit. The older Windows 3.1 operating system is 16-bit. Current 32-bit Windows operating systems are Windows NT, Windows 95, Windows 98, and Windows CE.
- Visual J++ is a 32-bit application designed for 32-bit Windows operating systems. The term *bit* refers to the width of a microprocessor's data bus.

- An application programming interface (API) is a library of methods and attributes that provide access to the features and functionality of an application or operating system. All 32-bit Windows operating systems share a common API called the Win32 API.
- Using the common functionality available in the Win32 API helps to maintain a consistent look for Windows-based applications.
- J/Direct translates the Win32 API syntax into a format that can be used with Java.
- Methods and other components in the Win32 API are contained in dynamic-link libraries (DLLs). DLLs are files with an extension of .dll.
- A directive tells the compiler in which DLL to find a method. There are three types of directives: @dll.import for declaring functions, @dll.struct for declaring structures, and @dll.structmap for declaring fixed-size strings and arrays embedded in structures.
- If you know the correct syntax for J/Direct calls and directives that you want to include in your Java application, then you can code them directly into a Java file. You can also insert J/Direct calls and directives into your code using the Visual J++ J/Direct Call Builder.
- Adding J/Direct calls removes a Java program's architectural neutrality—the program is no longer a true Java program since it can run only on Windows platforms. However, J/Direct allows you to use your Java programming skills to create Windows applications.
- Although programs you create that use only Java API packages will run on any platform for which there is a Java VM, the Java VM cannot use components that exist within an individual operating system.
- True Java applications will not run as fast as applications that are native to a specific operating system. This difference in speeds is a result of the Java VM. Although the Java VM allows Java applications to be architecturally neutral, it adds an extra translation layer that an application must pass through to run.
- Windows Foundation Classes (WFCs) for Java are packages of J/Direct calls that are translated into the Java programming language.
- The MessageBox class contains three constructors: the text to be displayed in the MessageBox, the dialog box title, and the dialog box style.
- Programs created in Visual J++ can include any combination of Java API packages, WFC packages, and J/Direct calls.
- If your application will run only in Windows, there is no reason not to use the Win32 API. However, if your program will be widely distributed on multiple platforms as a true Java application or applet, then you should avoid using the Win32 API.
- The Application Wizard is a tool in Visual J++ that walks you through the process of creating a Windows application project.
- WFC form files are very similar to standard Java files: They have an extension of .java, contain Java code, and can be edited using the Text Editor window.
- The package that contains the fundamental classes for WFC applications is com.ms.wfc.core. This class must be explicitly imported into every WFC application.
- The Form class in WFCs is similar to the Frame class in the AWT.
- WFC applications use a main() method, similar to Java applications, but must be executed using the run() method of the Application class found in the com.ms.wfc.app package.
- The Application.run() method is placed in the main() method of a WFC class with a single argument containing the keyword new and the name of the WFC class.
- The initForm() method is used to set the properties of the form and any components it contains. Any additional constructor code is placed after the call to the initForm() method.

- When you close a WFC application using the Close button , the `Application.exit` method of the `com.ms.wfc.app` package is called.
- You can also quit a WFC application by calling the `Application.exit` method from anywhere in the code.
- The `dispose()` method is similar to the `destroy()` method that is called in an applet when using a Web browser or `AppletViewer`.
- With WFC programming, components descend from the `Control` class of the `com.ms.wfc.ui` package.
- The WFC `Control` class descends from the WFC `Component` class. Like the AWT `Component` class, the WFC `Component` class contains several methods that you can use with any of its descendants, such as components that extend the WFC `Control` class.
- Many of the methods contained in the AWT `Component` class, such as the `setSize()` and `setVisible()` methods, are not available in the WFC `Component` class. Instead, they are located in the WFC `Control` class.
- In WFC programming, the `Control` class does not descend from the `Container` class. Instead, the `Control` class descends directly from the `Component` class. Therefore, components in the `Control` class are not containers as they are in the AWT `Component` class.
- The `Container` class in WFC logically organizes other components, whereas the `Container` class in the AWT physically organizes components.

