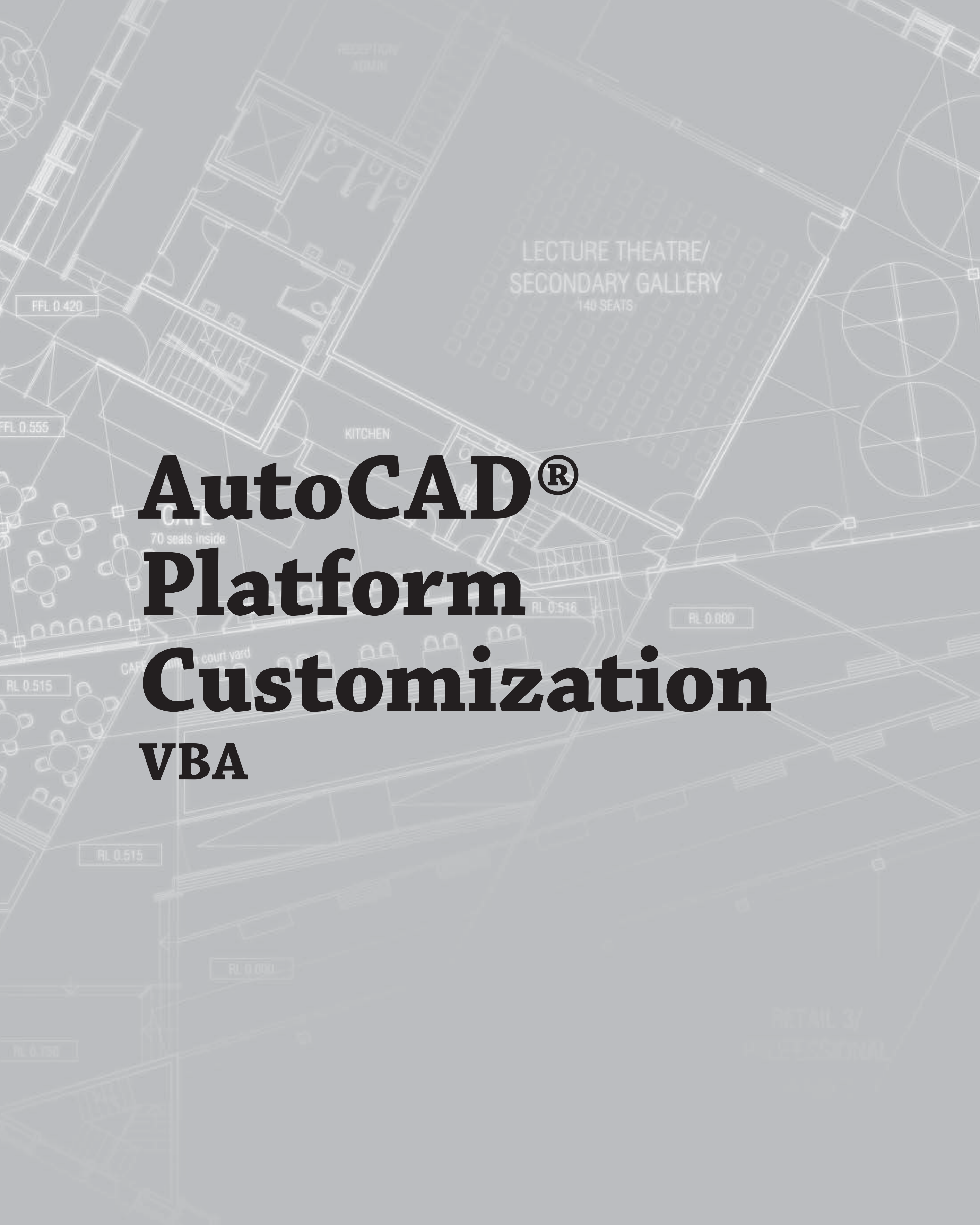
LEE AMBROSIUS

AutoCAD® Platform Customization

VBA

SYBEX
A Wiley Brand



AutoCAD® Platform Customization

VBA

AutoCAD® Platform Customization

VBA

Lee Ambrosius

Autodesk®
Official Training Guide

SYBEX®
A Wiley Brand

Senior Acquisitions Editor: Stephanie McComb
Development Editor: Mary Ellen Schutz
Technical Editor: Richard Lawrence
Production Editor: Dassi Zeidel
Copy Editor: Liz Welch
Editorial Manager: Pete Gaughan
Production Manager: Kathleen Wisor
Associate Publisher: Jim Minatel
Book Designers: Maureen Forys, Happenstance Type-O-Rama; Judy Fung
Proofreader: Candace Cunningham
Indexer: Ted Laux
Project Coordinator, Cover: Brent Savage
Cover Designer: Wiley
Cover Image: © Smileyjoanne/iStockphoto.com

ISBN: 978-1-118-90044-4 (ebk.)
ISBN: 978-1-118-90698-9 (ebk.)

Library of Congress Control Number: 2015936845

10 9 8 7 6 5 4 3 2 1

To my friend Kathy Enderby: You were one of the first people to encourage me to follow my passion for programming and sharing what I had learned with others. Thank you for believing in me all those years ago and for being there when I needed someone to bounce ideas off—especially during those late-night scrambles right before deploying a new software release.

Acknowledgments

I have to give a very special thanks to all the great folks at Sybex, especially Willem Knibbe, for working on and helping to get this project off the ground after a few years of talking about it. The next two people I would like to thank are Mary Ellen Schutz and Dassi Zeidel, the development and production editors on this book; you two made sure I stayed on track and delivered a high-quality book. I also want to thank Liz Welch (copyeditor), Candace Cunningham (proofreader), and Ted Laux (indexer) for the work you all did on this book.

Thanks to all the folks at Autodesk, who put in the long hours and are dedicated to the work they do on the Autodesk® AutoCAD® product. I cannot forget one of the most important individuals on this book, my technical editor, Richard Lawrence. Richard is a great friend who I met many years ago at Autodesk University. He is a passionate and driven user of AutoCAD and is always looking to improve the way he uses AutoCAD. Richard, I appreciate everything that you have done to make this book better. Congrats on making it through your first book as a technical editor.

About the Author

Lee Ambrosius first started working with AutoCAD R12 for DOS in 1994. As a drafter, he quickly discovered that every project included lots of repetition. Lee, not being one to settle for "this is just the way things are," set out on a path that would redefine his career. This new path would lead him into the wondrous world of customization and programming—which you might catch him referring to as "the rabbit hole."

In 1996, Lee began learning the core concepts of customizing the AutoCAD user interface and AutoLISP. The introduction of VBA in AutoCAD R14 would once again redefine how Lee approached programming solutions for AutoCAD. VBA made it much easier to communicate with external databases and other applications that supported VBA. It transformed the way information could be moved between project-management and manufacturing systems.

Not being content with VBA, in 1999 Lee attended his first Autodesk University and began to learn ObjectARX®. Autodesk University had a lasting impression on him. In 2001, he started helping as a lab assistant. He began presenting on customizing and programming AutoCAD at the event in 2004. Along the way he learned how to use the AutoCAD Managed .NET API.

In 2005, Lee decided cubicle life was no longer for him, so he ventured off into the CAD industry as an independent consultant and programmer with his own company, HyperPics, LLC. After he spent a few years as a consultant, Autodesk invited him to work on the AutoCAD team; he has been on the AutoCAD team since 2007. For most of his career at Autodesk, Lee has worked primarily on customization and end-user documentation. Recently, he has been working on the AutoLISP, VBA, ObjectARX, .NET, and JavaScript programming documentation.

In addition to working on the AutoCAD documentation, Lee has been involved as a technical editor or author for various editions of the *AutoCAD and AutoCAD LT Bible*, *AutoCAD for Dummies*, *AutoCAD & AutoCAD LT All-in-One Desk Reference for Dummies*, *AutoCAD 3D Modeling Workbook for Dummies*, and *Mastering AutoCAD for Mac.* He has also written white papers on customization for Autodesk and a variety of articles on customization and programming for *AUGIWorld*, published by AUGI®.

Contents at a Glance

Contents

Introduction

Welcome to *AutoCAD Platform Customization: VBA*! Have you ever thought to yourself, why doesn't the Autodesk® AutoCAD® program include every feature I need? Why isn't it streamlined for the type of work I perform? If so, you are not alone. AutoCAD at its core is a drafting platform that, through programming, can be shaped and molded to more efficiently complete the tasks you perform on a daily basis and enhance your company's workflows. Take a deep breath. I did just mention programming, but programming isn't something to fear. At first, just the idea of programming makes many people want to run in the opposite direction—myself included. The productivity gains are what propelled me forward. Programming isn't all that different from anything else you've tried doing for the first time.

In many ways, learning to program is much like learning a foreign language. For many new to Visual Basic for Applications (VBA), the starting place is learning the basics: the syntax of the programming language and how to leverage commands and system variables. Executing commands and working with system variables using the `SendCommand` and `PostCommand` methods can be a quick way to get started and become comfortable with VBA. After you are comfortable with the syntax of VBA and the `SendCommand` and `PostCommand` functions, you can begin to learn how to access the AutoCAD Object library to develop more complex and robust programs.

About This Book

AutoCAD Platform Customization: VBA provides you with an understanding of the VBA programming language and how it can be used in combination with the AutoCAD Object library to improve your productivity. This book is designed to be more than just an introduction to VBA and the AutoCAD Object library; it is a resource that can be used time and again when developing VBA programs for use with AutoCAD. As you page through this book, you will notice that it contains sample code and exercises that are based on real-world solutions.

This book is the third and final book in a series that focuses on customizing and programming AutoCAD. The three-book series as a whole is known as *AutoCAD Platform Customization: User Interface, AutoLISP, VBA, and Beyond*, which will be available as a printed book in 2015. Book 1 in the series, *AutoCAD Platform Customization: User Interface and Beyond*, was published in early 2014 and focused on CAD standards and general customization of AutoCAD; Book 2, *AutoCAD Platform Customization: AutoLISP*, was published in mid-2014 and covers the AutoLISP programming language.

Is This Book for You?

AutoCAD Platform Customization: VBA covers many aspects of VBA programming for AutoCAD on Windows. If any of the following are true, this book will be useful to you:

- You want to develop and load custom programs with the VBA programming language for use in the AutoCAD drawing environment.
- You want to automate the creation and manipulation of drawing objects.
- You want to automate repetitive tasks.
- You want to help manage and enforce CAD standards for your company.

NOTE VBA programming isn't supported for AutoCAD on Mac OS.

VBA in AutoCAD

VBA is often overlooked as one of the options available to extend the AutoCAD program. There is no additional software to purchase, but you must download and install a release-specific secondary component to use VBA. You can leverage VBA to perform simple tasks, such as inserting a title block with a specific insertion point, scale, and rotation and placing the block reference on a specific layer. To perform the same tasks manually, end users would have to first set a layer as current, choose the block they want to insert, and specify the properties of the block, which in the case of a title block are almost always the same.

The VBA programming language and AutoCAD Object library can be used to do the following:

- Create and manipulate graphical objects in a drawing, such as lines, circles, and arcs
- Create and manipulate nongraphical objects in a drawing, such as layers, dimension styles, and named views
- Perform mathematical and geometric calculations
- Request input from or display messages to the user at the Command prompt
- Interact with files and directories in the operating system
- Read from and write to external files
- Connect to applications that support ActiveX and COM
- Display user forms and get input from the end user

VBA code statements are entered into the Visual Basic Editor and stored in a DVB file. Once a VBA project has been loaded, you can execute the macros through the Macros dialog box. Unlike standard AutoCAD commands, macros cannot be executed from the Command prompt, but once executed, a macro can prompt users for values at the Command prompt or with a user form. It is possible to execute a macro from a command macro that is activated with a command button displayed in the AutoCAD user interface or as a tool on a tool palette.

What to Expect

This book is organized to help you learn VBA fundamentals and how to use the objects in the AutoCAD Object library. Additional resources and files containing the example code found throughout this book can be found on the companion web page, `www.sybex.com/go/autocadcustomization`.

Chapter 1: Understanding the AutoCAD VBA Environment In this chapter, you'll get an introduction to the Visual Basic Editor. I begin by showing you how to verify whether the VBA environment for AutoCAD has been installed and, if not, how to install it. After that, you are eased into navigating the VBA Editor and managing VBA programs. The chapter wraps up with learning how to execute macros and access the help documentation.

Chapter 2: Understanding the Visual Basic for Application In this chapter, you'll learn the fundamentals of the VBA programming language and how to work with objects. VBA fundamentals include a look at the syntax and structure of a statement, how to use a function, and how to work with variables. Beyond syntax and variables, you learn to group multiple statements into a custom procedure.

Chapter 3: Interacting with the Application and Documents Objects In this chapter, you'll learn to work with the AutoCAD application and manage documents. Many of the tasks you perform with an AutoCAD VBA program require you to work with either the application or a document. For example, you can get the objects in a drawing and even access end-user preferences. Although you typically work with the current document, VBA allows you to work with all open documents and create new documents. From the current document, you can execute commands and work with system variables from within a VBA program, which allows you to leverage and apply your knowledge of working with commands and system variables.

Chapter 4: Creating and Modifying Drawing Objects In this chapter, you'll learn to create and modify graphical objects in model space with VBA. Graphical objects represent the drawing objects, such as a line, an arc, or a circle. The methods and properties of an object are used to modify and obtain information about the object. When working with the objects in a drawing, you can get a single object or step through all objects in a drawing.

Chapter 5: Interacting with the User and Controlling the Current View In this chapter, you'll learn to request input from an end user and manipulate the current view of a drawing. Based on the values provided by the end user, you can then determine the end result of the program. You can evaluate the objects created or consider how a drawing will be output and use that information to create named views and adjust the current view in which objects are displayed.

Chapter 6: Annotating Objects In this chapter, you'll learn how to create and modify annotation objects. Typically, annotation objects are not part of the final product that is built or manufactured based on the design in the drawing. Rather, annotation objects are used to communicate the features and measurements of a design. Annotation can be a single line of text that is used as a callout for a leader, a dimension that indicates the distance between two drill holes, or a table that contains quantities and information about the windows and doors in a design.

Chapter 7: Working with Blocks and External References In this chapter, you'll learn how to create, modify, and manage block definitions. Model space in a drawing is a special named block definition, so working with block definitions will feel familiar. Once you create a block definition, you will learn how to insert a block reference and work with attributes along with dynamic properties. You'll complete the chapter by learning how to work with externally referenced files.

Chapter 8: Outputting Drawings In this chapter, you will learn how to output the graphical objects in model space or on a named layout to a printer, plotter, or electronic file. Named layouts will be used to organize graphical objects for output, including title blocks, annotation, floating viewports, and many others. Floating viewports will be used to control the display of objects from model space on a layout at a specific scale. After you define and configure a layout, you learn to plot and preview a layout. The chapter wraps up with covering how to export and import file formats.

Chapter 9: Storing and Retrieving Custom Data In this chapter, you will learn how to store custom information in a drawing or in the Windows Registry. Using extended data (Xdata), you will be able to store information that can be used to identify a graphical object created by your program or define a link to a record in an external database. In addition to attaching information to an object, you can store data in a custom dictionary that isn't attached to a specific graphical object in a drawing. Both Xdata and custom dictionaries can be helpful in making information available between drawing sessions; the Windows Registry can persist data between sessions.

Chapter 10: Modifying the Application and Working with Events In this chapter, you will learn how to customize and manipulate the AutoCAD user interface. You'll also learn how to load and access externally defined custom programs and work with events. Events allow you to respond to an action that is performed by the end user or the AutoCAD application. There are three main types of events that you can respond to: application, document, and object.

Chapter 11: Creating and Displaying User Forms In this chapter, you will learn how to create and display user forms. User forms provide a more visual approach to requesting input from the user.

Chapter 12: Communicating with Other Applications In this chapter, you will learn how to work with libraries provided by other applications. These libraries can be used to access features of the Windows operating system, read and write content in an external text or XML file, and even work with the applications that make up Microsoft Office.

Chapter 13: Handling Errors and Deploying VBA Projects In this chapter, you will learn how to catch and handle errors that are caused by the incorrect use of a function or the improper handling of a value that is returned by a function. The Visual Basic Editor provides tools that allow you to debug code statements, evaluate values assigned to user-defined variables, identify where within a program an error has occurred, and determine how errors should be handled. The chapter wraps everything up with covering how to deploy a VBA project on other workstations for use by individuals at your company.

Bonus Chapter 1: Working with 2D Objects and Object Properties In this chapter, you build on the concepts covered in Chapter 4, “Creating and Modifying Drawing Objects.” You will learn to create additional types of 2D objects and use advanced methods of modifying objects; you also learn to work with complex 2D objects such as regions and hatch fills. The management of layers and linetypes and the control of the appearance of objects are also covered.

Bonus Chapter 2: Modeling in 3D Space In this chapter, you learn to work with objects in 3D space and 3D with objects. 3D objects can be used to create a model of a drawing which can be used to help visualize a design or detect potential design problems. 3D objects can be viewed from different angles and used to generate 2D views of a model that can be used to create assembly directions or shop drawings.

Bonus Chapter 3: Development Resources In this chapter, you discover resources that can help expand the skills you develop from this book or locate an answer to a problem you might encounter. I cover development resources, as well as places you might be able to obtain instructor-led training and interact with fellow users on extending AutoCAD. The online resources listed cover general customization, AutoLISP, and VBA programming in AutoCAD.

NOTE Bonus Chapter 1, Bonus Chapter 2, and Bonus Chapter 3 are located on the companion website.

Companion Website

An online counterpart to this book, the companion website contains the sample files required to complete the exercises found in this book, in addition to the sample code and project files used to demonstrate some of the programming concepts explained in this book. In addition to the sample files and code, the website contains resources that are not mentioned in this book, such as the bonus chapters. The companion website can be found at `www.sybex.com/go/autocadcustomization`.

Other Information

This book assumes that you know the basics of your operating system and AutoCAD 2009 or later. When appropriate, I indicate when a feature does not apply to a specific operating system or release of AutoCAD. Most of the images in this book were taken using AutoCAD 2014 in Windows 8.

Neither AutoCAD LT® nor AutoCAD running on Mac OS support the VBA programming platform, none of the content in this book can be used if you are working on Mac OS.

Styles and Conventions of This Book

This book uses a number of styles and character formats—bold, italic, monotype face, and all uppercase or lowercase letters, among others—to help you distinguish between the text you read, sample code you can try, text that you need to enter at the AutoCAD Command prompt, or the name of an object class or method in one of the programming languages.

As you read through this book, keep the following conventions in mind:

- User-interface selections are represented by one of the following methods:
  - Click the Application button ➢ Options.
  - On the ribbon, click the Manage tab ➢ Customization ➢ User Interface.
  - On the menu bar, click Tools ➢ Customize ➢ Interface.
  - In the drawing window, right-click and click Options.
- Keyboard input is shown in bold (for example, type **`cui`** and press Enter).
- Prompts that are displayed at the AutoCAD Command prompt are displayed as monospace font (for example, `Specify a start point:`).
- AutoCAD command and system variable names are displayed in all lowercase letters with a monospace font (for example, `line` or `clayer`).
- VBA function and AutoCAD Object library member names are displayed in mixed-case letters with a monospace font (for example, `Length` or `SendCommand`).
- Example code and code statements that appear within a paragraph are displayed in monospace font. Code might look like one of the following:
  - `MsgBox "ObjectName: " & oFirstEnt.ObjectName`
  - The `MsgBox` method can be used to display a text message to the user
  - `' Gets the first object in model space`

Contacting the Author

I hope that you enjoy *AutoCAD Platform Customization: VBA* and that it changes the way you think about completing your day-to-day work. If you have any feedback or ideas that could improve this book, you can contact me using the following address:

Lee Ambrosius: `lee_ambrosius@hyperpics.com`

On my blog and website, you'll find additional articles on customization and samples that I have written over the years. You'll find these resources here:

Beyond the UI: `http://hyperpics.blogs.com`

HyperPics: `www.hyperpics.com`

If you encounter any problems with this publication, please report them to the publisher. Visit the book's website, `www.sybex.com/go/autocadcustomization`, and click the Errata link to open a form and submit the problem you found.

Chapter 1

Understanding the AutoCAD VBA Environment

More than 15 years ago, Visual Basic (VB) was the first modern programming language I learned. This knowledge was critical to taking my custom programs to a whole new level. VB allows you to develop stand-alone applications that can communicate with other programs using Microsoft's Object Linking and Embedding (OLE) and ActiveX technologies. Autodesk® AutoCAD® supports a variant of VB known as Visual Basic for Applications (VBA) that requires a host application to execute the programs you write; it can't be used to develop stand-alone executable files.

I found VB easier to learn than AutoLISP® for a couple of reasons. First, there are, in general, many more books and online resources dedicated to VB. Second, VB syntax feels more natural. By natural, I mean that VB syntax reads as if you are explaining a process to someone in your own words, and it doesn't contain lots of special characters and formatting like many other programming languages.

As with learning anything new, there will be a bit of hesitation on your part as you approach your first projects. This chapter eases you into the AutoCAD VBA environment and the VB programming language.

What Makes Up an AutoCAD VBA Project?

Custom programs developed with VBA implemented in the AutoCAD program are stored in a project that has a `.dvb` file extension. VBA projects contain various objects that define a custom program. These objects include the following:

- Code modules that store the custom procedures and functions that define the primary functionality of a custom program
- `UserForms` that define the dialog boxes to be displayed by a custom program
- Class modules that store the definition of a custom object for use in a custom program
- Program library references that contain the dependencies a custom program relies on to define some or all of the functionality

The AutoCAD VBA Editor is an integrated development environment (IDE) that allows for the creation and execution of macros stored in a project file. A macro is a named block of code that can be executed from the AutoCAD user interface or privately used within a project. You can also enter and execute a single VBA statement at the AutoCAD Command prompt using the `vbastmt` command.

The most recent generation of VB is known as VB.NET. Although VB and VB.NET have similar syntax, they are not the same. VBA, whether in AutoCAD or other programs such as Microsoft Word, is based on VB6 and not VB.NET. If you are looking for general information on VBA, search the Internet using the keywords VBA and VB6.

Real World Scenario

If You Have Conversations Like This, You Can Code Like This

The summer intern had one job—add a layer and a confidentiality note to a series of 260 production drawings. September arrived, the intern left for school, and now your manager is in your cubicle.

"Half of these drawings are missing that confidentiality note Purchasing asked for. I need you to **add** that new **layer**, name it **Disclaimer**, and then **add** the confidentiality note as **multiline text** to **model space**. The note should be located on the new **Disclaimer** layer at **0.25,0.1.75,0** with a **height** of **0.5,** and the text should read **Confidential: This drawing is for use by internal employees and approved vendors only**. Be sure to check to see **if paper space** is **active.** If it is, **then** set **model space active** per the new standards before you **save** each drawing," he says.

"I can do that," you respond.

"Can you manage it by close of day tomorrow? The parts are supposed to go out for quote on Wednesday morning."

"Sure," you tell him, knowing that a few lines of VBA code will allow you to make the changes quickly.

So, you sit down and start to code. The conversation-to-code translation flows smoothly. (Notice how many of the words in the conversation flow right into the actual VBA syntax.)

```
With ThisDrawing
  .Layers.Add "Disclaimer"

  Dim objMText As AcadMText

  Dim insPt(2) As Double
  insPt(0) = 0.25: insPt(1) = 1.75: insPt(2) = 0

  Set objMText = .ModelSpace.AddMText(insPt, 15, _
    "Confidential: This drawing is for use by internal " & _
    "employees and approved vendors only")

  objMText.Layer = "Disclaimer"
  objMText.Height = 0.5

  If .ActiveSpace = acPaperSpace Then
    .ActiveSpace = acModelSpace
   End If

  .Save
End With
```

What You'll Need to Start

To complete the exercises in this chapter and create and edit VBA project files, you must have the following:

- AutoCAD 2006 or later
- Autodesk AutoCAD VBA Enabler for AutoCAD 2010 or later

Beginning with AutoCAD 2010, the AutoCAD VBA Enabler is an additional component that must be downloaded and installed to enable VBA support in the AutoCAD drawing environment. (For AutoCAD 2000 through AutoCAD 2009, VBA capabilities were part of a standard install.)

NOTE The Autodesk website (`http://www.autodesk.com/vba-download`) allows you to download the Autodesk AutoCAD VBA Enabler for AutoCAD 2014 and 2015 (Microsoft Visual Basic for Applications Module). If you need the VBA Enabler for AutoCAD 2010 through 2013, you will want to check with your local Autodesk Value Added Reseller.

Without the VBA Enabler, you won't have access to the VBA Editor and can't create or execute VBA code contained in a DVB file with AutoCAD 2010 and later releases. All of the VBA commands were available without an additional download and install. Changes in the later AutoCAD releases were made due to Microsoft's planned deprecation of the VBA technology and editor, only to eventually extend its life cycle because of its continued importance to Microsoft Office. Microsoft planned to move to Visual Studio Tools for Applications (VSTA) as the replacement for VBA, but the company backed off because there was no easy migration from VBA to VSTA.

NOTE Although I mention AutoCAD 2006 or later, everything covered in this chapter should work without any problems going all the way back to AutoCAD 2000. The first release of the AutoCAD program that supported VBA was AutoCAD R14, and much has remained the same since then as well, with the exception of being able to work with multiple documents in AutoCAD 2000 and later.

Determine If the AutoCAD VBA Environment Is Installed

Prior to working with the AutoCAD VBA Editor, you must ensure that the VBA environment is installed on your workstation. The following steps explain how to determine whether VBA is installed and, if necessary, how to download the AutoCAD VBA environment for installation. These steps are important if you are using AutoCAD 2010 or later.

1. Launch AutoCAD if it isn't already running.
2. At the Command prompt, type **`vbaide`** and press Enter.
3. If the VBA - Not Installed message box is displayed, the AutoCAD VBA environment hasn't been installed. Continue to the next step.
4. Click the `http://www.autodesk.com/vba-download` link to open your system's default web browser to the download website.

5. Click the link for the AutoCAD VBA Enabler that matches the version of AutoCAD installed on your workstation.
6. Save the AutoCAD VBA Enabler to a folder on your local workstation.

Install the AutoCAD 2015 VBA Enabler

After downloading the AutoCAD 2015 VBA Enabler using the steps explained in the previous section, follow these steps to install it:

1. Close the AutoCAD program and double-click the downloaded self-extracting executable for the AutoCAD VBA module.
2. In the Extract To message, accept the default destination location and click OK.
3. When the AutoCAD VBA Enabler installer opens, click Install.
4. On the next page of the installer, accept the default destination location and click Install.
5. On the Installation Complete page, click Finish.
6. Launch AutoCAD.
7. At the Command prompt, type **`vbaide`** and press Enter.

   The VBA Editor is displayed, indicating that the AutoCAD VBA environment has been installed.

NOTE If you downloaded the VBA Enabler for a different release of the AutoCAD program, follow the on-screen instructions for that release of the VBA Enabler.

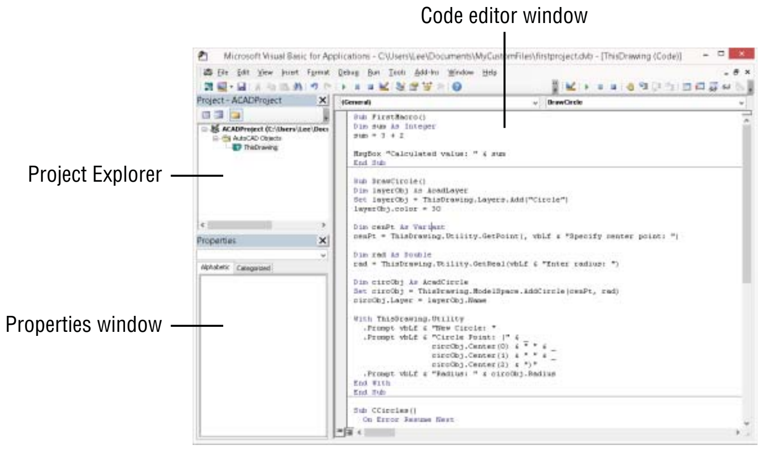
Getting Started with the VBA Editor

The VBA Editor (see Figure 1.1) is the authoring environment used to create custom programs that are stored in a VBA project. The following tasks can be performed from the VBA Editor:

- Access and identify the components in a VBA project
- View and edit the code and components stored in a loaded VBA project
- Debug the code of a procedure during execution
- Reference programming libraries
- Display contextual help based on the code or component being edited

Any of the following methods can be used to display the VBA Editor:

- On the ribbon, click the Manage tab ➢ Applications panel ➢ Visual Basic Editor.
- At the Command prompt, type **`vbaide`** and press Enter.
- When the VBA Manager is open, click Visual Basic Editor.
- When loading a VBA project, in the Open VBA Project dialog box, check the Open Visual Basic Editor check box before clicking Open.


Code editor window
Project Explorer
Properties window
Microsoft Visual Basic for Applications - C:\Users\Lee\Documents\MyCustomFiles\firstproject.dvb - [ThisDrawing (Code)]
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Project - ACADProject
ACADProject (C:\Users\Lee\Docs
AutoCAD Objects
ThisDrawing
Properties
Alphabetic Categorized
(General)
DrawCircle

```
Sub FirstMacro()
Dim sum As Integer
sum = 3 + 2

MsgBox "Calculated value: " & sum
End Sub

Sub DrawCircle()
Dim layerObj As AcadLayer
Set layerObj = ThisDrawing.Layers.Add("Circle")
layerObj.color = 30

Dim cenPt As Variant
cenPt = ThisDrawing.Utility.GetPoint(, vbLf & "Specify center point: ")

Dim rad As Double
rad = ThisDrawing.Utility.GetReal(vbLf & "Enter radius: ")

Dim circObj As AcadCircle
Set circObj = ThisDrawing.ModelSpace.AddCircle(cenPt, rad)
circObj.Layer = layerObj.Name

With ThisDrawing.Utility
  .Prompt vbLf & "New Circle: "
  .Prompt vbLf & "Circle Point: (" & _
                  circObj.Center(0) & ", " & _
                  circObj.Center(1) & ", " & _
                  circObj.Center(2) & ")"
  .Prompt vbLf & "Radius: " & circObj.Radius
End With
End Sub

Sub CCircles()
  On Error Resume Next
```

FIGURE 1.1 The VBA Editor allows for the development of a VBA program

Identifying the Components of a VBA Project

VBA supports four types of components to define the functionality of a custom program. Each component can be used to store code, but the code in each component serves a distinct purpose within a VBA project. Before you begin learning the basic features of the VBA Editor, you should have a basic understanding of the component types in a VBA project.

The following provides an overview of the component types:

Code Module Code modules, also referred to as standard code modules, are used to store procedures and define any global variables for use in the module or globally across the VBA project. I recommend using code modules to store procedures that can be executed from the AutoCAD user interface or used across multiple projects.

When you add a new code module to a VBA project, you should give the module a meaningful name and not keep the default name of `Module1`, `Module2`, and so on. Standard industry naming practice is to add the prefix of `bas` to the name of the module. For example, you might name a module that contains utility procedures as `basUtilities`. I explain how to define procedures and variables in the "Learning the Fundamentals of the VBA Language" section in Chapter 2, "Understanding Visual Basic for Applications."

Class Module Class modules are used to define a custom class—or *object*. Custom classes aren't as common as code modules in a VBA project, but they can be helpful in organizing and simplifying code. The variables and procedures defined in a class module are hidden from all other components in a VBA project, unless an instance of the class is created as part of a procedure in another component.

When you add a new class module to a VBA project, you should give the module a meaningful name and not keep the default name of `Class1`, `Class2`, and so on. Standard industry naming practice is to add the prefix of `cls` to the name of the module. For example, you might name a module that contains a custom class named employee as `clsEmployee`. I explain how to define procedures and variables and work with objects in the "Learning the Fundamentals of the VBA Language" section in Chapter 2.

ThisDrawing ThisDrawing is a specially named object that represents the current drawing and is contained in each VBA project. The ThisDrawing component can be used to define events that execute code based on an action performed by the user or the AutoCAD program. Variables and procedures can be defined in the ThisDrawing component, but I recommend storing only the variables and procedures related to the current drawing in the ThisDrawing component. All other code should be stored in a code module. I explain how to work with the current drawing and events in Chapter 3, "Interacting with the Application and Documents Objects," and Chapter 10, "Modifying the Application and Working with Events."

`UserForm` `UserForms` are used to define custom dialog boxes for use in a VBA program. A `UserForm` can contain controls that present messages to the user and allow the user to provide input. When you add a new `UserForm` to a VBA project, you should give the `UserForm` a meaningful name and not keep the default name of `UserForm1`, `UserForm2`, and so on. Standard industry naming practice is to add the prefix of `frm` to the name of the `UserForm`. For example, you might name a `UserForm` that contains a dialog box that draws a bolt as `frmDrawBolt`. I explain how to create and display a `UserForm` in Chapter 11, "Creating and Displaying User Forms."

The following explains how to add a new component to a VBA project and change its name:

1. In the VBA Editor with a project loaded, on the menu bar, click Insert.
2. Click UserForm, Module, or Class Module to add a component of that type to the VBA project.
3. In the Project Explorer, select the new component.
4. In the Properties window, in the (Name) field, type a new name and press Enter.

Using Components in Multiple VBA Projects

A component added to a VBA project can be exported, and then imported into another VBA project. Exporting a component creates a copy of that component; any changes to the component in the original VBA project don't affect the exported copy of the component. Importing the component into a VBA project creates a copy of the component in that VBA project.

The following steps can be used to export a VBA component to a file:

1. In the VBA Editor, Project Explorer, select the component to export.
2. On the menu bar, click File ➢ Export File.
3. In the Export File dialog box, browse to the location to store the exported file and enter a filename. Click Save.

The following steps can be used to import an exported file into a VBA project:

1. In the VBA Editor, Project Explorer, select a loaded project to set it current.
2. On the menu bar, click File ➢ Import File.
3. In the Import File dialog box, browse to and select the exported file. Click Open.

Navigating the VBA Editor Interface

The VBA Editor interface contains a variety of tools and windows that are used to manage and edit the components and code stored in a VBA project. While all of the tools and windows in the VBA Editor will be important over time, there are four windows that you should have a basic understanding of when first getting started:

- Project Explorer
- Properties window
- Code editor window
- Object Browser

Accessing Components in a VBA Project with the Project Explorer

The Project Explorer window (see Figure 1.2) lists all of the VBA projects that are currently loaded into the AutoCAD drawing environment and the components of each loaded project. By default, the Project Explorer should be displayed in the VBA Editor, but if it isn't you can display it by clicking View ➢ Project Explorer or pressing Ctrl+R.

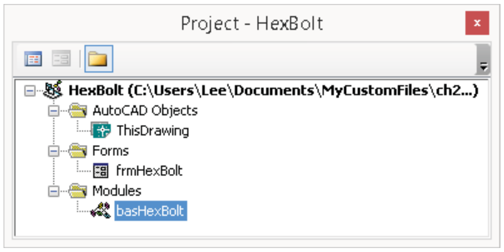


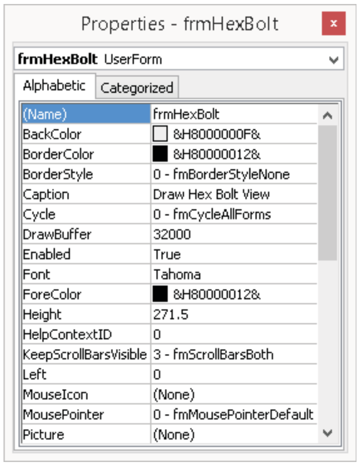
Figure 1.2
The Project Explorer lists loaded projects and components

When the Project Explorer is displayed, you can

- Select a project to set it as the current project; the name of the current project is shown in bold. Some tools in the VBA Editor work on only the current project.
- Expand a project to access its components.
- Toggle the display style for components; alphabetically listed or grouped by type in folders.
- Double-click a component to edit its code or `UserForm` in an editor window.
- Right-click to export, import, or remove a component.

Using the Properties Window

The Properties window (see Figure 1.3) allows you to change the name of a component in a loaded VBA project or modify the properties of a control or `UserForm`. Select a component or `UserForm` from the Project Explorer, or a control to display its properties in the Properties window. Click in a property field, and enter or select a new value to change the current value of the property. The Properties window is displayed by default in the VBA Editor, but if it isn't you can display it by clicking View ➢ Properties Window or pressing F4.



Properties - frmHexBolt

frmHexBolt UserForm

Alphabetic | Categorized

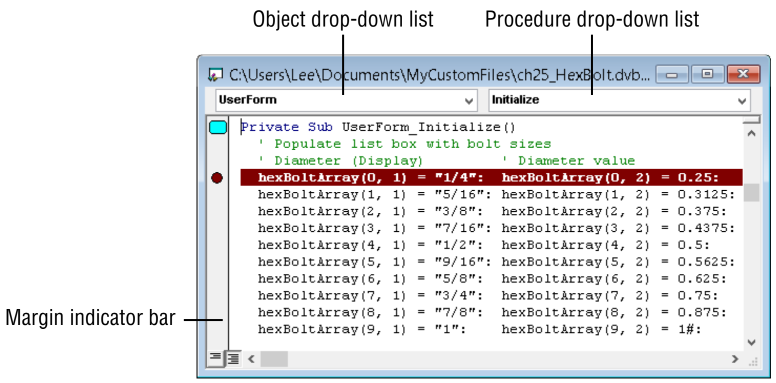
(Name)	frmHexBolt
BackColor	□ &H8000000F&
BorderColor	■ &H80000012&
BorderStyle	0 - fmBorderStyleNone
Caption	Draw Hex Bolt View
Cycle	0 - fmCycleAllForms
DrawBuffer	32000
Enabled	True
Font	Tahoma
ForeColor	■ &H80000012&
Height	271.5
HelpContextID	0
KeepScrollBarsVisible	3 - fmScrollBarsBoth
Left	0
MouseIcon	(None)
MousePointer	0 - fmMousePointerDefault
Picture	(None)

FIGURE 1.3 Modify the properties of a component, `UserForm`, or control

EDITING CODE AND CLASS MODULES IN EDITOR WINDOWS

A code editor window (see Figure 1.4) is where you will write, edit, and debug code statements that are used to make up a custom program. You display a code editor window by doing one of the following in the Project Explorer:

- Double-clicking a code or class module
- Right-clicking a `UserForm` and then clicking View Code



```
Private Sub UserForm_Initialize()
  ' Populate list box with bolt sizes
  ' Diameter (Display)          ' Diameter value
  hexBoltArray(0, 1) = "1/4":   hexBoltArray(0, 2) = 0.25:
  hexBoltArray(1, 1) = "5/16":  hexBoltArray(1, 2) = 0.3125:
  hexBoltArray(2, 1) = "3/8":   hexBoltArray(2, 2) = 0.375:
  hexBoltArray(3, 1) = "7/16":  hexBoltArray(3, 2) = 0.4375:
  hexBoltArray(4, 1) = "1/2":   hexBoltArray(4, 2) = 0.5:
  hexBoltArray(5, 1) = "9/16":  hexBoltArray(5, 2) = 0.5625:
  hexBoltArray(6, 1) = "5/8":   hexBoltArray(6, 2) = 0.625:
  hexBoltArray(7, 1) = "3/4":   hexBoltArray(7, 2) = 0.75:
  hexBoltArray(8, 1) = "7/8":   hexBoltArray(8, 2) = 0.875:
  hexBoltArray(9, 1) = "1":     hexBoltArray(9, 2) = 1#:
```

FIGURE 1.4 Edit code statements stored in a code or class module.