## Q U E S T I O N S

1. The term *bit* refers to \_\_\_\_\_.
  - a. the amount of memory in your computer
  - b. the width of a computer microprocessor's data bus
  - c. code statements in a WFC application
  - d. the size of a computer's hard drive
2. \_\_\_\_\_ is a 16-bit Windows operating system.
  - a. Windows NT
  - b. Windows 95
  - c. Windows 3.1
  - d. Windows 98
3. A library of methods and attributes that provide programmatic access to an application or operating system is called \_\_\_\_\_.
  - a. an application programming interface (API)
  - b. a Windows Foundation Class (WFC) program
  - c. a dynamic link library (DLL)
  - d. the Abstract Windows Toolkit (AWT)
4. You can directly access the Win32 API in Visual J++ by using \_\_\_\_\_.
  - a. standard Java statements
  - b. the Sun JDK
  - c. J/Direct calls
  - d. the `java.awt` package

5. A simple Win32 API dialog box that displays a message and an OK button is created with \_\_\_\_\_.
  - a. the `initform()` function
  - b. the `System.out.print()` method
  - c. the `MessageBox` function
  - d. the `Beep` function
6. The declaration that informs Java that a method is implemented in another programming language is called the \_\_\_\_\_ modifier.
  - a. `external`
  - b. `final`
  - c. `public`
  - d. `native`
7. A \_\_\_\_\_ tells the compiler in which DLL a method will be found.
  - a. parameter
  - b. `// TODO` comment
  - c. constructor
  - d. directive
8. You can type J/Direct calls directly into a file or insert them using \_\_\_\_\_.
  - a. Object Browser
  - b. J/Direct Call Builder
  - c. the Properties window
  - d. the Open dialog box
9. Java programs containing J/Direct calls or that import WFC packages can run \_\_\_\_\_.
  - a. on any platform for which there is a Java VM
  - b. on Windows NT but not on Windows 95
  - c. on Windows 95 and Windows 98, but not on Windows NT
  - d. on any 32-bit Windows platform
10. Windows Foundation Classes (WFCs) are \_\_\_\_\_.
  - a. special Java packages that make Java programs architecturally neutral
  - b. part of the `java.awt` package
  - c. packages of standard Java calls that are used in true Java applications
  - d. packages of J/Direct calls that are translated into the Java programming language
11. The package that should be imported into every WFC application is \_\_\_\_\_.
  - a. `com.ms.wfc.app`
  - b. `com.ms.wfc.core`
  - c. `com.ms.wfc.data`
  - d. `com.ms.wfc.html`
12. Programs created in Visual J++ can include \_\_\_\_\_.
  - a. Java API packages, but not WFC packages or J/Direct calls
  - b. WFC packages and J/Direct calls, but not Java API packages
  - c. only Java API packages
  - d. any combination of Java API packages, WFC packages, and J/Direct calls
13. Your program should include calls to the Win32 API when \_\_\_\_\_.
  - a. you are planning on distributing your program on a wide range of platforms
  - b. your program will run only on 32-bit Windows platforms

- c. your program will never run on 32-bit Windows platforms
  - d. you want your program to be architecturally neutral
14. The Win32 API is \_\_\_\_\_.
- a. the most recent version of Java
  - b. part of every Java program
  - c. a replacement for the Java programming language
  - d. intended only as a supplement to the Java programming language
15. The Application Wizard walks you through the process of creating \_\_\_\_\_.
- a. a console application project
  - b. a Windows application project
  - c. an Applet
  - d. a program's graphical user interface (GUI)
16. You create and edit the visual aspects of a WFC application using the \_\_\_\_\_.
- a. Object Browser window
  - b. Win32 API
  - c. Forms Designer window
  - d. Class Outline window
17. You create a form in a WFC application by extending the Form class of the \_\_\_\_\_ package into a subclass.
- a. com.ms.wfc.core
  - b. com.ms.wfc.data
  - c. com.ms.wfc.app
  - d. com.ms.wfc.ui
18. To execute a WFC application, you must include the \_\_\_\_\_ method in a program's main() method.
- a. Application.run()
  - b. Show()
  - c. DisplayForm()
  - d. MessageBox()
19. A form's properties and components are initialized using the \_\_\_\_\_.
- a. Form constructor
  - b. initForm() method
  - c. main() method
  - d. Application.run() method
20. A form is closed when \_\_\_\_\_.
- a. the program receives a close notification from the Win32 API
  - b. the last statement in the main() method is executed
  - c. the Application.exit() method is called
  - d. the Application.close() method is called
21. The \_\_\_\_\_ method removes any of the components created by a WFC application.
- a. remove()
  - b. dispose()
  - c. kill()
  - d. quit()

22. Specific components in WFCs, such as labels, buttons, and check boxes, are contained in the \_\_\_\_\_ class.
  - a. Form
  - b. Control
  - c. Component
  - d. Container
23. Controls in WFC applications descend from \_\_\_\_\_.
  - a. both the Component class and Container class
  - b. the Component class but not the Container class
  - c. the Container class but not the Component class
  - d. neither the Container class nor the Component class



## E X E R C I S E S

Save each of the programs that you create in the exercises in the Chapter.16 folder on your Student Disk.

1. Write a Java Windows application that prompts users to select which invitation they would like displayed: Birthday, Graduation, or Anniversary. Add three Strings to the program for each invitation type. The Birthday invitation should read “Please join us for our son’s birthday.” The Graduation invitation should read “We are celebrating our daughter’s graduation from college.” The Anniversary greeting should read “You are cordially invited to attend our 25<sup>th</sup> wedding anniversary.” Add J/Direct calls to the program that call the MessageBox and the Beep functions. Display a MessageBox containing the selected invitation, and then call the Beep function after the user presses the OK button. Name the project SelectAnInvitation.
2. Write a WFC application project that displays a form containing the words to any well-known song. Name the project Song.
3. Create a WFC application project named EmploymentApplication. Include Edit boxes for a name, address, and city. Create a ComboBox that allows users to select a single state. Fill the state ComboBox with the abbreviations of at least five states in your area. Create a ListBox that allows users to select a range containing their years of experience: 0-1, 1-3, 3-5, 5-10, or 10+. Allow the user to select only one entry at a time. Add a GroupBox for the type of employment the user is seeking. Include five RadioButtons in the GroupBox: Management, Clerical, Industrial, Technical, and Construction. At the bottom of the form, add three CheckBoxes: Willing to Relocate, Available for Overtime, and Available for Travel.
4. Windows applications usually contain an About box that displays legal and other information about the program. Use a WFC application to create an About box for the WindowsPartyPlanner application you created in this section. Look in the About boxes for other Windows applications on your computer for ideas of the type of information to include in your About box. You can display the About box for most Windows applications by selecting About from the Help menu. Experiment with fonts and the different controls available in the Toolbox.
5. Review some of the applets and applications you have created in previous chapters. Try to simulate the same designs in a WFC program. Do you find creating visual interfaces easier with the AWT or in WFC programs?

# SECTION B

## objectives

In this section you will learn:

- About WFC Events
- How to create and use packages
- How to use Visual J++ packaging to distribute programs

# WFC Events and Packaging

## WFC Events

In standard Java programming, you tell your program to listen for an event by implementing an event interface such as `ActionListener`. You have worked with event interfaces, such as `ActionListener`, since Chapter 7, so you know that the `ActionListener` interface listens for `ActionEvents`, which are the types of events that occur when a user clicks a `Button`. To implement `ActionListener`, you

- Import `java.awt.event.*`
- Add `implements ActionListener` to the right of the class header
- Add the `addActionListener()` method to each item, such as a `Button`, that responds to `ActionEvents`
- Add an `actionPerformed(ActionEvent e)` method to the class

When you work with Event objects, such as `ActionEvent`, you create only a single method to handle *all* instances of a specific event type. Whereas you may have two or more buttons in a class that are registered as `ActionListeners`, a class can contain only one `actionPerformed(ActionEvent e)` method.

Instead of accessing internal Java events contained in the `java.awt.event` package, components in WFC applications access standard Windows events that are available as methods in the `Control` class. You do not need to import a different package or implement an event interface. A variety of different Windows events are available for different types of controls. Figure 16-21 lists some common Windows events that are automatically available to a `Button` in a WFC application.

Event	Occurs When
Click	The Button is clicked
Enter	The Button gains focus
Leave	The Button loses focus

Figure 16-21: Common Windows events available to a Button

Event	Occurs When
MouseDown	The Button is clicked and the mouse is not released
MouseUp	The Button is clicked and the mouse is released after being held down

Figure 16-21: Common Windows events available to a Button (continued)



.....

Unlike the single `actionPerformed(Event e)` method in standard Java applications, each control in a WFC program can have its own Click method.

.....

You register AWT components to listen for events using the `add<event>Listener()` method, where `<event>` represents the specific type of event. For example, the `addActionListener()` method registers a component to listen for `ActionEvents`. You register WFC controls for events using the `addOn<event>()` method. The `addOn<event>()` method accepts a single parameter known as a delegate. A **delegate** is a wrapper class that is used for passing one method to another method.

.....



.....

You first encountered wrapper classes in Chapter 6, where you learned that a wrapper is a class or object that is “wrapped” around a simpler thing.

.....



.....

Delegates are similar in function to pointers found in other programming languages such as C, C++, and Pascal.

.....

WFC delegate classes are comparable to AWT event interfaces. For example, the generic `EventHandler` delegate handles roughly the same events as the AWT `ActionListener` interface. Other delegates include the `MouseEventHandler` delegate, which handles mouse events in the same manner as the AWT `MouseListener` class, and the `KeyEventHandler` delegate, which processes keypresses, similar to the AWT `KeyListener` class.