The code editor window supports many common editing tools: copy and paste, find and replace, and many others. In addition to common editing tools, it supports tools that are designed specifically for working with VBA code statements, and some of these tools allow you to accomplish the following:

- Autocomplete a word as you type
- Find and replace text across all components in a VBA project

- Comment and uncomment code statements
- Add bookmarks to allow you to move between procedures and code statements
- Set breakpoints for debugging

The text area is the largest area of the code editor window and where you will spend most of your time. The Object drop-down list typically is set to (General), which indicates you want to work with the General Declaration area of the code window. When working in the code editor window of a `UserForm`, you can select a control or the `UserForm` to work with from the Object drop-down list. The Object drop-down list is also used when working with events.

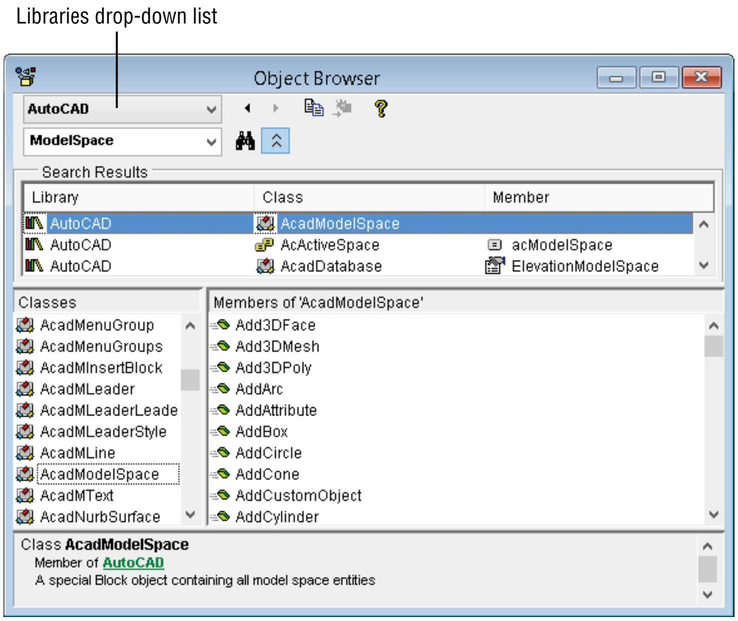
Once an object is selected, a list of available events or procedures for the selected object is displayed in the Procedure drop-down list. Select a procedure from the drop-down list to insert the basic structure of that procedure. Enter the code statements to execute when the procedure is executed. I explain how to work with events in Chapter 10 and `UserForms` in Chapter 11.

The margin indicator bar of the code editor window helps you know where a bookmark or breakpoint is inserted by displaying an oval for a bookmark or a circle for a breakpoint. I discuss more about breakpoints in Chapter 13, "Handling Errors and Deploying VBA Projects."

Exploring Loaded Libraries with the Object Browser

The Object Browser (see Figure 1.5) allows you to view the classes and enumerated constants defined in a referenced programming library. Each AutoCAD VBA project contains a reference to the VBA and AutoCAD Object libraries. I discuss referencing other libraries in Chapter 12, "Communicating with Other Applications." You can display the Object Browser by clicking View ➢ Object Browser or pressing F2.

Figure 1.5 Members of an object in a referenced library can be viewed in the Object Browser.



Library	Class	Member
AutoCAD	AcadModelSpace	
AutoCAD	AcActiveSpace	acModelSpace
AutoCAD	AcadDatabase	ElevationModelSpace

Classes: AcadMenuGroup, AcadMenuGroups, AcadMInsertBlock, AcadMLeader, AcadMLeaderLeade, AcadMLeaderStyle, AcadMLine, AcadModelSpace, AcadMText, AcadNurbSurface

Members of 'AcadModelSpace': Add3DFace, Add3DMesh, Add3DPoly, AddArc, AddAttribute, AddBox, AddCircle, AddCone, AddCustomObject, AddCylinder

Class **AcadModelSpace**
Member of AutoCAD
A special Block object containing all model space entities

A class is used to create an instance of an object, which I discuss in the "Working with Objects" section in Chapter 2. An enumerated constant is a set of integer values with unique names that can be used in a code statement. Using a constant name makes the integer value easier to understand, and also protects your code when values change. For example, the constant name of `acBlue` is equal to an integer value of 5. If the meaning of 5 were changed to mean a different color than blue, the constant of `acBlue` would be updated with the new integer and no changes to your code would need to be made if you used the constant.

When the Object Browser is displayed, you can select a class or enumerated constant from the Classes list. The Classes list contains all the classes and enumerated constants of the referenced libraries in the VBA project. You can filter the list by selecting a referenced library from the Libraries drop-down list located at the top of the Object Browser. Select a class or enumerated constant from the Classes list to see its members, which are methods, properties, events, or constant values. Select a member to learn more about it and press F1 to display its associated help topic. I explain how to access the AutoCAD VBA documentation in the "Accessing the AutoCAD VBA Documentation" section later in this chapter.

Working with Other Windows

The four windows that I described in the previous sections are the main windows of the VBA Editor; they are used the most frequently. You will use some additional windows on occasion. These are primarily used for creating `UserForms` and debugging VBA statements. (I discuss creating `UserForms` in Chapter 11 and debugging in Chapter 13.)

Here are the windows you will use when creating `UserForms` and debugging:

Immediate Window The Immediate window allows you to execute code statements in real time, but those code statements are not saved. Not all code statements can be executed in the Immediate window, such as statements that define a procedure and declare variables. Text messages and values assigned to a variable can be output to the Immediate window for debugging purposes with the `Print` method of the `Debug` object. I discuss more about the `Debug` object and Immediate window in Chapter 13.

Watches Window The Watches window allows you to monitor the current value assigned to the variables used in the procedures of your VBA project as they are being executed. When an array or object is assigned to a variable, you can see the values assigned to each element in the array and the current property values of the object in the Watches window. In the code editor window, highlight the variable you want to watch, and right-click. Click Add Watch and then when the Add Watch dialog box opens click OK. I discuss more about the Watches window in Chapter 13.

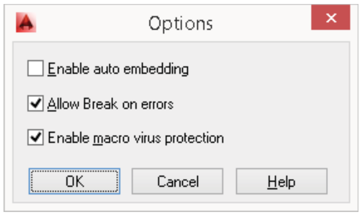
`UserForm` Editor Window The `UserForm` editor window allows you to add controls and organize controls on a `UserForm` to create a custom dialog box that can be displayed from your VBA project. You add controls to a `UserForm` from the Toolbox window. While the `UserForm` editor window is current, the Format menu on the menu bar contains tools to lay out and align the controls on a `UserForm`. I explain how to create and work with `UserForms` in Chapter 11.

Toolbox Window The Toolbox window contains the controls that can be added to a `UserForm` when displayed in the `UserForm` editor window. Click a tool and then drag it into the `UserForm` editor window to place an instance of the control. Right-click over one of the tools on the window and click Additional Controls to display the Additional Controls dialog box. Click any of the available controls to make it available for use in a `UserForm`. I explain how to add controls to a `UserForm` in Chapter 11.

Setting the VBA Environment Options

There are several settings that affect the behavior of the AutoCAD VBA environment and not just the currently loaded VBA projects. These settings can be changed in the Option dialog box of the VBA environment (see Figure 1.6), which can be displayed using one of the following methods:

- After the Macros dialog box has been opened with the `vbarun` command, click Options.
- At the Command prompt, type **`vbapref`** and press Enter.



☐ Enable auto embedding
☑ Allow Break on errors
☑ Enable macro virus protection

Figure 1.6 Changing the VBA environment settings

Here is an explanation of the settings in the Options dialog box:

Enable Auto Embedding The Enable Auto Embedding option creates a new empty VBA project each time a drawing file is opened and embeds that empty project into the drawing file. A new project is created and embedded only if the drawing opened doesn't already contain an embedded project. This option is disabled by default.

Allow Break On Errors The Allow Break On Errors option displays a message box that allows you to step into a procedure if an error is produced during execution. You can then use the debugging tools offered by the VBA Editor to locate and handle the error. I discuss debug procedures in Chapter 13. This option is enabled by default.

Enable Macro Virus Protection The Enable Macro Virus Protection option, when enabled, displays a message box during the loading of a DVB file. I recommend leaving this option enabled to ensure that a drawing file with an embedded VBA project isn't opened in the AutoCAD drawing environment. This reduces the risk of accidentally running malicious code. The option is enabled by default.

Managing VBA Programs

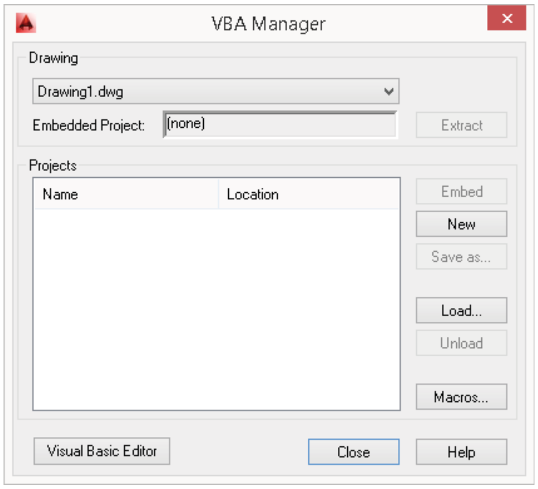
VBA programs developed in the AutoCAD VBA environment can be stored in a project file or embedded in a drawing file. VBA projects can also be embedded in a drawing template (DWT) or drawing standards (DWS) file. By default, VBA programs developed in the AutoCAD VBA environment are stored in a project file with a `.dvb` file extension and then are loaded into the AutoCAD drawing environment as needed.

DVB files can be managed externally from Windows Explorer or File Explorer, or from within AutoCAD whenever the file is loaded into the AutoCAD drawing environment. General file-management tasks on a DVB file can be performed using Windows Explorer or File Explorer.

Once the DVB file is loaded into the AutoCAD drawing environment, you can manage it using the VBA Manager (see Figure 1.7). The VBA Manager allows you to do the following:

- Create a new VBA project
- Save a VBA project to a DVB file
- Load a VBA project from a DVB file into the AutoCAD drawing environment
- Unload a VBA project from the AutoCAD drawing environment
- Edit the components and code stored in a VBA project
- Embed or extract a VBA project from a drawing file

FIGURE 1.7
Managing loaded VBA programs



There are two ways to display the VBA Manager in AutoCAD:

- On the ribbon, click the Manage tab ➢ Applications panel title bar and then click VBA Manager.
- At the Command prompt, type **vbaman** and press Enter.

Creating a New VBA Project

A new VBA project can be created automatically by the AutoCAD program or manually as needed. When the VBA environment is initialized the first time during an AutoCAD session, a new VBA project is created automatically unless a VBA project has already been loaded into memory. If you want to create a new project after the VBA environment has been initialized, do one of the following:

- When the VBA Manager is open, click New.
- At the Command prompt, type **vbanew** and press Enter.

Each new VBA project is assigned two default names: a project name and a location name. The project name is an internal name used by the AutoCAD program to differentiate the procedures and components in each loaded VBA project. The default project name for a new VBA project is `ACADProject`; I recommend assigning a descriptive project name for each VBA project you create. A project name can contain alphanumeric characters and underscores, but can't start with a number or underscore character.

The location name of a VBA project is the same as a filename and is used to specify where the `DVB` file is stored. Since a new VBA project exists only in memory, it is assigned the default location name of `Global1`. The location name is incremented by one for each new VBA project created during an AutoCAD session; thus the second and third VBA projects have location names of `Global2` and `Global3`, respectively. When you save VBA projects, they are stored in `DVB` files locally or on a network. To ensure that AutoCAD knows where the `DVB` files are located, you add the locations of your `DVB` files to the AutoCAD support file search and trusted paths. I discuss how to add a folder to the AutoCAD support file search and trusted paths in Chapter 13.

Saving a VBA Project

New VBA projects can be saved to disc using the Save As option in the VBA Manager or Save in the VBA Editor. When an existing project is loaded in memory, the Save As option can be used to create a copy of the project on disc or to overwrite an existing VBA project file. Typically, changes made to an existing project file that already has been loaded in the VBA environment are saved to the project file using the Save option in the VBA Editor. I discussed the VBA Editor earlier in the "Getting Started with the VBA Editor" section.

The following explains how to save a VBA project:

1. In the VBA Editor, click File ➢ Save. Alternatively, on the Standard toolbar click Save.
2. If the project hasn't been previously saved, the Save As dialog box is displayed. Otherwise, the changes to the VBA project are saved.
3. When the Save As dialog box opens, browse to the folder you wish to use to store the VBA project.
4. In the File Name text box, type a descriptive filename for the project and click Save.

NOTE A DVB file can be password-protected to restrict the editing of the components and code stored in the file. I discuss how to assign a password to a VBA project in Chapter 13.

Loading and Unloading a VBA Project

Before a VBA project can be edited and before the code stored in the project can be executed, the project must be loaded into the AutoCAD VBA environment. The process for loading a project into the AutoCAD VBA environment is similar to opening a drawing file.

Manually Loading a VBA Project

A VBA project can be manually loaded using the VBA Manager or the `vbaload` command. The following explains how to manually load a VBA project:

1. On the ribbon, click the Manage tab and then click the Applications panel title bar. Click Load Project. (As an alternative, at the Command prompt, type **`vbaload`** and press Enter.)
2. When the Open VBA Project dialog box opens, browse to and select `ch01_hexbolt.dvb`.
3. Clear the Open Visual Basic Editor check box and click Open.

4. If the File Loading - Security Concern dialog box is displayed, click Load to load the file into memory. (You can click Do Not Load to cancel a load operation.)
5. If the AutoCAD dialog box is displayed, click Enable Macro to allow the execution of the code in the project. (You can click Disable Macros to load a file but not allow the execution of the code, or Do Not Load to cancel without loading the project into memory.)

NOTE You can download the sample VBA project file `ch01_hexbolt.dvb` used in the following exercise from `www.sybex.com/go/autocadcustomization`.

Place the file in the `MyCustomFiles` folder within the `Documents` (or `My Documents`) folder or another location you are using to store custom program files.

As an alternative, DVB and other types of custom program files can be dragged and dropped onto an open drawing window in the AutoCAD drawing environment. When you drop a DVB file onto an open drawing window, AutoCAD prompts you to load the file and/or to enable the macros contained in the VBA project file.

Automatically Loading a VBA Project

The Load Project button in the VBA Manager and the `vbaload` command require input from the user to load a VBA project, which isn't ideal when you want to integrate your VBA projects as seamlessly as possible into the AutoCAD drawing environment. A script, custom AutoLISP program, or command macro from the AutoCAD user interface can all be used to load a VBA project without user input. The following outlines some of the methods that can be used to load a VBA project without user input:

- Call the `-vbaload` command. The `-vbaload` command is the command-line version of the `vbaload` command. When the `-vbaload` command is started, the `Open VBA Project:` prompt is displayed. Provide the name of the DVB file as part of the macro or script file.
- Call the AutoLISP `vl-vbaload` function. The AutoLISP `vl-vbaload` function can be used to load a DVB file from a custom AutoLISP program. If the DVB file that is passed to the `vl-vbaload` function isn't found, an error is returned that should be captured with the AutoLISP `vl-catch-all-apply` function.
- Create a VBA project file named `acad.dvb` and place it in one of the AutoCAD support file search paths. AutoCAD looks for a file named `acad.dvb` during startup and if the file is found, that file is loaded automatically.
- Use the Startup Suite (part of the Load/Unload Applications dialog box that opens with the `appload` command). When a DVB file is added to the Startup Suite, the file is loaded when the first drawing of a session is opened. Removing a file from the Startup Suite causes the file not to be loaded in any future drawings that are opened or in AutoCAD sessions. If you want to use the Startup Suite to load DVB files, you must add the files to the Startup Suite on each workstation and AutoCAD user profile.
- Create a plug-in bundle. Plug-in bundles allow you to load DVB and other custom program files in AutoCAD 2013 or later. A plug-in bundle is a folder structure with a special name and metadata file that describes the files contained in the bundle.

I discuss each of these methods in greater detail in Chapter 13.

Manually Unloading a VBA Project

When a VBA project is no longer needed, it can be unloaded from memory to release system resources. A VBA project can be manually unloaded from memory using the VBA Manager or the `vbaunload` command. The following explains how to unload the `ch01_hexbolt.dvb` file with the VBA Manager:

1. On the ribbon, click the Manage tab and then click the Applications panel title bar to expand the panel. Click VBA Manager. (If the ribbon isn't displayed or the release of the AutoCAD program you are using doesn't support the ribbon, at the Command prompt type **`vbaman`** and press Enter.)
2. When the VBA Manager dialog box opens, in the Projects list select HexBolt and click Unload.
3. If prompted to save changes to the VBA project, click Yes if you made changes that you wish to save or No to discard any changes.

Automatically Unloading a VBA Project

If you want to unload a DVB file as part of a script, custom AutoLISP program, or command macro from the AutoCAD user interface, you will need to use the `vbaunload` command. When the `vbaunload` command starts, the `Unload VBA Project:` prompt is displayed. Provide the filename and full path of the DVB file you want to unload; the path you specify must exactly match the path for the DVB file that was loaded into the AutoCAD drawing environment. If it doesn't, the unload fails and an error message will be displayed. A failed execution of the `vbaunload` command doesn't cause the program calling the command to fail.

TIP I recommend using the AutoLISP `findfile` function to locate the DVB file in the AutoCAD support file search paths when loading and unloading a DVB file to ensure that the correct path is provided.

Embedding or Extracting a VBA Project

A VBA project can be embedded in a drawing file to make the components and code in the project available when the drawing file is opened in the AutoCAD drawing environment. Only one VBA project can be embedded in a drawing file at a time. Embedding a VBA project in a file can be helpful to make specific tools available to anyone who opens the file, but there are potential problems using this approach. Here are the main two problems with embedding a VBA project file into a drawing:

- Embedding a VBA project triggers a security warning each time a drawing file is opened, which could impact sharing drawing files. Many companies will not accept drawings with embedded VBA projects because of potential problems with viruses and malicious code.
- Embedding a VBA project that is stored in a DVB file results in a copy of that project being created and stored in the drawing file. The embedded project and the original DVB file are kept separately. This can be a problem if the project is embedded in hundreds of drawing

files and needs to be revised. Each drawing file would need to be opened, the project extracted, and the revised project re-embedded.

So, while you can embed a VBA project, I don't recommend doing it.

Embedding a VBA Project

The following explains how to embed a VBA project in a current drawing file:

1. On the ribbon, click the Manage tab ➢ Applications panel title bar and then click VBA Manager.
2. When the VBA Manager opens, select a VBA project to embed from the Projects list. Load the VBA project you want to embed if it isn't already loaded.
3. Click Embed.

Extracting a VBA Project

Extracting a VBA project reverses the embedding process. After a project is selected for extraction, you can either export the project to a DVB file or discard the project. The following explains how to extract a VBA project from a current drawing file:

1. On the ribbon, click the Manage tab and then click Applications panel title bar to expand the panel. Click VBA Manager.
2. When the VBA Manager opens, in the Drawing section click Extract. (If the Extract button is disabled, there is no VBA project embedded in the current drawing.)
3. In the AutoCAD message box, click Yes to remove and export the VBA project to a DVB file. Specify a filename and location for the project you wish to extract. Click No if you wish to remove the VBA project from the drawing file without saving the project.

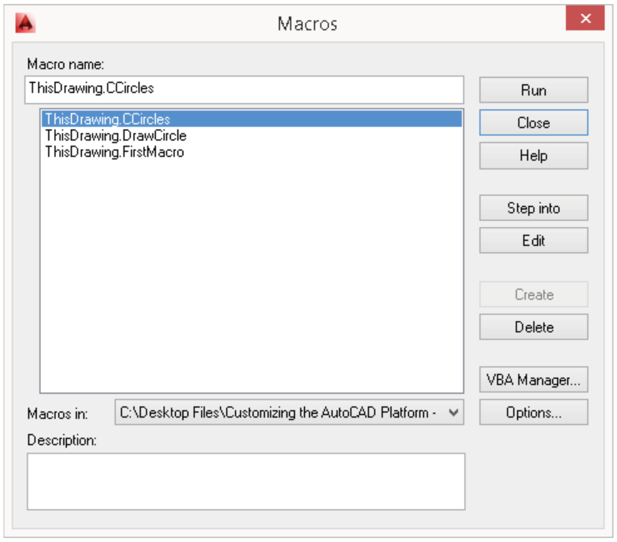
Executing VBA Macros

VBA projects contain components that organize code and define user forms and custom classes. A component can contain one or more procedures that are used to perform a task on the objects in a drawing or request input from an end user. Most procedures are defined so they are executed from other procedures in a VBA project and not from the AutoCAD user interface. A procedure that can be executed from the AutoCAD user interface is known as a *macro*. I explain how to define a procedure in Chapter 2.

A macro can be executed using the Macros dialog box (see Figure 1.8). In addition to executing a macro, the Macros dialog box can also be used to do the following:

- ◆ Execute and begin debugging a macro
- ◆ Open the VBA Editor and scroll to a macro's definition
- ◆ Create the definition of a new macro based on the name entered in the Macro Name text box

- Remove a macro from a loaded project
- Display the VBA Manager
- Change the VBA environment options



Macro name: ThisDrawing.CCircles

ThisDrawing.CCircles
ThisDrawing.DrawCircle
ThisDrawing.FirstMacro

Macros in: C:\Desktop Files\Customizing the AutoCAD Platform -

Description:

FIGURE 1.8
Executing a macro stored in a VBA project

The following methods can be used to display the Macros dialog box:

- On the ribbon, click the Manage tab ➢ Applications panel ➢ Run VBA Macro.
- At the Command prompt, type **`vbarun`** and press Enter.
- When the VBA Manager opens, click Macros.

The Macros dialog box requires input from the user to execute a macro in a loaded VBA project. If you want to execute a macro as part of a script, custom AutoLISP program, or command macro from the AutoCAD user interface you can use one of the following methods:

Command Line The `-vbarun` command is the command-line version of the `vbarun` command. When the `-vbarun` command is started, the `Macro name:` prompt is displayed.

AutoLISP The AutoLISP `vl-vbarun` function can be used to execute a macro in a loaded DVB file from a custom AutoLISP program. If the macro isn't found, an error message is displayed but the error doesn't cause the program to terminate.

The name of the macro to execute with the `-vbarun` command or `vl-vbarun` function must be in the following format:

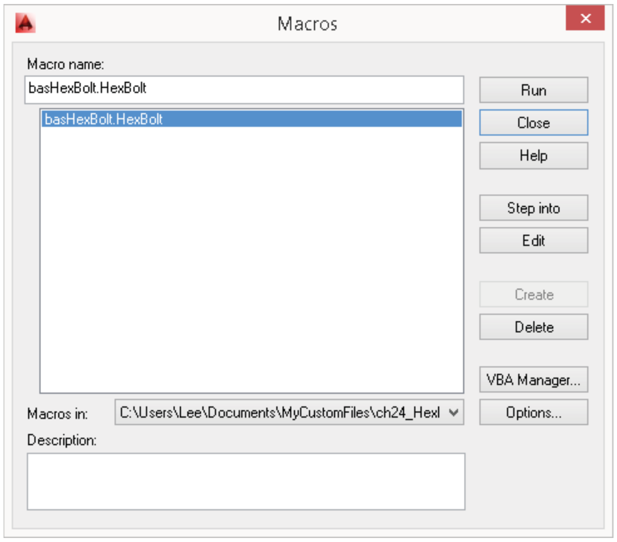
```
DVBFilename.ProjectName!MacroName
```

For example, you would use the string value `firstproject.dvb!ThisDrawing.CCircles` to execute the `CCircle` macro in the `ThisDrawing` component of the `firstproject.dvb` file.

These steps explain how to execute the macro named `hexbolt`:

1. On the ribbon, click the Manage tab ➢ Applications panel ➢ Run VBA Macro (or at the Command prompt, type **`vbarun`** and press Enter).
2. When the Macros dialog box opens, click the Macros In drop-down list and choose `ch01_hexbolt.dvb`.

   Figure 1.9 shows the macro that is stored in and can be executed from the `ch01_hexbolt.dvb` file with the Macros dialog box.



Macro name: basHexBolt.HexBolt

basHexBolt.HexBolt

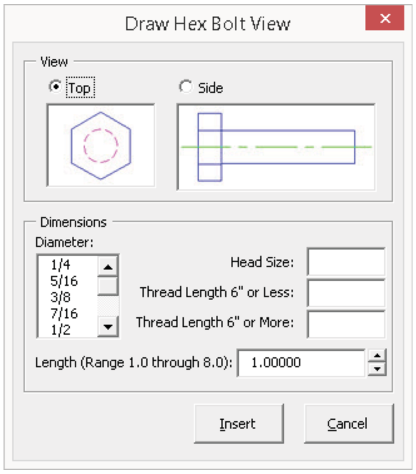
Macros in: C:\Users\Lee\Documents\MyCustomFiles\ch24_HexI

Description:

FIGURE 1.9 Edit, debug, and execute macros from the Macros dialog box.

3. In the Macros list, choose basHexBolt.HexBolt and click Run.

   The Draw Hex Bolt View dialog box, shown in Figure 1.10, is displayed.



View: [✓ Top] Side

Dimensions

Diameter: 1/4, 5/16, 3/8, 7/16, 1/2

Head Size:

Thread Length 6" or Less:

Thread Length 6" or More:

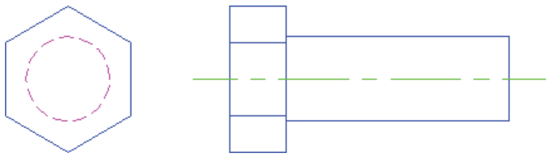
Length (Range 1.0 through 8.0): 1.00000

FIGURE 1.10 Custom dialog box used to draw a top or side view of a hex bolt

4. In the Diameter list box, choose 3/8 and click Insert.
5. At the `Specify center of bolt head:` prompt, specify a point in the drawing area to draw the top view of the hex bolt.
6. When the Draw Hex Bolt View dialog box reappears, in the View section click the Side option or image. Click Insert.
7. At the `Specify middle of bolt head:` prompt, specify a point in the drawing area to draw the side view of the hex bolt.
8. When the Draw Hex Bolt View dialog box reappears again, click Cancel.

   Figure 1.11 shows the top and side views of the hex bolt that were drawn with the macro.

FIGURE 1.11
Views of the completed hex bolt



Accessing the AutoCAD VBA Documentation

The AutoCAD VBA documentation is available from the AutoCAD product Help landing page and the VBA Editor. The documentation is composed of two documentation sets: the AutoCAD Object Library Reference and the ActiveX Developer's Guide. Although this book is designed to make it easy to learn how to use the AutoCAD Object library and the VBA programming language, you will want to refer to the documentation that is provided with the AutoCAD product too, as it just isn't possible to cover every function and technique here.

The topics of the AutoCAD Object Library Reference explain the classes, methods, properties, and constants that make up the AutoCAD Object library. The ActiveX Developer's Guide topics can be used to explore advanced techniques and features that aren't covered in this book.

You can see the AutoCAD VBA and ActiveX documentation written for AutoCAD 2015 here:

```
http://help.autodesk.com/view/ACD/2015/ENU/
```

On the Autodesk AutoCAD 2015 Help landing page, click the Developer Home Page link. On the AutoCAD Developer Help Home Page, use the AutoCAD Object Library Reference and Developer's Guide links under the ActiveX/VBA section to access the AutoCAD VBA and ActiveX documentation.

When working in the VBA Editor, you can access the AutoCAD Object Library Reference and Microsoft Visual Basic for Applications Help by doing the following:

1. In a code editor window, highlight the keyword, statement, data type, method, property, or constant that you want to learn more about.
2. Press F1.

Help can also be accessed from the Object Browser. In the Object Browser, select a class, method, property, or constant and then press F1 to open the associated help topic. I discussed the Object Browser earlier, in the "Exploring Loaded Libraries with the Object Browser" section.

Chapter 2

Understanding Visual Basic for Applications

The Visual Basic for Applications (VBA) programming language is a variant of the Visual Basic 6 (VB6) programming language that was introduced in 1998. Though similar, VB6 isn't exactly the same as the current version of Visual Basic (known as VB.NET). Unlike VB6, which allows you to develop stand-alone applications, VBA programs require a host application. The host application provides the framework in which the VBA program can be executed; Microsoft Word and the Autodesk® AutoCAD® program are examples of host applications.

VBA was first introduced as a preview technology and modern programming alternative to AutoLISP® and ObjectARX® with AutoCAD Release 14 back in 1997. It was not until after the release of AutoCAD R14.01 that VBA was officially supported. The implementation of VBA in the AutoCAD program at that time was huge to the industry, as the learning curve between AutoLISP and C++ was steep, and the number of developers who knew VBA was growing rapidly.

Here are some of the reasons I recommend using VBA for your custom programs:

- Individuals with VB/VBA experience often can be found in-house (check in your company's IS/IT department); finding someone fluent in AutoLISP or ObjectARX is much rarer.
- VB/VBA resources are easier to locate—on the Internet or at your local library.
- Connecting to external applications and data sources is simpler using VB/VBA.
- VBA programs are relatively low maintenance; programs written for the last release of the AutoCAD program (even those written a decade ago) often run in the latest release with little to no change.

Learning the Fundamentals of the VBA Language

Before you learn to use VBA to automate the AutoCAD drawing environment, it is essential to have a basic understanding of the VBA or VB6 programming language. If you are not familiar with the VBA or VB6 programming language, I recommend reading this chapter before moving on.

In addition to this chapter, the Microsoft Visual Basic for Applications Help from the Help menu on the VBA Editor's menu bar and your favorite Internet search engine can be great resources for information on the VBA programming language. The following are a couple of web resources that can help you get started on locating additional information on VBA and VB6:

- Microsoft's Programming Resources for Visual Basic for Applications page (`http://support.microsoft.com/kb/163435`)
- Microsoft Developer Network: Visual Basic 6.0 Language Reference (`http://msdn.microsoft.com/en-us/library/aa338033(v=vs.60).aspx`)

Creating a Procedure

Most of the code you write in VBA will be grouped into a named code block called a *procedure*. If you are familiar with AutoLISP or another programming language, you might be familiar with the terms *function* or *method*. VBA supports two types of procedures:

Subroutine (or Sub) A named code block that doesn't return a value

Function A named code block that does return a value

The definition of a procedure always starts with the keyword `Sub` or `Function` followed by its designated name. The procedure name should be descriptive and should give you a quick idea of the purpose of the procedure. The naming of a procedure is personal preference—I like to use title casing for the names of the functions I define to help identify misspelled function names. For example, I use the name `CreateLayer` for a function that creates a new layer. If I enter **`createlayer`** in the VBA Editor window, the VBA Editor will change the typed text to `CreateLayer` to match the procedure's definition.

After the procedure name is a balanced set of parentheses that contains the arguments that the procedure expects. Arguments aren't required for a procedure, but the parentheses must be present. The `End Sub` or `End Function` keywords (depending on the type of procedure defined) must be placed after the last code statement of the procedure to indicate where the procedure ends.

The following shows the basic structures of a `Sub` procedure:

```
Sub ProcedureName()

End Sub

Sub ProcedureName(Arg1 As DataType, ArgN As DataType)

End Sub
```

Here's an example of a custom procedure named `MyDraftingAids` that changes the values of two system variables—`osmode` to `35` and `orthomode` to `1`.

```
Sub MyDraftingAids()
  ThisDrawing.SetVariable "osmode", 35
  ThisDrawing.SetVariable "orthomode", 1
End Sub
```

When defining a procedure of the `Function` type, you must indicate the type of data that the procedure will return. In addition to indicating the type of data to return, at least one code statement in the procedure must return a value. You return a value by assigning the value to the procedure's name.

The following shows the basic structures of a `Function` procedure:

```
Function ProcedureName() As DataType

  ProcedureName = Value
End Function

Function ProcedureName(Arg1 As DataType, ArgN As DataType) As DataType

  ProcedureName = Value
End Function
```

The arguments and return values you specify as part of a procedure follow the structure of dimensioning a variable. I explain how to dimension a variable in the next section.

Arguments can be prefixed with one of three optional keywords: `Optional`, `ByRef`, or `ByVal`. The `Optional` keyword can be used to pass a value to a procedure that might be needed only occasionally. You can learn more about these keywords from the VBA Editor Help system. The following demonstrates the definition of a function named `CreateLayer` that accepts an optional color argument using the `Optional` keyword:

```
Function CreateLayer(lyrName As String, _
                     Optional lyrColor As ACAD_COLOR = acGreen) As AcadLayer
  Dim objLayer As AcadLayer

  Set objLayer = ThisDrawing.Layers.Add(lyrName)
  objLayer.color = lyrColor

  Set CreateLayer = objLayer
End Function
```

The value returned by that function is the new layer created by the `Add` method based on the name passed to the *`lyrName`* argument. The `Add` method returns an object of the `AcadLayer` type. After the layer is created, the color passed to *`lyrColor`* is assigned to the new layer's `Color` property. Finally, the new layer is returned by the assigning the value to `CreateLayer`. Since an object is being returned, the `Set` statement must be placed to the left of the variable name to assign the object to the variable. I discuss the `Set` statement in the "Working with Objects" section later in this chapter.

The following demonstrates how to use the `CreateLayer` procedure:

```
Dim newLayer as AcadLayer
Set newLayer = CreateLayer("Object", acWhite)
```

Another concept that can be used when defining an optional argument is setting a default value. A default value is assigned to an argument using the equal symbol (=). In the previous example, the default value of the *`lyrColor`* argument is assigned the value of `acGreen`, which is a constant in the AutoCAD COM library that represents the integer value of 3. The optional value is used if no color value is passed to the `CreateLayer` function.

NOTE The keywords `Public` and `Private` can be added in front of the `Sub` and `Function` keywords used to define a procedure. The `Public` keyword allows a procedure to be accessed across most code modules in a VBA project, whereas the `Private` keyword limits the procedure to be accessed only from the module in which it is defined. I explain these keywords further in the "Controlling the Scope of a Procedure or Variable" section later in this chapter.

Real World Scenario

YOU WANT ME TO MEASURE HUNGARY? REALLY?

No, I really don't. However, as you learn about VBA, you'll be exposed to some new (and seemingly strange) terms.

When instructions in a procedure want you to declare and define a variable in VBA, you'll be asked to dimension the variable. This will be accomplished using a `Dim` statement that looks something like this: `Dim objLayer As AcadLayer`.

When you begin working with user forms and variables, you'll be asked to add a Hungarian notation prefix, which helps you to identify `UserForm` objects and controls or the data type that variables are declared. Hungarian notation is a shorthand used by programmers to quickly provide identifying information. Here are a few common prefixes and their uses:

- `c` or `str`: string data
- `d`: double
- `i`: integer
- `o` or `obj`: object
- `btn`: button
- `cbo` or `cmb`: combo box
- `lbl`: label
- `txt`: text box

Just remember, you're learning a new language. I'll do my best to explain the new terms in plain English in context as I use them, although you might have to wait until later in the chapter or book to get all the details.

Declaring and Using Variables

Variables are used to allocate space in memory to store a value, which can then be retrieved later using the name assigned to the variable. You can declare variables to have a specific value or assign it a new value as a program is being executed. I explain the basics of working with variables in the following sections.

DECLARING A VARIABLE

VBA by default allows you to dynamically create a variable the first time it is used within a procedure, but I don't recommend using this approach. Although it can save you time, the VBA Editor isn't able to assist in catching issues related to incorrect data types in a code statement.

The proper approach to declaring a variable is to use the `Dim` keyword and follow the keyword with the name of the variable to dimension. The `Option` statement can be helpful in ensuring that all variables are declared before being used. I mention the `Option` statement in the "Forcing the Declaration of Variables" sidebar.

Unlike procedure names, the industry uses Hungarian notation as a standard for naming variables in VBA programs. For example, you would add `c` or `str` in front of a variable name to represent a `string` or `d` for a `double`. The variable name for a layer name might look like `cName` or `strLayerName`, whereas a variable name that holds a double number for the radius of a circle might be `dRadius`.

The following shows the minimal syntax used to declare a variable:

```
Dim VariableName
```

That syntax would declare a variable of the variant data type. The variant data type can hold a value of any type; though that might sound convenient, the VBA Editor isn't able to assist in catching issues related to the usage of an incorrect data type. It is good practice to use the `As` keyword and follow it with a specific type of data. The following shows the syntax used to declare a variable:

```
Dim VariableName As DataType
```

The following declares a variable named *`strName`* as a string and *`iRow`* as an integer:

```
Dim strName As String
Dim iRow As Integer
```

I discuss the general types of data that VBA supports in the "Exploring Data Types" section.

NOTE The `Dim` keyword is used when defining a variable as part of a procedure, but in the General Declaration of a VBA code module you must use the `Public`, `Global`, and `Private` keywords. The General Declaration is located at the very top of a code module before the first procedure definition. The `Public` and `Global` keywords allow a variable to be accessed across all code modules in a VBA project, whereas the `Private` keyword limits the access of a variable to the module where it is defined. I explain these keywords further in the "Controlling the Scope of a Procedure or Variable" section.

Assigning a Value to and Retrieving a Value from a Variable

After a variable has been declared, a value can be assigned to or retrieved from a variable using the = symbol. A value can be assigned to a variable by placing the name of the variable on the left side of the = symbol and placing the value to be assigned on the right. The value could be a static value or a value returned by a procedure.

For example, the following shows how to assign a string value of "Error: Bad string" to a variable named *`strMsg`* and the value of 5 to the variable named *`iRow`*.

```
strMsg = "Error: Bad string"
iRow = 5
```

The value of a variable can be retrieved by using it as one of the arguments that a procedure expects. The following demonstrates how to display the value of the *`strMsg`* variable in a message box with the `MsgBox` function:

```
MsgBox strMsg
```

The `MsgBox` function is part of the VBA programming language and is used to display a basic message box with a string and set of predefined buttons and icons. I cover providing feedback to the user in Chapter 5, "Interacting with the User and Controlling the Current View." You can also learn more about the `MsgBox` function from the Microsoft VBA Help system. In the VBA Editor, click Help ➢ Microsoft Visual Basic For Applications Help.

Declaring Constant Variables

There are special variables known as *constants* that can only be assigned a value in the editor window and cannot be changed when the VBA program is executed. A constant variable is declared using the `Const` statement; the `Const` statement is used instead of the `Dim` statement. After the data type is assigned to the variable, you then assign the value to the constant variable using the = symbol. I recommend adding a prefix of `c_` to the name of a constant variable. Adding the prefix can be a helpful reminder that the value of the variable can't be updated.

The following shows the syntax to declare a constant variable:

```
Const VariableName As DataType = Value
```

Here's an example of declaring a constant variable named `c_PI` of the double data type and then assigning it the value of `3.14159`:

```
Const c_PI as Double = 3.14159
```

Forcing the Declaration of Variables

The VBA environment supports a statement named `Option`. The `Option` statement is used to control several coding practices at the code module level. For example, entering the `Option` statement with the `Explicit` keyword in the General Declaration of a module forces you to declare all variables before they can be used. To force the declaration of variables, you type `Option Explicit` in the General Declaration; the keyword always follows the `Option` statement. The `Option` statement also supports the following keywords:

`Base`—Specifies if the lower limit of an array should be 0 or 1. By default, arrays start at index 0 in VBA. I discuss arrays in the "Storing Data in Arrays" section. Example statement: `Option Base 1`
`Compare`—Specifies the default string comparison method used within a code module. Valid values are Binary, Database, or Text. Example statement: `Option Compare Text` `Private`—All procedures that are declared with the `Public` keyword in a code module are available only within the current project and are not accessible when the project is referenced by other projects.

Controlling the Scope of a Procedure or Variable

Procedures and variables can be designated as being global in scope or local to a VBA project, component, or procedure. Global scope in VBA is referred to as *public*, whereas local scope is referred to as *private*. By default, a procedure that is defined with the `Sub` or `Function` statement is declared as public and is accessible from any module in the VBA project; in the case of a class module, the procedure can be used when an instance of the class is created.