When you register a component in a WFC application, you essentially “delegate” the responsibility of the event to another method, known as an event handler. An **event handler** is a method created to run in response to a particular event. The following code adds a new `EventHandler` to a Button named `myButton` and passes the event to an event handler method called `myButton_Click()`:

```
Button myButton = new Button();
myButton.addOnClick(new EventHandler(this.myButton_Click));
```

An event handler receives two items from a delegate: a reference to the component that initiated the event, and the event object itself. Any method that you


intend to use as an event handler must include arguments to access these two items. For example, the following code shows a typical click event handler for a button:

```
private void myButton_click(Object source, Event e)
{
    // Add statements here
}
```

When you work with events in WFC applications, Visual J++ automatically creates an event's delegate and event handler method for you. You can also create a new event handler method from scratch. When Visual J++ automatically creates an event handler method for you, the name of the event handler method is associated with the name of the component for which it was created. For example, the default name of the event handler for myButton is myButton\_click. If you intend to use a single, automatically created event handler method with multiple controls, you may want to rename the event handler with a more logical name. When you create a new method from scratch that will be used as an event handler, be sure to include the `Object source, Event e` arguments in the method's header.

Next you will add functionality to the WindowsPartyPlanner application. To do so, you will add events that will calculate the price of the event based on the items that users select. You will also enable the Exit button so that clicking it ends the application.

#### To add events to the WindowsPartyPlanner application:

- 1 If necessary, open the WindowsPartyPlanner project from the Chapter.16 folder on your Student Disk, then open the WindowsPartyPlanner.java file in the Forms Designer window.
- 2 Click the **Exit** button in the Forms Designer window, then click the **Events button**  in the Properties window. A list of events that are available for a WFC Button appears. Double-click anywhere on the line for the Click event. Visual J++ switches to code view and automatically generates an event handler skeleton for the Exit button's Click event.
- 3 Locate the buttonExit\_click() method and add **Application.exit();** between the method's curly brackets. This statement allows you to call the internal Application.exit() method.



You can also create a default event handler skeleton for a control by double-clicking it in the Forms Designer window. The default event depends on the control. For example, a Button control's default event is the Click event, while a GroupBox control's default event is the Enter event.

- 4** In the Code window, locate the `WindowsPartyPlanner` class. Immediately following the class's opening curly brackets, add the following variables for the event prices:

```
int cocktailPrice = 300, dinnerPrice = 600;
int beefPrice = 100, fishPrice = 75;
int[] actPrice = {0, 725, 325, 125};
int[] favorPrice = {8,10,25,35};
```

- 5** Locate the class's main method, and enter the following `eventOptionsChanged()` method immediately after the `main()` method. The `eventOptionsChanged` method will update the event price whenever an option is changed in the form. All of the form controls that can change the event's pricing will call the `eventOptionsChanged()` method as a delegate.

```
public void eventOptionsChanged(Object source, Event e)
{
    int totalPrice = 200;
    if(cocktailBox.getCheckState() == 1)
        totalPrice += cocktailPrice;
    if(dinnerBox.getCheckState() == 1)
    {
        totalPrice += dinnerPrice;
        if(beefButton.getChecked())
            totalPrice += beefPrice;
        else if(fishButton.getChecked())
            totalPrice += fishPrice;
        else
            chickenButton.setChecked(true);
    }
    int actNum = entertainmentChoice.getSelectedIndex();
    if(actNum != -1)
        totalPrice += actPrice[actNum];
    int[] favorNums = partyFavorList.getSelectedIndices();
    for(int x = 0; x < favorNums.length; ++x)
        totalPrice += favorPrice[favorNums[x]];
    totalPriceLabel.setText(Integer.toString(totalPrice));
}
```

- 6** Save the project so the `eventOptionsChanged()` method is available to the controls in the Forms Design window.

**help**

From working with similar code based on the AWT, you should recognize the methods in the preceding code. However, some of the method names differ slightly from their counterparts in the AWT.

- 7 Return to the Forms Designer window, click the **Cocktails** CheckBox, then click the drop-down arrow to the right of the Click event's Settings box in the Properties window. A list displays containing the two valid event handler methods: `buttonExit_Click` and `eventOptionsChanged()`. Click the **eventOptionsChanged()** method. Visual J++ automatically places the insertion point in the `eventOptionsChanged()` method in the Code window. Scroll to the gray-screened code that is being generated by Visual J++ until you find the statement that reads `cocktailBox.addOnClick(new EventHandler(this.eventOptionsChanged));`. Notice that the delegate calls the `eventOptionsChanged()` method. This statement is the only code necessary to call the `eventOptionsChanged()` method as a Click event for the Cocktails CheckBox.
- 8 Repeat Step 6 to add a `selectedIndexChanged` event that calls the `eventOptionsChanged()` method for the Dinner Checkbox, the PartyFavor ListBox, and each of the three entrée Radio Buttons.
- 9 Add a `selectedIndexChanged` event for the Entertainment Choice ComboBox that also calls the `eventOptionsChange()` method.
- 10 Rebuild and save the project. If necessary, correct any syntax errors and rebuild again. Execute the program by selecting **Start** from the **Debug** menu, and test each of the components.

## Creating and Using Packages

Just as Java's creators have provided you with packages, such as `java.util` and `java.awt`, you can create your own packages. When you create your own classes, you can place these classes in packages so that you or other programmers can easily import related classes into new programs.



.....

**Creating packages encourages others to reuse software because packages make it convenient to import many related classes at once.**

.....

When you create classes for others to use, most often you do not want to provide users with your `.java` files, which contain your source code. You expend significant effort in developing workable code for your programs. If you provide your source code to people, other programmers will be able to copy your programs, make minor changes, and market the new product themselves, profiting from your effort. Instead of providing users with your `.java` files, you provide users with the `.class` compiled files, which allow users to run the program you have developed. Similarly, when other programmers use classes you have developed, they need only the completed compiled code in the `.class` files to import into their programs. You place the `.class` files in a package so other programmers can import them.

To place the compiled code into a package, you include the `package` statement at the beginning of your class file, outside the class definition. The statement `package com.course.animals;` indicates that the compiled file should be placed in a folder named `com.course.animals`. The statement includes the keyword `package` followed by the path of the folder that should contain the `.class` file. In this case, the `package` statement indicates that the compiled file will be stored in the `animals` subfolder inside the `course` subfolder inside the `com` subfolder (or `com\course\animals`). The path name can contain as many levels as you want.



**The `package` statement, import statements, and comments are the only statements that appear outside class definitions in Java program files.**

Building a project creates the new package. If the `Animal` class file in a project named `myProject` contains the statement `package com.course.animals;`, then the `Animal.class` file will be placed in the `myProject\com\course\animals` folder. If any of the subfolders do not exist within the `myProject` project folder, Visual J++ will create them. You can then move the `com` folder to the root of a drive. For example, moving the `com` folder to the root of the C drive creates a directory structure of `c:\com\course\animals`. If you then package compiled files for `Dog.java`, `Cow.java`, and so on, future programs only need to use the statement `import com.course.animals.*` to be able to use all the related classes. Alternatively, you can list each class separately, as in `import com.course.Dog;` and `import com.course.Cow;`. Usually, if you want to use only one or two classes in a package, you use separate import statements for each class. If you want to use many classes in a package, it is easier to import the entire package, even if you do not use some of the classes.



**You cannot import more than one package in one statement; for example, `import com.*` does not work.**

Before you import a custom package into a project, you must use the Project Properties dialog box to specify where Visual J++ should look for the package. You use the Classpath tab in the Project Properties dialog box to specify the location of packages that your project will import. An example of the Classpath tab of the Project Properties dialog box is displayed in Figure 16-22.

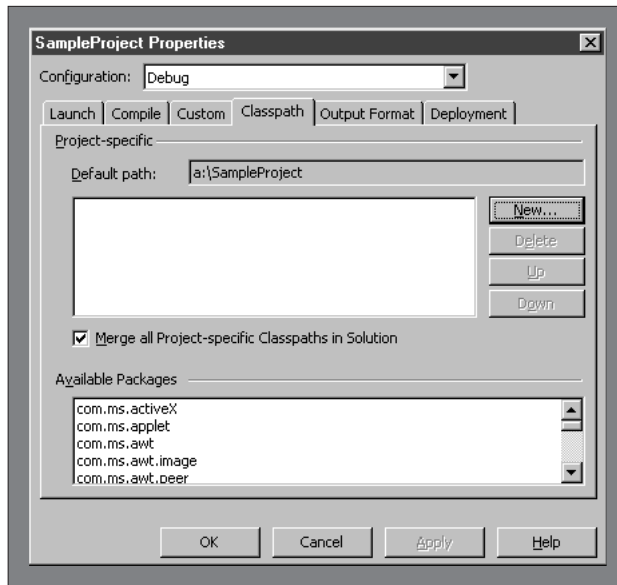


Figure 16-22: Classpath tab of the Project Properties dialog box

The Available Packages list at the bottom of the dialog box displays the default Java packages that are installed on your system. If you scroll through the list, you will recognize several of the packages, including the `com.ms.wfc.ui` and the `java.awt` packages. The Default path text box displays the path of the current project folder. Immediately below the Default path text box is a list of project-specific packages that are required by the current project.