You typically want to limit the procedures that are public because a public procedure can be executed by a user from the AutoCAD user interface with the `vbarun` or `-vbarun` command. The `Public` and `Private` keywords can be added in front of a `Sub` or `Function` statement to control the scope of the variable. Since all procedures have a public scope by default, the use of the `Public` keyword is optional. However, if you want to make a procedure only accessible from the module in which it is defined, use the `Private` keyword.

The following shows how to define a public Sub and private Function procedure:

```
Public Sub HelloWorld()
  CustomMsg "Hello World!"
End Sub

Private Function CustomMsg(strMsg As String) _
                            As VbMsgBoxResult
  CustomMsg = MsgBox(strMsg)
End Function
```

The CustomMsg function is executed from the Hello subroutine. Because the CustomMsg function is private, it cannot be executed from the AutoCAD user interface with the vbarun or -vbarun command.

All variables declared within a procedure are local to that procedure and can't be accessed from another procedure or component. If you want to define a public variable, the variable must be declared in a module's General Declarations at the very top of a module. When declaring a variable that can be accessed from any module in a project or just all procedures in a module, use the Public or Private keyword, respectively, instead of Dim.

A Dim statement in the General Declarations can be used to declare a public variable, though. The Public or Private keyword can also be placed in front of the Const statement to declare a public or private constant variable, which by default is declared as private and is accessible only from the module in which it is defined. The Public keyword can be used only in a code module, not in a class module or user form, when declaring a constant variable.

NOTE When you're defining a variable in the General Declaration, I recommend adding a prefix of g_ to help you identify that the variable is in the global scope of a code module or VBA project.

The following example shows how to declare a public variable that can hold the most recent generated error:

```
Public g_lastErr As ErrObject
```

The next example shows how to declare a private constant variable that holds a double value of 3.14159:

```
Private Const c_PI As Double = 3.14159
```

This last example shows how to declare a private variable that holds a layer object:

```
Private objLyr As AcadLayer
```

If you want to make a value accessible to multiple projects or between AutoCAD sessions, you can write values to a custom dictionary or the Windows Registry. I explain how to work with custom dictionaries and use the Windows Registry in Chapter 9, "Storing and Retrieving Custom Data."

Continuing Long Statements

A code statement is typically a single line in the editor window that can result in relatively long and harder-to-read code statements. The underscore character can be placed anywhere within a code statement to let the VBA environment know a code statement continues to the next line. A space must be placed in front of the underscore character as well—otherwise the VBA editor will display an error message.

The following shows a code statement presented on a single line:

```
Set objCircle = ThisDrawing.ModelSpace.AddCircle(dCenPt, 2)
```

The following shows several ways the underscore character can be used to continue the statement to the next line:

```
Set objCircle = _
    ThisDrawing.ModelSpace.AddCircle(dCenPt, 2)

Set objCircle = ThisDrawing.ModelSpace. _
                AddCircle(dCenPt, 2)

Set objCircle = ThisDrawing.ModelSpace.AddCircle(dCenPt, _
                                                 2)
```

Adding Comments

As a veteran programmer of more than 16 years, I can honestly say that I formed my fair share of bad habits early on. One of the habits that I had to correct was adding very few comments (or not adding any) to my code. Comments are nonexecutable statements that are stored as part of code in a VBA project. The concept of comments is not specific to VBA; it is part of most modern programming languages. The syntax used to indicate a comment does vary from programming language to programing language.

The following are common reasons why and when you might want to add comments to your code:

- To document when the program or component was created and who created it.
- To maintain a history of changes made to the program—what changes were made, when, and by whom.
- To indicate copyright or legal statements related to the code contained in a code module.
- To explain how to use a procedure, the values each argument might expect.
- To explain what several code statements might be doing; you might remember the task several code statements perform today, but it can become more of a challenge to remember what they are doing months or years later.
- To mask a code statement that you currently don't want to execute; during testing or while making changes to a program, you might want to temporarily not execute a code statement but keep the expression for historical purposes.

Comments in VBA are typically denoted with the use of an apostrophe (') or the `Rem` keyword added to the beginning of a code statement. When using the `Rem` keyword, the keyword must be followed by a space. Although a space isn't required after the use of the apostrophe character, I recommend adding one. Code statements and text to the right of the apostrophe or `Rem` keyword are not executed; this allows you to add comments on a line by themselves or even on the same line after a code statement.

The following example demonstrates the use of the comments in a code module. The comments are used to explain when the procedure was added and what the procedure does.

```
' Last updated: 7/13/14
' Updated by: Lee Ambrosius

' Revision History:
' HYP1 (7/13/14) - Added optional color argument

' Module Description:
' Shared utility code module that contains many
' procedures that are reusable across VBA projects.

' Creates a new layer and returns the AcadLayer object
' that was created.
' Revision(s): HYP1
Function CreateLayer(strLyrName As String, _
                     Optional nLyrColor As ACAD_COLOR = acGreen) _
                     As AcadLayer

  ' Create a variable to hold the new layer
  Dim objLayer As AcadLayer

  ' Create the new layer
  Set objLayer = ThisDrawing.Layers.Add(strLyrName)

  objLayer.color = nLyrColor ' Assign the color to the new layer

  'MsgBox "Layer created."

  ' Return the new layer
  Set CreateLayer = objLayer
End Function
```

Understanding the Differences Between VBA 32- and 64-Bit

The VBA programming language is supported on both Windows 32-bit and 64-bit systems, but there are a few differences that you will need to consider. The following outlines a few of these differences:

- The `LongLong` data type is supported on 64-bit systems to allow larger numbers compared to the `Long` data type. I recommend using the `LongPtr` data type when possible to allow your program to use either the `Long` or `LongLong` data type based on the system it is executing on.
- Not all third-party libraries and `UserForm` controls work on both 32-bit and 64-bit systems. Some third-party libraries and controls are only supported on 32-bit systems, so be sure to test your programs on both 32-bit and 64-bit systems if possible.
- Prior to the AutoCAD 2014 release, the AutoCAD COM library had separate procedures that were required when working on a 32-bit or 64-bit system.

Because of potential problems with library and control references, I recommend creating a 32-bit and 64-bit version of your VBA projects. Then when you make changes in one project, export and import the changed code modules and `UserForms` between projects. The examples and exercises shown in this book are designed to work on 32-bit and 64-bit systems.

Exploring Data Types

Programming languages use *data types* to help you identify the type of data:

- Required by a procedure's argument
- Returned by a procedure defined as a function

Table 2.1 lists the basic data types that VBA supports. The Data Type column lists the name of a data type and the Hungarian notation that is commonly added as a prefix to a variable that is declared with that data type. I mentioned the purpose of Hungarian notation earlier in this chapter. The Range column gives a basic understanding of the values a data type supports, and the Description column offers a brief introduction to the data type.

Table 2.1: VBA data types

Data Type (Hungarian Notation)	Range	Description
Byte (by)	0 to 255	Binary data or small integer
Boolean (b)	True or False	True or False value; used to condition code statements
Date (`dt`)	January 1, 100 to December 31, 9999	Date and time as a double value
Double (d)	1.80×10^{308} to -4.94×10^{-324} for negative numbers and 4.94×10^{-324} to 1.80×10^{308} for positive numbers	Large decimal number with an accuracy of up to 16 places
Integer (n)	-32,768 to 32,767	Numeric value without a decimal point
Long (`l`)	-2,147,483,648 to 2,147,483,647	Large numeric value without a decimal point
String (`c` or `str`)	0 to 65,400 for fixed-length strings, or 0 to approximately 2 billion for variable-length strings	One or more characters enclosed in quotation marks
Variant (v)	Same as the data type of the value assigned to the variable	Value of any data type

NOTE The double data type in VBA is referred to as a real or a float in other programming languages.

Objects and arrays are two other data types that are commonly found in a VBA program. I cover these two data types in the next sections.

You can use the TypeName and VarType functions to identify the type of data returned by a function or assigned to a variable. These two procedures are commonly used to determine how to handle the data assigned to a variable with conditionalized expressions, which I discuss in the "Conditionalizing and Branching Statements" section. The TypeName function returns a string value, and the VarType function returns an integer that represents the data type of a value.

The following shows the syntax of the TypeName and VarType functions:

```
retVal = TypeName(value)
retVal = VarType(value)
```

The *value* argument represents any valid procedure that returns a value or variable name. The string or integer value returned by the TypeName or VarType function is represented by the *retVal* variable. The variable name you use in your programs doesn't need to be named *retVal*.

NOTE Each integer value returned by the VarType function has a specific meaning. For example, a value of 2 represents an integer data type, whereas a value of 8 represents a string data type. You can learn about the meaning of each integer value that is returned by looking up the VbVarType constant in the Object Browser of the VBA Editor. I explained how to use the Object Browser in the Chapter 1, "Understanding the AutoCAD VBA Environment."

Here are examples of the TypeName and VarType functions:

```
' Displays a message box with the text String
MsgBox TypeName("Hello World!")

' Displays a message box with the text Double
MsgBox TypeName(1.0)

' Displays a message box with the text Integer
MsgBox TypeName(1)

' Displays a message box with the text 8
MsgBox VarType("Hello World!")

' Displays a message box with the text 5
MsgBox VarType(1.0)

' Displays a message box with the text 2
MsgBox VarType(1)
```

I explain more about the MsgBox procedure and other ways of providing feedback to the user in Chapter 5.

Working with Objects

An object represents an instance of a *class* from a referenced library, which might be a layer in a drawing or a control on a user form. A class is a template from which an object can be created, and it defines the behavior of an object. A new object can be created with

- A procedure, such as `Add` or `AddObject`. (The procedure you use depends on the object being created.)
- The `New` keyword when declaring a variable.

The following syntax creates a new object of the specific object data type with the `New` keyword:

```
Dim VariableName As New ObjectType
```

An object can't simply be assigned to a variable with the = symbol like a string or integer value can be. The `Set` statement must precede the name of the variable when you want to assign an object to a variable. The following shows the syntax of assigning an object to a variable:

```
Set VariableName = object
```

The following example shows how to create a new circle object in model space and assign the new circle to a variable named *`objCircle`*:

```
Dim dCenPt(0 To 2) As Double
dCenPt(0) = 0: dCenPt(1) = 0: dCenPt(2) = 0

Dim objCircle As AcadCircle
Set objCircle = ThisDrawing.ModelSpace.AddCircle(dCenPt, 2)
```

Once a reference to an object is obtained, you can query and modify the object using its properties and methods. Place a period after a variable name that contains a reference to an object to access one of the object's properties or methods. The following shows the syntax for accessing a property or method of an object:

```
VariableName.PropertyName
VariableName.MethodName
```

You can assign a new value to a property using the same approach as assigning a value to a variable. The following shows how to assign the string `Objects-Light` to the `Layer` property of a circle:

```
objCircle.Layer = "Objects-Light"
```

The current value assigned to a property can be retrieved by placing the object and its property name on the right side of the = symbol. The following shows how to retrieve the current value of the `Name` property of a circle object and assign it to a variable that was declared as a string:

```
Dim strLayerName as String
strLayerName = objCircle.Layer
```

When you create a new object with the `New` keyword, you should release the object from memory when it is no longer needed. The VBA environment will automatically free up system resources when it can, but it is best to assign the `Nothing` keyword to a variable that contains an object before the end of the procedure where the object was created. It is okay to assign `Nothing` to variables that contain an object reference; the value of the variable will be cleared but might not free up any system memory. The following shows how to free up the memory allocated for the creation of a new AutoCAD Color Model object:

```
Dim objColor As New AcadAcCmColor
Set objColor = Nothing
```

Using an Object Across Multiple Statements

The `With` statement can be used to work with a referenced object across multiple statements and can help to reduce the amount of code that needs to be written. The following shows the syntax of the `With` statement:

```
With variable
  statementsN
End With
```

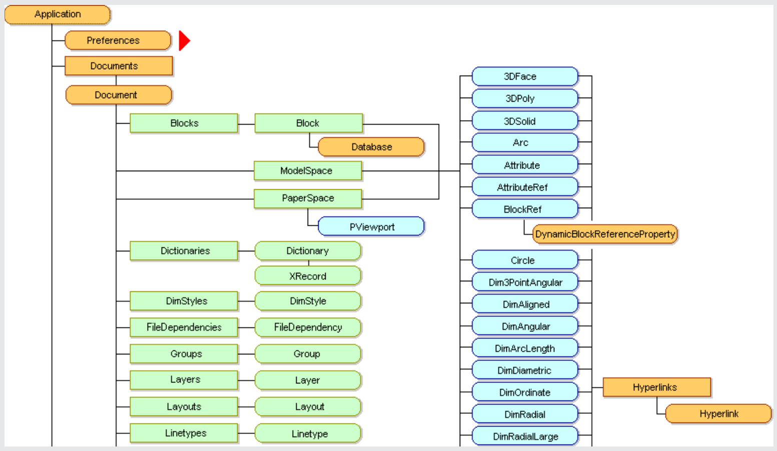
The *`variable`* argument represents the object that can be referenced throughout the `With` statement. You type a period between the `With` and `End With` statements to access the object's methods or properties. Here is an example of using the `ThisDrawing` object with the `With` statement to set the value of multiple system variables:

```
With ThisDrawing
  .SetVariable "BLIPMODE", 0
  .SetVariable "OSMODE", 32
  .SetVariable "ORTHOMODE", 1
End With
```

Exploring the AutoCAD Object Model

The AutoCAD Object library is designed to have a hierarchical structure, with the AutoCAD Application object at the top. From the AutoCAD Application object, you can access and open drawing files in the AutoCAD drawing environment. Once you have a reference to a drawing file, you can then access its settings, as well as the graphical and nongraphical objects stored in the drawing.

You can use the Object Browser to explore the classes and their members of the AutoCAD Object library, but it simply provides you with a flat listing of the available classes. The AutoCAD VBA documentation offers an object model map (shown in the following graphic) that allows you to graphically see the relationship between each object in the AutoCAD Object library. Clicking a node on the object model displays the object's topic in the Autodesk AutoCAD: ActiveX Reference Guide.



You can display the AutoCAD Object Model by following these steps:

1. Open your web browser and navigate to `http://help.autodesk.com/view/ACD/2015/ENU/`. If you are using AutoCAD 2015, display the AutoCAD product Help system.
2. On the Autodesk AutoCAD 2015 Help landing page, click the Developer Home Page link.
3. On the AutoCAD Developer Help home page, use the AutoCAD Object Model link under the ActiveX/VBA section.

If you open the Autodesk AutoCAD: ActiveX Reference Guide, scroll to the top of the Contents list and expand the Object Model node to access the Object Model topic.

Accessing Objects in a Collection

A collection is a container object that holds one or more objects of the same type. For example, the AutoCAD Object library contains collections named `Layers` and `DimStyles`. The `Layers` collection provides access to all the layers in a drawing, whereas `DimSyles` provides access to the dimension styles stored in a drawing. Since collections are objects, they have properties and methods as well, which are accessed using a period, as I explained in the previous section.

Objects in a collection have a unique index or key value assigned to them. Most collections start with an index of 0, but some start with an index of 1; you will need to refer to the documentation for the collection type to know the index of its first item. The following example shows how to set the first layer in the `Layers` collection and set it as the current layer:

```
ThisDrawing.ActiveLayer = ThisDrawing.Layers.Item(0)
```

The `Item` method returns the layer object at the index of 0 and the layer is then assigned to the `ActiveLayer` property. The `Item` method is the default method of most collections, so the previous code example could be written as follows:

```
ThisDrawing.ActiveLayer = ThisDrawing.Layers(0)
```

A key value is a string that is unique to the object in the collection. The `Item` method can accept a key value in addition to an integer that represents an index, as shown in the following examples:

```
ThisDrawing.ActiveLayer = ThisDrawing.Layers.Item("Objects-Light")
ThisDrawing.ActiveLayer = ThisDrawing.Layers("Objects-Light")
```

In addition to the `Item` method, the exclamation point (`!`) can be used to reference a key value in a collection. When using the `!` symbol, a key value that contains spaces must be enclosed in square brackets. The following shows how to access a key value in the Layers collection using the `!` symbol:

```
ThisDrawing.ActiveLayer = ThisDrawing.Layers!CenterLine
ThisDrawing.ActiveLayer = ThisDrawing.Layers![Center Line]
```

The `Item` method and examples I have shown in this section return a specific object from a collection. If you want to step through all the objects in a collection, you can use the `For` statement. I introduce the `For` statement in the “Repeating and Looping Expressions” section. To learn about using collections in the AutoCAD Object library, see the following chapters:

- Chapter 3, “Interacting with the Application and Documents Objects,” for working with the Documents collection
- Chapter 5, “Interacting with the User and Controlling the Current View,” for working with the Views collection
- Chapter 6, “Annotating Objects,” for working with the DimStyles and TextStyles collections
- Bonus Chapter 1, “Working with 2D Objects and Object Properties,” for working with the Layers and Linetypes collections in *AutoCAD Platform Customization: User Interface, AutoLISP, VBA, and Beyond*
- Bonus Chapter 2, “Modeling in 3D Space,” for working with the UCSs and Materials collections in *AutoCAD Platform Customization: User Interface, AutoLISP, VBA, and Beyond*

Storing Data in Arrays

An array is not really a data type but a data structure that can contain multiple values. Unlike the objects in a collection, the elements of an array can be of different data types and do not all need to be of the same data type. The first element in an array typically starts at an index of 0, but you can specify the index of the first element using a range. Arrays are used to represent a coordinate value in a drawing, to specify the objects used to define a closed boundary when creating a `Region` or `Hatch` object, or to specify the data types and values that make up the XData attached to an object.

The processes for declaring an array and variable are similar—with one slight difference. When declaring an array, you add opening and closing parentheses after the variable name. The value in the parentheses determines whether you declare a fixed-length or dynamic array.

NOTE The `Option Base 1` statement can be used to change the default index of `0` to `1` for the lower limit of an array. I explained the `Option` statement earlier in the “Forcing the Declaration of Variables” sidebar.

Declaring a Fixed-Length Array

A fixed-length array, as its name implies, is an array that can hold a specific number of values. When declaring a fixed-length array, you can specify a single- or multidimensional array. You define a single-dimensional array by specifying the number of rows, whereas you typically define a multidimensional array by specifying the number of rows and columns for the array. Rows and columns are based on the first row or column having an index of 0, the second one having an index of 1, and so on. The first row or column in an array is known as the *lower limit*, and the last row or column is known as the *upper limit* of the array.

Entering a single integer value within the parentheses when declaring an array specifies the *upper limit* of the rows in the array. Remember, the first row is 0, so specifying an upper limit of 1 declares an array of two rows. The following shows how to declare a fixed-length array with two rows, a single column, and a starting index of 0:

```
Dim names(1) As String
```

As an alternative, you can specify the lower and upper limit of an array. Enter the starting and ending index of the array separated by the `To` keyword. The following shows how to declare a fixed-length array with three rows, a single column, and a starting index of 1:

```
Dim centerPt(1 To 3) As Double
```

An array with a single dimension is the most common, but a multidimensional array can be used to create an in-memory data grid of values. You can specify the upper limit or range of the columns in the array. The following three examples show how to declare a fixed-length array that is four rows by four columns with a starting index of 0:

```
Dim matrix(3, 3) As Double
Dim matrix(0 To 3, 1 To 4) As Double
```

Declaring a Dynamic Array

A dynamic array is an array that isn't declared with a specific lower or upper limit, so it isn't initialized with any elements. Unlike a fixed-length array, the lower and upper limit of a dynamic array can be changed as needed. Typically, the upper limit of an array is increased to allow additional values to be added to the array. The `ReDim` statement, short for redimension, is used to decrease or increase the number of elements in an array by changing the lower and upper limits of the array. The following shows how to declare a dynamic array and then redimension the array to five elements:

```
Dim names() As String
ReDim names(4)
```

When you use the `ReDim` statement, all values that have been assigned to the array are lost. The current values of the elements remaining from the original array can be retained by using the `Preserve` keyword. The following shows how to increase an array to seven elements and retain any current values:

```
ReDim Preserve names(6)
```

The following shows how to decrease an array to four elements and retain any current values:

```
ReDim Preserve names(3)
```

In the previous example, any values in elements 4 through 6 are lost, but all other values would be retained. It is possible to dynamically resize an array by starting with the array's current lower and upper limits. I explain how to get the lower and upper limits of an array in the "Getting the Size of an Array" section.

Working with Array Elements

After an array has been declared and the number of elements established, you can assign a value to and retrieve a value from an element. Working with an element in an array is similar to working with a variable with the exception of needing to specify an element index.

The following shows how to declare a three-element array that represents a coordinate value of 0,5,0:

```
Dim dCenPt(0 To 2) As Double
dCenPt(0) = 0
dCenPt(1) = 5
dCenPt(2) = 0
```

You retrieve the value of an element by using it as an argument of a procedure or placing it to the right of the `=` symbol. The following shows how to get the current value of the element in the *`dCenPt`* with an index of 0 and display it in a message box with the `MsgBox` procedure:

```
MsgBox dCenPt(0)
```

If you want to step through all the elements in an array, you can use a `For` statement. I introduce the `For` statement in the "Repeating and Looping Expressions" section.

Getting the Size of an Array

When you want to resize an array or step through all the elements of an array, you need to know how many elements are in an array. The `LBound` and `UBound` procedures are used to return an integer that represents the lower and upper limits of an array, respectively.

The following shows the syntax of the `LBound` and `UBound` procedures:

```
LBound array [, dimension]
UBound array [, dimension]
```

Here are the arguments:

`array` The *`array`* argument represents the variable that contains the array of the lower or upper limit you want to return.

`dimension` The *`dimension`* argument is optional and represents the dimension in a multi-dimensional array that you want to query. When no argument value is provided, the upper limit of the first dimension is queried. The first dimension in an array has an index of 1 and not 0 like the elements of an array.

The following shows examples of the `LBound` and `UBound` procedures:

```
' Declares a single dimension array
Dim dCenPt(0 To 2) As Double

' Displays 0
Debug.Print LBound(dCenPt)
```

```
' Displays 2
Debug.Print UBound(dCenPt)

' Declares a multi-dimensional array
Dim matrix(0 To 3, 1 To 4) As Double

' Displays 3 which is the upper-limit of the first dimension
Debug.Print UBound(matrix, 1)

' Displays 4 which is the upper-limit of the second dimension
Debug.Print UBound(matrix, 2)
```

The output is displayed in the Output window of the VBA Editor with the `Print` procedure of the `Debug` object. You'll learn more about using the `Debug` object in Chapter 13, "Handling Errors and Deploying VBA Projects."

Calculating Values with Math Functions and Operators

When working with AutoCAD, you must consider the accuracy with which objects are placed and the precision with which objects are created in a drawing. The same is true with using VBA. You must consider both accuracy and precision when creating and modifying objects. The VBA math functions allow you to perform a variety of basic and complex calculations. You can add or multiply two numbers, or even calculate the sine or arctangent of an angle.

Table 2.2 lists many of the math functions and operators that you will use with VBA in this book.

Table 2.2: VBA math functions and operators

Function/ Operator	Description
+	Returns the sum of two numeric values. Syntax: *`retVal = number + number`*
-	Returns the difference between two numeric values. Syntax: *`retVal = number - number`*
*	Returns the product of two numeric values. Syntax: *`retVal = number * number`*
/	Returns the quotient after dividing two numeric values. A double value is returned. Syntax: *`retVal = number / number`*

Function/ Operator	Description
`\`	Returns the quotient after dividing two numeric values. A double value is returned. Syntax: `retVal = number \ number`
`Mod`	Returns the remainder after dividing two numeric values. Syntax: `retVal = number Mod number`
`Atn`	Calculates the arctangent of an angular value expressed in radians. Syntax: `retVal = Atn(number)`
`Cos`	Returns the cosine of an angular value expressed in radians. Syntax: `retVal = Cos(number)`
`Exp`	Returns a numeric value that has been raised to its natural antilogarithm. Syntax: `retVal = Exp(number)`
`Log`	Calculates the natural logarithm of a numeric value. Syntax: `retVal = Log(number)`
`Rnd`	Generates a random value of the single data type, which is similar to a double data value with less precision. Syntax: `retVal = Rnd([seed])` The optional `seed` argument is used to generate the same random number.
`Sin`	Returns the sine of an angular value expressed in radians. Syntax: `retVal = Sin(number)`
`Sqr`	Gets the square root of a numeric value. Syntax: `retVal = Sqr(number)`
`Tan`	Calculates the tangent of an angular value expressed in radians. Syntax: `retVal = Tan(number)`

For more information on these functions, see the Microsoft Visual Basic for Applications Help system.

Manipulating Strings

Strings are used for a variety of purposes in VBA, from displaying command prompts and messages to creating annotation objects in a drawing. String values in a VBA program can have a static or fixed value that never changes during execution, or a value that is more dynamic and is changed by the use of string manipulation functions.

Table 2.3 lists many of the string manipulation functions and operators that you will use with VBA in this book.

TABLE 2.3: VBA string manipulation functions and operators

FUNCTION/ OPERATOR	DESCRIPTION
`+`	Concatenates two strings together. Syntax: `retVal = string + string`
`&`	Concatenates two strings together. Syntax: `retVal = string & string`
`LCase`	Converts the characters in a string to all lowercase. Syntax: `retVal = UCase(string)`
`Left`	Returns a substring based on a specified number of characters from the left side of a string. Syntax: `retVal = Left(string, length)`
`Len`	Returns an integer that represents the number of characters in a string. Syntax: `retVal = Len(string)`
`LTrim`	Removes leading spaces from a string. Syntax: `retVal = LTrim(string)`
`Mid`	Returns a substring based on a starting position from the left side of a string and going to the right for a specified number of characters. Syntax: `retVal = Mid(string, start, length)` A starting position of 1 indicates the substring should start at the first character of the string.
`Right`	Returns a substring based on a specified number of characters from the right side of a string. Syntax: `retVal = Right(string, length)`
`RTrim`	Removes trailing spaces from a string. Syntax: `retVal = RTrim(string)`
`Space`	Returns a string containing the specified number of spaces. Syntax: `retVal = Space(number)`
`Split`	Returns an array of strings based on the delimited character. Syntax: `retVal = Split(string [, delimiter [, limit [, comparison]]])`

Function/ Operator	Description
`StrConv`	Returns a string based on a specified conversion option. Syntax: `retVal = StrConv(string, mode, localeID)` For a list of supported conversion modes and locale IDs, see the `StrConv` Function topic in the Microsoft Visual Basic for Applications Help system.
`String`	Returns a string containing a character repeated a specified number of times. Syntax: `retVal = String(number, character)`
`StrReverse`	Inverts the characters in a string and returns a new string. Syntax: `retVal = StrReverse(string)`
`Trim`	Removes leading and trailing spaces from a string. Syntax: `retVal = Trim(string)`
`UCase`	Converts the characters in a string to all uppercase. Syntax: `retVal = UCase(string)`

For more information on these functions, see the Microsoft Visual Basic for Applications Help system.

The + and & operators are used to concatenate two strings together into a single string. In addition to concatenating two strings, you can concatenate special constants that represent an ASCII value to a string. For example, you can add a tab or linefeed character. Table 2.4 lists the special constants that the VBA programming language supports.

Table 2.4: Special constants with ASCII values

Constant	Description
`vbBack`	Backspace character equal to `Chr(8)`
`vbCr`	Carriage return character equal to `Chr(13)`
`vbCrLf`	Carriage return and linefeed characters equal to `Chr(13) + Chr(10)`
`vbLf`	Linefeed character equal to `Chr(10)`
`vbTab`	Tab character equal to `Chr(9)`

The `Chr` function is used to return the character equivalent of the ASCII code value that is passed to the function. I introduce the `Chr` function and other data conversion functions in the next section.

The following code statements use the `vbLf` constant to break a string into two lines before displaying it in a message box with the `MsgBox` function:

```
' Displays information about the active linetype
MsgBox "Name: " & ThisDrawing.ActiveLinetype.Name & vbLf & _
       "Description: " & ThisDrawing.ActiveLinetype.Description
```

Converting Between Data Types

Variables in VBA hold a specific data type, which helps to enforce data integrity and communicate the type of data an argument expects or a function might return. As your programs become more complex and you start requesting input from the user, there will be times when a function returns a value of one data type and you want to use that value with a function that expects a different data type. VBA supports a wide range of functions that can convert a string to a number, a number to a string, and most common data types to a specific data type.

Table 2.5 lists many of the data conversion functions that you will use with VBA in this book.

TABLE 2.5: VBA data conversion functions

FUNCTION	DESCRIPTION
`Abs`	Returns the absolute value of a numeric value, integer, or double number. The absolute value is a positive value—never negative. Syntax: `retVal = Abs(number)`
`Asc`	Returns an integer that represents the ASCII code value of the string character that is passed to the function. Syntax: `retVal = Asc(string)`
`CBool`	Converts a value to a Boolean value. Syntax: `retVal = CBool(value)`
`CByte`	Converts a value to a byte value. Syntax: `retVal = CByte(value)`
`CCur`	Converts a value to a currency value. Syntax: `retVal = CCur(value)`
`CDate`	Converts a value to a date value. Syntax: `retVal = CDate(value)`
`CDbl`	Converts a value to a double value. Syntax: `retVal = CDbl(value)`
`CDec`	Converts a value to a decimal value. Syntax: `retVal = CDec(value)`

Function	**Description**
`Chr`	Returns the character equivalent of the ASCII code value that is passed to the function. Syntax: `retVal = Chr(number)`
`CInt`	Converts a value to an integer value. Syntax: `retVal = CInt(value)`
`CLng`	Converts a value to a long value. Syntax: `retVal = CLng(value)`
`CLngLng`	Converts a value to a `LongLong` value that is valid on 64-bit systems only. Syntax: `retVal = CLngLng(value)`
`CLngPtr`	Converts a value to a long value on 32-bit systems or a `LongLong` value on 64-bit systems. Syntax: `retVal = CLngPtr(value)`
`CSng`	Converts a value to a single value. Syntax: `retVal = CSng(value)`
`CStr`	Converts a value to a string value. Syntax: `retVal = CStr(value)`
`CVar`	Converts a value to a variant value. Syntax: `retVal = CVar(value)`
`Fix`	Returns the nearest integer of a double number after discarding the fractional value after the decimal. When a negative double value is passed to the function, the first negative number greater than or equal to the number passed is returned. Syntax: `retVal = Fix(number)`
`Format`	Returns a string that contains a formatted numeric or date value. Syntax: `retVal = Format(value[, format[,firstweekday [, firstweekofyear]]])` The optional *`format`* argument controls the number or date formatting, and the optional *`firstweekday`* and *`firstweekofyear`* specify the first day of the week or first week of the year.
`Hex`	Returns a hexadecimal value as a string based on the number provided. Syntax: `retVal = Hex(number)`
`Int`	Returns the nearest integer of a double number after discarding the fractional value after the decimal. When a negative double value is passed to the function, the first negative number less than or equal to the number passed is returned. Syntax: `retVal = Int(number)`
`Oct`	Returns an octal value as a string based on the number provided. Syntax: `retVal = Oct(number)`

For more information on these functions, see the Microsoft Visual Basic for Applications Help system.

Comparing Values

As the complexity of a program grows, so too does the need to perform conditional tests, also referred to as test conditions. Test conditions are used to compare values or settings in the AutoCAD environment against a known condition. VBA operators and functions that are used to test conditions return a Boolean value of `True` or `False`. The VBA operators and functions used to test a condition allow you to

- Compare two values for equality
- Determine if a value is numeric, zero, or negative
- Compare two values to see which is greater or less than or equal to the other
- Check for a value being `Nothing`, an array, or an object

Testing Values for Equality

Testing for equality is probably the most common test condition you will perform in most of your programs. For example, you might want to see if the user provided any input with one of the `GetXXXX` functions that are part of the AutoCAD COM library. In this case, you could check to see if the value returned is expected. The VBA `=` (equal to) and `<>` (not equal to) operators are how values are commonly compared to each other. The `=` operator returns `True` if the values are equal; otherwise, `False` is returned. The `<>` operator returns `True` if the values are not equal; `False` is returned if the values are equal.

The following shows the syntax of the `=` and `<>` operators:

```
value1 = value2
value1 <> value2
```

Here are examples of the `=` and `<>` operators:

```
' Returns True, numbers are equal
1 = 1
1 = 1.0

' Returns True, strings are equal
"ABC" = "ABC"

' Returns False, numbers are not equal
1 <> 2

' Returns False, strings are not equal
"ABC" = "abc"
```

In addition to the `=` operator, the `Like` operator can be used to compare string values. I discuss the `Like` operator in the next section.

TIP The `Not` operator can be used to invert a Boolean value returned by an operator or function. A value of `True` is returned as `False`, whereas a value of `False` is returned as `True`.

The = operator isn't ideal for comparing to see if two objects are the same. If you want to compare two objects for equality, you use the Is operator. The syntax for using the Is operator is the same as for using the = operator. A value of True is returned if both objects are the same when using the Is operator; otherwise, False is returned.

Here are examples of the Is operator:

```
' Gets the current layer of the drawing
Dim objCurLayer as AcadLayer
Set objCurLayer = ThisDrawing.ActiveLayer

' Creates a new layer
Dim objNewLayer as AcadLayer
Set objNewLayer = ThisDrawing.Layers.Add("ABC")

' Returns True since both objects are the same
objCurLayer Is ThisDrawing.ActiveLayer

' Returns False since both objects are not the same
objCurLayer Is objNewLayer
```

Comparing String Values

The = operator isn't the only way to compare two string values. The Like operator allows you to compare a string value to a string pattern that can contain one or more wildcard characters. If the string matches the pattern, True is returned, and False is returned if the string doesn't match the pattern.

The following shows the syntax of the Like operator:

```
retVal = string Like pattern
```

Here are examples of the Like operator:

```
' Returns True since both strings match
"ABC" Like "ABC"

' Returns False since both strings don't match
"ABC" Like "AC"

' Returns True since both strings match the pattern
"DOOR_DEMO" Like "DOOR*"
```

The StrComp and InStr functions can be used to compare two string values using an optional comparison mode. The StrComp and InStr functions don't return a Boolean value like the = operator; instead they return an integer value based on the comparison mode passed to the function. 0 is returned if the strings are equal, 1 is returned if the binary value of the first string is greater than the second string or the two strings are not equal when doing a textual comparison, and -1 is returned if the binary value of the first string is less than the second string.

The following shows the syntax of the StrComp function:

```
retVal = StrComp(string1, string2[, comparison])
```

For more information on the *StrComp* function and a list of values that the *comparison* argument expects, see the Microsoft Visual Basic for Applications Help.

The InStr function is similar to the StrComp function with one exception: it has an optional *start* argument, which specifies the location within the first string that the comparison should start. The following shows the syntax of the InStr function:

```
retVal = InStr([start, ][string1, ][string2, ][comparison])
```

Determining If a Value Is Greater or Less Than Another

The values that a user provides or the settings that define the AutoCAD environment aren't always easily comparable for equality. Values such as the radius of a circle or the length of a line are often compared to see if a value is greater or less than another. The VBA > (greater than) and < (less than) operators can be used to ensure that a value is or isn't greater than or less than another value.

These two operators are great for making sure a value is within a specific range, more than a value, or less than a value. You can also use the > and < operators with the Do and While statements to count down or up and make sure that while incrementing or decrementing a value you don't exceed a specific value. You might also use the > and < operators with a logical grouping operator to make sure a value is within a specific range of values. I discuss logical groupings in the "Grouping Comparisons" section.

The > (greater than) operator returns True if the first number is greater than the second number; otherwise, False is returned. The < (less than) operator returns True if the first number is less than the second number; otherwise, False is returned. If the values being compared are equal, then False is returned.

The following shows the syntax of the > and < operators:

```
value1 > value2
value1 < value2
```

In addition to comparing values to see if a value is greater or less than another, you can check for equality at the same time. The >= (greater than or equal to) and <= (less than or equal to) operators allow you to check to see if a value is greater or less than another or if the two values are equal. The syntax and return values for the >= and <= operators are the same as for the > and < operators, except True is returned if the values being compared are equal to each other.

Here are examples of comparing values with the >, <, >=, and <= operators, along with the values that are returned:

```
' Returns True as 2 is greater than 1
2 > 1

' Returns False as the values are equal
1 > 1.0

' Returns False as 2 is not less than 1
2 < 1

' Returns False as the values are equal
```

```
1 < 1.0

' Returns True as the values are equal
1 >= 1.0

' Returns False as 1 is not greater than or equal to 2
1 >= 2

' Returns True as the values are equal
1 <= 1.0

' Returns True as 1 is less than or equal to 2
1 <= 2
```

TIP You can compare a value within a range of values by using logical groupings, which I cover in the "Grouping Comparisons" section.

Checking for Null, Empty, or Nothing Values

Values assigned to a variable or returned by a statement can be checked to see whether they evaluate to null, empty, or nothing. A null value occurs when no valid data is assigned to a variable. The `IsNull` function returns `True` if a value is null; otherwise, `False` is returned. A variable can be set to a value of null using this syntax:

```
variable = Null
```

A variable declared with the variant data type can hold any type of data, but if it is not initialized and assigned a value, it is empty. The `IsEmpty` function returns `True` if a value is empty; otherwise, `False` is returned. A variable can be set to a value of empty using this syntax:

```
variable = Empty
```

Values that are of an object type can't be compared for a null or empty value, but rather you compare them against a value of nothing. Unlike checking for a null or empty value, there is no `IsNothing` function that can be used to check for a value of nothing. Checking for a `Nothing` value requires the use of the `Is` operator, which I mentioned in the "Testing Values for Equality" section. The following syntax shows how to compare an object for a value of nothing:

```
' Creates new variable of the AcadLayer object type
Dim objCurLayer as AcadLayer

' Evaluates to True since no object has been assigned to the variable
objCurLayer Is Nothing

' Gets the current layer of the drawing
Set objCurLayer = ThisDrawing.ActiveLayer

' Evaluates to False since the current layer has been assigned to the variable
Debug.Print objCurLayer Is Nothing
```

A variable can be set to a value of nothing using the syntax:

```
Set variable = Nothing
```

Validating Values

Prior to using a variable, I recommend testing to see if the variable holds the type of value that you might reasonably expect. Although it does increase the complexity of a program, the additional statements used to test variables are worth the effort; they help to protect your programs from unexpected values. The following lists some of the functions that can be used to test the values of a variable:

`IsArray`: Determines if a value represents a valid array; returns `True` or `False`.

`IsDate`: Determines if a value represents a valid calendar date or time; returns `True` or `False`.

`IsMissing`: Checks to see if an optional argument of a procedure was provided; returns `True` or `False`.

`IsNumeric`: Determines if a value is a number; returns `True` or `False`.

`IsObject`: Determines if a value is an object; returns `True` or `False`.

`Sgn`: Determines the sign of a numeric value; 1 is returned if the value is greater than zero, 0 is returned if equal to zero, or –1 is returned if the number is less than zero.

For more information on these functions, see the Microsoft Visual Basic for Applications Help system.

Grouping Comparisons

There are many times when one test condition isn't enough to verify a value. One of the best examples of when you want to use more than one test condition is to see if a value is within a specific numeric range. Logical grouping operators are used to determine if the results of one or more test conditions evaluates to `True`.

The `And` and `Or` operators are the two logical grouping operators that can be used to evaluate two or more test conditions. The `And` operator returns `True` if all test conditions in a grouping return `True`; otherwise, `False` is returned. The `Or` operator returns `True` if at least one test condition in a grouping returns `True`; otherwise it returns `False`.

The following shows the syntax of the `And` and `Or` operators:

```
test_condition1 And test_condition2
test_condition1 Or test_condition2
```

The *`test_condition1`* and *`test_condition2`* arguments represent the test conditions that you want to group together and evaluate.