Because the Java programming language is used extensively on the Internet, it is important to give every package a unique name. Sun Microsystems, the creator of the Java programming language, has defined a convention for naming packages. Under this convention, you use your Internet domain name in reverse order. For example, if your domain name is `course.com`, then you begin all of your package names with `com.course`. Then you organize your packages into subfolders in the `course` directory. Using this convention ensures that your package names will not conflict with those of any other Java code providers.

## Using Visual J++ Packaging to Distribute Programs

The most common way to distribute (or *deploy*) a Java applet is to include it on a Web page using the `<APPLET>` tag. For someone to use the program you created, they need only use a Web browser to access the Web page containing your applet. In contrast, you distribute standard Java applications (not WFCs) on floppy disk, CD-ROM, or a network drive. Additionally, when someone wants to use a Java application (as opposed to a Java applet) that you wrote, he or she must also have a copy of `JVIEW` or `WJVIEW`. You also distribute WFC applications on floppy

disk, CD-ROM, or a network drive, and people who want to use the application must also have a copy of JVIEW or WJVIEW.



.....

The JVIEW program that comes with your version of Visual J++ is freely distributable to your users. You can download JVIEW from <http://www.microsoft.com/java/>.

.....

Distributing a Java applet or application consisting of a single class file is fairly simple, especially if the program accesses only internal Java packages. However, distributing programs consisting of multiple classes that import packages other than the internal Java packages can be much trickier. You now know how to create and import Java packages. To make distribution easier, Visual J++ also enables you to compile your project in several types of **output formats**, or packages. The Standard edition of Visual J++ allows you to create Microsoft Cabinet (CAB) files or Windows executable (EXE) files. A **CAB file** is a Microsoft format that contains compressed files and a security feature called digital signature that is used with ActiveX controls and Web browsers. A **Windows EXE file** is a standard file that starts applications on Windows platforms. Whenever you select a program to run using the Windows Start menu, you are usually launching a Windows EXE file. You use the Visual J++ Properties dialog box to select a project's output format.



.....

The Professional and Enterprise editions of Visual J++ also allow you to create COM DLL, Setup, and ZIP package types.

.....

So far, you have used the Properties dialog box to select which file to load when running your program and whether to run the program as a console application. You have also used the Properties dialog box to add new class paths to your project so that you can import custom packages. You can also use the Properties dialog box to select your project's build configuration. A **build configuration** contains the options that the compiler uses when you build your project. There are two types of build configurations: debug and release. A debug build includes full debugging information that allows you to catch release-build errors and check for memory errors, for example. A release build is the version of your application that you will release to users.

Next you will create a release build of the WindowsPartyPlanner project as a Windows EXE package.

#### **To create a release build of the WindowsPartyPlanner project as a Windows EXE package:**

- 1** If necessary, open the **WindowsPartyPlanner** project from the Chapter.16 folder on your Student Disk.
- 2** After opening the project, select **WindowsPartyPlanner Properties** from the **Project** menu. The WindowPartyPlanner Properties dialog box displays.

- 3 At the very top of the dialog box is the Configuration drop-down list box. It contains three entries: Release, Debug, and All Configurations. Select **Release** from the **Configuration** drop-down list box.



The All Configurations option allows you to apply properties to both release and debug property sets simultaneously.

- 4 Click the **Compile** tab, which is used for changing compiler settings. The default option for release configurations removes settings that create debug information and selects the Optimize compiled code check box, which helps Java applications run faster. Click the Output Directory text box and type **A:\Chapter.16\OutputDirectory**, where A: is the drive containing your Student Disk. The Output Directory is where Visual J++ places the necessary files to run an application.
- 5 Click the **Output Format** tab. Make sure the **Enable Packaging** check box is selected, then select **Windows EXE** from the **Packaging type** drop-down list box. In the File name text box, change the drive, directory, and filename to **A:\Chapter.16\OutputDirectory\WindowsPartyPlanner.exe**.
- 6 Select the **Advanced** button. The Advanced...Properties dialog box displays. The EXE/DLL Options tab in the Advanced...Properties dialog box is used for setting version information which can be viewed in Windows by users of your application. An example of the EXE/DLL Options tab is displayed in Figure 16-23.