Here are examples of the `And` and `Or` operators, along with the values that are returned:

```
' Checks to see if a number is between 1 and 5
Dim num as Integer
```

```
' Evaluates to and displays True since num is 3 and between 1 and 5
num = 3
MsgBox 5 >= num And 1 <= num

' Evaluates to and displays False since num is 6 and is not between 1 and 5
num = 6
MsgBox 5 >= num And 1 <= num

' Checks to see if values are numeric or strings
Dim val1, val2
val1 = 1.5: val2 = "1.5"

' Evaluates to and displays True since va11 is a double or integer
MsgBox VarType(val1) = vbDouble Or VarType(val1) = vbInteger

' Evaluates to and displays False since va12 is not a double or integer
MsgBox VarType(val2) = vbDouble Or VarType(val2) = vbInteger
```

I discussed the `VarType` function in the "Exploring Data Types" section.

Conditionalizing and Branching Statements

The statements in a procedure are executed sequentially, in what is commonly known as a linear program. In a linear program, execution starts with the first statement and continues until the last statement is executed. Although statements are executed in a linear order, a procedure can contain branches. Think of a branch as a fork in the road.

Branches allow a procedure to make a choice as to which statements should be executed next based on the results of a test condition. I covered test conditions in the "Comparing Values" section. The `If` and `Select Case` statements are used to branch the statements in a procedure.

Evaluating If a Condition Is Met

The operators and functions discussed in the previous sections allow a program to compare and test values to determine which expressions to execute by using a programming technique called *branching*. The most common branching method is the `If…Then` statement. Using the `If…Then` statement, a set of statements can be executed if the test condition is evaluated as `True`.

The following shows the syntax of the `If…Then` statement:

```
If test_condition Then
  true_statementsN
End If
```

Here are the arguments:

`test_condition` The *`test_condition`* argument represents the test condition that you want to evaluate and determine which statements to execute.

`then_statementN` The *`then_statementN`* argument represents the statements to evaluate if the *`test_condition`* argument evaluates to `True`.

The If…Then statement supports an optional Else statement, which can be used to execute a set of statements when the test condition is evaluated as False. The following shows the syntax of the If…Then statement with the Else statement:

```
If test_condition Then
  true_statementsN
Else
  else_statementN
End If
```

The *else_statementN* argument represents the statements that should be executed if the *test_condition* argument evaluates to False. In addition to the Else statement, the If…Then statement can support one or more optional ElseIf statements. An ElseIf statement allows for the evaluation of additional test conditions. The following shows the syntax of the If…Then statement with the inclusion of the ElseIf and Else statements:

```
If test_condition Then
  true_statementsN
[ElseIf test_condition Then
  elseif_statementN]
[Else
  else_statementN]
End If
```

When the *test_condition* argument of the If…Then statement evaluates to a value of False, the *test_condition* of the ElseIf statement is evaluated. If the *test_condition* of the ElseIf statement evaluates to a value of True, the set of statements after it is executed. If the *test_condition* of the ElseIf statement evaluates to a value of False, the next ElseIf statement is evaluated if one is present. If no other ElseIf statements are present, the Else statement is executed if one is present.

The following is an example of an If…Then statement that uses the ElseIf and Else statements to compare the value of a number entered:

```
' Prompts the user for a number
Dim num As Integer
num = CInt(InputBox("Enter a number: "))

' Checks to see if the number is greater than, less than, or equal to 4
If num > 4 Then
  MsgBox "Number is greater than 4"
ElseIf num < 4 Then
  MsgBox "Number is less than 4"
Else
  MsgBox "Number is equal to 4"
End If
```

Validating for an Object of a Specific Type

You can use the `TypeOf` *`object`* `Is` *`objecttype`* clause of the `If` statement to determine an object's type. This can be helpful if your program expects the user to select or work with a specific type of object. Selection filters, discussed in Chapter 5, can be used to allow only the user to select an object of a specific type.

The following example displays one of two messages based on whether the first object in model space is a circle:

```
' Gets the first object in model space
Dim oFirstEnt As AcadEntity
Set oFirstEnt = ThisDrawing.ModelSpace(0)

' Display a message based on if the
' first object is a circle or not
If TypeOf oFirstEnt Is AcadCircle Then
  MsgBox "Object is a circle."
Else
  MsgBox "The object isn't a circle."
End If
```

Testing Multiple Conditions

The `If`…`Then` statement allows a procedure to execute one or more possible sets of statements based on the use of the `ElseIf` and `Else` statements. In addition to the `If`…`Then` statement, the `Select Case` statement can be used to evaluate multiple test conditions. The `Select Case` statement is a more efficient approach to testing multiple conditions when compared to the `If`…`Then` statement.

Each test condition of a `Select Case` statement starts with the `Case` statement and can be used to compare more than one value. Similar to the `If`…`Then` statement, the `Select Case` statement also supports an optional statement if none of the test conditions are valued as `True`; the optional statement is named `Case Else`.

The following shows the syntax of the `Select Case` statement:

```
Select Case
  Case test_condition
    case_statementsN
  [Case test_condition
    case_statementsN]
  [Case Else
    else_statementN
  ]
End Select
```

test_condition The *test_condition* argument represents the test condition that you want to evaluate and determine which statements to execute.

case_statementsN The *case_statementsN* argument represents the statements to evaluate if the *test_condition* argument evaluates to True.

else_statementsN The *else_statementsN* argument represents the expressions to evaluate if none of the test conditions represented by the *Case* statements evaluates to True. The Case Else statement must also be used.

The following is an example of the Select Case statement:

```
' Displays a message based on the number entered
Select Case CInt(InputBox("Enter a number: "))
  Case 1
    MsgBox "1 was entered"
  Case 2 To 4
    MsgBox "2 to 4 was entered"
  Case 5, 6
    MsgBox "5 or 6 was entered"
  Case Is >= 7
    MsgBox "7 or greater was entered"
  Case Else
    MsgBox "0 or less was entered"
End Select
```

Repeating and Looping Expressions

Repetition helps to form habits and learn how to perform a task, but repetition can also be counterproductive. If you know a task is repeated many times a day and you know how to complete that task, it is ideal to automate and simplify the process as much as possible, if not eliminate the process altogether. VBA—and most programming languages, for that matter—have no problem with repetition because they support a concept known as *loops*. Loops allow for a set of expressions to be executed either a finite number of times or infinitely while a condition is met.

Repeating Expressions a Set Number of Times

The easiest way to loop a set of expressions in VBA is to use the For statement. The first argument of the For statement is known as the *counter*, which is a variable name that is incremented or decremented each time the For statement is executed. The initial value of the counter and number of times the For statement should be executed are determined by a range of two values.

Typically, the range starts with 0 or 1 and the difference between the start and ending of the range is used to specify how many times the For statement is executed. By default, the counter is incremented by 1 each time the For statement is executed. Optionally, the For statement supports the Step keyword, which can be used to specify a larger increment value than the default of 1 or a decrement value to count down instead of up.

The following shows the syntax of the For statement:

```
For counter = start To end [Step stepper]
  statementN
Next [counter]
```

Its arguments are as follows:

counter The *counter* argument represents the variable name that is assigned to the current loop counter. The variable should be of a number data type, such as an integer or short. When the `For` statement is executed the first time, the counter variable is assigned the value passed to the *start* argument.

start The *start* argument represents the start of the numeric range.

end The *end* argument represents the end of the numeric range.

stepper The *stepper* argument is optional and represents the numeric value that *counter* should be stepped each time the `For` statement is executed. Use a positive number to increment *counter* or a negative number to decrement *counter*.

statementN The *statementN* argument represents the statements that should be executed each time the loop is started.

NOTE The `Exit For` statement can be used to end a `For` statement before the counter reaches the end of the specified range.

The following is an example of using the `For` statement:

```
' Executes the statements 5 times, the variable
' cnt is incremented by 1 with each loop
Dim cnt as Integer

For cnt = 1 To 5
  Debug.Print cnt
Next cnt
```

Here is the output that the previous statements create:

```
1
2
3
4
5
```

Stepping Through an Array or Collection

The `For Each` statement is similar to the `For` statement described in the previous section. Instead of specifying a counter variable, a range, and an optional step, the `For Each` statement requires an element variable and a grouping, such as an array or a collection object. When the `For Each` statement is executed, the first value of the array or object of the collection is assigned to the element variable. As the `For Each` statement continues to be executed, the next value or object is assigned to the variable until all values or objects have been retrieved.

The following shows the syntax of the `For Each` statement:

```
For Each element In grouping
  statementN
Next [element]
```

Its arguments are as follows:

element The *element* argument represents the variable name that is assigned to the current loop element. When the For Each statement is executed the first time, the element variable is assigned the first value or object of the *grouping* argument.

grouping The *grouping* argument represents the array or collection object that you want to step through one value or object at a time.

statementN The *statementN* argument represents the statements that should be executed each time the loop is started.

NOTE The Exit For statement can be used to end a For statement before the last value or object in an array or a collection is retrieved.

The following is an example of using the For Each statement:

```
' Steps through all layer objects in the Layers collection
' of the current drawing and displays the names of each layer
Dim objLayer as AcadLayer

For Each objLayer In ThisDrawing.Layers
  Debug.Print objLayer.Name
Next objLayer
```

Here is the output that the previous statements create:

```
0
Plan_Walls
Plan_Doors
Plan_Cabinets
Plan_Furniture
Labels
Panels
Surfaces
Storage
Defpoints
Dimensions
```

The order in which values or objects are retrieved is the same in which they were added to the array or collection.

Performing a Task While or Until a Condition Is Met

The For and For Each statements, as I mentioned in the previous sections, can be used to execute a set of statements a finite number of times. However, it isn't always easy to know just how many times a set of statements might need to be executed to get the desired results. When you are unsure of the number of times a set of statements might need to be executed, you can use the Do or While statement.

The Do and While statements use a test condition, just like the If statement, to determine whether the set of statements should be executed. The set of statements are executed as long as the test condition returns True. The test conditions that can be used are the same ones mentioned earlier in the "Comparing Values" and "Grouping Comparisons" sections.

There are two uses for the Do statement. The first is to evaluate a test condition before it executes any statements, whereas the other executes a set of statements and then evaluates a test condition to determine whether the statements should be executed again. Which version you use simply depends on whether you want to execute the statements at least once each time the Do statement is executed.

A Do statement also requires the use of either the While or Until keyword. The While keyword indicates that the Do statement should execute until the test condition is no longer True, and the Until keyword indicates that the Do loop should execute while the test is False.

The following shows the syntax of the Do statement that evaluates a test condition to determine whether the set of statements should be executed:

```
Do [{While | Until} test_condition]
  statementN
Loop
```

The next example shows the syntax of the Do statement that executes a set of statements before evaluating a test condition:

```
Do
  statementN
Loop [{While | Until} test_condition]
```

Its arguments are as follows:

test_condition The *test_condition* argument represents the statement that should be used to determine if the expressions represented by the *statementN* argument should be executed or continue to be executed.

statementN The *statementN* argument represents the statements that should be executed each time the loop is started.

The following are examples of the Do function:

```
' Executes the statements 5 times, the variable
' cnt is decremented by 1 with each loop
Dim cnt As Integer
cnt = 5

Do While cnt > 0
  Debug.Print cnt

  cnt = cnt - 1
Loop
```

Here is the output that the previous statements create:

```
5
4
3
2
1
```

```
' Executes the statements once since the test condition
' only returns True while cnt is greater than 4
Dim cnt As Integer
cnt = 5

Do
  Debug.Print cnt

  cnt = cnt + 1
Loop Until cnt > 4
```

Here is the output that the previous statements create:

```
5
```

NOTE The `Exit Do` statement can be used to end a `Do` statement before the test condition returns `True` or `False` based on whether the `While` or `Until` keyword is used.

The `While` statement is similar to the `Do` statement with the `While` keyword when evaluating a test condition before it executes a set of statements. The one difference between the `Do` and `While` statements is that the `While` statement doesn't support the ability to end early with the use of the `Exit` statement. Ending a `While` statement early would require statements to manipulate the test condition being used to determine when to end the looping.

The following shows the syntax of the `While` statement:

```
While test_condition
  statementN
Wend
```

The *`test_condition`* and *`statement`* arguments are the same as those in the `Do` statement. Here is an example of the `While` function:

```
' Executes the statements 5 times, the variable
' cnt is decremented by 1 with each loop
Dim cnt As Integer
cnt = 5

While cnt > 0
  Debug.Print cnt

  cnt = cnt - 1
Wend
```

Here is the output that the previous statements create:

```
5
4
3
2
1
```

Chapter 3

Interacting with the Application and Documents Objects

The top object in the AutoCAD® Object library is the `AcadApplication` object, which allows you to access and manipulate the AutoCAD application window. From the `AcadApplication` object, you can also access the `AcadDocuments` collection, which allows you to work with not only the current drawing but all open drawings in the current AutoCAD session. As mentioned in earlier chapters, the `ThisDrawing` object can be used to access the current drawing.

Working with the Application

The `AcadApplication` object is the topmost object in the AutoCAD Object library. Although it isn't the most used object, it does provide access to the many features that you will use in VBA projects. All objects in the AutoCAD Object library provide access to the `AcadApplication` object with the object's `Application` property. You can access the `AcadApplication` object from the `ThisDrawing` object with the following code statement in the VBA Editor:

```
ThisDrawing.Application
```

You can also use the following code statement to access the `AcadApplication` object from a code, class, or `UserForm module`:

```
AcadApplication.Application
```

The following tasks can be performed once you have a reference to the `AcadApplication` object:

- Get the current drawing or the `AcadDocuments` collection object to work with all open drawings (see the section "Managing Documents" later in this chapter for more information).
- List, load, and unload ObjectARX® applications.
- Load and unload VBA project files and execute a macro (see Chapter 13, "Handling Errors and Deploying VBA Projects").
- Manipulate the menus on the menu bar and toolbars in the user interface (see Chapter 10, "Modifying the Application and Working with Events").
- Monitor changes to the application, system variables, commands, and more using event handlers (see Chapter 13).
- Update the display in the drawing window or zoom in the current viewport (see Chapter 5, "Interacting with the User and Controlling the Current View").

- Access application preferences (see the section "Querying and Setting Application and Document Preferences" later in this chapter for more information).
- Get the name of and path to an application executable.
- Manipulate the size and position of the application window.

The following shows a few code statements that allow you to query or manipulate an application:

```
' Gets and displays the caption of the application window
MsgBox ThisDrawing.Application.Caption

' Zooms to the extents of all drawing objects in the current viewport
AcadApplication.Application.ZoomExtents

' Maximizes the application window
ThisDrawing.Application.WindowState = acMax
```

For a full list of the methods and properties that the `AcadApplication` object supports, look up the `AcadApplication` class in the Object Browser of the VBA Editor and the AutoCAD Help system.

Getting Information about the Current AutoCAD Session

The properties of the `AcadApplication` object can be used to access information about the current instance of the application. You can learn the application name and where the executable is stored, as well as which drawing is current or which drawings are open.

Table 3.1 lists the properties of the `AcadApplication` object that can be used to get information about the AutoCAD executable.

Table 3.1: Application-related properties

Property	Description
`FullName`	Returns a string that contains the full name of the executable file used to start the application. This property is read-only.
`HWND`	Returns a long integer that contains the handle of the application in memory. A handle is a unique value assigned to an application by Windows while it is executing in memory. A different number is assigned to the application each time it is started. This property is read-only.
`HWND32`	Returns a long integer that contains the handle of the application in memory on a Windows 64-bit platform. This property is read-only and is available in AutoCAD 2014 and earlier releases that didn't support a true implementation of VBA 64-bit on the Windows 64-bit platform.
`LocaleId`	Returns an integer that represents the locale or language being used in the current session. This property is read-only.

Property	**Description**
`Name`	Returns a string that contains the name and file extension of the executable file used to start the application. This property is read-only.
`Path`	Returns a string that contains the path of the executable file used to start the application. This property is read-only.
`Version`	Returns a string that contains the version number of the application. This property is read-only.

The following demonstrates how to display a message box containing the name, path, and version number of the application:

```
Sub DisplayAppInfo()

  MsgBox "Name: " & ThisDrawing.Application.Name & vbLf & _
         "Path: " & ThisDrawing.Application.Path & vbLf & _
         "FullName : " & ThisDrawing.Application.FullName & vbLf & _
         "Version : " & ThisDrawing.Application.Version, _
         vbInformation, "Application Info"
End Sub
```

TIP The `FullName` and `Path` properties can be helpful in identifying whether the current AutoCAD session was started from a plain or from a vertical AutoCAD installation. For example, the installation path might be `C:\Program Files\Autodesk\AutoCAD 2015\ACA`, which lets you know that instance of AutoCAD 2015 should have access to the AutoCAD® Architecture features. You can also use the `product` system variable to check whether the current AutoCAD instance is a vertical-based product. I discuss working with system variables in the "Working with System Variables" section later in this chapter.

Manipulating the Placement of the Application Window

Some properties of the `AcadApplication` object can be used to resize, reposition, or even hide the AutoCAD application window from the user.

Table 3.2 lists the `AcadApplication` object properties that can be used to resize and get information about the application window.

Table 3.2: Application window–related properties

Property	**Description**
`Caption`	Returns a string that contains the title of the application window. This property is read-only.
`Height`	Specifies the height of the application window. The value returned is an integer and represents the window height in pixels.

Table 3.2: Application window–related properties *(continued)*

Property	Description
`Visible`	Specifies the visibility of the application window. The value returned is Boolean. `True` indicates that the application window is visible, whereas `False` indicates the window is hidden.
`Width`	Specifies the width of the application window. The value is an integer and represents the window width in pixels.
`WindowLeft`	Specifies the location of the application window's left edge. The value is an integer. 0 sets the window to the leftmost visible position. A negative value moves the window to the left and off the screen, whereas a value greater than 0 moves the window to the right.
`WindowState`	Returns an integer value that represents the current state of the application window. The integer values allowed are defined as part of the `AcWindowState` enumerator. A value of 1 (or `acNorm`) indicates the window is neither minimized nor maximized, whereas a value of 2 (or `acMin`) or 3 (or `acMax`) indicates the window is minimized or maximized, respectively.
`WindowTop`	Specifies the location of the application window's top edge. The value is an integer. 0 sets the window to the topmost visible position. A negative value moves the window up and off the screen, whereas a value greater than 0 moves the window down.

For more information on these properties, use the Object Browser in the VBA Editor or check the AutoCAD Help system.

Managing Documents

When a drawing file is opened in the AutoCAD drawing environment, it is presented in a drawing window. A drawing window in the AutoCAD Object library is referred to as a *document* and is represented by an `AcadDocument` object. The `AcadDocument` object provides access to the objects in a drawing file and the window in which the drawing is displayed.

The `AcadDocuments` collection object is used to manage all the drawings open in the current AutoCAD session. You can access the `AcadDocuments` collection object with the `Documents` property of the `AcadApplication` object. In addition to working with drawings in the current session, you can create new and open existing drawing files, save and close open drawings, and get information from an open drawing.

NOTE As I explained in Chapter 2, "Understanding Visual Basic for Applications," the `For` statement can be used to step through and get each drawing in the `AcadDocuments` collection object. The `Item` method can also be used to get a specific document in the `AcadDocuments` collection object.

Working with the Current Drawing

The `ThisDrawing` object is the most common way to access the current drawing from a VBA project. `ThisDrawing` is equivalent to using the code statement `AcadApplication .ActiveDocument`.

From the current drawing, you can perform the following tasks:

- Add, query, and modify graphical and nongraphical objects (see Chapter 4, "Creating and Modifying Drawing Objects," Chapter 6, "Annotating Objects," and Chapter 7, "Working with Blocks and External References").
- Set a nongraphical object as current (see Chapter 4, Chapter 6, and Chapter 7).
- Use utility functions to get user input and perform geometric calculations (see Chapter 5).
- Monitor changes to the drawing, commands, objects, and more using event handlers (see Chapter 13).
- Select objects using selection sets (see Chapter 4).
- Get the name and path to the drawing file stored on disc.
- Access and modify drawing properties.

The following shows a few example code statements that access the properties of a current drawing:

```
' Sets the model space elevation to 10.0
ThisDrawing.ElevationModelSpace = 10#

' Displays a message box with the name of the current layer
MsgBox ThisDrawing.ActiveLayer.Name

' Maximizes the drawing window
ThisDrawing.WindowState = acMax
```

For a full list of the methods and properties that the `ThisDrawing` object supports, look up the `AcadDocument` class in the Object Browser of the VBA Editor or check the AutoCAD Help system.

Creating and Opening Drawings

Not only can you work with the current drawing, but you can create a new drawing or open an existing drawing file that had been stored on disc. The `Add` method of the `AcadDocuments` collection object can be used to create a new drawing from scratch or based on a drawing template (DWT) file. If you don't pass the name of a drawing template file to the `Add` method, the measurement units of the new drawing is determined by the current value of the `measureinit` system variable. A value of 0 for the `measureinit` system variable indicates the new drawing will use imperial units, whereas a value of 1 indicates the use of metric units. The `Add` method returns an `AcadDocument` object that represents the new drawing file that has been created in memory.

The following example code statements show how to create a new drawing that uses metric units from scratch or based on the `Tutorial-iArch.dwt` file that is installed with AutoCAD:

```
' Set the measurement system for new drawings to metric
ThisDrawing.SetVariable "measureinit", 1

' Create a new drawing from scratch
Dim newDWG1 As AcadDocument
Set newDWG1 = Application.Documents.Add

' Create a new drawing based on Tutorial-iArch.dwt
Dim newDWG2 As AcadDocument
Set newDWG2 = Application.Documents.Add("Tutorial-iArch.dwt")
```

NOTE If the DWT file passed to the `Add` method isn't located in a path listed under the Drawing Template File Location node on the Files tab of the Options dialog box, you must specify the full path to the DWT file. The `TemplateDwgPath` property of the `AcadPreferencesFiles` object can be used to add additional paths for AutoCAD to look in for DWT files. I discuss application preferences later, in the "Querying and Setting Application and Document Preferences" section.

The `New` method of an `AcadDocument` object can also be used to create a new drawing file when the AutoCAD drawing environment is in single document interface (SDI) mode. Autodesk doesn't recommend using SDI mode; it affects the functionality of some features in the AutoCAD drawing environment. You can determine whether AutoCAD is in SDI mode by checking the value of the `sdimode` system variable or checking the `SingleDocumentMode` property of the `AcadPreferencesSystem` object. The `New` method returns an `AcadDocument` object that represents the new drawing file that has been created in memory.

When you want to work with an existing drawing file that is stored on a local or network drive, use the `Open` method of the `AcadDocument` or `AcadDocuments` collection object. Here's the syntax of the `Open` methods:

```
retVal = document.Open(fullname [, password])
retVal = documents.Open(fullname [, read-only] [, password])
```

The arguments are as follows:

fullname The *fullname* argument is a string that represents the DWG file you want to open. You can also open a DWS or DWT file.

password The *password* argument is an optional string that represents the password that is required to open a password-protected DWG file.

read-only The *read-only* argument is an optional Boolean that specifies whether the drawing should be open for read-write or read-only. A value of `True` indicates the drawing should be open for read-only access.

retVal The *retVal* argument specifies a user-defined variable that you want to assign the AcadDocument object that is returned by the Open method.

The following example code statements show how to open a DWG file named Building_Plan.dwg stored at C:\Drawings, first for read-write and then for read-only access:

```
' Open Building_Plan.dwg for read-write
Dim objDoc1 As AcadDocument
set objDoc1 = ThisDrawing.Open("c:\drawings\building_plan.dwg")

' Open Building_Plan.dwg for read-only
Dim objDoc2 As AcadDocument
set objDoc2 = Application.Documents.Open("c:\drawings\building_plan.dwg", True)
```

NOTE Before you try to use a DWT file or open a DWG file, you should make sure the file exists on your workstation. The VBA Dir method can be used to check for the existence of a file or folder. I explain how to work with files in Windows in Chapter 12, "Communicating with Other Applications."

Saving and Closing Drawings

After you create a new drawing or make changes to an existing drawing file, you most likely will want to save the drawing to a file. Saving a drawing can be accomplished with the Save or SaveAs method of the AcadDocument object. Similar to the user interface, the Save method should be used to save a drawing file that was opened from disc or was previously saved with the SaveAs method.

The Save method accepts no arguments and saves a drawing to the location it was opened from. If you use the Save method on a new drawing that wasn't previously saved, the Save method saves the drawing to the location stored in the Path property of the AcadDocument object. You can determine whether the drawing was previously saved by checking the FullName property of the AcadDocument object; if the property returns an empty string, the drawing hasn't been saved to disc yet.

When you want to save a new drawing, save an existing drawing with a new name or location or in a different file format, or change the password protection, save the drawing with the SaveAs method. Here's the syntax of the SaveAs method:

```
document.SaveAs(fullname [, SaveAsType] [,SecurityParams])
```

The arguments are as follows:

fullname The *fullname* argument is a string that represents the name and path of the drawing or drawing interchange file on disc.

SaveAsType The *SaveAsType* argument is an optional integer that represents one of the supported file formats. The supported values can be found in the Object Browser of the VBA Editor under the enumerator named AcSaveAsType or in the SaveAs Method topic in the AutoCAD Help system. When not provided, the default format (the native drawing file format for the AutoCAD release you are using) is used. For AutoCAD 2013 and later, the default file format is the AutoCAD 2013 DWG file format.

SecurityParams The *SecurityParams* argument is an optional AcadSecurityParams object that specifies the password or digital signature settings to apply to the drawing. For information on the AcadSecurityParams object, see the AutoCAD Help system.

Before saving a drawing, you should check to see if the file was opened as read-only or if the drawing already has been saved. The ReadOnly property returns a Boolean value of True when the drawing is opened for read-only access, and the Saved property returns a Boolean value of True if the drawing doesn't need to be saved.

The following example demonstrates how to save a DWG file named SampleVBASave.dwg to the Documents (or My Documents) folder:

```
' Check to see if the drawing is read-write
If ThisDrawing.ReadOnly = False Then
  ' Check to see if the drawing file was previously saved
  If ThisDrawing.FullName = "" Then
    ' Drawing wasn't previously saved
    ThisDrawing.SaveAs ThisDrawing.GetVariable("MyDocumentsPrefix") & _
                       "\SampleVBASave.dwg"
  Else
    ' Drawing was previously saved to disc

    ' Check to see if the drawing has been modifed
    If ThisDrawing.Saved = False Then
      ThisDrawing.Save
    End If
  End If
End If
```

Once a drawing file no longer needs to remain open in the AutoCAD drawing environment, you can close it using the Close method of the AcadDocument object. Alternatively, you can use the Close method of the AcadDocuments collection object, which will close all open drawings and ignore any changes that haven't previously been saved.

Here's the syntax of the Close methods:

```
document.Close([SaveChanges] [, fullname])
documents.Close
```

Here are the arguments:

SaveChanges The *SaveChanges* argument is an optional Boolean that specifies whether the changes made to the drawing should be saved or discarded.

fullname The *fullname* argument is an optional string that represents a new name and path to use when saving the drawing file if SaveChanges was passed a value of True.

TIP If you want to close all open drawings, I recommend using the For statement with the AcadDocuments collection object and then close each drawing one at a time with the Close method of the AcadDocument object returned by the For statement. This approach will give you a chance to specify how changes are handled for each drawing as it is closed.

The following example demonstrates a procedure that mimics some of the functionality available with the AutoCAD `closeall` command:

```
Sub CloseAll()
  Dim oDoc As AcadDocument

  For Each oDoc In Application.Documents
    ' Activates the document window
    oDoc.Activate

    ' Close the drawing if no changes have been made since last save
    If oDoc.Saved = True Then
      oDoc.Close False
    Else
      Dim nRetVal As Integer
      nRetVal = MsgBox("Save changes to " & _
                 oDoc.Path & "\" & oDoc.Name & "?", vbYesNoCancel)
      Select Case nRetVal
        Case vbYes
          ' Save the drawing using its default name or last saved name
          ' if not open as read-only.
          If oDoc.ReadOnly = False Then
            oDoc.Save

            ' Close the drawing
            oDoc.Close
          Else
            ' Close file and discard changes if file is read-only
            If vbYes = MsgBox("File is read-only." & vbLf & vbLf & _
                         "Discard changes and close file?", vbYesNo) Then
              oDoc.Close False
            End If
          End If

          ' You should prompt the user here if the file was not previously
          ' saved to disc for a file name and path, or how read-only files
          ' should be handled.
        Case vbNo
          ' Close file and discard changes
          oDoc.Close False
        Case vbCancel
          ' Exit the procedure and return to AutoCAD
          Exit Sub
      End Select
    End If
  Next oDoc
End Sub
```

NOTE The previous example doesn't handle all situations that might be encountered when closing and saving changes to a drawing. The `Ch03_ExSamples.dvb` file, which you can download from `www.sybex.com/go/autocadcustomization`, contains a more comprehensive and complete solution. Place the file in the `MyCustomFiles` folder within the `Documents` (or `My Documents`) folder, or the location you are using to store the DVB files. Then load the VBA project into the AutoCAD drawing environment to use it. This sample file also contains a custom class that wraps two functions that can be used to display an open or save file-navigation dialog box.

Accessing Information about a Drawing

The properties of an `AcadDocument` object can be used to access information about the drawing file it represents. You can learn where the drawing file is stored, identify the graphical and nongraphical objects stored in a drawing file, and access the drawing properties that are used to identify a drawing file. I discuss how to access graphical and nongraphical objects later in this book.

Table 3.3 lists the properties of the `AcadDocument` object that can be used to get the name, location, and drawing properties of a drawing file open in the current AutoCAD session.

Table 3.3: Drawing file–related properties

Property	Description
`FullName`	Returns a string that contains the full name of the drawing file when it is stored on disc. If the drawing has not been saved yet, this property returns an empty string. This property is read-only.
`Name`	Returns a string that contains the name and file extension of the drawing file. If the drawing has not been saved yet, it returns the default name assigned to the drawing file (that is, `Drawing1.dwg`, `Drawing2.dwg`, …). This property is read-only.
`Path`	Returns a string that contains the path of the drawing file when it is stored on disc or the `Documents` (or `My Documents`) folder by default if the drawing has not been saved. This property is read-only.
`SummaryInfo`	Returns a reference to an `AcadSummaryInfo` object, which represents the drawing properties that can be displayed and modified using the AutoCAD `dwgprops` command. This property is read-only.

NOTE Use the `SaveAs` method of the `AcadDocument` object to save a drawing file with a new name or location.

The following demonstrates how to display a message box containing the path and name of the current drawing:

```
Sub DisplayDWGName()
  MsgBox "Name: " & ThisDrawing.Name & vbLf & _
         "Path: " & ThisDrawing.Path & vbLf & _
         "FullName : " & ThisDrawing.FullName, _
         vbInformation, "File Name and Path"
End Sub
```

To query and set the `Author` and `Comments` properties of the `AcadSummaryInfo` object for the current drawing, you'd use this code:

```
Sub DWGSumInfo()
  Dim oSumInfo As AcadSummaryInfo
  Set oSumInfo = ThisDrawing.SummaryInfo

  MsgBox "Author: " & oSumInfo.Author & vbLf & _
         "Comments: " & oSumInfo.Comments, _
         vbInformation, "Drawing Properties"

  oSumInfo.Author = "Drafter"
  oSumInfo.Comments = "Phase 1: Demolishion of first floor"

  MsgBox "Author: " & oSumInfo.Author & vbLf & _
         "Comments: " & oSumInfo.Comments, _
         vbInformation, "Drawing Properties"
End Sub
```

For more information on the `AcadSummaryInfo` object, use the Object Browser in the VBA Editor or check the AutoCAD Help system.

NOTE The `Ch03_ExSamples.dvb` sample file, which you can download from `www.sybex.com/go/autocadcustomization`, contains two procedures—named `AssignSumInfo` and `QuerySumInfo`—that demonstrate a more comprehensive solution for working with drawing properties. Place the file in the `MyCustomFiles` folder within the `Documents` (or `My Documents`) folder, or the location you are using to store the DVB files. Then load the VBA project into the AutoCAD drawing environment to use it.

Manipulating a Drawing Window

In addition to getting information about a drawing file, you can query and manipulate the window in which a drawing file is displayed. The `Active` property and the `Activate` method are helpful when you are working with the `AcadDocuments` collection object. You can use the `Active` property to determine whether a document is the current object, and the `Activate` method lets you set a document as current.

Table 3.4 lists the `AcadDocument` object properties that can be used to resize and get information about a drawing window.

Table 3.4: Drawing window–related properties

Property	Description
`Height`	Specifies the height of the drawing window. The value is an integer and represents the window height in pixels.
`Width`	Specifies the width of the drawing window. The value is an integer and represents the window width in pixels.
`WindowState`	Returns an integer value that represents the current state of the drawing window. The integer values allowed are defined as part of the `AcWindowState` enumerator. A value of 1 (or `acNorm`) indicates the window is neither minimized nor maximized, whereas a value of 2 (or `acMin`) or 3 (or `acMax`) indicates the window is minimized or maximized, respectively.
`WindowTitle`	Returns a string that contains the title of the drawing window. This property is read-only.

For more information on these properties and methods, use the Object Browser in the VBA Editor or check the AutoCAD Help system.

Working with System Variables

System variables are used to alter the way commands work, describe the current state of a drawing or AutoCAD environment, and specify where support files are stored. Many of the settings that are exposed by system variables are associated with controls in dialog boxes and palettes; other settings are associated with various command options. For example, many of the settings in the Drafting Settings dialog box (which you display using the `dsettings` command) are accessible from system variables.

A system variable can store any one of the basic data types that VBA supports (see "Exploring Data Types" in Chapter 2). You can see the hundreds of system variables and the type they hold by using the AutoCAD Help system. Whereas you might normally use the `setvar` command to list or change the value of a system variable at the AutoCAD Command prompt, with the AutoCAD Object library you use the `GetVariable` and `SetVariable` methods of an `AcadDocument` object to query and set the value of a system variable, respectively.

Here's the syntax of the `SetVariable` and `GetVariable` methods:

```
document.SetVariable sysvar_name, value
retVal = document.GetVariable(sysvar_name)
```

The arguments are as follows:

`sysvar_name` The *`sysvar_name`* argument specifies the name of the system variable you want to query or set.

`value` The *`value`* argument specifies the data that you want to assign to the system variable.

retVal The *retVal* argument specifies the user-defined variable that you want to assign the current value of the system variable.

The next exercise demonstrates how to query and set the value of the osmode system variable, which controls the object snap drafting aid that is currently running. This setting is available in the Drafting Settings dialog box:

1. Create a new VBA project or use the empty VBA project that is available by default when the VBA Editor is started.
2. In the Project Explorer, double-click the ThisDrawing component.
3. In the code editor window, type the following:

```
Sub WorkingWithSysVars()
  ' Get and store the current value of osmode
  Dim nCurOsmode As Integer
  nCurOsmode = ThisDrawing.GetVariable("osmode")
  MsgBox "Current value of osmode: " & CStr(nCurOsmode)

  ' Set osmode to a value of 33
  ThisDrawing.SetVariable "osmode", 33
  MsgBox "Current value of osmode: " & _
         CStr(ThisDrawing.GetVariable("osmode"))

  ' Restore osmode to its previous value
  ThisDrawing.SetVariable "osmode", nCurOsmode
  MsgBox "Current value of osmode: " & _
         CStr(ThisDrawing.GetVariable("osmode"))
End Sub
```

4. Switch to the AutoCAD application window.
5. On the ribbon, click the Manage tab ➢ Applications panel ➢ Run VBA Macro.
6. When the Macros dialog box opens, select the macro name that ends with WorkingWithSysVars. Click Run.
7. Review the value in the message box and click OK to continue the execution of the procedure.

   The current value of the osmode system variable is displayed after the colon in the message box.
8. Review the message and click OK in the next two message boxes.

   The value of the osmode system variable is changed to 33, and the change is reflected after the colon in the message box. The final message box reflects the original value.

TIP The AutoCAD Help system is a great resource for learning about system variables. However, if you need to support multiple AutoCAD releases, you will need to reference the documentation for each release. To make it easier to identify which system variables are supported in the current and previous AutoCAD releases, Shaan Hurley (http://autodesk.blogs.com/between_the_lines/) and I compiled a list of system variables that spans AutoCAD releases from 2004 through the present; you can view the list here: www.hyperpics.com/system_variables/.

Querying and Setting Application and Document Preferences

System variables provide access to many application and document settings, but there are some settings that are not accessible using system variables. The `AcadApplication` and `AcadDocument` objects both offer a property named `Preferences` that allows you to access additional settings that are not accessible using system variables. The `Preferences` property of the `AcadApplication` object contains a reference to an `AcadPreferences` object. The `AcadPreferences` object provides access to 10 properties that provide access to different preference objects that are used to organize the available preferences. The 10 preference objects represent many of the tabs in the Options dialog box (which you open using the `option` command).

Table 3.5 lists the preference objects that are used to organize application preferences.

Table 3.5: Preference objects accessible from the application

Class/Object	Description
`AcadPreferencesDisplay`	Provides access to settings that control the display and color of user-interface elements, scroll bars, drawing windows, and crosshairs.
`AcadPreferencesDrafting`	Provides access to the AutoSnap and AutoTracking settings.
`AcadPreferencesFiles`	Provides access to the support-file locations, such as Support File Search Path and Drawing Template File Location.
`AcadPreferencesOpenSave`	Provides access to the default drawing format used when saving a drawing with the `save` and `qsave` commands, in addition to settings used to control the loading of Xrefs and ObjectARX applications.
`AcadPreferencesOutput`	Provides access to settings that control the plotting and publishing of drawing files.
`AcadPreferencesProfiles`	Provides access to methods used to manage profiles defined in the AutoCAD drawing environment, as well as a property used to get or switch the active profile.
`AcadPreferencesSelection`	Provides access to settings that control the display of grips and the pickbox.
`AcadPreferencesSystem`	Provides access to application settings that control the display of message boxes and whether the `acad.lsp` file is loaded once per AutoCAD session or into each drawing.
`AcadPreferencesUser`	Provides access to settings that control the default insertion units used with the `insert` command and the behavior of the shortcut menus in the drawing area.

The `Preferences` property of the `AcadDocument` object doesn't provide access to a reference of an `AcadPreferences` object but instead provides access to an `AcadDatabasePreferences` object. The `AcadDatabasePreferences` object can be used to control the display of lineweights, object selection sorting, and the number of contour lines per surface, among many other settings.

The following examples show how to query and set application and drawing preferences:

```
' Sample used to control Application preferences
With ThisDrawing.Application.Preferences
  ' Displays a message box with the current support file search paths
  MsgBox .Files.SupportPath

  ' Displays a message box with all available profiles
  Dim vName As Variant, vNames As Variant, strNames As String
  .Profiles.GetAllProfileNames vNames

  For Each vName In vNames
    strNames = strNames & vName & ","
  Next vName

  MsgBox "Available profile names: " & strNames

  ' Sets the crosshairs to 100
  .Display.CursorSize = 100

  ' Sets the background color of model space color to light gray
  .Display.GraphicsWinModelBackgrndColor = 12632256
End With

' Sample used to control Document preferences
With ThisDrawing.Preferences
  ' Turns off solid fill mode
  .SolidFill = False

  ' Turns on quick text display mode
  .TextFrameDisplay = True
End With
```

Executing Commands

The AutoCAD Object library allows you to automate most common tasks without the use of a command, but there might be times when you will need to use an AutoCAD or third-party command. A command can be executed using the `SendCommand` method of the `AcadDocument` object.

NOTE While using a command might seem like a quick and easy choice over using the methods and properties of the objects in the AutoCAD Object library, you should avoid using commands whenever possible. The execution of an AutoCAD command can be slower and more limited than using the same approach with the AutoCAD Object library and VBA. The behavior of commands is affected by system variables, and ensuring system variables are set to specific values before calling a command can result in you having to write additional code statements that can complicate your programs.