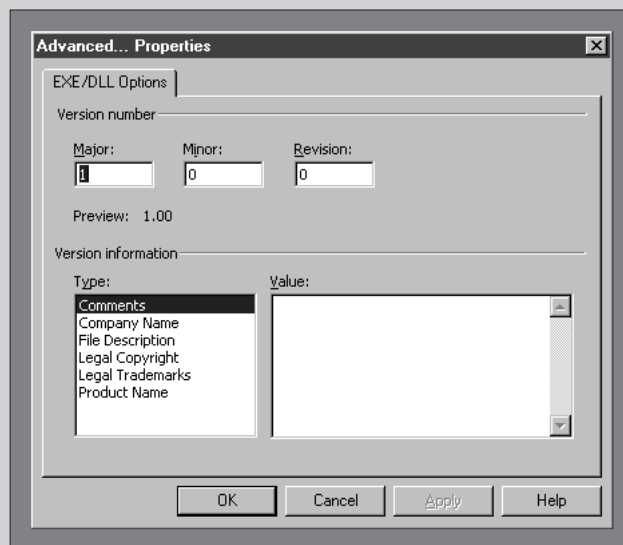


Figure 16-23: EXE/DLL Options tab

help

The Advanced...Properties dialog box is shared by the COM DLL output format, which is a package type that is available only in the Professional and Enterprise editions of Visual J++.

- 7** Select the **Major** text box, and change the version number to **2**. You set the version number to 2 instead of 1 since your program is actually the second version of the PartyPlanner application—you created the first version in Chapter 10.
- 8** Select **Comments** in the **Type** list, then select the **Value** box and type **This is my first Windows application!**
- 9** Select **Company Name** in the **Type** list, then select the **Value** box and type **Event Handlers Incorporated.**
- 10** Select **Product Name** in the **Type** list, then select the **Value** box and type **Windows Party Planner.**
- 11** Click the **OK** button to close the Advanced...Properties dialog box.
- 12** You use the Package Contents section of the Output Format to select the files that will be packaged with your application. The default option includes class files and other types of files your application may require, such as graphics and sound files. The drop-down list box contains several other combinations of file types you can select. Instead of using default file types, select the **These outputs** button, which allows you to select the individual files you want to include with your program. The two files listed are WindowsPartyPlanner.Java and WindowsPartyPlanner.class. Since you do not want to give users access to your source code, deselect **WindowsPartyPlanner.java**, then click the **OK** button to close the WindowsPartyPlanner Properties dialog box.
- 13** On the **Build** menu, point to **Build Configuration** and click **Release**. This command instructs Visual J++ to create a release version of the application the next time you build the project.
- 14** Build the project. When the project is finished building, open Windows Explorer and click the OutputDirectory folder in the Chapter.16 folder on your Student Disk. You should see two files: WindowsPartyPlanner.exe and WindowsPartyPlanner.class. Right click once on the **WindowsParty Planner.exe** file and select **Properties** from the shortcut menu. The WindowsPartyPlanner.exe Properties dialog box displays. Click the **Version** tab, and then click each of the entries in the **Item name** list. You should see the information from Steps 7, 8, 9, and 10 that you entered. Some of the entries are created automatically, such as the Language item. Click **OK** to close the WindowsPartyPlanner.exe Properties dialog box.
- 15** Execute the WindowsPartyPlanner.exe file by double-clicking **Windows PartyPlanner.exe**. The Windows Party Planner should run the same as when you ran it from within Visual J++.



## S U M M A R Y

- Components in WFC applications can access standard Windows events that are available as methods in the Control class.
- Unlike the single actionPerformed(Event e) method in standard Java applications, each control in a WFC program can have its own Click method.
- A delegate is a wrapper class that is used for passing one method to another method.
- WFC delegate classes are comparable to AWT event interfaces.
- When you register a component in a WFC application, you “delegate” the responsibility for the event to an event handler method.
- An event handler is a method created to run in response to a particular event. An event handler receives two arguments from a delegate: a reference to the component that initiated the event, and the event object itself.
- In WFC applications, Visual J++ automatically creates an event’s delegate and event handler method for you.
- When you create a number of classes that inherit from each other, you can place these classes in a package.
- When you create classes for others to use, you most often do not want to provide users with your source code in .java files. Rather, you provide users with compiled .class files.
- You place .class files in a package so other programmers can import them.
- To place compiled code in a specific folder, you include the package statement at the beginning of your .class file.
- Package statements, import statements, and comments are the only statements that appear outside class definitions in Java program files.
- Because the Java programming language is used extensively on the Internet, it is important to give every package a unique name. The convention for naming packages involves using your Internet domain name in reverse order.
- The JVIEW program that comes with Visual J++ is freely distributable to users of your programs.
- Visual J++ enables you to compile your project in several types of packages, or output formats, to make distribution easier. A CAB file is a Microsoft format that contains compressed files. A Windows EXE file is a standard file that starts applications on Windows platforms.
- Visual J++ has two types of build configurations: debug and release. A debug build includes information that assists in the debugging process. A release build is the version that is distributed to your program’s users and does not contain debugging information.



## Q U E S T I O N S

1. WFC controls are registered for events using the \_\_\_\_\_ method.
  - a. registerEvent()
  - b. add<event>Listener()
  - c. addOn<event>()
  - d. <event>Register()

2. A wrapper class that is used for passing one method to another method is called a \_\_\_\_\_.
  - a. parameter
  - b. constructor
  - c. function
  - d. delegate
3. Which is the correct syntax to add a delegate to a component named myButton?
  - a. `myButton.addOnClick(new EventHandler());`
  - b. `myButton.addOnClick(new EventHandler(Click));`
  - c. `myButton.addOnClick(new EventHandler(this.myButton_Click));`
  - d. `myButton. (new EventHandler(this.myButton_Click));`
4. The WFC delegate that is most similar to the AWT ActionListener interface is \_\_\_\_\_.
  - a. EventHandler
  - b. MouseEventHandler
  - c. KeyEventHandler
  - d. KeyListener
5. A(n) \_\_\_\_\_ is a method created to run in response to a particular event.
  - a. event controller
  - b. response function
  - c. control routine
  - d. event handler
6. An event handler receives two arguments from a delegate: a reference to the component that initiated the event, and the \_\_\_\_\_.
  - a. type of Windows platform
  - b. Java version number
  - c. event object itself
  - d. name of the application
7. Which is the correct syntax for the header of a click event handler for a component named myButton?
  - a. `private void myButton_click(Event e)`
  - b. `private void myButton_click(Object source)`
  - c. `private void myButton_click(Object source, Event e)`
  - d. `private void myButton_click()`
8. You place class groups in \_\_\_\_\_ so you or other programmers can easily import them into new programs.
  - a. abstract classes
  - b. interfaces
  - c. packages
  - d. bundles
9. The files you usually place in packages are files with the extension \_\_\_\_\_.
  - a. .doc
  - b. .class
  - c. .java
  - d. .javac

10. You include a(n) \_\_\_\_\_ statement at the beginning of a class file to place the compiled class code into the indicated folder.
  - a. build
  - b. bundle
  - c. export
  - d. package
11. The location of custom packages that are imported into a Java file must be specified in the \_\_\_\_\_ tab of the Project Properties dialog box.
  - a. Classpath
  - b. Launch
  - c. Compiler
  - d. Custom Build Rules
12. If your Internet domain name is mycompany.com, then your package names should begin with \_\_\_\_\_.
  - a. mycompanycom
  - b. commycompany
  - c. com.mycompany
  - d. mycompany.com
13. The Standard edition of Visual J++ can package files as Windows EXE and \_\_\_\_\_ package types.
  - a. Setup
  - b. COM DLL
  - c. ZIP
  - d. CAB
14. Visual J++ projects have two types of build configurations: debug and \_\_\_\_\_.
  - a. final
  - b. release
  - c. completion
  - d. non-debug



## E X E R C I S E S

Save each of the programs that you create in the exercises in the Chapter.16 folder on your Student Disk.

1. Create a WFC program named ButtonEvents and add a button to the program's form. Add an event handler for at least three of the button's events. Use a MessageBox to display the name of the event each time it is executed.
2. Create a WFC program named SharedEvents. Add three buttons to the program: Button1, Button2, and Button3. Also add a label. Add to each button a delegate that calls a single event handler method. Each time you click a button, display the button's name in the label.

3.
  - a. Create a WFC program that calculates the weekly salary for an individual based on a regular hourly rate. Allow the user to input the hourly rate and then pay the user for a 40-hour week. Use two forms. Each form should have a text box. Use one form for input and one for output showing the total salary for the week. On the input form, include a button that displays the output form, and on the output form, include a button that displays the input form.
  - b. Write a WFC program that calculates the weekly salary for an individual based on a regular hourly rate, plus a premium percent for overtime. Allow the user to input the hourly rate, the overtime premium percent, the regular hours worked, and the overtime hours worked.
4. Write a WFC program that calculates the new balance of a checking account based on the current balance, a check, or a deposit. Allow the user to enter the current balance, a check amount, and a deposit amount.
5. Create a Java class file named `stringPackage` and declare a `String` named `packageString`. Assign the text “You have imported a package!” to `packageString`. Write a public method named `getPackageString` that returns `packageString`. Export the class to a package named `com.stringPackage`. Create a WFC application that imports the `stringPackage`. Add to the WFC program’s main form a button that calls the `getPackageString` method in the `stringPackage` class.