The `SendCommand` method expects a single string that represents the command, options, and values that would be entered at the Command prompt. A space in the string represents the single press of the Enter key. When the Enter key must be pressed, such as when providing a string value that supports spaces, use the `vbCr` constant.

The following statements show how to draw a rectangle and a single-line text object using commands:

```
' Draws a rectangle 0,0 to 10,4
ThisDrawing.SendCommand "._rectang 0,0 10,4 "

' Draws a single line text object with middle center justification
' at 5,2 with a height of 2.5 units and the text string D101

ThisDrawing.SendCommand "._-text _j _mc 5,2 2.5 0 D101" & vbCr
```

Figure 3.1 shows the result of drawing a rectangle with the `rectang` command and single-line text placed inside the rectangle with the `-text` command.

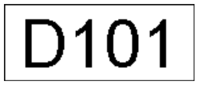


Figure 3.1
Rectangle and text drawn using commands

The string sent by the `SendCommand` method to the `AcadDocument` object is executed immediately in most cases. Typically, the string isn't executed immediately when the `SendCommand` method is called from an event handler. I discuss event handlers in Chapter 10.

Starting with AutoCAD 2015, you can postpone the execution of the commands and options in the string until after the VBA program finishes by using the `PostCommand` method instead of the `SendCommand` method. Unlike with the `SendCommand` method, you don't need to have all commands, options, and values in a single string.

The following statements show how to draw a rectangle with the `PostCommand` method:

```
' Draws a rectangle 0,0 to 10,4
ThisDrawing.PostCommand "._rectang "
ThisDrawing.PostCommand "0,0 10,4 "
```

Exercise: Setting Up a Project

Before a product is manufactured or a building is constructed, it starts as an idea that must be documented. In AutoCAD, documenting is known as drafting or modeling. Similar to a Microsoft Word document, a drawing must be set up to ensure that what you want to design appears and outputs as intended. Although you can create a number of drawing template (DWT) files to use when creating a new drawing, it can be better and more flexible to design an application that can adapt to your company's needs instead of creating many DWT files to try to cover every type of drawing your company might create.

In this section, you will create and set up a new drawing file using some of the concepts that have been introduced in this chapter. The key concepts that are covered in this exercise are as follows:

Managing Documents Create and save a new drawing file.

Assigning and Creating Drawing Properties Assign values to standard drawing properties and create custom drawing properties that can be used to populate text in a title block.

Setting System Variables and Preferences Changes can be made to system variables and preferences that are stored with the application or a drawing to affect the behavior of drafting aids and other AutoCAD features.

Performing Tasks with a Command AutoCAD commands can be used to create and modify graphical and nongraphical objects in a drawing.

Creating the *DrawingSetup* Project

A project is used to store any and all VBA code that is to be executed in the AutoCAD drawing environment. The following steps explain how to create a project named `DrawingSetup` and save it to a file named `drawingsetup.dvb`:

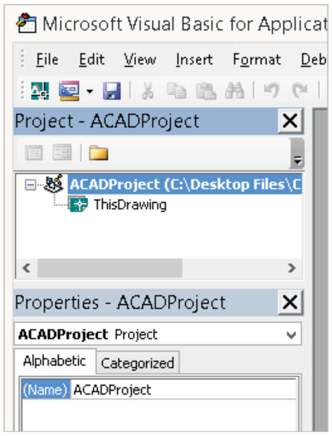
1. On the ribbon, click the Manage tab ➢ Applications panel title bar and then click VBA Manager (or at the Command prompt, type **`vbaman`** and press Enter).
2. When the VBA Manager opens, select the first project in the Projects list and click Unload. If prompted to save the changes, click Yes if you wish to save the changes, or click No to discard the changes.
3. Repeat step 2 for each VBA project in the list.
4. Click New.

   The new project is added to the list with a default name of `ACADProject` and a location of `Global1`, `Global2`, and so on based on how many projects have been created in the current AutoCAD session.
5. Select the new project from the Projects list and click Save As.
6. When the Save As dialog box opens, browse to the `MyCustomFiles` folder within the `Documents` (or `My Documents`) folder, or the location you are using to store custom program files.
7. In the File Name text box, type **`drawingsetup`** and click Save.
8. In the VBA Manager dialog box, click Visual Basic Editor.

The next steps explain how to change the project name from `ACADProject` to `DrawingSetup` and add a new code module named `basDrawingSetup`:

1. When the VBA Editor opens, select the project node labeled `ACADProject` (shown in Figure 3.2) from the Project Explorer. If the Project Explorer isn't displayed, click View ➢ Project Explorer on the menu bar in the VBA Editor.

Figure 3.2
Navigating the new project with the Project Explorer


Microsoft Visual Basic for Applicat
File Edit View Insert Format Deb
Project - ACADProject
Properties - ACADProject
Alphabetic
Categorized

ACADProject (C:\Desktop Files\C
ThisDrawing

ACADProject Project

(Name) ACADProject

2. In the Properties window, select the field named (Name) and double-click in the text box adjacent to the field. If the Properties window isn't displayed, click View ➢ Properties Window on the menu bar in the VBA Editor.
3. In the text box, type **`DrawingSetup`** and press Enter.
4. On the menu bar, click Insert ➢ Module.
5. In the Project Explorer, select the new module named `Module1`.
6. In the Properties window, change the current value of the (Name) property to **`basDrawingSetup`**.
7. On the menu bar, click File ➢ Save.

Creating and Saving a New Drawing from Scratch

Designs created with the AutoCAD drawing environment are stored in a DWG file, which can then be shared with others in your organization or external vendors and clients. The `New` method of the `AcadDocuments` collection object gives you the most flexibility in creating a new drawing file.

In the following steps, you define a procedure named `newDWGFromScratch`, which will be used to create a drawing from scratch with imperial units and return the `AcadDocument` object that represents the new drawing. Once the drawing is created, the `SaveAs` method of the new `AcadDocument` object is used to save the drawing with the name of `ACP-D1.B.dwg` to the `MyCustomFiles` folder within the `Documents` (or `My Documents`) folder, or the location you are using to store files from this book.

1. In the Project Explorer, double-click the code module named `basDrawingSetup`.
2. When the code editor opens, type the following:

```
' Creates a new drawing from scratch
' Function accepts an optional value of:
' 0 - Creates an imperial units based drawing
' 1 - Creates a metric units based drawing
Private Function newDWGFromScratch _
                 (Optional nMeasureInit As Integer = 0) As AcadDocument
  ' Get the current value of the MEASUREINIT system variable
  Dim curMInit As Integer
  curMInit = ThisDrawing.GetVariable("measureinit")

  ' Set the measurement system for new drawings to metric
  ThisDrawing.SetVariable "measureinit", nMeasureInit

  ' Create a new drawing from scratch
  Dim newDWGFromScratch As AcadDocument
  Set newDWGFromScratch = Application.Documents.Add

  ' Restore the previous value
  ThisDrawing.SetVariable "measureinit", curMInit
End Function
```

3. On the menu bar, click File ➢ Save.

Since the procedure `newDWGFromScratch` is designated as private, it can't be executed from the AutoCAD user interface with the `vbarun` command. In the next steps, you will create a public procedure named `Main` that will be used to execute the various procedures that will make up the final functionality of the `DrawingSetup` project.

1. In the code editor, click after the `End Function` code statement of the `newDWGFromScratch` procedure and press Enter twice.
2. Type the following:

```
Public Sub Main()
  ' Executes the newDWGFromScratch to create a new drawing from sratch
  Dim newDWG As AcadDocument
  Set newDWG = newDWGFromScratch

  ' Saves the new drawing
  ThisDrawing.SaveAs ThisDrawing.GetVariable("mydocumentsprefix") & _
                     "\MyCustomFiles\acp-d1_b.dwg"
End Sub
```

3. On the menu bar, click File ➢ Save.
4. In the code editor, click after the code statement that starts with `Public Sub Main`.

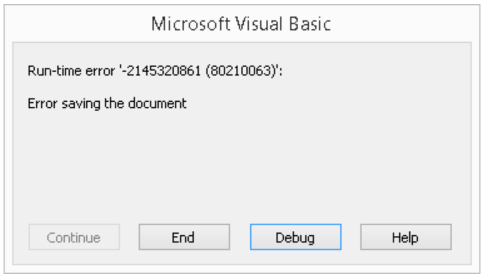
5. On the menu bar, click Run ➢ Run Sub/UserForm. If the Macros dialog box opens, select Main and click Run. If you clicked inside the Main procedure definition, the Macro dialog box will not be displayed.

   The new drawing is created and saved to the file named `acp-d1_b.dwg`. If an error message is displayed, make sure that the `MyCustomFiles` folder exists under the `Documents` (or `My Documents`) folder, or update the code to reflect the folder you are using to store the files for this book.

6. On the Windows taskbar, click the AutoCAD application icon and verify that the new drawing was created.

7. Try executing the `Main` procedure again.

   This time an error message is displayed, as shown in Figure 3.3, which indicates that the drawing couldn't be saved. The drawing couldn't be saved because it was already open in AutoCAD and the file was locked on the local disc. I cover how to handle errors in Chapter 13.



```
Run-time error '-2145320861 (80210063)':

Error saving the document
```

FIGURE 3.3 Error message generated as a result of AutoCAD not being able to save the drawing

8. In the Microsoft Visual Basic error message box, click End to terminate the execution of the code.

Inserting a Title Block with the *insert* Command

The next steps insert a title block into the current drawing. The `insert` command is sent to the current drawing with the `SendCommand` method.

NOTE From `www.sybex.com/go/autocadcustomization` you can download the drawing file `b-tblk.dwg` used by the `insert` command in the following steps. Place the file in the `MyCustomFiles` folder within the `Documents` (or `My Documents`) folder, or the location you are using to store custom program files. If you are storing the files for this book in a different folder other than `MyCustomFiles` under the `Documents` (or `My Documents`) folder, update the code in the following steps as needed.

1. In the code editor, click after the `End Sub` code statement of the `Main` procedure and press Enter twice.

2. Type the following:

```
Private Sub insertTitleBlock()
  With ThisDrawing
    ' Gets the current layer name
    Dim sLyrName As String
    sLyrName = .GetVariable("clayer")

    ' Creates a new layer named TBlk with the ACI value 8
    .SendCommand "._-layer _m " & "TBlk" & vbCr & "_c 8 " & "TBlk" & vbCr & vbCr

    ' Inserts the title block drawing
    .SendCommand "._-insert " & .GetVariable("mydocumentsprefix") & _
                 "\MyCustomFiles\b-tblk" & vbCr & "0,0 1 1 0" & vbCr

    ' Zooms to the extents of the drawing
    .SendCommand "._zoom _e" & vbCr

    ' Restores the previous layer
    .SetVariable "clayer", sLyrName
  End With
End Sub
```

3. Scroll up and add the statements shown in bold to the `Main` procedure:

```
Public Sub Main()
  ' Executes the newDWGFromScratch to create a new drawing from scratch
  Dim newDWG As AcadDocument
  Set newDWG = newDWGFromScratch

  ' Saves the new drawing
  ThisDrawing.SaveAs ThisDrawing.GetVariable("mydocumentsprefix") & _
                     "\MyCustomFiles\acp-d1_b.dwg"

  ' Insert the title block
  insertTitleBlock
End Sub
```

4. On the menu bar, click File ➢ Save.
5. Close all open drawing files and then create a new drawing file.
6. Execute the `Main` procedure with the `vbarun` command or by clicking Run ➢ Run Sub/UserForm from the VBA Editor menu bar.

   A new drawing should be created and the title block drawing `b-tblk` is inserted on the `TBlk` layer, as shown in Figure 3.4.

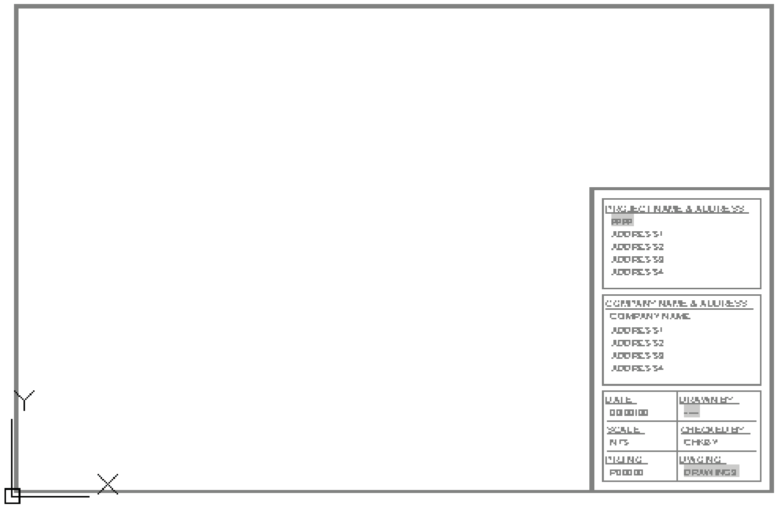


FIGURE 3.4 New drawing with a title block

7. Switch to the AutoCAD application to view the new drawing and title block.

Adding Drawing Properties

Drawing properties can be a great way to populate values of a title block using fields or even to help you locate a drawing file years later. (You will remember that drawing properties can be searched using the Search field in Windows Explorer and File Explorer, making it possible to find a drawing based on an assigned property value.) Drawing properties are stored with a DWG file and are accessible with the `dwgprops` command or the `AcadSummaryInfo` object of the AutoCAD Object library.

In the following steps, you will define a procedure named `addDWGProps`, which adds some static values to some of the standard drawing properties and creates a few custom drawing properties. A few of the values are used by the fields in the title block that was inserted with the `insertTitleBlock` procedure added in the previous section.

1. In the AutoCAD drawing window, zoom into the lower-right area of the title block.

   You should notice a few values with the text `----`. This text is the default value of a field value that can't be resolved.

2. In the code editor, click after the `End Sub` code statement of the `insertTitleBlock` procedure and press Enter twice.

3. Type the following:

```
Private Sub addDWGProps()
  With ThisDrawing.SummaryInfo
    ' Set the author and comment properties
    .Author = "[Replace this text with your initials here]"
```

```
      .Comments = "Phase 1: 1st Floor Furniture Plan"

      ' Add custom properties to a drawing
      Dim sProject As String
      Dim sPhase As String

      On Error Resume Next

      .GetCustomByKey "ProjectName", sProject

      If Err.Number <> 0 Then
        ' Property doesn't exist
        .AddCustomInfo "ProjectName", "ACP Renovation"
        Err.Clear
      Else
        ' Property exists, so update the value
        .SetCustomByKey "ProjectName", "ACP Renovation"
      End If
    End With

    ' Regen the drawing to update the fields
    ThisDrawing.Regen acActiveViewport
  End Sub
```

4. Scroll up and add the statements shown in bold to the `Main` procedure:

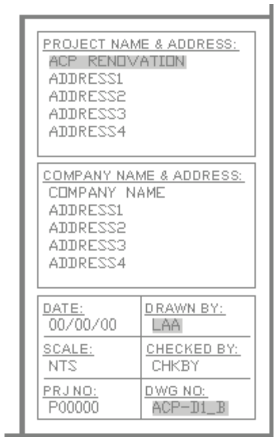
```
Public Sub Main()
  ' Executes the newDWGFromScratch to create a new drawing from scratch
  Dim newDWG As AcadDocument
  Set newDWG = newDWGFromScratch

  ' Saves the new drawing
  ThisDrawing.SaveAs ThisDrawing.GetVariable("mydocumentsprefix") & _
                     "\MyCustomFiles\acp-d1_b.dwg"

  ' Insert the title block
  insertTitleBlock

  ' Add the drawing properties
  addDWGProps
End Sub
```

5. On the menu bar, click File ➢ Save.
6. Close all open drawing files and then create a new drawing file.

7. Execute the `Main` procedure.

   The "project name" and "drafted by" values are populated in the title block by the property values assigned to the drawing. The property values of the drawing are assigned using the methods and properties of the `AcadSummaryInfo` object, as shown in Figure 3.5.

Figure 3.5 Field values populated by drawing properties



PROJECT NAME & ADDRESS:
ACP RENOVATION
ADDRESS1
ADDRESS2
ADDRESS3
ADDRESS4

COMPANY NAME & ADDRESS:
COMPANY NAME
ADDRESS1
ADDRESS2
ADDRESS3
ADDRESS4

DATE: 00/00/00	DRAWN BY: LAA
SCALE: NTS	CHECKED BY: CHKBY
PRJ NO: P00000	DWG NO: ACP-D1_B

Setting the Values of Drafting-Related System Variables and Preferences

System variables and the preferences of the application or drawing can be used to affect many of the commands and drafting aids in the AutoCAD drawing environment.

In the following steps, you define a procedure named `setDefDraftingAids`, which specifies the values of system variables and application preferences.

1. In the code editor, click after the `End Sub` code statement of the `addDWGProps` procedure and press Enter twice.
2. Type the following:

```
Private Sub setDefDraftingAids()
  ' Set the values of drafting-related system variables
  With ThisDrawing
    .SetVariable "orthomode", 1
    .SetVariable "osmode", 35
    .SetVariable "gridmode", 0
    .SetVariable "snapmode", 0
    .SetVariable "blipmode", 0
  End With

  ' Set display-related preferences
  With ThisDrawing.Application.Preferences.Display
    .CursorSize = 100
  End With
```

```
    ' Set drafting-related preferences
    With ThisDrawing.Application.Preferences.Drafting
      .AutoSnapAperture = True
      .AutoSnapApertureSize = 10
    End With

    ' Set selection-related preferences
    With ThisDrawing.Application.Preferences.Selection
      .DisplayGrips = True
      .PickFirst = True
    End With
  End Sub
```

3. Scroll up and add the statements shown in bold to the `Main` procedure:

```
Public Sub Main()
  ' Executes the newDWGFromScratch to create a new drawing from scratch
  Dim newDWG As AcadDocument
  Set newDWG = newDWGFromScratch

  ' Saves the new drawing
  ThisDrawing.SaveAs ThisDrawing.GetVariable("mydocumentsprefix") & _
                     "\MyCustomFiles\acp-d1_b.dwg"

  ' Insert the title block
  insertTitleBlock

  ' Add the drawing properties
  addDWGProps

  ' Sets the values of system variables and application preferences
  setDefDraftingAids

  ' Saves the changes to the drawing
  ThisDrawing.Save
End Sub
```

4. On the menu bar, click File ➢ Save.
5. Close all open drawing files and then create a new drawing file.
6. Execute the `Main` procedure.

   The system variables and application preferences are changed, and the changes to the drawing file are saved.

Chapter 4

Creating and Modifying Drawing Objects

All drawings start off as an idea. Maybe it's just in your head, or maybe it became a sketch done on a napkin over lunch. The idea or sketch is then handed over to a drafter or engineer, who creates a set of drawings that will be used to communicate the final design. The final design is then used to manufacture the parts or construct the building. A drafter or engineer completes a design using a variety of objects, from lines to circles, and even splines and hatch patterns.

Although adding objects to a drawing is how most designs start off, those objects are often used to create new objects or are modified to refine a design. Most users of the AutoCAD® program, on average, spend more time modifying objects than creating new objects. When automating tasks with VBA, be sure to look at tasks related not only to *creating* objects but also to *modifying* objects.

In this chapter, you will learn to create 2D graphical objects and how to work with nongraphical objects, such as layers and linetypes. Along with creating objects, you will learn how to modify objects.

Due to limitations on the number of pages available for this book, additional content that covers working with 2D and 3D objects is presented in the bonus chapters available on the companion website. The companion website is located here: `www.sybex.com/go/autocadcustomization`.

Understanding the Basics of a Drawing-Based Object

Each drawing contains two different types of objects: nongraphical and graphical. Nongraphical objects represent the layers, block definitions, named styles, and other objects that are stored in a drawing but aren't present in model space or on a named layout. Nongraphical objects can, and typically do, affect the display of graphical objects.

Although model space and named layouts are typically not thought of as nongraphical objects, they are. Model space is a special block definition, whereas a layout is an object that is based on a plot configuration—commonly called a page setup—with a reference to a block definition. Graphical objects are those objects that are added to model space or a named layout, such as lines, circles, and text. Every graphical object added to a drawing references at least one nongraphical object and is owned by one nongraphical object. The nongraphical object that each graphical object references is a layer, and each graphical object is owned by model space or a named layout.

In the AutoCAD Object library, any object that can be added to a drawing is derived from or based on the `AcadObject` object type. For example, an `AcadLine` object that represents a line segment and an `AcadLayer` object that represents a layer have the same properties and methods as the `AcadObject` object type. You can think of the `AcadObject` as a more general or generic object in the terms of AutoCAD objects, much like you might use the term *automobile* to describe a vehicle with four wheels. Figure 4.1 shows the object hierarchy of nongraphical and graphical objects.

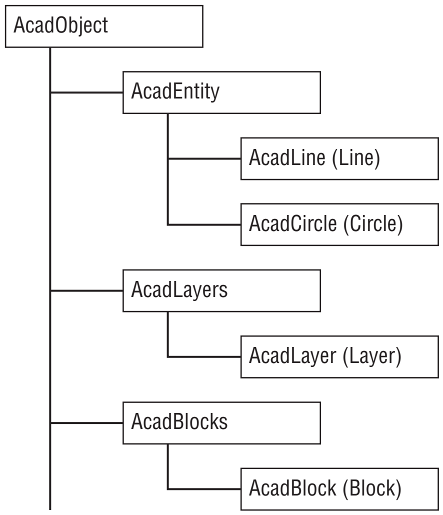


Figure 4.1
Drawing object hierarchy

Table 4.1 lists the properties of the `AcadObject` object that you use to get information about an object in a drawing.

Table 4.1: Properties related to the `AcadObject` object

Property	**Description**
`Application`	Returns the `AcadApplication` object that represents the current AutoCAD session. I discussed working with the `AcadApplication` object in Chapter 3, "Interacting with the Application and Documents Objects."
`Document`	Returns the `AcadDocument` object that represents the drawing in which the object is stored. I discussed working with the `AcadDocument` object in Chapter 3.
`Handle`	Returns a string that represents a unique hexadecimal value that differentiates one object from another in a drawing; think of it along the lines of a database index. An object's handle persists between drawing sessions. A handle, while unique to a drawing, can be assigned to another object in a different drawing.
`HasExtensionDictionary`	Returns `True` if an extension dictionary has been attached to the object. I discuss extension dictionaries in Chapter 9, "Storing and Retrieving Custom Data."

Property	Description
`ObjectID`	Returns a unique integer that differentiates one object from another in a drawing; think of it along the lines of a database index. Unlike a handle, the object ID of an object might be different each time a drawing is loaded into memory.
`ObjectID32`	Same as the `ObjectID` property, but must be used on 64-bit releases of Windows. This property is only valid with AutoCAD 2009 through 2014. Use the `ObjectID` property for earlier releases and AutoCAD 2015.
`ObjectName`	Returns a string that represents the object's internal classname. This value can be used to distinguish one object type from another as part of a conditional statement.
`OwnerID`	Returns the object ID of the object's parent. For example, the parent of a line might be model space or a named layout whereas the text style symbol table is the parent of a text style.
`OwnerID32`	Same as the `OwnerID` property, but must be used on 64-bit releases of Windows. This property is only valid with AutoCAD 2009 through 2014. Use the `OwnerID` property for earlier releases and AutoCAD 2015.

Determining a Drawing Object's Type

The `ObjectName` property and VBA `TypeOf` statement can be used to determine an object's type. The following code statements demonstrate how to display an object's name in a message box and use the `TypeOf` statement to determine whether an object is based on the `AcadCircle` class:

```
' Gets the first object in model space
Dim oFirstEnt As AcadEntity
Set oFirstEnt = ThisDrawing.ModelSpace(0)

' Display an object's name in a message box
MsgBox "ObjectName: " & oFirstEnt.ObjectName

' Check to see if the object is a circle, and
' display a message based on the results
If TypeOf oFirstEnt Is AcadCircle Then
  MsgBox "Object is a circle."
Else
  MsgBox "The object isn't a circle."
End If
```

In addition to the properties that are shared across all drawing-based objects, several methods are shared. The `Delete` method is used to remove an object from a drawing; it is the AutoCAD Object library equivalent of the `erase` command. The other three shared methods are used to work with extension dictionaries and extended data (Xdata). These three methods are `GetExtensionDictionary`, `GetXData`, and `SetXData`, and I cover them in Chapter 9.

All graphical objects in a drawing are represented by the `AcadEntity` object. The `AcadEntity` object inherits the properties and methods of the `AcadObject` object and adds additional properties and methods that all graphical objects have in common. For example, all graphical objects can be assigned a layer and moved in the drawing. The `Layer` property of the `AcadEntity` object is used to specify the layer in which an object is placed, and the `Move` method is used to relocate an object in the drawing. Objects based on the `AcadEntity` object can be added to model space, a named layout, or a block definition.

Table 4.2 lists the properties of the `AcadEntity` object that you can use to get information about and control the appearance of a graphical object in a drawing.

TABLE 4.2: Properties related to the `AcadEntity` object

PROPERTY	DESCRIPTION
`EntityTransparency`	Specifies the transparency for an object. See Bonus Chapter 1, "Working with 2D Objects and Object Properties."
`Hyperlinks`	Returns the `AcadHyperlinks` collection object assigned to an object. See Bonus Chapter 1.
`Layer`	Specifies the layer for an object. See Bonus Chapter 1.
`Linetype`	Specifies the linetype for an object. See Bonus Chapter 1.
`LinetypeScale`	Specifies the linetype scale for an object. See Bonus Chapter 1.
`Lineweight`	Specifies the lineweight for an object. See Bonus Chapter 1.
`Material`	Specifies the name of the material to use when an object is rendered. See Bonus Chapter 2, "Modeling in 3D Space."
`PlotStyleName`	Specifies the name of the plot style for an object. See Bonus Chapter 1.
`TrueColor`	Specifies the color assigned to an object. See Bonus Chapter 1.
`Visible`	Specifies the visibility for an object. See Bonus Chapter 1.

INHERITING DEFAULT PROPERTY VALUES

When new objects are added to a drawing, they inherit many of their default property values from system variables; this occurs whether you are using an AutoCAD command or the AutoCAD Object library. For example, the `clayer` system variable holds the name of the layer that is assigned to the `Layer` property of each new graphical object. If your functions need to create multiple objects on a specific layer, it is best to set that layer current before adding new graphical objects and then restore the previous layer after the objects have been added.

The following lists other system variables that affect the default properties of new graphical objects:

`cecolor:`	Color assigned to the `TrueColor` property
`celtype:`	Linetype assigned to the `Linetype` property
`celweight:`	Lineweight assigned to the `Lineweight` property
`celtscale:`	Linetype scale assigned to the `LinetypeScale` property
`cetransparency:`	Transparency assigned to the `EntityTransparency` property
`cmaterial:`	Material assigned to the `Material` property
`cplotstyle:`	Plot style name assigned to the `PlotStyleName` property

Table 4.3 lists the methods that all graphical objects inherit from the `AcadEntity` object.

TABLE 4.3: Methods related to the `AcadEntity` object

METHOD	**DESCRIPTION**
`ArrayPolar`	Creates a polar array from an object. See Bonus Chapter 1.
`ArrayRectangular`	Creates a rectangular array from an object. See Bonus Chapter 1.
`Copy`	Duplicates an object. See the “Copying and Moving Objects” section.
`GetBoundingBox`	Returns an array of doubles that represents the lower and upper points of an object’s extents. See Bonus Chapter 1.
`Highlight`	Highlights or unhighlights an object. See Bonus Chapter 1.
`IntersectWith`	Returns an array of doubles that represents the intersection points between two objects. See Bonus Chapter 1.
`Mirror`	Mirrors an object along a vector. See Bonus Chapter 1.
`Mirror3D`	Mirrors an object about a plane. See Bonus Chapter 2.
`Move`	Moves an object. See the “Copying and Moving Objects” section.
`Rotate`	Rotates an object around a base point. See the “Rotating Objects” section.
`Rotate3D`	Rotates an object around a vector. See Bonus Chapter 2.
`ScaleEntity`	Uniformly increases or decreases the size of an object. See Bonus Chapter 1.
`TransformBy`	Applies a transformation matrix to an object. A transformation matrix can be used to scale, rotate, move, and mirror an object in a single operation. See Bonus Chapter 1.
`Update`	Instructs AutoCAD to recalculate the display of an object; similar to the `regen` command but it only affects the object in which the method is executed. See the “Modifying Objects” section.

Accessing Objects in a Drawing

Before working with nongraphical and graphical objects, you must understand where objects are located in the AutoCAD Object hierarchy. Nongraphical objects are stored in collection objects that are accessed from an `AcadDocument` or `ThisDrawing` object. Even graphical objects displayed in model space, on named layouts, or in a block definition require you to work with a collection object. I explained how to work with the `AcadDocument` object in Chapter 3.

To access the collection objects of a drawing, use the properties of an `AcadDocument` object. Collection objects may also be referred to as symbol tables or dictionaries the AutoLISP and Managed .NET programming languages (Table 4.4).

Table 4.4: Properties used to access the collection objects of an `AcadDocument` object

Property	**Description**
`Blocks`	Returns an `AcadBlocks` collection object that contains the block definitions stored in a drawing, even model space, paper space, and those used for named layouts. See Chapter 7, "Working with Blocks and External References," for more information.
`Dictionaries`	Returns an `AcadDictionaries` collection object that contains the named dictionaries stored in a drawing. See Chapter 9 for more information.
`DimStyles`	Returns an `AcadDimStyles` collection object that contains the dimension styles stored in a drawing. See Chapter 6, "Annotating Objects," for more information.
`FileDependencies`	Returns an `Acad FileDependencies` collection object that contains the external file dependencies used by a drawing. See Chapter 7 for more information.
`Groups`	Returns an `AcadGroups` collection object that contains the named groups defined in a drawing. See Bonus Chapter 1.
`Layers`	Returns an `AcadLayers` collection object that contains the layers stored in a drawing. See Bonus Chapter 1.
`Layouts`	Returns an `AcadLayouts` collection object that contains the named layouts stored in a drawing. See Chapter 8, "Outputting Drawings," for more information.
`Linetypes`	Returns an `AcadLinetypes` collection object that contains the linetypes stored in a drawing. See Bonus Chapter 1.

Property	**Description**
`Materials`	Returns an `AcadMaterialss` collection object that contains the names of the materials stored in a drawing. See Bonus Chapter 2.
`ModelSpace`	Returns an `AcadBlock` object that is a reference to model space in the drawing. See the "Working with Model or Paper Space" section.
`PaperSpace`	Returns an `AcadBlock` object that is a reference to paper space in the drawing. See the "Working with Model or Paper Space" section.
`PlotConfigurations`	Returns an `Acad PlotConfigurations` collection object that contains the named plot configurations stored in a drawing. See Chapter 8 for more information.
`RegisteredApplications`	Returns an `AcadRegisteredApplications` collection object that contains the names of all registered applications that store custom data in a drawing. See Chapter 9 for more information.
`TextStyles`	Returns an `AcadTextStyles` collection object that contains the text styles stored in a drawing. See Chapter 6 for more information.
`UserCoordinateSystems`	Returns an `AcadUCSs` collection object that contains the user coordinate systems saved in a drawing. See Bonus Chapter 2.
`Viewports`	Returns an `AcadViewports` collection object that contains the named arrangements of tiled viewports for use in model space. See Chapter 5, "Interacting with the User and Controlling the Current View," for more information.

NOTE Not all named styles are accessible from a property of the `AcadDocument` object. For example, table and multileader styles are stored as dictionaries and accessed from the `Dictionaries` property.

Working with Model or Paper Space

Graphical objects created by the end user or with the AutoCAD Object library are all added to a block definition. Although this might seem a bit confusing at first, model space is nothing more than a block definition that is edited using the drawing area displayed in the drawing window. The same is true with paper space and the named layouts stored in a drawing. Before you can add or modify an object in a drawing file, you must determine which block definition to work with.

Model space and paper space are accessed using the ModelSpace and PaperSpace properties of an AcadDocument or ThisDrawing object. You use the ModelSpace property to get a reference to an AcadModelSpace object, which is actually a reference to the block definition named *MODEL_SPACE. The PaperSpace property returns a reference to an AcadPaperSpace object, which is a reference to the most recently accessed paper space block. The initial paper space block is named *PAPER_SPACE or *PAPER_SPACE0. Switching named layouts changes which paper space block is returned by the PaperSpace property.

You use the AcadModelSpace and AcadPaperSpace objects to access the graphical objects in a drawing. A majority of the methods that these two objects support are related to adding new graphical objects. To add new graphical objects to a drawing, use the methods whose names start with the prefix Add. I explain how to add graphical objects to model space in the next section and I cover how to access the objects already in model space in the "Getting an Object in the Drawing" section. You can learn more about the properties and methods specific to block definitions in Chapter 7 and the properties and methods specific to named layouts in Chapter 8.

The standard commands of AutoCAD typically work in the current context of the drawing. If model space is active and the line command is started, the line object is added to model space. However, if the line command is started when a named layout is current, the line is added to the named layout. The AutoCAD Object library isn't concerned with the active space. Model space might be active, but objects can be added to paper space and vice versa. The active space can be bypassed with the AutoCAD Object library and VBA because you have direct access to the objects in a drawing's database.

The active space doesn't matter so much when adding and modifying objects with the AutoCAD Object library, but users still expect macros to be executed in the current context of the drawing. The ActiveSpace property can be used to determine which space is active. A constant value of acModelSpace or acPaperSpace is returned by the ActiveSpace property; acModelSpace is returned when model space is current. You can also use the ActiveSpace property to switch the active space; assign the constant value of acPaperSpace to switch to paper space when model space is current.

The following code statements display a message containing the number of objects in the current space:

```
Dim nCnt As Integer
nCnt = 0

Select Case ThisDrawing.ActiveSpace
  Case acModelSpace
    nCnt = ThisDrawing.ModelSpace.Count
  Case acPaperSpace
    nCnt = ThisDrawing.PaperSpace.Count
End Select

MsgBox "Number of objects in current space: " & CStr(nCnt)
```

TIP As an alternative to specifying model space or paper space, you can use the ActiveLayout property of an AcadDocument or ThisDrawing object. The ActiveLayout property can be helpful when you want to draw objects on the current layout. Using the Block property of the AcadLayout object that is returned by the ActiveLayout property, you can get a reference to model space or paper space. When the Model layout is current, ActiveLayout.Block returns a reference to the AcadModelSpace object. I discuss more about layouts in Chapter 8.

Creating Graphical Objects

Graphical objects are used to communicate a design, whether a mechanical fastener or a new football stadium. AutoCAD supports two types of graphical objects: straight and curved. Straight objects, such as lines, rays, and xlines, contain only straight segments. Curved objects can have curved segments, but as an option can have straight segments, too. Arcs, circles, splines, and polylines with arcs are considered examples of curved objects. I cover commonly used straight and curved objects in the "Adding Straight Line Segments" and "Working with Curved Objects" sections. Polylines are discussed in the "Working with Polylines" section.

NOTE As a reminder, graphical objects inherit many of their properties and methods from the `AcadEntity` object. For that reason, I only focus on the properties and methods specific to an object as they are introduced going forward. I covered the `AcadEntity` object in the "Understanding the Basics of a Drawing-Based Object" section earlier in this chapter.

Adding Straight Line Segments

Straight objects are used in a variety of drawings created by drafters and engineers. You can use a straight object to represent the following:

- The top of a bolt head
- The tooth of a gear
- A wire in a wiring diagram
- The edge of a student desk
- The face of a wall for a building

Lines are straight objects with a defined start point and endpoint and are represented by the `AcadLine` object. The `AddLine` function allows you to create a line object drawn between two points. The following shows the syntax of the `AddLine` function:

```
retVal = object.AddLine(startPoint, endPoint)
```

Its arguments are as follows:

retVal The *retVal* argument represents the new `AcadLine` object returned by the `AddLine` function.

object The *object* argument represents the `AcadModelSpace` collection object.

startPoint The *startPoint* argument is an array of three doubles that defines the start point of the new line.

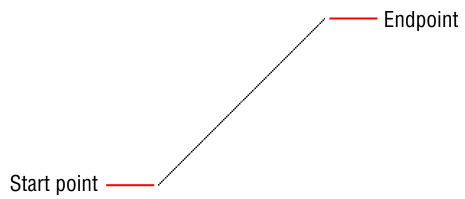
endPoint The *endPoint* argument is an array of three doubles that defines the endpoint of the new line.

The following code statements add a new line object to model space (see Figure 4.2):

```
' Defines the start and endpoint for the line
Dim dStartPt(2) As Double, dEndPt(2) As Double
dStartPt(0) = 0: dStartPt(1) = 0: dStartPt(2) = 0
dEndPt(0) = 5: dEndPt(1) = 5: dEndPt(2) = 0
```

```
Dim oLine As AcadLine
Set oLine = ThisDrawing.ModelSpace.AddLine(dStartPt, dEndPt)
```

FIGURE 4.2
Definition of a line



Using the `AcadLine` object returned by the `AddLine` function, you can obtain information about and modify the line's properties. In addition to the properties that the `AcadLine` object shares in common with the `AcadEntity` object, you can use the properties listed in Table 4.5 when working with an `AcadLine` object.

TABLE 4.5: Properties related to an `AcadLine` object

PROPERTY	DESCRIPTION
`Angle`	Returns a double that represents the angle of the line expressed in radians. All angles are stored in a drawing file as radians.
`Delta`	Returns an array of three double values that represent the delta of the line: the difference between the line's start and endpoints.
`EndPoint`	Specifies the endpoint of the line.
`Length`	Returns a double that represents the length of the line.
`Normal`	Specifies the normal vector of the line. The normal vector is an array of three double values, which defines the positive Z-axis of the line.
`StartPoint`	Specifies the start point of the line.
`Thickness`	Specifies the thickness assigned to the line; the value must be numeric. The default is 0; anything greater than 0 results in the creation of a 3D planar object.

Working with Curved Objects

Straight objects are used in many designs, but they aren't the only objects. Curved objects are used to soften the edges of a design and give a design a more organic look. You can use a curved object to represent any of the following:

- A hole in a plate
- A fillet on a metal bracket

- A round edge on the top of a desk
- A cross section of a shaft or hub

I discuss how to create and modify circles in the upcoming sections.

NOTE Ellipse and spline objects are covered in Bonus Chapter 1.

Creating and Modifying Circles

Circles are one of the most commonly used curved objects in mechanical designs, but they are less frequently used in architectural and civil designs. Drill holes in the top view of a model, the center of a gear, or the grommet in the side of a desk are typically circular and are drawn using circles. Circles in a drawing are represented by the `AcadCircle` object in the AutoCAD Object library. The `AddCircle` function allows you to create a circle object based on a center point and radius value, and the function returns an `AcadCircle` object that represents the new circle. The following shows the syntax of the `AddCircle` function:

```
retVal = object.AddCircle(centerPoint, radius)
```

Its arguments are as follows:

retVal The *retVal* argument represents the new `AcadCircle` object returned by the `AddCircle` function.

object The *object* argument represents the `AcadModelSpace` collection object.

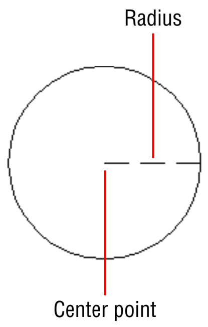
centerPoint The *centerPoint* argument is an array of three doubles that defines the center point of the new circle.

radius The *radius* argument is a double that specifies the radius of the new circle. If you know the diameter of the circle you want to create, divide that value in half to get the radius for the circle.

The following code statements add a new circle object to model space (see Figure 4.3):

```
' Defines the center point for the circle object
Dim dCenPt(2) As Double
dCenPt(0) = 2.5: dCenPt(1) = 1: dCenPt(2) = 0

' Adds the circle object to model space with a radius of 4
Dim oCirc As AcadCircle
Set oCirc = ThisDrawing.ModelSpace.AddCircle(dCenPt, 4)
```

FIGURE 4.3 Definition of a circle



The properties and methods of the `AcadCircle` object returned by the `AddCircle` function can be used to obtain information about and modify the circle. An `AcadCircle` object shares properties and methods in common with the `AcadEntity` object, but it has additional properties that describe the circle object. Table 4.6 lists the properties specific to the `AcadCircle` object.

Table 4.6: Properties related to an `AcadCircle` object

Property	Description
`Area`	Returns a double that represents the calculated area of the circle.
`Center`	Specifies the center point of the circle. That value is expressed as an array of three doubles.
`Circumference`	Returns a double that represents the circumference of the circle.
`Diameter`	Specifies the diameter of the circle; the value is a double.
`Normal`	Specifies the normal vector of the line. The normal vector is an array of three doubles that defines the positive Z-axis for the circle.
`Radius`	Specifies the radius of the circle; the value is a double.
`Thickness`	Specifies the thickness assigned to the circle; the value must be numeric. The default is 0; anything greater than 0 results in the creation of a 3D cylinder object.

Adding and Modifying Arcs

Fillets and rounded corners are common in many types of designs, and they are drawn using arcs. Arcs are partial circles represented by the `AcadArc` object. An arc is added to a drawing with the `AddArc` function. Unlike drawing arcs with the `arc` command, which offers nine options, the `AddArc` function offers only one approach to adding an arc, and that is based on a center point, two angles (start and end), and a radius. The `AddArc` function returns an `AcadArc` object that represents the new arc added to the drawing. The following shows the syntax of the `AddArc` function:

```
retVal = object.AddArc(centerPoint, radius, startAngle, endAngle)
```

Its arguments are as follows:

retVal The *retVal* argument represents the new `AcadArc` object returned by the `AddArc` function.

object The *object* argument represents the `AcadModelSpace` collection object.

centerPoint The *centerPoint* argument is an array of three doubles that defines the center point of the new arc.

radius The *radius* argument is a double that specifies the radius of the new arc.

startAngle* and *endAngle The *startAngle* and *endAngle* arguments are doubles that specify the starting and end angle of the new arc, respectively. A start angle larger than the end angle results in the arc being drawn in a counterclockwise direction. Angles are measured in radians.

The following code statements add a new arc object to model space (see Figure 4.4):

```
' Defines the center point for the arc object
Dim dCenPt(2) As Double
dCenPt(0) = 2.5: dCenPt(1) = 1: dCenPt(2) = 0

' Sets the value of PI
Dim PI As Double
PI = 3.14159265

' Adds the arc object to model space with a radius of 4
Dim oArc As AcadArc
Set oArc = ThisDrawing.ModelSpace.AddArc(dCenPt, 4, PI, 0)
```

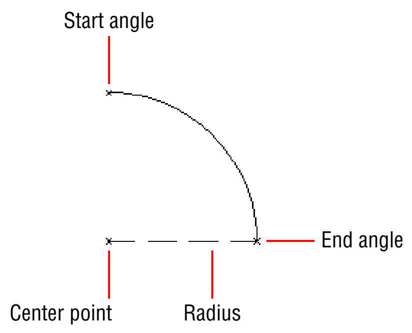


FIGURE 4.4
Definition of an arc

The `AcadArc` object returned by the `AddArc` function can be used to obtain information about and modify the object's properties and methods. In addition to the properties that the `AcadArc` object shares in common with the `AcadEntity` object, you can use the properties listed in Table 4.7 when working with an `AcadArc` object.

TABLE 4.7: Properties related to an `AcadArc` object

PROPERTY	DESCRIPTION
`ArcLength`	Returns a double that represents the length along the arc.
`Area`	Returns a double that represents the calculated area of the arc.
`Center`	Specifies the center point of the arc. The value is expressed as an array of three doubles.
`EndAngle`	Specifies a double that represents the end angle of the arc.
`EndPoint`	Returns an array of doubles that represents the endpoint of the arc.
`Normal`	Specifies the normal vector of the line. The normal vector is an array of three doubles that defines the positive Z-axis for the arc.

TABLE 4.7: Properties related to an AcadArc object *(CONTINUED)*

PROPERTY	**DESCRIPTION**
`Radius`	Specifies the radius of the arc; the value is a double.
`StartAngle`	Specifies a double that represents the start angle of the arc.
`StartPoint`	Returns an array of doubles that represents the start point of the arc.
`Thickness`	Specifies the thickness assigned to the circle; the value must be numeric. The default is 0; anything greater than 0 results in the creation of a curved 3D object.
`TotalAngle`	Returns a double that represents the angle of the arc: the end angle minus the start angle.

Working with Polylines

Polylines are objects that can be made up of multiple straight and/or curved segments. Although lines and arcs drawn end to end can look like a polyline, polylines are more efficient to work with. Because a polyline is a single object made up of multiple segments, it is easier to modify. For example, all segments of a polyline are offset together instead of individually. If you were to offset lines and arcs that were drawn end to end, the resulting objects wouldn't be drawn end to end like the original objects (see Figure 4.5).

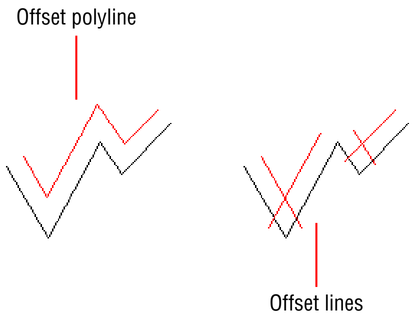


FIGURE 4.5
Offset polylines and lines

There are two types of polylines that you can create and modify:

Polyline Legacy polylines were available in AutoCAD R13 and earlier releases, and they are still available in AutoCAD R14 and later releases. This type of polyline object supports 3D coordinate values, but it uses more memory and increase the size of a drawing file.

Lightweight Polyline Lightweight polylines, or LWPolylines, were first introduced in AutoCAD R14. They are more efficient in memory and require less space in a drawing file. Lightweight polylines support only 2D coordinate values.

NOTE Autodesk recommends using lightweight polylines in a drawing instead of legacy polylines when possible.

Legacy polylines are represented by the `AcadPolyline` object type and can be added to a drawing with the `AddPolyline` function. LWPolylines are represented by the `AcadLWPolyline` object type and can be added to a drawing with the `AddLightWeightPolyline` function. The `AddPolyline` and `AddLightWeightPolyline` functions both require you to specify a list of vertices.

A vertices list is defined using an array of doubles. The number of elements in the array varies by the type of polyline you want to create or modify. To create an `AcadPolyline` object, you define an array of doubles in multiples of three, whereas an array of doubles must be in multiples of two to create an `AcadLWPolyline` object. For example, an `AcadPolyline` object with three vertices would require an array with nine elements (three elements × three vertices). For an LWPolyline, each vertex requires two elements in an array, so an `AcadLWPolyline` object with three vertices would require a vertices list with six elements (two elements × three vertices).

The following shows an example of a six-element array that defines three 2D points representing the corners of a triangle:

```
' Defines a six element array of doubles
Dim dVecList(5) As Double

' Sets the first corner
dVecList(0) = 0#: dVecList(1) = 0#

' Sets the second corner
dVecList(2) = 3#: dVecList(3) = 0#

' Sets the third corner
dVecList(4) = 1.5: dVecList(5) = 2.5981
```

The following shows the syntax of the `AddLightWeightPolyline` and `AddPolyline` functions:

```
retVal = object.AddLightWeightPolyline(vecList)
retVal = object.AddPolyline(vecList)
```

The arguments are as follows:

retVal The *retVal* argument represents the new `AcadLWPolyline` or `AcadPolyline` object returned by the `AddLightWeightPolyline` or `AddPolyline` function.

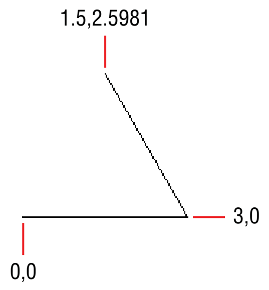
object The *object* argument represents the `AcadModelSpace` collection object.

vecList The *vecList* argument is an array of doubles that defines the vectors of the polyline. For the `AddLightWeightPolyline` function, the array must contain an even number of elements since each vertex is defined by two elements. Specify an array in three-element increments when using the `AddPolyline` function since each vertex is defined by three elements.

The following code statements add a new lightweight polyline object to model space (see Figure 4.6):

```
' Adds a lightweight polyline
Dim oLWPoly As AcadLWPolyline
Set oLWPoly = ThisDrawing.ModelSpace.AddLightWeightPolyline(dVecList)
```

Figure 4.6
Polyline with three vertices



Using the `AcadLWPolyline` or `AcadPolyline` object returned by the `AddLightWeightPolyline` or `AddPolyline` function, you can obtain information about and modify the polyline's properties. In addition to the properties that the `AcadLWPolyline` or `AcadPolyline` object share in common with the `AcadEntity` object, you can use the properties listed in Table 4.8 when working with an `AcadLWPolyline` or `AcadPolyline` object.

Table 4.8: Properties related to an `AcadLWPolyline` or `AcadPolyline` object

Property	**Description**
`Area`	Returns a double that represents the calculated area of the polyline.
`Closed`	Specifies whether the polyline is open or closed. A value of `True` closes the polyline if the object contains more than two vertices.
`ConstantWidth`	Specifies the global width for all segments of the polyline.
`Coordinate`	Specifies the coordinate value of a specific vertex in the polyline.
`Coordinates`	Specifies the coordinate values for all vertices of the polyline.
`Elevation`	Specifies the elevation at which the polyline is drawn.
`Length`	Returns a double that represents the length of the polyline.
`LinetypeGeneration`	Specifies whether the linetype pattern assigned to the polyline is generated across the polyline as one continuous pattern, or whether the pattern begins and ends at each vertex. A value of `True` indicates that the linetype pattern should be generated across the polyline as one continuous pattern.
`Normal`	Specifies the normal vector of the polyline. The normal vector is an array of three doubles that defines the positive Z-axis for the polyline.
`Thickness`	Specifies the thickness assigned to the polyline; the value must be numeric. The default is 0; anything greater than 0 results in the creation of a 3D planar object.

In addition to the properties listed in Table 4.8, an `AcadLWPolyline` or `AcadPolyline` object contains methods that are specific to polylines. Table 4.9 lists the methods that are unique to polylines.

Table 4.9: Methods related to an `AcadLWPolyline` or `AcadPolyline` object

Method	Description
`AddVertex`	Adds a new 2D point at the specified vertex in the LWPolyline (supported by `AcadLWPolyline` objects only).
`AppendVertex`	Appends a new 3D point to the polyline (supported by `AcadPolyline` objects only).
`Explode`	Explodes the polyline and returns an array of the objects added to the drawing as a result of exploding the polyline.
`GetBulge`	Gets the bulge–curve–value at the specified vertex. The bulge is a value of the double data type.
`GetWidth`	Gets the width of the segment at the specified vertex. The width is a value of the double data type.
`SetBulge`	Sets the bulge–curve–value at the specified vertex.
`SetWidth`	Sets the width of the segment at the specified vertex.

TIP You use the `AddVertex` or `AppendVertex` method to add a new vertex to a polyline, but it isn't exactly obvious how you might remove a vertex. To remove a vertex from a polyline, use the `Coordinates` property to get the vertices of the polyline. Then create a new vertices list of the points you want to keep and assign the new vertices list to the `Coordinates` property.

Real World Scenario

Defining Parallel Line Segments

Polylines make it easy to create parallel straight and curved segments. Parallel line segments can also be created with an `AcadMLine` object. Multilines (or mlines) allow you to create multiple parallel line segments and each parallel line can have a different format. The formatting of an mline is inherited from an mline style. You can use mlines to draw the walls of a building and even the foundation in plan view where the outermost lines might represent the footing and the inner lines represent the actual foundation walls. Although mlines have their use, they aren't common in drawings because they can be hard to edit. Mlines are added to a drawing with the `AddMLine` function. You can learn more about the `AddMLine` function and `AcadMLine` object in the AutoCAD Help system.

Getting an Object in the Drawing

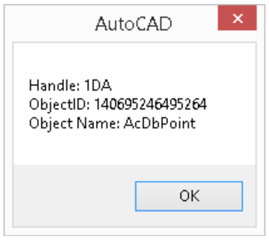
Modifying an object after it has been added to a drawing is fairly straightforward; you use the properties and methods of the object that is returned by one of the `Add*` functions described in the previous sections. If you want to modify an existing object in a drawing, you must locate it in the AcadModelSpace or AcadPaperSpace collection object or a block definition represented by an AcadBlock object. I explain how to work with block definitions in Chapter 7 and with paper space in Chapter 8.

The Item method and a For statement are the most common ways to access an object in the AcadModelSpace collection object. I explained how to use the Item method and For statement in Chapter 2, "Understanding Visual Basic for Applications." Use the Item method when you want to access a specific object in model space based on its index value; the first object in model space has an index of 0. Here are example code statements that get the handle and object type name of the first object in model space:

```
Dim oEnt As AcadEntity
Set oEnt = ThisDrawing.ModelSpace(0)

MsgBox "Handle: " & oEnt.Handle & vbLf & _
       "ObjectID: " & CStr(oEnt.ObjectID) & vbLf & _
       "Object Name: " & oEnt.ObjectName
```

The values displayed in the message box by the example code will vary from drawing to drawing. Figure 4.7 shows an example of a message box with the values from a first object in model space; the values reflected are of a point object.

FIGURE 4.7 Message box containing the handle and object ID of an object



```
Handle: 1DA
ObjectID: 140695246495264
Object Name: AcDbPoint
```

A For statement is the most efficient way to step through all the objects in model space or any other collection object you might need to work with. The following code statements step through model space and return the center point and radius of each circle object:

```
Dim oEnt As AcadEntity
Dim oCircle As AcadCircle

' Displays a general message
ThisDrawing.Utility.Prompt vbLf & "Circles in model space"

' Steps through model space
For Each oEnt In ThisDrawing.ModelSpace
  ' Checks to see if the object is a circle
  If TypeOf oEnt Is AcadCircle Then
    Set oCircle = oEnt

    ' outputs the center point and radius of the circle
    ThisDrawing.Utility.Prompt vbLf & "Center point: " & _
                               CStr(oCircle.Center(0)) & "," & _
                               CStr(oCircle.Center(1)) & "," & _
```

```
                                 CStr(oCircle.Center(2)) & _
                                 vbLf & "Radius: " & _
                                 CStr(oCircle.Radius)
  End If
Next oEnt

ThisDrawing.Utility.Prompt vbLf
```

Here is an example of the output created by the previous code statements:

```
Circles in model space
Center point: 5,2,0
Radius: 2.5
Center point: 6,2.5,0
Radius: 0.125
Center point: 3,1,0
Radius: 5
```

NOTE The `Item` method and `For` statement are useful when stepping through all objects in model space or paper space, but they don't allow the user to interactively select an object. I discuss how to prompt a user for objects in Chapter 5.

Modifying Objects

Adding new objects is critical to completing a design, but more time is often spent by a drafter or engineer modifying existing objects than adding new objects. The AutoCAD Object library contains methods that are similar to many of the standard AutoCAD commands used to modify objects. The modifying methods of the AutoCAD Object library can be used to erase, move, scale, mirror, and rotate objects, among other tasks. I explain how to erase, copy, move, and rotate graphical objects in the following sections using the methods that are inherited by the `AcadEntity` object. I discuss how to scale, mirror, offset, array, and control the visibility of objects in Bonus Chapter 1 on the companion website at `www.sybex.com/go/autocadcustomization`.

When a change is made to an object, I recommend that you update the display of that object. The AutoCAD command `regen` is used to regenerate the display of all objects in the current space, but with the AutoCAD Object library you can update the display of a single graphical object or all objects in a drawing. Use the `Update` method to update the display of a single graphical object. The `Update` method doesn't accept any argument values.

If you want to update the display of all objects in a drawing, use the `Regen` method of the `AcadDocument` or `ThisDrawing` object. The `Regen` method expects a constant value from the `AcRegenType` enumerator. You use the `acActiveViewport` constant to regenerate the objects in the current viewport or the `acAllViewports` constant to regenerate all objects in a drawing.

The following code statements show how to update the display of the first object in model space and all objects in the current viewport:

```
' Update the first object in model space
ThisDrawing.ModelSpace(0).Update

' Update all objects in the current viewport
Thisdrawing.Regen acActiveViewport
```

Deleting Objects

All graphical and most nongraphical objects can be removed from a drawing when they are no longer needed. The only objects that can't be removed are any nongraphical objects that are referenced by a graphical object, such as a text or dimension style, and nongraphical objects that represent symbol tables, such as the Layers and Blocks symbol tables. The `Delete` method is used to remove—or erase—an object. The method doesn't accept any arguments. If an object can't be removed, an error is generated. I explain how to trap and handle errors in Chapter 13, "Handling Errors and Deploying VBA Projects."

The following code statement removes the first object in model space:

```
' Removes the first object in model space
ThisDrawing.ModelSpace(0).Delete
```

Removing All Unreferenced Nongraphical Objects

Although the `Delete` method can be used to remove a nongraphical object that isn't currently being referenced by a graphical object in a drawing, the `PurgeAll` method can be used to purge all unreferenced nongraphical objects. The `PurgeAll` method is a member of the `AcadDocument` or `ThisDrawing` object, and it doesn't accept any argument values.

Here's an example of the `PurgeAll` method:

```
ThisDrawing.PurgeAll
```

Copying and Moving Objects

The `copy` and `move` commands are used to duplicate and relocate objects in a drawing. When working with the AutoCAD Object library, use the `Copy` function to duplicate an object. The `Copy` function doesn't accept any arguments, but it does return a reference to the new duplicate object. The `Move` method can be used to relocate an object. It expects two arrays of three doubles that define the base and destination points to control the distance and angle at which the object should be moved.

The following code statements draw a circle, duplicate the circle, and then move the duplicated circle 5 units along the X-axis in the positive direction:

```
' Defines the center point for the circle
Dim dCenPt(2) As Double
dCenPt(0) = 5: dCenPt(1) = 5: dCenPt(2) = 0

' Adds a new circle to model space
Dim oCirc As AcadCircle
Set oCirc = ThisDrawing.ModelSpace.AddCircle(dCenPt, 2)

' Creates a copy of the circle
Dim oCircCopy As AcadCircle
Set oCircCopy = oCirc.Copy
```

```
' Moves the circle 5 units along the X axis
Dim dToPt(2) As Double
dToPt(0) = oCircCopy.Center(0) + 5
dToPt(1) = oCircCopy.Center(1)
dToPt(2) = oCircCopy.Center(2)

oCircCopy.Move dCenPt, dToPt
```

Rotating Objects

The angle and orientation of an object can be changed by rotating the object around a base point or axis. Rotating an object around a base point is performed with the `Rotate` method, whereas rotating an object around an axis is performed with the `Rotate3D` method. I discuss the `Rotate3D` method in Bonus Chapter 2 on the companion website. The base point you pass to the `Rotate` method must be defined as an array of three doubles. The angle in which the object is rotated must be expressed in radians.

The following code statements draw a line from 5,5 to 7,9 and then create a copy of the line. The new line object that is copied is then rotated 90 degrees to a value of 1.570796325 radians (see Figure 4.8):

```
' Defines the start and endpoints of the line
Dim dStartPt(2) As Double, dEndPt(2) As Double
dStartPt(0) = 5: dStartPt(1) = 5: dStartPt(2) = 0
dEndPt(0) = 7: dEndPt(1) = 9: dEndPt(2) = 0

' Adds a new line to model space
Dim oLine As AcadLine
Set oLine = ThisDrawing.ModelSpace.AddLine(dStartPt, dEndPt)

' Copies the line
Dim oLineCopy As AcadLine
Set oLineCopy = oLine.Copy

' Rotates the copied line by 1.570796325 radians
oLineCopy.Rotate dStartPt, 1.570796325
```

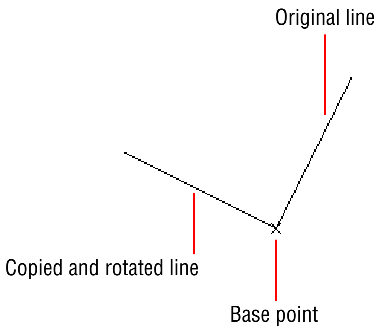


FIGURE 4.8 Rotated line object around a base point

The angular measurement of radians isn't as frequently used as degrees, but all angular values in a drawing are stored as radians; this is why the `Rotate` method expects radians. Radians are also expected or returned by most methods or properties in the AutoCAD Object library. Listing 4.1 shows two custom functions that can be used to convert degrees to radians and radians to degrees.

LISTING 4.1: Converting angular measurements

```
Const PI As Double = 3.14159265

Private Function Degrees2Radians(dDegrees As Double)
  Degrees2Radians = dDegrees * PI / 180
End Function

Private Function Radians2Degrees(dRadians As Double)
  Radians2Degrees = dRadians * 180 / PI
End Function
```

Here are a few examples of using the custom functions in Listing 4.1:

```
Dim dAngle As Double

' Converts 1.570796325 radians to 90 degrees
dAngle = Radians2Degrees(PI / 2)

' Converts 180 degrees to 3.14159265 radians
dAngle = Degrees2Radians(180)
```

Changing Object Properties

All graphical objects are derived from the `AcadEntity` object—that is, all graphical objects inherit the properties and methods of the `AcadEntity` object. For example, even though the `AcadLine` object represents a single line segment and the `AcadCircle` object represents a circle, they share the properties named `Layer` and `Linetype`, among many others.

The properties that all graphical objects have in common are known as the *general properties* of an object. In the AutoCAD user interface, an object's general properties can be modified from the Properties panel on the ribbon or the Properties palette (displayed with the `properties` command). The general properties shared by all graphical objects were listed in Table 4.2.

The following code statements assign the layer named TitleBlk to the first object in model space and override the color of the layer by directly assigning the color 3 (green) to the object:

```
' Assigns the TitleBlk layer to the first object in model space
ThisDrawing.ModelSpace(0).Layer = "TitleBlk"

' Assigns the ACI color Green to the first object in model space
```

```
Dim oClr As AcadAcCmColor
Set oClr = ThisDrawing.ModelSpace(0).TrueColor
oClr.ColorMethod = acColorMethodByACI
oClr.ColorIndex = acGreen
ThisDrawing.ModelSpace(0).TrueColor = oClr
```

I explain how to work with and manage layers and linetypes in Bonus Chapter 1 on the companion website. In addition to working with layers and linetypes, I explain how to work with true and color book colors, along with assigning a plot style and transparency to a layer or object.

Exercise: Creating, Querying, and Modifying Objects

In this section, you will create two new projects that create, query, and modify objects. One project will define a macro that allows you to draw a mounting plate with 2D objects, and the second project will use a similar set of logic to create a 3D model of a mounting plate. Along with the two projects, you will create a utility class that contains common functions that can be used across both projects and even in other projects later in this book.

The key concepts I cover in this exercise are as follows:

Creating and Modifying Graphical Objects Graphical objects are the backbone of any design; they are used to communicate what the building or product should look like when built or manufactured. When you want to add or modify graphical objects, you must decide whether to work with model space or paper space, or even a custom block definition.

Working with Layers All graphical objects are placed on a layer. Layers are used to organize graphical objects and control many of the general properties that all graphical objects have in common.

Creating and Using a Custom Class The VBA programming language supports the ability to create a custom class. Custom classes can be used to organize functions and manage global variables. A custom class when created in a project can be exported and used across many projects.

NOTE The steps in this exercise don't rely on the completion of an earlier exercise in this book. Later exercises in this book will rely on the completion of this exercise, though. If you don't complete this exercise, you can obtain the completed files from `www.sybex.com/go/autocadcustomization`.

Creating the *DrawPlate* Project

The following steps explain how to create a project named `DrawPlate` and to save it to a file named `drawplate.dvb`:

1. On the ribbon, click Manage tab ➢ Applications panel title bar and then click VBA Manager (or at the Command prompt, type **`vbaman`** and press Enter).
2. When the VBA Manager opens, click New.

The new project is added to the list with a default name of ACADProject and a location of Global1, Global2, and so on based on how many projects have been created in the current AutoCAD session.

3. Select the new project from the Projects list and click Save As.
4. When the Save As dialog box opens, browse to the MyCustomFiles folder within the Documents (or My Documents) folder, or the location you are using to store custom program files.
5. In the File Name text box, type **drawplate** and click Save.
6. In the VBA Manager dialog box, click Visual Basic Editor.

The next steps explain how to change the project name from ACADProject to DrawPlate:

1. When the VBA Editor opens, select the project node labeled ACADProject from the Project Explorer.
2. In the Properties window, select the field named (Name) and double-click in the text box adjacent to the field.
3. In the text box, type **DrawPlate** and press Enter.
4. On the menu bar, click File ➪ Save.

Creating the Utilities Class

Separating the custom functions you create into logical groupings can make debugging code statements easier and allow you to reuse code in other products. Custom classes are one way of sharing functions and protecting global variables from the functions of your main project. One of the benefits of using a custom class over just a code module is that you gain the advantage of type-ahead in the Visual Basic Editor, which reduces the amount of text you need to type.

In these steps, you add a new custom class module named clsUtilities to the DrawPlate project:

1. On the menu bar, click Insert ➪ Class Module.
2. In the Project Explorer, select the new module named Class1.
3. In the Properties window, change the current value of the (Name) property to **clsUtilities**.
4. On the menu bar, click File ➪ Save.

The clsUtilities class module will contain functions that define common and reusable functions for use with the main function of the DrawPlate project along with other projects later in this book. The following steps add two functions to the clsUtilities class that are used to work with system variables.

Working with one system variable at a time isn't always efficient when you need to set or restore the values of multiple system variables. You will define two functions named GetSysvars and SetSysvars. The GetSysvars function will return an array of the current values for multiple system variables, and the SetSysvars function will be used to set the values of multiple system variables.

The following steps explain how to add the `GetSysvars` and `SetSysvars` functions:

1. In the Project Explorer, double-click the `clsUtilities` component.
2. In the text editor area of the `clsUtilities` component, type the following. (The comments are here for your information and don't need to be typed.)

```
' GetSysvars function returns an array of the current values
' for each system variable in the array it is passed.
Public Function GetSysvars(sysvarNames) As Variant
  Dim nIdxTotal As Integer
  nIdxTotal = UBound(sysvarNames)

  Dim aVals() As Variant
  ReDim aVals(UBound(sysvarNames) - LBound(sysvarNames))

  Dim nCnt As Integer

  For nCnt = LBound(sysvarNames) To UBound(sysvarNames)
    aVals(nCnt) = ThisDrawing.GetVariable(sysvarNames(nCnt))
  Next

  GetSysvars = aVals
End Function

' SetSysvars function sets the values of the system variables
' in the array that the function is passed.
' Function expects two arrays.
Public Sub SetSysvars(sysvarNames, sysvarValues)
  Dim nCnt As Integer

  For nCnt = LBound(sysvarNames) To UBound(sysvarNames)
    ThisDrawing.SetVariable sysvarNames(nCnt), sysvarValues(nCnt)
  Next
End Sub
```

3. Click File ⇨ Save.

New graphical objects must be added to model space, paper space, or a block definition. In most situations, you want to add new objects to the current layout. You can create custom functions to combine multiple code statements and reduce the amount of code that needs to be otherwise entered. The following steps add three functions to the `clsUtilities` class that can be used to create a closed polyline and circle in the current layout, and a new layer.

1. In the text editor area of the `clsUtilities` component, type the following. (The comments are here for your information and don't need to be typed.)

```
' CreateRectangle function draws a closed LWPolyline object.
' Function expects an array that represents four points,
' but can accept more points.
```

```
Public Function CreateRectangle(ptList As Variant) As AcadLWPolyline
  Set CreateRectangle = ThisDrawing.ActiveLayout.Block. _
                      AddLightWeightPolyline(ptList)
  CreateRectangle.Closed = True
End Function

' CreateCircle function draws a Circle object.
' Function expects a center point and radius.
Public Function CreateCircle(cenPt As Variant, circRadius) As AcadCircle
  Set CreateCircle = ThisDrawing.ActiveLayout.Block. _
                   AddCircle(cenPt, circRadius)
End Function

' CreateLayer function creates a layer and returns an AcadLayer object.
' Function expects a layer name and color.
Public Function CreateLayer(sName As String, _
                            nClr As ACAD_COLOR) As AcadLayer

  On Error Resume Next

  ' Try to get the layer first and return it if it exists
  Set CreateLayer = ThisDrawing.Layers(sName)

  ' If layer doesn't exist create it
  If Err Then
    Err.Clear

    Set CreateLayer = ThisDrawing.Layers.Add(sName)
    CreateLayer.color = nClr
  End If
End Function
```

2. Click File ➢ Save.

Defining the *CLI_DrawPlate* Function

The main function of the DrawPlate project draws a rectangular mounting plate with four bolt holes. The outside edge of the mounting plate is defined using a closed lightweight polyline that is drawn using the CreateRectangle function defined in the clsUtilities class. Each of the bolt holes is drawn using the CreateCircle function of the clsUtilities class. Since objects in a drawing are organized using layers, you will place the rectangle and circles on different layers; the layers will be added to the drawing with the CreateLayer function.

In these steps, you add a new custom module named basDrawPlate to the DrawPlate project:

1. On the menu bar, click Insert ➢ Module.
2. In the Project Explorer, select the new module named Module1.

3. In the Properties window, change the current value of the (Name) property to **basDrawPlate**.
4. On the menu bar, click File ➢ Save.

The following steps explain how to add the CLI_DrawPlate function, which is the macro users will use to create the mounting plate:

1. In the Project Explorer, double-click the basDrawPlate component.
2. In the text editor area of the basDrawPlate component, type the following:

```
Private myUtilities As New clsUtilities
```

   The clsUtilities.cls file code statement defines a global variable named myUtilities. The myUtilities variable is then assigned a new instance of the clsUtilities class that you defined earlier. When you want to reference a function defined in the clsUtilities class, you will use the myUtilities variable.

3. In the text editor area of the basDrawPlate component, press Enter and type the following. (The comments are here for your information and don't need to be typed.)

```
Public Sub CLI_DrawPlate()
  Dim oLyr As AcadLayer


  On Error Resume Next

  ' Store the current value of the system variables to be restored later
  Dim sysvarNames As Variant, sysvarVals As Variant
  sysvarNames = Array("nomutt", "clayer", "textstyle")

  sysvarVals = myUtilities.GetSysvars(sysvarNames)

  ' Set the current value of system variables
  myUtilities.SetSysvars sysvarNames, Array(0, "0", "STANDARD")

  ' Define the width and height for the plate
  Dim dWidth As Double, dHeight As Double
  dWidth = 5#
  dHeight = 2.75

  Dim basePt(2) As Double
  basePt(0) = 0: basePt(1) = 0: basePt(2) = 0

  ' Create the layer named Plate or set it current
  Set oLyr = myUtilities.CreateLayer("Plate", acBlue)
  ThisDrawing.ActiveLayer = oLyr

  ' Create the array that will hold the point list
  ' used to draw the outline of the plate
  Dim dPtList(7) As Double
  dPtList(0) = basePt(0): dPtList(1) = basePt(1)
  dPtList(2) = basePt(0) + dWidth: dPtList(3) = basePt(1)
  dPtList(4) = basePt(0) + dWidth: dPtList(5) = basePt(1) + dHeight
  dPtList(6) = basePt(0): dPtList(7) = basePt(1) + dHeight

  ' Draw the rectangle
  myUtilities.CreateRectangle dPtList

  ' Create the layer named Holes or set it current
  Set oLyr = myUtilities.CreateLayer("Holes", acRed)
  ThisDrawing.ActiveLayer = oLyr

  ' Define the center points of the circles
  Dim cenPt1(2) As Double, cenPt2(2) As Double
  Dim cenPt3(2) As Double, cenPt4(2) As Double
  cenPt1(0) = 0.5: cenPt1(1) = 0.5: cenPt1(2) = 0
  cenPt2(0) = 4.5: cenPt2(1) = 0.5: cenPt2(2) = 0
  cenPt3(0) = 0.5: cenPt3(1) = 2.25: cenPt3(2) = 0
  cenPt4(0) = 4.5: cenPt4(1) = 2.25: cenPt4(2) = 0

  ' Draw the four circles
  myUtilities.CreateCircle cenPt1, 0.1875
  myUtilities.CreateCircle cenPt2, 0.1875
  myUtilities.CreateCircle cenPt3, 0.1875
  myUtilities.CreateCircle cenPt4, 0.1875

  ' Restore the saved system variable values
  myUtilities.SetSysvars sysvarNames, sysvarVals
End Sub
```

4. Click File ➢ Save.

Running the *CLI_DrawPlate* Function

Now that the `CLI_DrawPlate` function has been defined with the necessary code statements to draw the mounting plate, it can be executed from the AutoCAD user interface. In these steps, you run the `CLI_DrawPlate` function from the Macros dialog box.

1. Switch to AutoCAD by clicking on its icon in the Windows taskbar or by clicking View ➢ AutoCAD from the menu bar in the Visual Basic Editor.
2. In AutoCAD, at the Command prompt, type **`vbarun`** and press Enter.
3. When the Macros dialog box opens, select the `DrawPlate.dvb!basDrawPlate.CLI_DrawPlate` macro from the list and click Run.

   The new mounting plate is drawn, as shown in Figure 4.9. The mounting plate measures 5×2.75, which was defined in the `CLI_DrawPlate` function. In Chapter 5, you will learn to accept user input to control the size of the mounting plate that should be drawn.

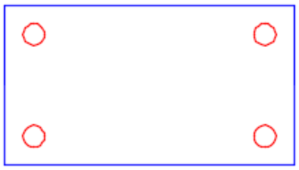


FIGURE 4.9
New mounting plate

If you don't see the mounting plate, use the `zoom` command and zoom to the extents of the drawing area.

Exporting the Utilities Class

The functions in the `clsUtilities` class can be used in other projects. By exporting the class module out of the `DrawPlate` project, you can then import it into other projects. Class modules aren't the only components that can be exported from a project; you can export code modules and User Forms that define dialog boxes in a project as well.

The following steps explain how to export the `clsUtilities` class module from the `drawplate.dvb` file:

1. In the VBA Editor, in the Project Explorer, right-click the `clsUtilities` component and choose Export File.
2. When the Export File dialog box opens, browse to the `MyCustomFiles` folder.
3. Keep the default filename of `clsUtilities.cls` and click Save.

   The `clsUtilities.cls` file is exported from the `DrawPlate` project.

Chapter 5

Interacting with the User and Controlling the Current View

Static values in a custom program are helpful in executing a set of code statements consistently each time the program is run. However, using static values only prevents the user from providing input during execution. Your users might need to specify the location of the corner of a mounting plate, an insertion point for a block reference, or which objects to modify. The AutoCAD® Object library provides a variety of functions that allow you to request input at the Command prompt or with controls in a user form. I cover working with user forms in Chapter 11, "Creating and Displaying User Forms."

Some values obtained from a user can be directly assigned to an object without any changes, whereas other values might need to be manipulated first. The AutoCAD Object library contains functions that can be used to manipulate coordinate and angular values. I discussed converting values from one data type to another in Chapter 2, "Understanding Visual Basic for Applications."

Getting input from a user will be helpful in creating dynamic and flexible programs, but so will the manipulation of the current view. The programs you write can pan and change the zoom factor or area that is visible in the drawing window. In addition to panning and zooming, you can work with named views, tiled viewports, and visual styles. In this chapter, I explain how to request input from the user at the Command prompt, calculate geometric values, and manipulate the current view in model space.

Interacting with the User

There are times when you will want users to provide a value instead of simply deciding which values a program should use each time it is executed. The AutoCAD Object library contains functions that can be used to request input from the user. The values returned by the user can then be validated using test conditions before the values are passed as arguments to a function. I explained how to use test conditions with comparison and logical grouping operators in Chapter 2.

In addition to getting input from the user, a custom program can provide textual feedback to the user, letting the user know the current state of a program or when an error occurs. Textual feedback can be provided at the Command prompt or in a message box. Whether getting input from the user or providing textual messages at the Command prompt, you will need to work with the `AcadUtility` object. The `AcadUtility` object can be accessed from the `Utility` property of an `AcadDocument` or `ThisDrawing` object.

The following code statements get the `AcadUtility` object of the `ThisDrawing` object:

```
Dim oUtil as AcadUtility
Set oUtil = ThisDrawing.Utility
```

Requesting Input at the Command Prompt

With the functions of the `AcadUtility` object, you can request input from the user at the Command prompt. Input requested can be any of the following:

- Integer or double (real) numeric value
- Distance or angular value
- String or keyword
- 2D or 3D point

Before requesting input from the user, you will want to define a *prompt*. A prompt is a short text message that provides the user with an idea of the input expected and whether any options are available. I discuss recommended etiquette for creating a prompt in the sidebar "Guidelines for Prompts."

TIP Use the `On Error Resume Next` statement before using the `Getxxx` functions mentioned in this section. An error is generated by most of the functions if the user presses Enter without providing a value or presses Esc. After a `Getxxx` function, be sure to check the value of the `Err` object. I explain more about error handling in Chapter 13, "Handling Errors and Deploying VBA Projects."

Getting Numeric Values

Numbers play an important role in creating and modifying objects in a drawing, whether it is the radius of a circle, part of a coordinate value, or the number of rows in a rectangular array. VBA supports two types of numbers: integers and doubles (or reals). Integers are whole numbers without a decimal value, and doubles are numbers that support a decimal value. You can use the `GetInteger` and `GetReal` functions to request a numeric value from the Command prompt. The number entered by the user is the value returned by the function, but if the user presses the spacebar or Enter without providing a value, an error is generated. When an incorrect value is provided, the function re-prompts the user to try again.

The following shows the syntax of the `GetInteger` and `GetReal` functions:

```
retVal = object.GetInteger([msg])
retVal = object.GetReal([msg])
```

Their arguments are as follows:

`retVal` The *`retVal`* argument represents the integer or double value returned.

`object` The *`object`* argument represents the `AcadUtility` object.

`msg` The *`msg`* argument is an optional string that defines the prompt message to display at the Command prompt. The *`msg`* argument is optional, but I recommend always providing one.

The following are examples of the GetInteger and GetReal functions, and the values that are returned:

```
nRetVal = oUtil.GetInteger(vblf & "Enter number of line segments: ")
oUtil.Prompt vbLf & "Value=" & CStr(nRetVal) & vbLf

Enter number of line segments: Type 3.5 and press Enter
Requires an integer value.
Enter number of line segments: Type 3 and press Enter
3

dRetVal = oUtil.GetReal(vblf & "Enter angle of rotation: ")
oUtil.Prompt vbLf & "Value=" & CStr(dRetVal) & vbLf

Enter number of line segments: Type 22.5 and press Enter
22.5
```

NOTE When the user is prompted for a double value with the GetReal function and enters a whole number, a double value is returned. For example, entering **1** results in 1.0 being returned.

Acquiring a Point Value

The GetPoint function allows the user to specify a point in the drawing area based on an optional base point. When an optional base point is provided, a rubber-band line is drawn from the base point to the current position of the cursor. Figure 5.1 shows the rubber-band line effect used when getting a point based on the optional base point. A variant containing an array of three doubles, representing a point, is returned by the GetPoint function if the user successfully specifies a point in the drawing area. If the user presses the spacebar or Enter without specifying a point, an error is generated.

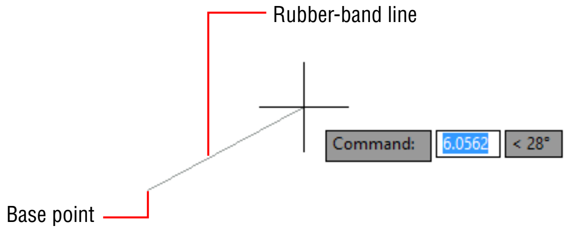


Figure 5.1 Rubber-band line effect used when specifying a point from a base point

In addition to the GetPoint function, the GetCorner function can be used to request a point. There are differences between the GetPoint and GetCorner functions:

- The GetCorner function requires a base point.
- The GetPoint function draws a rubber-band line from a base point to the cursor, whereas the GetCorner function draws a rectangle from the base point to the cursor, as shown in Figure 5.2.

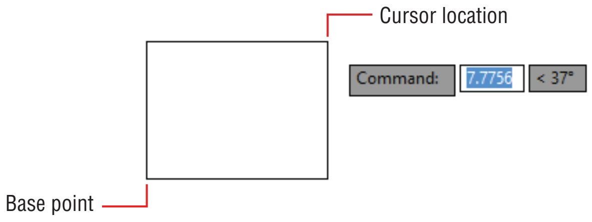


FIGURE 5.2 The rubber-band effect used when specifying the opposite corner with the `GetCorner` function

The following shows the syntax of the `GetPoint` and `GetCorner` functions:

```
retVal = object.GetPoint([basePoint], [msg])
retVal = object.GetCorner(basePoint, [msg])
```

Their arguments are as follows:

retVal The *retVal* argument represents the variant value returned by the function. This variant is an array of three doubles representing the point specified.

object The *object* argument represents the `AcadUtility` object.

basePoint The *basePoint* argument specifies the base point from which a rubber-band line or rectangle is drawn to the current position of the cursor. This argument value must be an array of three doubles and is optional for the `GetPoint` function.

msg The *msg* argument is an optional string that defines the prompt message to display at the Command prompt. The *msg* argument is optional, but I recommend always providing one.

The following are examples of the `GetPoint` and `GetCorner` functions:

```
Dim vPt As Variant
vPt = oUtil.GetPoint(, vbLf & "Specify first corner: ")

oUtil.Prompt vbLf & "X=" & CStr(vPt(0)) & _
                    " Y=" & CStr(vPt(1)) & _
                    " Z=" & CStr(vPt(2)) & vbLf

Dim vCornerPt As Variant
vCornerPt = oUtil.GetCorner(vPt, vbLf & "Specify opposite corner: ")

oUtil.Prompt vbLf & "X=" & CStr(vCornerPt(0)) & _
                    " Y=" & CStr(vCornerPt(1)) & _
                    " Z=" & CStr(vCornerPt(2)) & vbLf
```

Here is an example of values entered at the prompts displayed for the previous example code statements and the values returned:

```
Specify first corner: 0,0
X=0 Y=0 Z=0
Specify opposite corner: @5,5
X=5 Y=5 Z=0
```

Getting the Distance Between Points

Although the `GetReal` function can be used to request a value that might represent a distance or angular value, the AutoCAD Object library contains several functions that are better suited for acquiring distance or angular values. (I explain how to get angular values in the next section.) The `GetDistance` function can be used to get a distance between two points. The distance between the two points is returned as a double value. Optionally, the user can type a double value instead of specifying two points. If the user presses the spacebar or Enter without providing a value, an error is generated.

The following shows the syntax of the `GetDistance` function:

```
retVal = object.GetDistance([basePoint], [msg])
```

Its arguments are as follows:

`retVal` The *`retVal`* argument represents the double that is the result of the function calculating the distance between the two points specified.

`object` The *`object`* argument represents the `AcadUtility` object.

`basePoint` The *`basePoint`* argument is an optional argument that determines if a rubber-band line is drawn from the current position of the cursor to the coordinate value specified by the *`basePoint`* argument. This argument value must be an array of three doubles. If a base point isn't provided, the user must specify two points instead of one.

`msg` The *`msg`* argument is an optional string that defines the prompt message to display at the Command prompt. The *`msg`* argument is optional, but I recommend always providing one.

The following are examples of the `GetDistance` function and the values that are returned:

```
Dim dRetVal as Double
dRetVal = oUtil.GetDistance(, vblf & "Enter or specify a width: ")
oUtil.Prompt vbLf & "Distance=" & CStr(dRetVal) & vbLf

Enter or specify a width: Pick a point in the drawing area, enter a coordinate
value,
or enter a distance
Specify second point: If a point was specified, pick or enter a second point
Distance=6.25

Dim vPt As Variant, dRetVal As Double
vPt = oUtil.GetPoint(, vbLf & "Specify first point: ")
dRetVal = oUtil.GetDistance(vPt, vbLf & "Specify second point: ")
oUtil.Prompt vbLf & "Distance=" & CStr(dRetVal) & vbLf

Specify first point: Pick a point in the drawing area
Specify second point: Pick a point in the drawing area
Distance=7.0
```

TIP The `lunits` system variable affects the formatting of the linear distance that can be entered when the `GetDistance` function is executed. For example, when `lunits` is set to 2, the user can enter only decimal values and not values formatted in inches and feet. If `lunits` is set to 4, the user can enter either decimal or architectural formats for linear distances.

A double value can be converted to a string that reflects the formatting of a supported linear distance with the `RealToString` function. The `RealToString` function accepts a double value of the distance to format as a string, a constant value from the `AcUnits` enumerator to specify the linear format to apply, and an integer that indicates the precision in which the double should be formatted.

It is also possible to convert a string that is formatted as a supported linear distance to a double value with the `DistanceToReal` function. The `DistanceToReal` function accepts a string value and a constant value from the `AcUnits` enumerator that indicates the linear formatting of the string. For more information about the `RealToString` and `DistanceToReal` functions, see the AutoCAD Help system.

Getting the Angular Difference Between Points

The `GetAngle` and `GetOrientation` functions are used to obtain the angular difference between a vector defined by two points and the positive X-axis. The angular difference is expressed in radians, not decimal degrees or other angular measurement, and is returned as a double. If the user presses the spacebar or Enter without providing a value, an error is generated. The angular value returned by both functions is affected by the current value of the `angdir` system variable, which defines the direction in which positive angles are measured: counterclockwise or clockwise.

The `GetOrientation` function is also affected by the `angbase` system variables. The angular value returned by `GetOrientation` is calculated by adding the value specified by the user and that of the `angbase` system variable. For example, changing `angbase` to 45 and entering a value of 0 for the `GetOrientation` function returns a value of 0.785398, which is the current value of `angbase`. 0.785398 is the radians equivalent of 45 decimal degrees.

The following shows the syntax of the `GetAngle` and `GetOrientation` functions:

```
retVal = object.GetAngle([basePoint], [msg])
retVal = object.GetOrientation([basePoint], [msg])
```

The arguments of the two functions are the same as those of the `GetDistance` function explained in the previous section. The following are examples of the `GetAngle` function, and the values that are returned:

```
Dim dRetVal as Double
dRetVal = oUtil.GetAngle(, vblf & "Enter or specify an angle: ")
oUtil.Prompt vbLf & "Angle=" & CStr(dRetVal) & vbLf

Enter or specify an angle: Pick a point in the drawing area,
enter a coordinate value, or enter an angle
Specify second point: If a point was specified, pick or enter a second point
Angle= 0.785398
```

```
Dim vPt As Variant, dRetVal As Double
vPt = oUtil.GetPoint(, vbLf & "Specify first point: ")
dRetVal = oUtil.GetAngle(vPt, vbLf & "Specify second point: ")
oUtil.Prompt vbLf & "Angle=" & CStr(dRetVal) & vbLf
```

```
Specify first point: Pick a point in the drawing area
Specify second point: Pick a point in the drawing area
Angle=3.14159
```

Although AutoCAD uses and stores values in radians, users often think in decimal degrees. Listing 5.1 is a set of custom functions that can be used to convert radians to decimal degrees and decimal degrees to radians.

LISTING 5.1: Decimal degrees to radians and radians to decimal degrees

```
Const PI = 3.14159265358979

' Convert Radians to Decimal Degrees
' Usage: dRetval = rtd(0.785398)
Private Function rtd(dRadius As Double) As Double
  rtd = (dRadius / PI) * 180
End Function

' Convert Decimal Degrees to Radians
' Usage: dRetval = dtr(45.0)
Private Function dtr(dDegrees As Double) As Double
  dtr = (PI / 180) * dDegrees
End Function
```

A double that represents an angular value can be converted to a string that reflects the formatting of a supported angular measurement with the `AngleToString` function. The `AngleToString` function accepts a double value of the angle to format as a string, a constant value from the `AcAngleUnits` enumerator to specify the angular format to apply, and an integer that sets the precision in which the string should be formatted.

You can also convert a string that is formatted with a supported angular measurement to a double value with the `AngleToReal` function. The `AngleToReal` function accepts a string value and a constant value from the `AcAngleUnits` enumerator to specify the angular formatting of the string. For more information about the `AngleToString` and `AngleToReal` functions, see the AutoCAD Help system.

Guidelines for Prompts

Prompts explain the type of data that is being requested along with how that data might be used. Most of the commands you start in the AutoCAD program that don't open a dialog box will display a prompt that follows a common structure. I recommend structuring your prompts like the ones displayed by AutoCAD commands to make your prompts feel familiar to the user. Prompts commonly have two or more of the following elements:

Message The message is typically formatted as a statement that begins with a verb, such as specify or enter. I recommend using `Specify` when the user can pick one or more points in the drawing area to define a value or enter a value, and using `Enter` when the user can only type a value at the Command prompt. Messages can also be formatted as questions, but this is much less common. I recommend avoiding a conversational tone, which might use words such as please and thanks, in the message. Special character constants can also be used as part of a message; `vbLf` forces the text that follows it onto a new line, and `vbTab` and `"""` represent the Tab and quotation mark characters, respectively. The `vbBack` constant can be useful in removing the `Command:` text from the Command prompt; use 9 `vbBack` constants in a row to remove `Command:`. The exercise at the end of this chapter demonstrates how to create a constant and remove `Command:` from the Command prompt. For a full list of supported constants that can be used in strings, search on the "Miscellaneous Constants" topic in the Microsoft VBA Help system. In the VBA Editor, click Help ➢ Microsoft Visual Basic For Applications Help.

Option List The option list identifies which keywords are available in addition to the main data type of the `Getxxx` function. An opening ([) and a closing (]) square bracket denote the start and end of the option list. Each keyword in the option list should be separated by a forward slash (/), and the capitalization should match that of the keywords listing in the `InitializeUserInput` method that is evaluated just prior to the next `Getxxx` function. The option list should come after the main message of the prompt. I discuss the `InitializeUserInput` method in the "Initializing User Input and Keywords" section later in this chapter.

Default Value The default value that should be used if the user doesn't provide a value before pressing Enter is commonly displayed in a set of angle brackets (<>). The `Getxxx` function doesn't automatically return the value in the angle brackets if Enter is pressed before a value is provided. You must check for an error and return the desired default value. I demonstrate how to implement a prompt with a default value in the exercise at the end of this chapter.

Colon A colon should be the last character in a prompt, followed by a space to provide some separation between the prompt and value entered.

The following is the recommended structure of a prompt:

```
Message [Option list] <Default value>:
```

The following are examples of different prompts that follow my recommendations:

```
"Specify next point: "
"Specify rotation or [Reference] <45.000>: "
"Enter a number or press Backspace to clear: "
"Enter color option [Blue/Green/Red] <Blue>: "
```

The following are examples of prompts that shouldn't be used:

```
"Next point: "
"Pick a color (blue green black):"
"Specify next point"
"Enter color option or <Blue> [Blue/Green/Red]: "
```

Prompting for String Values

String values are used to represent the prompts that should be displayed when requesting input, a block name, or path, and even the text to be added to an annotation object. You can use the `GetString` function to request a string value at the Command prompt and control whether spaces are allowed in the string returned. The entered string is returned by the function, but if the user presses Enter without providing a value, an empty string (`""`) is returned.

The following shows the syntax of the `GetString` function:

```
retVal = object.GetString(allow_spaces, [msg])
```

Its arguments are as follows:

`retVal` The *`retVal`* argument represents the string that is returned by the function.

`object` The *`object`* argument represents the `AcadUtility` object.

`allow_spaces` The *`allow_spaces`* argument determines whether the spacebar acts like the Enter key or if it allows the entering of a space character. By default, pressing the spacebar is the same as pressing Enter. Provide a value of `True` to allow the user to enter a space character, or use `False` to not allow spaces in the text entered. A conditional expression that evaluates to `True` or `False` can also be used.

`msg` The *`msg`* argument is an optional string that defines the prompt message to display at the Command prompt. The *`msg`* argument is optional, but I recommend always providing one.

The following is an example of the `GetString` function and the value that is returned:

```
Dim sRetVal As String
sRetVal = oUtil.GetString(True, vbLf & "Enter your name: ")
oUtil.Prompt vbLf & "Value=" & sRetVal & vbLf

Type your first and last (or family) name, then press Enter
"Lee Ambrosius"
```

Initializing User Input and Keywords

The behavior of the `Getxxx` functions can be modified with the `InitializeUserInput` method of the `AcadUtility` object. When you want to enable one or more of the alternate behaviors of a `Getxxx` function, you include the `InitializeUserInput` method before the `Getxxx` function. In addition to controlling the alternate behaviors of the `Getxxx` functions, `InitializeUserInput` can be used to set up keyword usage for a function.

The following shows the syntax of the `InitializeUserInput` method:

```
object.InitializeUserInput(flags, [keywords_list])
```

The *`flags`* argument represents a bit-coded value that controls the type of input a `Getxxx` function can accept. The *`flags`* argument can contain one or more of the bits described in Table 5.1. Additional bits are available and described in the AutoCAD Help system; search on the keywords "InitializeUserInput method."

Table 5.1: Bit codes available for the `InitializeUserInput` method

Bit code	Description
1	User is not allowed to press Enter without first providing a value. Not supported for use with the `GetString` function.
2	Zero can't be entered when requesting a numeric value.
4	A negative value can't be entered when requesting a numeric value.
32	Rubber-band lines and rectangular boxes are shown as dashed instead of the default setting as solid.
64	Coordinate input is restricted to 2D points.
128	Arbitrary input is allowed; text values can be entered when using any of the `Getxxx` functions.

The *`keywords_list`* argument represents the keywords that the next `Getxxx` function can support. The keywords must be placed in a string and each keyword separated by a space. The letters you want a user to be able to enter without typing the full keyword must be in uppercase, and I recommend that they be consecutive; all other letters in a keyword must be lowercase.

The *keywords_list* argument is optional. Examples of keyword lists are "Blue Green Red" and "Azul Verde Rojo_Blue Green Red". The second example represents a keyword list that supports both localized and global languages; here the localized language is Spanish and the global language is typically English.

The global language value is used when an underscore is placed in front of a letter combination at the Command prompt. For example, typing **A** for the Azul option when the Spanish-language version of your program is loaded would work just fine but would fail if the English version was loaded. Entering **_B** instead would work with either the Spanish or English version of the program.

When a user enters a value that represents a keyword, an error is generated. Use the On Error Resume Next statement to keep the VBA environment from displaying an error message. After the Getxxx function is executed, check the value of the Err object to determine if the user entered a keyword, pressed Enter without providing a value, or pressed Esc. If a keyword is entered, the name of the keyword can be obtained with the GetInput function. The GetInput function doesn't accept any arguments and returns a string that represents the keyword the user choose.

The following is an example of the InitializeUserInput method that forces the user to provide a numeric value or enter a keyword option of Diameter with the GetDistance function. The If statement is used to determine if an error occurred and, if so, which error. Was the error caused by entering the keyword or by pressing Esc? The GetInput function is used to return the keyword value.

```
On Error Resume Next

' Disables pressing Enter without first
' entering a number or Diameter keyword
oUtil.InitializeUserInput 1, "Diameter"

Dim vRetVal As Variant
vRetVal = oUtil.GetDistance(, vbLf & "Specify radius or [Diameter]: ")

' Check to see if the user entered a value or option
If Err.Number = -2145320928 Then
  oUtil.Prompt vbLf & "Option=" & oUtil.GetInput & vbLf
ElseIf Err.Number = -2147352567 Then
  oUtil.Prompt vbLf & "User pressed Esc" & vbLf
Else
  oUtil.Prompt vbLf & "Distance=" & CStr(vRetVal) & vbLf
End If
```

Here are examples of values entered at the prompt displayed for the previous example code statement and the values returned:

```
Specify radius or [Diameter]: Type D and press Enter
Option=Diameter

Specify radius or [Diameter]: Type 7.5 and press Enter
Distance=7.5
```

The following is an example of the InitializeUserInput method that restricts the user's input to positive and nonzero values:

```
On Error Resume Next

' Disables pressing Enter without first entering a number,
' and limits input to positive and nonzero values
oUtil.InitializeUserInput 7

Dim vRetVal As Variant
vRetVal = oUtil.GetInteger(vbLf & "Enter a number: ")

' Check to see if the user entered a value
If Not Err Then
  oUtil.Prompt vbLf & "Value=" & CStr(vRetVal) & vbLf
End If
```

Here are examples of values entered at the prompt displayed for the previous example code statement, and the values returned:

```
Enter a number: Type -1 and press Enter
Value must be positive and nonzero.
Enter a number: Type 4 and press Enter
4
```

In addition to using keywords with the Getxxx functions, you can use the GetKeyword function to prompt the user for just keyword values. The GetKeyword function accepts input only in the form of a keyword value unless arbitrary input is enabled with the 128 bit-code of the InitializeUserInput method; in that case, the function can accept any string input. The GetKeyword function can return only a string value—it can't return numbers or arrays representing coordinate values. The InitializeUserInput method must be used to set up the keywords that the GetKeyword function can accept.

NOTE All Getxxx functions except the GetString function support keywords.

The following shows the syntax of the GetKeyword function:

```
retVal = object.GetKeyword([msg])
```

The *msg* argument represents the textual message to display at the Command prompt. The *msg* argument is optional, but I recommend always providing one.

The following is an example of the GetKeyword function and the value that is returned:

```
On Error Resume Next

' Sets up the keywords for the GetKeyword function
oUtil.InitializeUserInput 0, "Color LTYpe LWeight LTScale"

Dim vRetVal As Variant
```

```
vRetVal = oUtil.GetKeyword( _
   vbLf & "Enter option [Color/LTYpe/LWeight/LTScale] <Color>: ")

' Check to see if the user specified an option
If Err.Number = -2145320928 Then
  oUtil.Prompt vbLf & "Option=" & oUtil.GetInput & vbLf
ElseIf Err.Number = -2147352567 Then
  oUtil.Prompt vbLf & "User pressed Esc" & vbLf
Else
  If vRetVal = "" Then
    oUtil.Prompt vbLf & "Enter pressed w/o an option" & vbLf
  Else
    oUtil.Prompt vbLf & "Value=" & vRetVal & vbLf
  End If
End If
```

Here are examples of values entered at the prompt displayed for the previous example code statement, and the values returned:

```
Enter option [Color/LTYpe/LWeight/LTScale] <Color>: Type C and press Enter
Option=Color

Enter option [Color/LTYpe/LWeight/LTScale] <Color>: Type L and press Enter
Ambiguous response, please clarify...
LTYpe or LWeight or LTScale? Type LW and press Enter
Option=LWeight
```

Providing Feedback to the User

Although a program can simply request information from users and go on its way, it is best to acknowledge users and provide them with some feedback. Now this doesn't mean you need to make small talk with the person on the other side of the screen; it also doesn't mean you should share your life story. Based on the tasks your program might perform, you may want to provide information to the user when a macro does one of the following:

Starts Consider displaying the default settings or options that your program will be using, similar to the informational text that is displayed before the first prompt when using the `fillet` or `style` command.

Executes When processing a large dataset or number of objects, consider displaying a counter that helps the user know that something is still happening.

Causes an Error If something happens internally in your program, you should let users know what went wrong so they can let you (the programmer) know or try to fix the problem themselves.

Completes In most cases, you don't need to display information when a macro is done executing. However, you might want to let the user know if the information from a set of objects was successfully extracted or how many objects were modified.

Displaying Messages at the Command Prompt

In the "Requesting Input at the Command Prompt" section earlier, you learned how to display a message when requesting input from the user with one of the `Getxxx` functions. Messages can also be displayed at the Command prompt with the `Prompt` method of the `AcadUtility` object.

The following shows the syntax of the `Prompt` method:

```
object.Prompt(msg)
```

The *msg* argument represents the textual message to display at the Command prompt. As part of the textual message, you can use the constant `vbLf` to force the message on a new line, `vbTab` to add a Tab character, and `"""` to represent a quotation mark character. The `vbBack` constant, which emulates a press of the Backspace key, can also be useful in removing the `Command:` text from the Command prompt, thereby giving you a completely clean Command prompt. Use nine `vbBack` constants in a row to remove `Command:`. For a full list of supported constants that can be used in strings, search on the "Miscellaneous Constants" topic in the Microsoft VBA Help system. In the VBA Editor, click Help ➢ Microsoft Visual Basic For Applications Help.

The following are examples of the `Prompt` method and the values that are returned:

```
Dim oUtil As AcadUtility
Set oUtil = ThisDrawing.Utility

oUtil.Prompt vbLf & "Current OSMODE value: " & _
             CStr(ThisDrawing.GetVariable("OSMODE")) & vbLf
Current OSMODE value: 4133

oUtil.Prompt vbLf & "Drawing Name: "
oUtil.Prompt CStr(ThisDrawing.GetVariable("DWGNAME")) & vbLf
Drawing Name: Drawing1.dwg
```

TIP I recommend adding a `vbLf` constant to the start and end of all messages displayed with the `Prompt` function. The `vbLf` constant ensures that your message is displayed on a new line and that the user is always returned to a clean Command prompt.

Displaying Messages in a Message Box

A message at the Command prompt is a common way of displaying information to the user when you don't want to interrupt the user's workflow. However, you can also display information in a message box (which the user must acknowledge before the program continues).

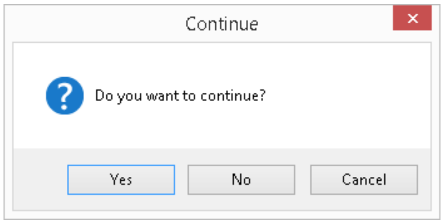
The `MsgBox` function of the VBA programming language can display a simple message box with a custom message and only an OK button. Message boxes can also contain a standard icon and button configuration that contains more than just an OK button. The `MsgBox` function returns a value that you can use to determine which button the user clicked. You can learn about the icons and button configurations that the `MsgBox` function supports in the Microsoft VBA Help system. In the VBA Editor, click Help ➢ Microsoft Visual Basic For Applications Help.

NOTE You can create a user form that displays additional information to the user that cannot be displayed with the `MsgBox` function. For example, you could display a picture or allow the user to click a link in the message displayed.

The following is an example of displaying a message with the `MsgBox` function and how to determine which button the user pressed. Figure 5.3 shows the first message box that is displayed when the example code is executed.

```
Dim nRetVal As Integer
nRetVal = MsgBox("Do you want to continue?", _
                 vbYesNoCancel + vbQuestion, "Continue")

Select Case nRetVal
  Case vbYes
    MsgBox "Yes was clicked"
  Case vbNo
    MsgBox "No was clicked"
  Case vbCancel
    MsgBox "Cancel was clicked"
End Select
```

FIGURE 5.3
Message displayed with the `MsgBox` function


Continue

Do you want to continue?

Selecting Objects

The AutoCAD Object library enables you to step through all of the objects in a drawing or allow the user to interactively select objects in the drawing area. I explained how to get an object from model space without user input in Chapter 4, "Creating and Modifying Drawing Objects." Using the selection techniques supported by the AutoCAD Object library, the user can be prompted to select a single object or a selection set can be created and the user allowed to select multiple objects.

Selecting an Individual Object

The user can be prompted to select a single object in the drawing area with the `GetEntity` method of the `AcadUtility` object. The `GetEntity` method returns two values: the selected object and the center point of the pick box when the object was selected. If no object is selected, an error is generated that must be handled to continue execution.

The following shows the syntax of the `GetEntity` method:

```
object.GetEntity(selectedObject, pickPoint, [msg])
```

Its arguments are as follows:

`object` The *`object`* argument represents the `AcadUtility` object.

`selectedObject` The *`selectedObject`* argument represents the variable that will be assigned the object that the user selected. The value assigned to the variable is of the Object data type.

pickPoint The *pickPoint* argument represents the variable that will be assigned the center point of the pick box when the object was selected. The value assigned to the variable is an array of three doubles.

msg The *msg* argument is an optional string that defines the prompt message to display at the Command prompt. The *msg* argument is optional, but I recommend always providing one.

NOTE The GetEntity method supports the use of keywords with the InitializeUserInput method. See the "Initializing User Input and Keywords" section earlier in this chapter for more information on using keywords.

The following is an example of the GetEntity method. The example prompts the user for an object and displays a message with the name of the object selected or a general message if no object was selected.

```
' Continue on error
On Error Resume Next

' Prompt the user for an object
Dim vObj As Object, vPt As Variant
ThisDrawing.Utility.GetEntity vObj, vPt, vbLf & "Select an object: "

' If an object was selected, display its object name
If Not vObj Is Nothing Then
  MsgBox "Type of object selected: " & vObj.ObjectName
Else
  MsgBox "No object selected."
End If
```

TIP If you want the user to select a specific type of object, you must use a selection method that supports selection filtering. I describe how to use selection filtering in the "Filtering Objects" section later in this chapter. The TypeOf statement can be used to validate the type of object selected. If the user selected the wrong type of object with the GetEntity method, you could use a While statement to continue prompting the user for an object until they select the correct type of object, select nothing, or press Enter.

The GetEntity method allows you to select an object as a whole, but not an entity inside of an object known as a *subentity*. The GetSubEntity method is similar to GetEntity except that GetSubEntity allows you to select an entire object or a subentity within an object such as an old-style polyline, dimension, or block. When the GetSubEntity method is used, it expects four arguments and can accept an optional prompt message. The four values that the GetSubEntity method returns are (in this order):

- The object that represents the subentity selected by the user; a value of the Object data type is returned
- The center point of where the pick box was positioned when the user selected the object; an array of three doubles

- A transformation matrix for the subentity; a multi-element array of doubles
- The object IDs of the subentities in the selected object or subentity; an array of long integers that represent the object IDs

For more information on the GetSubEntity method, see the AutoCAD Help system.

Working with Selection Sets

A grouping of selected objects in the AutoCAD drawing environment is known as a *selection set*. A selection set is a named container that holds references to objects in a drawing and exists only while a drawing remains open. From the AutoCAD user interface, a selection set is created when a user selects one or more objects at the `Select objects:` prompt.

In the AutoCAD Object library, a selection set is represented by an AcadSelectionSet object and all selection sets in a drawing are stored in the AcadSelectionSets collection object. The AcadSelectionSets collection object of a drawing is accessed using the SelectionSets property of an AcadDocument or ThisDrawing object.

In addition to the SelectionSets property, an AcadDocument or ThisDrawing object has two other properties that are related to selection sets: ActiveSelectionSet and PickfirstSelectionSet. Both properties are read-only. The ActiveSelectionSet property returns an AcadSelectionSet object that represents the active selection set of the drawing. The PickfirstSelectionSet property returns an AcadSelectionSet object that contains a selection set of the objects contained in the pickfirst selection. The pickfirst selection is made up of the objects that were selected before the execution of the VBA macro.

Managing Selection Sets

A selection set must be created or obtained before a user can be requested to select objects. The Add function of the AcadSelectionSets collection object creates a new selection set with the provided name and returns an AcadSelectionSet object. If you want to work with an existing selection set, use the Item method or a For statement on the AcadSelectionSets collection object to obtain an AcadSelectionSet object. When a selection set is no longer needed, use the Delete method of the AcadSelectionSet object to be removed.

NOTE When you try to create most nongraphical objects, such as a layer or linetype, with the Add function, the existing object with the same name is returned by the function. However, the same doesn't happen when creating a selection set. An error is generated by the Add function of the AcadSelectionSets collection object if you try to create a selection set with a name that already exists. When the error occurs, use the Item function to get the selection set. If you want to reuse an existing named selection set, clear the items that are contained in the set with the Clear method before adding new objects. By clearing the selection set, you can use a selection set with the same name across many different functions. This can be helpful for keeping your code simple and for cleaning up afterward. For example, you might create a selection set named SSBlocks that is used to keep a running record in memory of all blocks in a drawing.

The following example creates a new selection set or returns an existing selection if one already exists with the same name:

```
On Error Resume Next

' Create a new selection set named NewSS
```

```
Dim oSSet As AcadSelectionSet
Set oSSet = ThisDrawing.SelectionSets.Add("NewSS")

' Check for an error, if so get the existing selection set
If Err Then
  Err.Clear
  Set oSSet = ThisDrawing.SelectionSets.Item("NewSS")

  ' Reset the selection set
  oSSet.Clear
End If

' Perform selection tasks here and work with the objects selected

' When done with a selection set, it is best to remove it
oSSet.Delete
```

Adding and Removing Objects in a Selection Set

After a selection set has been created or an existing one obtained from the `AcadSelectionSets` collection object, you can work with the objects in the selection set or prompt the user to select objects in a drawing. The `AddItems` method of an `AcadSelectionSet` object allows you to add an array of objects to a selection set. Table 5.2 lists additional methods that can be used to manually add objects to a selection set by their placement in the drawing area.

Table 5.2: Object selection methods

Method	Description
`Select`	Adds objects to a selection set by selection mode: all objects, crossing window, last object added to a drawing, previous selected objects, or window. The method expects a selection mode that is a constant value from the `AcSelect` enumerator, and two optional arrays of three doubles that represent points in the drawing area.
`SelectAtPoint`	Adds an object to a selection set at a point in the drawing; the object selected is the topmost in the draw order at that point. The method expects an array of three doubles that represents a point in the drawing area.
`SelectByPolygon`	Adds objects to a selection set by selection mode: crossing polygon, fence, or window polygon. The method expects a selection mode that is a constant value from the `AcSelect` enumerator, and an array of doubles that represents multiple point values in the drawing area.

The `Select`, `SelectAtPoint`, and `SelectByPolygon` methods support object selection filtering with two optional arguments. I discuss object selection filtering in the next section.

For more information on adding objects to a selection set with the Select, SelectAtPoint, and SelectByPolygon methods, see the AutoCAD Help system.

Although adding objects manually to a selection set has its benefits, it is more common to prompt the user to select the objects that should be modified or queried. The SelectOnScreen method of an AcadSelectionSet object allows the user to interactively select objects in the drawing area using the standard selection methods. The SelectOnScreen method also supports object selection filtering.

The following shows the syntax of the SelectOnScreen method:

```
object.SelectOnScreen([filterType, filterData])
```

Its arguments are as follows:

object The *object* argument represents the AcadSelectionSet object.

filterType The *filterType* argument is an optional array of integers that represents the DXF code groups that you want to filter objects on.

filterData The *filterData* argument is an optional array of variants that represents the values that you want to filter objects on.

I explain how to define the arrays used to filter objects during selection in the "Filtering Objects" section later in this chapter.

Objects are typically only added to a selection set, but they can also be removed from a selection set. You might want to remove one or more objects from a selection set that don't meet certain criteria. One or more objects can be removed from a selection set with the RemoveItems method. The RemoveItems method is similar to the AddItems method, and it accepts an array of objects that should be removed from the selection set.

The following example prompts the user to select objects using the SelectOnScreen method, and adds the first and last objects in the drawing to the selection set named NewSS with the AddItems and Select methods. The last object is also removed to demonstrate the use of the RemoveItems method.

```
' Prompt the user for objects
ThisDrawing.Utility.Prompt vbLf & "Select objects to list: "
oSSet.SelectOnScreen

' Add the first object in model space to the selection set
Dim arObj(0) As AcadEntity
Set arObj(0) = ThisDrawing.ModelSpace(0)
oSSet.AddItems arObj

' Add the last object in the drawing to the selection set
oSSet.Select acSelectionSetLast

' Remove the last object in model space from
' the selection set
Set arObj(0) = ThisDrawing.ModelSpace( _
                   ThisDrawing.ModelSpace.Count - 1)
oSSet.RemoveItems arObj
```

Accessing Objects in a Selection Set

A selection set isn't any different than any other collection object. You can use the `Item` function of an `AcadSelectionSet` object to get a specific object in a selection set or a `For` statement to step through all the objects in a selection set. In addition to the `Item` function and `For` statement, you can use a `While` statement in combination with the `Item` function to step through all the objects in a selection set. The `Count` property of an `AcadSelectionSet` object lets you know how many objects are in a selection set; this value can be helpful when you are using the `Item` function or a `While` statement.

The following example steps through all the objects in a selection set and outputs the object name for each object to the command-line window:

```
' Step through each object in the selection set and output
' the name of each object with the Prompt method
Dim oEnt As AcadEntity

ThisDrawing.Utility.Prompt vbLf & "Objects in " & _
                           oSSet.Name & " selection set:"

For Each oEnt In oSSet
  ThisDrawing.Utility.Prompt vbLf & " " & oEnt.ObjectName
Next oEnt

' Return the user to a blank Command prompt
ThisDrawing.Utility.Prompt vbLf & ""
```

Here is an example of the output that might be displayed in the command-line window:

```
Objects in NewSS selection set:
 AcDbLine
 AcDbLine
 AcDbLine
 AcDbLine
 AcDbCircle
 AcDbArc
```

NOTE In most cases, you can step through a selection set and make changes to each object one at a time. However, there are three methods of an `AcadSelectionSet` object that can be used to manipulate all objects in a selection set. The `Erase` method can be used to remove all objects in a selection set from a drawing. The `Highlight` method can be used to highlight or unhighlight an object. The `Update` method is used to regenerate all the objects in the selection set.

Filtering Objects

The particular objects that are added to a selection set can be affected through the use of an optional selection filter. A selection filter can be used to limit the objects added to a selection set by type and property values. Filtering is defined by the use of two arrays with the same number of elements. Selection filters are supported by the `Select`, `SelectAtPoint`, `SelectByPolygon`, and `SelectOnScreen` methods of the `AcadSelectionSet` object. The two arrays are passed to the *`filterType`* and *`filterData`* arguments of the methods.

The first array of a selection filter contains only integer values that represent DXF group codes and the types of data that will be used to restrict object selection. The second array defines the actual values for the selection filter. The type of data to filter on can be a string, integer, or double, among other data types. When selection filter is used, objects are only selected when all conditions of the selection set are `True`.

For example, if you filter on circles that are placed on the Holes layer, only circles placed on the Holes layer will be added to the selection set. Lines and other objects placed on the layer named Holes will not be selected; circles on other layers will not be selected.

The following is an example of a selection filter that can be used to select the circles placed on the layer named Holes:

```
Dim arDXFCodes(1) As Integer, arValues(1) As Variant
' Object type
arDXFCodes(0) = 0: arValues(0) = "circle"

' Object layer
arDXFCodes(1) = 8: arValues(1) = "Holes"

' Prompt for and restrict the selection of objects with a selection filter
ThisDrawing.Utility.Prompt
            vbLf & "Select circles with a radius between 1 and 5: "
oSSet.SelectOnScreen arDXFCodes, arValues
```

In the previous example, the *`arDXFCodes`* variable contains an array of integer values that includes two DXF group codes. The DXF group code 0 represents an object's type, and the DXF group code 8 represents the name of the layer which an object is placed. For more information on DXF group codes, use the AutoCAD Help system and search on the keywords "dxf codes."

Object types and properties are not the only values that can be used to filter objects—a filter can also include logical grouping and comparison operators. Logical grouping and comparison operators allow for the selection of several object types, such as both text and `MText` objects, or allow for the selection of circles with a radius in a given range. Logical grouping and comparison operators are specified by string values with the DXF group code -4. For example, the following filter allows for the selection of circles with a radius in the range of 1 to 5:

```
Dim arDXFCodes(6) As Integer, arValues(6) As Variant
' Object type
arDXFCodes(0) = 0: arValues(0) = "circle"

' Start AND grouping
arDXFCodes(1) = -4: arValues(1) = "<and"

' Select circles with a radius between 1 and 5
arDXFCodes(2) = -4: arValues(2) = "<="
arDXFCodes(3) = 40: arValues(3) = 5#
arDXFCodes(4) = -4: arValues(4) = ">="
arDXFCodes(5) = 40: arValues(5) = 1#

' End AND grouping
arDXFCodes(6) = -4: arValues(6) = "and>"
```

Selection filters support four logical grouping operators: `and`, `or`, `not`, and `xor`. Each logical grouping operator used in a selection filter must have a beginning and ending operator. Beginning operators start with the character < and ending operators end with the character >. In addition to logical operators, you can use seven different comparison operators in a selection filter to evaluate the value of a property: `=` (equal to), `!=` (not equal to), `<` (less than), `>` (greater than), `<=` (less than or equal to), `>=` (greater than or equal to), and `*` (wildcard for string comparisons).

In addition to object types and property values, selection filters can filter on objects with attached extended data (Xdata). Xdata is used to add custom information to an object in a drawing. I discuss working with and selecting objects that have attached Xdata in Chapter 9, "Storing and Retrieving Custom Data."

Performing Geometric Calculations

The math functions of the VBA programming language are great for calculating numeric values based on other numeric values, but they aren't specifically designed to work with geometric values. With the AutoCAD Object library and standard math formulas, you can calculate the following:

- A new coordinate value based on a starting point, and at a specific angle and distance
- The distance value between two points
- An angular value from the X-axis

Calculating a Coordinate Value

When you create or modify an object, you frequently need to calculate a new point based on another point on or near an existing graphical object. Although you could prompt the user to specify a point you might need, that could lead to unnecessary steps in a workflow, so it is always best to calculate any and all points that you can with minimal input from the user.

The `PolarPoint` function returns a 2D or 3D point in the current UCS, based on an angle and distance from a point. The result of the `PolarPoint` function is similar to specifying a relative polar coordinate from the AutoCAD user interface.

The following shows the syntax of the `PolarPoint` function:

```
retVal = object.PolarPoint(point, angle, distance)
```

Its arguments are as follows:

`retVal` The *`retVal`* argument represents the variant value that contains the new coordinate point that was calculated as an array of two or three doubles.

`object` The *`object`* argument represents the `AcadUtility` object.

`point` The *`point`* argument represents the coordinate point in the drawing that you want to calculate the new point from. If a 2D point is specified, a 2D point is returned; specifying a 3D point results in a 3D point being returned.

`angle` The *`angle`* argument represents the angle, in radians, by which the new point should be separated from the coordinate point specified with the *`point`* argument.

distance The *distance* argument represents the distance at which the new point should be calculated from the *point* argument and along the angle specified by the *angle* argument.

The following is an example of the `PolarPoint` function:

```
Dim oUtil As AcadUtility
Set oUtil = ThisDrawing.Utility

Dim pt1(2) As Double
pt1(0) = 0: pt1(1) = 0: pt1(2) = 0

Dim vPt As Variant
vPt = oUtil.PolarPoint(pt1, 0.785398, 5#)

' Returns the calculated coordinate value
oUtil.Prompt vbLf & "X=" & CStr(vPt(0)) & _
                    " Y=" & CStr(vPt(1)) & _
                    " Z=" & CStr(vPt(2)) & vbLf

X=3.53553448362991 Y=3.53553332823547 Z=0
```

NOTE A coordinate value can be translated from one coordinate system to another with the `TranslateCoordinates` function. For example, you can convert a coordinate value from the World Coordinate System (WCS) to a User Coordinate System (UCS). Refer to the AutoCAD Help system for information on the `TranslateCoordinates` function.

Measuring the Distance Between Two Points

The AutoCAD Object library doesn't provide a function to calculate the distance between two points; instead you must rely on a geometric formula. The geometric formula is shown in Figure 5.4, and the VBA equivalent is as follows:

```
' Distance of 3D points
Sqr((X2 - X1) ^ 2 + (Y2 - Y1) ^ 2 + (Z2 - Z1) ^ 2)
```

FIGURE 5.4
Formula for calculating the distance between two points

$$\sqrt{(X2 - X1)^2 + (Y2 - Y1)^2 + (Z2 - Z1)^2}$$

If you need to calculate the distance between 2D points, the code statement in VBA might be as follows:

```
' Distance of 2D points
Sqr((X2 - X1) ^ 2 + (Y2 - Y1) ^ 2)
```

Listing 5.2 shows a custom function named `Distance` that can be used to calculate the distance between two points in the drawing area. The value returned is a double number.

Listing 5.2: Calculating the distance between two points

```
Private Function Distance(Point1 As Variant, Point2 As Variant) As Double
  ' Check to see if the points are 2D or 3D
  If UBound(Point1) - LBound(Point1) = 1 Then
    ' Distance of 2D points
    Distance = Sqr((Point1(0) - Point2(0)) ^ 2 + _
                   (Point1(1) - Point2(1)) ^ 2)
  Else
    ' Distance of 3D points
    Distance = Sqr((Point1(0) - Point2(0)) ^ 2 + _
                   (Point1(1) - Point2(1)) ^ 2 + _
                   (Point1(2) - Point2(2)) ^ 2)
  End If
End Function
```

Here is an example of using the custom `Distance` function from Listing 5.2:

```
Dim pt1(2) As Double, pt2(2) As Double
pt1(0) = 0: pt1(1) = 0: pt1(2) = 0
pt2(0) = 2: pt2(1) = 2: pt2(2) = 2

ThisDrawing.Utility.Prompt vbLf & _
   "Distance=" & CStr(Distance(pt1, pt2)) & vbLf

Distance=3.46410161513775
```

Calculating an Angle

When you draw or modify an object, you commonly need to know the angle at which an object should be drawn in relationship to the X-axis or other objects in a drawing. The `AngleFromXAxis` function accepts two arrays of three elements that define the line from which you want to calculate the angular value.

The following shows the syntax of the `AngleFromXAxis` function:

```
retVal = object.AngleFromXAxis(fromPoint, toPoint)
```

Its arguments are as follows:

retVal The *retVal* argument represents the angular value expressed in radians from the X-axis. The value is returned as a double.

object The *object* argument represents the `AcadUtility` object.

fromPoint The *fromPoint* argument is an array of three doubles that defines the start point of the line.

toPoint The *toPoint* argument is an array of three doubles that defines the end point of the line.

The following is an example of the `AngleFromXAxis` function:

```
Dim oUtil As AcadUtility
Set oUtil = ThisDrawing.Utility
```

```
Dim pt1(2) As Double, pt2(2) As Double
pt1(0) = 0: pt1(1) = 0: pt1(2) = 0
pt2(0) = 5: pt2(1) = 5: pt2(2) = 0

oUtil.Prompt vbLf & _
   "Angle=" & CStr(oUtil.AngleFromXAxis(pt1, pt2)) & vbLf

Angle=0.785398163397448
```

Changing the Current View

The view of model space can be adjusted to show a specific area of a drawing or the full extents of all objects in model space. You can adjust the area and magnification of the current view, and store a view that can later be restored in model space or applied to a floating viewport on a named layout. In addition to managing named views, you can divide model space into multiple viewports known as *tiled viewports*. Each tiled viewport can display a different view of model space and can be helpful when modeling in 3D. Visual styles can also be used to affect the way objects appear in a view or viewport.

Zooming and Panning the Current View

You can manipulate the current model space view by adjusting its scale and center in which objects should be displayed; this is typically known as *zooming* and *panning*. When you want to zoom or pan the current view, you will use the zoom-related methods of the `AcadApplication` object. You can get a reference to the `AcadApplication` object with the `Application` property of an `AcadDocument` or `ThisDrawing` object. Table 5.3 lists the different zoom-related methods that are available from the `AcadApplication` object.

Table 5.3: Zoom-related methods

Method	Description
`ZoomAll`	Fills the current view with the extents of the drawing limits or all graphical objects depending on which is largest.
`ZoomCenter`	Defines the center point of the current view, and increases or decreases the objects based on a specified magnification.
`ZoomExtents`	Fills the current view with the extents of all graphical objects.
`ZoomPickWindow`	Prompts the user for two points. The points define the area of the drawing and magnification in which the objects should be displayed.
`ZoomPrevious`	Restores the most recent view.

Table 5.3: Zoom-related methods *(continued)*

Method	Description
`ZoomScaled`	Increases or decreases the magnification of the current view; the center point of the view remains unchanged.
`ZoomWindow`	Defines the area of the drawing and magnification in which the objects should be displayed.

For specifics on the arguments that each of the methods listed in Table 5.3 expects, see the AutoCAD Help system. The following is an example of the `ZoomExtents` method:

```
' Set model space current
ThisDrawing.ActiveSpace = acModelSpace

Dim dPt1(2) As Double, dPt2(2) As Double
dPt1(0) = 1: dPt1(1) = 5: dPt1(2) = 0
dPt2(0) = 7: dPt2(1) = 3: dPt2(2) = 0

' Add a line to model space
ThisDrawing.ModelSpace.AddLine dPt1, dPt2

' Zoom to the extents of model space
ThisDrawing.Application.ZoomExtents
```

Although it might not seem obvious, you can use the `ZoomCenter` method to pan the current view. The following example gets the center point and magnification of the current view with the `viewctr` and `viewsize` system variables. Once the center point is obtained from the `viewctr` system variable, the point is adjusted to pan the current view 10 units to the right. The new center point and current magnification are passed to the `ZoomCenter` method to cause the current view to be panned and not zoomed.

```
' Get the current values of the viewctr
' and viewsize system variables
Dim vViewPt As Variant, dViewSize As Double
vViewPt = ThisDrawing.GetVariable("viewctr")
dViewSize = ThisDrawing.GetVariable("viewsize")

' Pan the viewport 10 drawing units to the right
vViewPt(0) = vViewPt(0) - 10
ThisDrawing.Application.ZoomCenter vViewPt, dViewSize
```

Zooming to an Object

There are times when you might want to zoom to a specific object in a drawing. Maybe you want to update the information in a table or dimension text. There is no `ZoomObject` method like there is an Object option for the `zoom` command. However, you can use a combination of the `ZoomWindow` and `ZoomScaled` methods to zoom to the extents of an object. The extents of an object can be obtained using the GetBoundingBox method that all graphical objects have in common. I discuss the GetBoundingBox method in Bonus Chapter 1, "Working with 2D Objects and Object Properties."

The following code statements zoom to the extents of the first object in model space:

```
' Gets the first object in model space
Dim oEnt As AcadEntity
Set oEnt = ThisDrawing.ModelSpace(0)

' Gets the extents of the objects' bounding box
Dim vExtMin As Variant, vExtMax As Variant
oEnt.GetBoundingBox vExtMin, vExtMax

' Zooms to the extents of the object
ThisDrawing.Application.ZoomWindow vExtMin, vExtMax

' Zooms out by 5%
ThisDrawing.Application.ZoomScaled 0.95, acZoomScaledRelative
```

Working with Model Space Viewports

The Model tab in the AutoCAD user interface is used to view and interact with the graphical objects of the model space block. By default, the objects in model space are displayed in a single tiled viewport named *Active. Tiled viewports aren't the same as the viewports displayed on a named layout tab; they do share some properties and methods in common, though. You use tiled viewports to view different areas or angles of the same drawing, whereas you use viewports on a named layout to control which model space objects are plotted, the angle in which objects are viewed, and at which scale. I discuss the viewports that can be added to a named layout in Chapter 8, "Outputting Drawings."

Each tiled viewport in model space can be split into two or more smaller viewports, but only one viewport can be active at a time. Unlike with the AutoCAD user interface, you can't join viewports back together again once they have been split; instead, you need to create a new configuration that reflects the desired layout and set it as current. Use the name of the active viewport to determine which viewports are part of the active viewport configuration.

You can access the active model space viewport with the ActiveViewport property of an AcadDocument or Thisdrawing object. The ActiveViewport property returns an AcadViewport object that represents a tiled viewport in model space. Not only is the ActiveViewport property used to get the active viewport, but it is also used to set a viewport configuration as active. Once you have the active viewport, you can modify the drafting aids that are viewport specific along with the current model view.

In addition to working with the active viewport, you can create and manage named viewport configurations with the AcadViewports collection object. You use the Add function of the AcadViewports collection object to create a new viewport configuration, and the Item function or a For statement to step through all the viewports of a viewport configuration. Named viewport configurations that are no longer needed can be removed using the DeleteConfiguration method on the `AcadViewports` collection object, not the `Delete` method of the `AcadViewport` object like other collection objects.

The following code statements split the current active viewport vertically into two viewports and then change some of the drafting aids related to the active viewport:

```
' Get the name of the current viewport configuration
Dim sVpName As String
sVpName = ThisDrawing.ActiveViewport.Name

' Create a new viewport with the same name
' as the active viewport
Dim oVPort As AcadViewport
Set oVPort = ThisDrawing.Viewports.Add(sVpName)

' Split the active viewport vertically
oVPort.Split acViewport2Vertical

' Turn off the grid and snap in the new viewport
oVPort.GridOn = False
oVPort.SnapOn = False

' Turn on Ortho mode
oVPort.OrthoOn = True

' Set the viewport active
ThisDrawing.ActiveViewport = oVPort

' Set snap style to rectangular
ThisDrawing.SetVariable "snapstyl", 0
```

Using the `AcadViewport` object returned by the `Add` function of the `AcadViewports` collection object or the `ActiveViewport` property, you can obtain information about the current view and some of the drafting aids that are enabled. Table 5.4 lists the properties of the `AcadViewport` object.

Table 5.4: Properties related to an `AcadViewport` object

Property	Description
`ArcSmoothness`	Specifies the smoothness for curved model space objects. Enter a value from 1 to 20,000.
`Center`	Specifies an array of three double values that represents the center point of the view in the viewport.
`Direction`	Specifies the view direction of the model space objects. View direction is expressed as an array of three double values.

Property	Description
`GridOn`	Specifies whether grid display is enabled. A Boolean value of `True` indicates the grid display is on.
`Height`	Specifies the height of the view in drawing units, not pixels. This value corresponds to the magnification factor of the current view. The value returned or expected is a double.
`LowerLeftCorner`	Specifies an array of two double values that represents the lower-left corner of the viewport.
`Name`	Specifies the name of the configuration in which the viewport is associated.
`OrthoOn`	Specifies whether Ortho mode is enabled. A Boolean value of `True` indicates Ortho mode is on.
`SnapBasePoint`	Specifies an array of two double values that represents the base point of the snap grid for the viewport.
`SnapOn`	Specifies whether snapping is enabled. A Boolean value of `True` indicates snapping is on.
`SnapRotationAngle`	Specifies the angle in which the snap grid is rotated. The value returned or expected is a double that represents the angle in radians.
`Target`	Specifies the target point of the current view in the viewport. View direction is expressed as an array of three double values.
`UCSIconAtOrigin`	Specifies whether the UCS icon is displayed at the origin of the drawing. A Boolean value of `True` indicates the UCS icon is displayed at the drawing's origin, or in the lower-left corner of the drawing area if the origin is off the screen.
`UCSIconOn`	Specifies whether the UCS icon is displayed in the drawing area. A Boolean value of `True` indicates the UCS icon is displayed.
`UpperRightCorner`	Specifies an array of two double values that represents the upper-right corner of the viewport.
`Width`	Specifies the width of the view in drawing units, not pixels. This value corresponds to the magnification factor of the current view. The value returned or expected is a double.

In addition to the `GridOn` and `SnapOn` properties that allow you to turn on grid display and enable snapping to grid, you can use the `GetGridSpacing` and `GetSnapSpacing` methods to get the current grid and snap spacing. Both of the methods expect two arguments that are used to return the X and Y spacing values for the grid or snap. To change the spacing of the grid and snap, use the `SetGridSpacing` and `SetSnapSpacing` methods, which expect two double values that represent the X and Y spacing values for the grid or snap.

A named view can be assigned to a model space viewport using the `SetView` function. I explain how to work with named views in the next section. For more information on working with tiled viewports, see the AutoCAD Help system.

Creating and Managing Named Views

Named views are areas in a drawing with a user-defined name that can later be restored to improve navigation around a large drawing and even help to output various areas of a drawing with viewports on a named layout. Many users associate named views with 3D modeling, but they can be just as helpful with designs that consist of just 2D objects. Named views are stored in the `AcadViews` collection object, which you can access from the Views property of the `AcadDocument` or `ThisDrawing` object. Each view stored in the `AcadViews` collection object is represented by an `AcadView` object.

You can create a new named view with the `Add` function of the `AcadViews` collection object. If you want to work with an existing view, use the `Item` function of the `AcadViews` collection object or a `For` statement to get the `AcadView` object that represents the named view you want to modify or query. Once a named view has been created, you can pass the `AcadView` object to the `SetView` method of an `AcadViewport` or `AcadPViewport` object to restore the view. If you no longer need a named view, you can use the `Delete` method of the `AcadView` object to be removed.

Table 5.5 lists the properties of an `AcadView` object that can be used to modify or query a named view.

Table 5.5: Properties related to an `AcadView` object

Property	**Description**
`CategoryName`	Specifies a category name for the view. The category name is used to group multiple views on the ShowMotion bar when it is pinned and controls how named views are organized in sheet sets.
`Center`	Specifies an array of three double values that represents the center point of the view.
`Direction`	Specifies the direction from which the objects in the model space should be viewed. View direction is expressed as an array of three double values.
`HasVpAssociation`	Specifies whether the view is associated with a viewport. A Boolean value of `True` indicates that the view is associated with a viewport placed from the Sheet Set Manager.
`Height`	Specifies the height of the view in drawing units, not pixels. The value returned or expected is a double.
`LayerState`	Specifies the name of the layer state that should be restored when the view is restored. I discussed layer states in Chapter 4.
`LayoutId`	Specifies the object ID of the layout that the view is associated with. Model space views can't be used on a named layout and a named layout can't be used on the Model tab.

Property	Description
`Name`	Specifies the name of the named view.
`Target`	Specifies the target point of the view. The target is expressed as an array of three double values.
`Width`	Specifies the width of the view in drawing units, not pixels. This value corresponds to the magnification factor of the view. The value returned or expected is a double.

For more information on working with named views, see the AutoCAD Help system.

Applying Visual Styles

Visual styles affect the way 2D and 3D objects are displayed on screen and how they are plotted. The AutoCAD Object library offers very limited support when it comes to managing visual styles. Using the AutoCAD Object library, you can obtain a listing of which visual styles are stored in a drawing by accessing the `ACAD_VisualStyles` dictionary. I explain how to work with dictionaries in Chapter 9.

If you need to create or update a visual style using the AutoCAD Object library, set as current the visual style that you want to base the new visual style on or modify with the `vscurrent` command. Once the visual style is current, modify the values of the system variables related to visual styles. Many of the system variables that are related to visual styles begin with the prefix `VS`.

Use the `SetVariable` and `GetVariable` methods to work with the system variables. After the variables have been updated, use the `vssave` command to save the new visual style or overwrite an existing visual style with the same name. You can assign a visual style to model space with the `vscurrent` command, or use the `VisualStyle` property of an `AcadPViewport` object, which represents a floating viewport on a named layout. I explain how to work with floating viewports in Chapter 8.

Exercise: Getting Input from the User to Draw the Plate

In this section, you will continue to build on the `DrawPlate` project that was introduced in Chapter 4. The key concepts I cover in this exercise are as follows:

Requesting Input Input functions can be used to get values from the user at the Command prompt.

Creating a New Point Value Values from different point lists can be used to create new coordinate values.

Using Conditional Statements Conditional statements are a great way to check the data provided by a user.

Looping Until a Condition Is Met Loops allow you to execute a set of expressions a specific number of times or while a condition remains `True`. You can use a loop to keep allowing the user to provide input.

NOTE The steps in this exercise depend on the completion of the steps in the "Exercise: Creating, Querying, and Modifying Objects" section of Chapter 4. If you didn't complete the steps, do so now or start with the `ch05_drawplate.dvb` sample file available for download from `www.sybex.com/go/autocadcustomization`. Place the sample file in the `MyCustomFiles` folder within the `Documents` (or `My Documents`) folder, or the location where you are storing the DVB files. Also, remove `ch05_` from the filename before you begin working.

Revising the *CLI_DrawPlate* Function

The changes to the `CLI_DrawPlate` function implement the use of user input to get points and distances. The points and distances provided by the user are used to specify the size and location of the plate in the drawing. The following steps have you replace the `CLI_DrawPlate` function with a newer version in the `drawplate.dvb` project file:

1. Load the `drawplate.dvb` file into the AutoCAD drawing environment and display the VBA Editor.
2. In the VBA Editor, in the Project Explorer, double-click the `basDrawPlate` component.
3. In the code editor window, replace all of the code statements in the code module with the following code statements; the comments are here for your information and don't need to be typed:

```
Private myUtilities As New clsUtilities

Private g_drawplate_width As Double
Private g_drawplate_height As Double

' Constants for PI and removal of the "Command: " prompt msg
Const PI As Double = 3.14159265358979
Const removeCmdPrompt As String = vbBack & vbBack & vbBack & _
                                  vbBack & vbBack & vbBack & _
                                  vbBack & vbBack & vbBack & vbLf

Public Sub CLI_DrawPlate()
  Dim oLyr As AcadLayer

  On Error Resume Next

  Dim sysvarNames As Variant, sysvarVals As Variant
  sysvarNames = Array("nomutt", "clayer", "textstyle")

  ' Store the current value of system variables to be restored later
  sysvarVals = myUtilities.GetSysvars(sysvarNames)

  ' Set the current value of system variables
  myUtilities.SetSysvars sysvarNames, Array(0, "0", "STANDARD")

  ' Define the width and height for the plate
```

```
If g_drawplate_width = 0 Then g_drawplate_width = 5#
If g_drawplate_height = 0 Then g_drawplate_height = 2.75

' Get recently used values from the global variables
Dim width As Double, height As Double
width = g_drawplate_width
height = g_drawplate_height

' Prompt for the current values
ThisDrawing.Utility.Prompt removeCmdPrompt & "Current width: " & _
                           Format(ThisDrawing.Utility. _
                             RealToString(width, acDecimal, 4), _
                             "0.0000") & _
                           "  Current height: " & _
                           Format(ThisDrawing.Utility. _
                             RealToString(height, acDecimal, 4), _
                             "0.0000") & _
                           vbLf

Dim basePt As Variant

' Continue to ask for input until a point is provided
Do
  Dim sKeyword As String
  sKeyword = ""
  basePt = Null

  ' Set up default keywords
  ThisDrawing.Utility.InitializeUserInput 0, "Width Height"

  ' Prompt for a base point, width, or height value
  basePt = ThisDrawing.Utility.GetPoint(, _
           removeCmdPrompt & _
           "Specify base point for plate or [Width/Height]: ")

  ' If an error occurs, the user entered a keyword or pressed Enter
  If Err Then
    Err.Clear

    sKeyword = ThisDrawing.Utility.GetInput

    Select Case sKeyword
      Case "Width"
        width = ThisDrawing.Utility. _
                GetDistance(, removeCmdPrompt & _
                              "Specify the width of the plate <" & _
                              Format(ThisDrawing.Utility. _
                                RealToString(width, acDecimal, 4), _
                                "0.0000") & _
                              ">: ")
      Case "Height"
        height = ThisDrawing.Utility. _
                 GetDistance(, removeCmdPrompt & _
                               "Specify the height of the plate <" & _
                               Format(ThisDrawing.Utility. _
                                 RealToString(height, acDecimal, 4), _
                                 "0.0000") & _
                               ">: ")
    End Select
  End If

  ' If a base point was specified, then draw the plate
  If IsNull(basePt) = False Then
    ' Create the layer named Plate or set it current
    Set oLyr = myUtilities.CreateLayer("Plate", acBlue)
    ThisDrawing.ActiveLayer = oLyr

    ' Create the array that will hold the point list
    ' used to draw the outline of the plate
    Dim dPtList(7) As Double
    dPtList(0) = basePt(0): dPtList(1) = basePt(1)
    dPtList(2) = basePt(0) + width: dPtList(3) = basePt(1)
    dPtList(4) = basePt(0) + width: dPtList(5) = basePt(1) + height
    dPtList(6) = basePt(0): dPtList(7) = basePt(1) + height

    ' Draw the rectangle
    myUtilities.CreateRectangle dPtList

    ' Create the layer named Holes or set it current
    Set oLyr = myUtilities.CreateLayer("Holes", acRed)
    ThisDrawing.ActiveLayer = oLyr

    Dim cenPt1 As Variant, cenPt2 As Variant
    Dim cenPt3 As Variant, cenPt4 As Variant
    Dim dAng As Double, dDist As Double

    ' Calculate the placement of the circle in the lower-left corner.
    ' Calculate a new point at 45 degrees and distance of 0.7071 from
    ' the base point of the rectangle.
    cenPt1 = ThisDrawing.Utility.PolarPoint(basePt, PI / 4, 0.7071)
    myUtilities.CreateCircle cenPt1, 0.1875

    ' Calculate the distance between the first
```

```
        ' and second corners of the rectangle.
        dDist = myUtilities.Calc2DDistance(dPtList(0), dPtList(1), _
                                           dPtList(2), dPtList(3))

        ' Calculate and place the circle in the lower-right
        ' corner of the rectangle.
        dAng = myUtilities.Atn2(dPtList(2) - dPtList(0), _
                                dPtList(3) - dPtList(1))
        cenPt2 = ThisDrawing.Utility.PolarPoint(cenPt1, dAng, dDist - 1)
        myUtilities.CreateCircle cenPt2, 0.1875

        ' Calculate the distance between the second
        ' and third corners of the rectangle.
        dDist = myUtilities.Calc2DDistance(dPtList(2), dPtList(3), _
                                           dPtList(4), dPtList(5))

        ' Calculate and place the circle in the upper-right
        ' corner of the rectangle.
        dAng = myUtilities.Atn2(dPtList(4) - dPtList(2), _
                                dPtList(5) - dPtList(3))
        cenPt3 = ThisDrawing.Utility.PolarPoint(cenPt2, dAng, dDist - 1)
        myUtilities.CreateCircle cenPt3, 0.1875

        ' Calculate and place the circle in the upper-left
        ' corner of the rectangle.
        dAng = myUtilities.Atn2(dPtList(6) - dPtList(0), _
                                dPtList(7) - dPtList(1))
        cenPt4 = ThisDrawing.Utility.PolarPoint(cenPt1, dAng, dDist - 1)
        myUtilities.CreateCircle cenPt4, 0.1875
      End If
    Loop Until IsNull(basePt) = True And sKeyword = ""

    ' Restore the saved system variable values
    myUtilities.SetSysvars sysvarNames, sysvarVals

    ' Save previous values to global variables
    g_drawplate_width = width
    g_drawplate_height = height
  End Sub
```

4. Click File ➤ Save.

Revising the *Utilities* Class

The changes to the `Utilities` class add a new constant named *PI* that holds the mathematical value of PI and introduce two new functions: `Calc2DDistance` and `Atn2`. The `Calc2DDistance` function returns a double value that is the distance between two 2D points, and the `Atn2` function returns an angular value in radians between two points. The following steps have you adding the constant value and two functions to the `clsUtilities` class module:

1. In the VBA Editor, in the Project Explorer, double-click the `clsUtilities` component.
2. In the code editor window, click to the left of the first comment or code statement and press Enter twice.
3. Click in the first blank line of the code module and type the following code statement:

```
Const PI As Double = 3.14159265358979
```

4. Scroll to the bottom of the code editor window and click to the right of the last code statement. Press Enter twice.
5. Type the following code statements; the comments are here for your information and don't need to be typed:

```
' Returns the 2D distance between two points.
' Function expects four double numbers that represent the
' X and Y values of the two points.
Public Function Calc2DDistance(X1, Y1, X2, Y2) As Double
  Calc2DDistance = Sqr((X2 - X1) ^ 2 + (Y2 - Y1) ^ 2)
End Function

' Returns the radians angular value between the differences of the
' X and Y delta values.
' Function expects the X and Y delta differences between two points.
Function Atn2(dDeltaX As Double, dDeltaY As Double) As Double
  Select Case dDeltaX
    Case Is > 0
      Atn2 = Atn(dDeltaY / dDeltaX)
    Case Is < 0
      Atn2 = Atn(dDeltaY / dDeltaX) + PI * Sgn(dDeltaY)
      If dDeltaY = 0 Then Atn2 = Atn2 + PI
    Case Is = 0
      Atn2 = (PI / 2) * Sgn(dDeltaY)
    End Select
End Function
```

6. Click File ➢ Save.

The following steps explain how to export the `clsUtilities` class module from the `drawplate.dvb` file:

1. In the VBA Editor, in the Project Explorer, right-click the `clsUtilities` component and choose Export File.
2. When the Export File dialog box opens, browse to the `MyCustomFiles` folder.
3. Keep the default filename of `clsUtilities.cls` and click Save.

   The `clsUtilities.cls` file is exported from the `DrawPlate` project.

4. In the Confirm Save As dialog box, click Yes to replace the previously exported version of the `Utilities` class.

Using the Revised *drawplate* Function

Now that that the `drawplate.dvb` project file has been revised, you can test the changes that have been made. The following steps explain how to use the revised `drawplate` function:

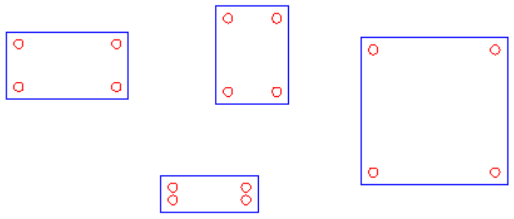
1. Switch to AutoCAD by clicking on its icon in the Windows taskbar or click View ➢ AutoCAD from the menu bar in the Visual Basic Editor.
2. In AutoCAD, at the Command prompt, type **`vbarun`** and press Enter.
3. When the Macros dialog box opens, select the `DrawPlate.dvb!basDrawPlate.CLI_DrawPlate` macro from the list and click Run.
4. Press F2 to expand the command-line window. The current width and height values for the plate are displayed in the command-line history.

   ```
   Current width: 5.0000  Current height: 2.7500
   ```

5. At the `Specify base point for the plate or [Width/Height]:` prompt, type **`w`** and press Enter.
6. At the `Specify the width of the plate <5.0000>:` prompt, type **`3`** and press Enter.
7. At the `Specify base point for the plate or [Width/Height]:` prompt, type **`h`** and press Enter.
8. At the `Specify the height of the plate <2.7500>:` prompt, type **`4`** and press Enter.
9. At the `Specify base point for the plate or [Width/Height]:` prompt, pick a point in the drawing area to draw the plate and holes based on the width and height values specified.
10. Type **`'zoom`** and press Enter, and then type **`e`** and press Enter.

    Figure 5.5 shows a number of different plates that were drawn at various sizes with the `CLI_DrawPlate` macro.

FIGURE 5.5
Completed plates



11. Continue trying the `CLI_DrawPlate` macro with different input values.
12. Press Enter to exit the macro when you are done.

Chapter 6

Annotating Objects

Annotation plays an important role in most designs; it is used to communicate measurements and design features that might require explanation. The Autodesk® AutoCAD® program offers a variety of annotation objects that include stand-alone text, dimensions, leaders, and tables. Each annotation object type is affected by specially named styles that control its appearance. Blocks can also include attributes, which are a form of annotation that can be updated when an instance of a block reference is inserted into a drawing. I discuss blocks and attributes in Chapter 7, "Working with Blocks and External References."

In this chapter, you will learn to create and modify stand-alone text objects and other types of annotation objects, such as dimensions, leaders, and tables. Along with creating and modifying annotation objects, you will also learn to control the appearance of annotation objects with named styles and create field values that can be used in multiline text objects and table cells.

Working with Text

Stand-alone text is often used for adding labels below a viewport and detail, general disclaimers, and revision comments. You can create two types of stand-alone text: single-line and multiline. Single-line text (`Text`) is used when you only need to add a few words or a short comment to a drawing, whereas multiline text (`MText`) is used when you want to create a bullet list or a paragraph of text.

`MText` supports a wider range of formatting options and features than single-line text. Even though `MText` is designed for formatting text in paragraphs, it can be used in place of single-line text. The appearance of stand-alone text is controlled by its assigned text style.

Creating and Modifying Text

Single-line text and `MText` is represented by the `AcadText` and `AcadMText` objects. The `AddText` function allows you to create a single-line text object based on a text string, an insertion point, and text height. The text height passed to the `AddText` function is used only if the `Height` property of the text style assigned to the text object is set to `0`. I discuss text styles in the "Controlling Text with Text Styles" section later in this chapter. You use the `AddMText` function to create a new `MText`. The `AddMText` function is similar to the `AddText` function with one exception: the `AddMText` function expects a value that defines the width of the bounding box of the text area instead of a value that defines the height of the text object. The following shows the syntax of the `AddText` and `AddMText` functions:

```
retVal = object.AddText(textString, insertionPoint, height)
retVal = object.AddMText(insertionPoint, width, textString)
```

Their arguments are as follows:

retVal The *retVal* argument represents the new `AcadText` or `AcadMText` object returned by the function.

object The *object* argument represents an `AcadModelSpace`, `AcadpaperSpace`, or `AcadBlock` collection object.

textString The *textString* argument is a string that contains the text that should be added to the text object. The text string can contain special character sequences to format text and insert special characters; see the "Formatting a Text String" section later in this chapter for some of the supported character sequences.

insertionPoint The *insertionPoint* argument is an array of three doubles that defines the insertion point of the text object.

height* or *width The *height* and *width* arguments are doubles that define the height of the text for an `AcadText` object or overall width of the boundary box of an `AcadMText` object.

The following code statements add two new single-line text objects to model space (see Figure 6.1):

```
' Defines the insertion point and height for the text object
Dim dInsPt(2) As Double, dHeight As Double
dInsPt(0) = 0: dInsPt(1) = 0: dInsPt(2) = 0
dHeight = 0.25

' Creates a new text object
Dim oText As AcadText
Set oText = ThisDrawing.ModelSpace.AddText( _
    "NOTE: ADA requires a minimum turn radius of", dInsPt, dHeight)

' Adjusts the insertion point for the second text object
dInsPt(0) = 0: dInsPt(1) = dHeight * -1.6065: dInsPt(2) = 0
Set oText = ThisDrawing.ModelSpace.AddText( _
    "60"" (1525mm) diameter for wheelchairs.", dInsPt, dHeight)
```

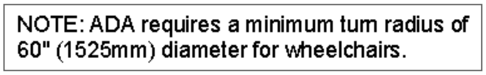

NOTE: ADA requires a minimum turn radius of
60" (1525mm) diameter for wheelchairs.

FIGURE 6.1
Basic note created with single-line text

The following code statements add an `MText` object to model space (see Figure 6.2):

```
' Defines the insertion point and width for the text object
Dim dInsPt(2) As Double, dWidth As Double
dInsPt(0) = 0: dInsPt(1) = 0: dInsPt(2) = 0
dWidth = 5.5
```

```
' Creates a new text object
Dim oMText As AcadMText
Set oMText = ThisDrawing.ModelSpace.AddMText(dInsPt, dWidth, _
    "NOTE: ADA requires a minimum turn radius of " & _
    "60"" (1525mm) diameter for wheelchairs.")
```

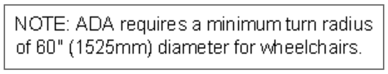

NOTE: ADA requires a minimum turn radius
of 60" (1525mm) diameter for wheelchairs.

Figure 6.2
Basic note created with an MText object

The properties of the AcadText and AcadMText objects can be used to adjust the justification of the text, the direction in which the text is drawn, and much more. For information on the properties of the two text objects, see the AutoCAD Help system or the Object Browser in the VBA Editor. Like other graphical objects, the AcadText and AcadMText objects also inherit the properties and methods of an AcadEntity object, which I discussed in Chapter 4, "Creating and Modifying Drawing Objects."

The following code statements add a new single-line text object to model space and center the text:

```
' Defines the insertion point and height for the text object
Dim dInsPt(2) As Double, dHeight As Double
dInsPt(0) = 5: dInsPt(1) = 5: dInsPt(2) = 0
dHeight = 0.25

' Creates a new text object
Dim oText As AcadText
Set oText = ThisDrawing.ModelSpace.AddText( _
    "Center Justified", dInsPt, dHeight)

' Sets the justification of the text to middle center
oText.Alignment = acAlignmentMiddleCenter

' Moves the alignment point of the justified text
' to the original insertion point
oText.TextAlignmentPoint = dInsPt
```

NOTE After changing the justification of an AcadText object, you will need to update the TextAlignmentPoint property to move the location to the correct position.

In addition to the methods the AcadText and AcadMText objects inherit from an AcadEntity object, the objects also support a function named FieldCode. I explain the FieldCode function in the "Creating Fields" section later in this chapter.

Formatting a Text String

Alphanumeric characters are used to create the text string that an AcadText object displays, but how those characters are arranged can impact how the text appears. The use of the percent symbol has a special meaning in a text string. You use a percent symbol to indicate the use of special control codes and field values. Special control codes can be used to toggle underlining or overscoring for part or all of a text string and to insert special symbols. Table 6.1 lists the control codes that are supported in the text string of an `AcadText` object.

TABLE 6.1: Control codes for `AcadText` objects

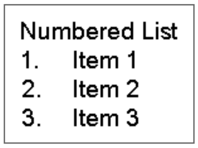
CONTROL CODE	DESCRIPTION
%%c	Adds a diameter symbol to the text.
%%d	Adds a degree symbol to the text.
%%nnn	Adds the ASCII character represented by the character value *nnn*. For example, `%%169` adds the Copyright symbol.
%%o	Toggles the use of overscoring. The first instance of `%%o` in a text string turns overscoring on, and the second turns it off.
%%p	Adds a plus or minus symbol (±) to the text.
%%u	Toggles the use of underscoring. The first instance of `%%u` in a text string turns underscoring on, and the second turns it off.
%%%	Adds a percent symbol to the text.
%< and >%	Defines the start and end of a field value. I discuss working with field values in in the "Creating Fields" section later in this chapter.

The text string of an `AcadMText` object can be very basic, but it can be very complex as well. You can control the formatting of each character in a text string with special control codes. Unlike the special control codes that are supported by an `AcadText` object, those used by an `AcadMText` object are much more complicated and harder to figure out at first. However, the AutoCAD `list` command will be your friend if you want to create complexly formatted text strings.

The best process for learning how to format the text string of an `AcadMText` object is to use the `mtext` command in AutoCAD and create a sample text string that you want to create with your VBA macro. Once the `MText` object is added to the drawing, use the `list` command and look at the value after the Contents label in the output. For example, the following is an example of the output displayed by the `list` command for an `MText` object that contains a numbered list with three items (see Figure 6.3):

```
Contents:
  Numbered List\P\pxi-3,l3,t3;1.      Item 1\P2. Item 2\P3.     Item 3
```

FIGURE 6.3
Numbered list in an `MText` object



The long spaces in the example are actually tab characters. To create the numbered list shown in Figure 6.3 with VBA, the code statements would look like the following:

```
' Defines the insertion point and width for the MText object
Dim dInsPt(2) As Double, dWidth As Double
dInsPt(0) = 0: dInsPt(1) = 0: dInsPt(2) = 0
dWidth = 5.5

' Creates a new MText object with a numbered list
Dim oMText As AcadMText
Set oMText = ThisDrawing.ModelSpace.AddMText(dInsPt, dWidth, _
    "Numbered List\P\pxi-3,l3,t3;1." & vbTab & _
    "Item 1\P2." & vbTab & "Item 2\P3." & vbTab & "Item 3")
```

Most of the control codes you will need to use take a combination of the `list` and `mtext` commands to initially figure out, but there a few control codes that are much easier to add to the text string of `MText`. The `AcadMText` object supports the `%%d`, `%%c`, and `%%p` control codes that are also supported by the `AcadText` object. If you want to add a special character to a text string of an `AcadMText` object, use the control sequence of `\U+nnn`, which adds a character based on its Unicode value instead of the `%%nnn` that an `AcadText` object supports. For example, to insert the Copyright symbol you would use the sequence of `\U+00A9`.

TIP You can use the Windows Character Map to get the Unicode value of a character for a specific font. If you need to use a character from the font that isn't assigned to the text style applied to the `MText` object, you must provide the proper control codes to indicate the font you want to use for that character. For example, the following indicates that the Copyright symbol of the Arial font should be added:

```
{\fArial|b0|i0|c186|p34;\U+00A9}
```

As I mentioned before, it is best to use the `mtext` command to first create an `MText` object and then use the `list` command to see the contents of that object. Then you will know the code control codes and sequences required.

Real World Scenario

Checking Spelling

The AutoCAD Object library doesn't support the ability to check the spelling or grammar of a text string. However, with some help from the Microsoft Word Object library you can check the spelling and grammar of a text string. The following outlines an approach you can take using the Word Object library to check the spelling or grammar of a text string:

1. Create a Word Document object.
2. Add the text you want to check.
3. Perform the spelling and grammar check.
4. Update the text in the drawing.
5. Close and discard the changes to the Word Document object.

I introduce how to work with the Word Object library in Chapter 12, "Communicating with Other Applications."

Controlling Text with Text Styles

Text styles are used to control the appearance of the characters in a text string for an `AcadText` or `AcadMText` object. Some of the characteristics that are controlled by a text style are font filename, bold and italic font faces, and character sets. A text style is represented by the `AcadTextStyle` object, and the text styles stored in a drawing are accessed from the `AcadTextStyles` collection object. Use the `TextStyles` property of an `AcadDocument` or `ThisDrawing` object to get a reference to the `AcadTextStyles` collection object.

Creating and Managing Text Styles

New text styles are created with the `Add` method of the `AcadTextStyles` collection object. The `Add` method of the `AcadTextStyles` collection object requires you to provide the name of the new text style and returns an `AcadTextStyle` object. The `Item` method of the `AcadTextStyles` collection object is used to get an existing text style in the drawing; if the text style doesn't exist, an error is generated. I discuss how to handle errors in Chapter 13, "Handling Errors and Deploying VBA Projects."

The `Item` method accepts a string that represents the name of the text style you want to work with or an integer value. The integer value represents the index of the text style in the `AcadTextStyles` collection object you want to return. The index of the first text style in the drawing starts with `0`, and the highest index is one less than the number of text styles in the `AcadTextStyles` collection object returned by the `Count` property. If you want to step through all the text styles in the drawing, you can use a `For` statement.

The following sample code statements check for the existence of a text style named General; if the text style doesn't exist, it is created:

```
On Error Resume Next

' Gets the TextStyles collection
Dim oStyles As AcadTextStyles
Set oStyles = ThisDrawing.TextStyles

' Gets the text style named General
Dim oStyle As AcadTextStyle
Set oStyle = oStyles("General")

' If an error is returned, create the text style
If Err Then
  Err.Clear

  ' Creates a new text style
  Set oStyle = oStyles.Add("General")
End If
```

NOTE Although the `Add` method won't return an error if a text style with the same name already exists, I recommend using the `Item` method of the `AcadTextStyles` collection object to check whether a text style already exists.

After you have an AcadTextStyle object, you can get its current font and character set with the GetFont method. The SetFont method is used to set the font and character set among other settings of the text style. In addition to the GetFont and SetFont methods, you can use the fontFile and BigFontFile properties of the AcadTextStyle object to specify the TrueType font (TTF) and Shape (SHX) file that should be used by the text style. The BigFontFile property is helpful if you need to support the double-byte characters that are used mainly for Asian languages.

If you want text to be drawn at a specific height each time the text style is used, you set the height value to the Height property of the text style. Other properties of a text style allow you to specify the oblique angle and direction in which the text should be drawn, among other settings with the properties of the AcadTextStyle object. For information on the properties of the two text objects, see the AutoCAD Help system or the Object Browser in the VBA Editor.

NOTE Text styles are used by dimension, mleader, and table styles. If a text style will be used by other named annotation styles, I recommend that you set the Height property of the text style to 0. When you use a height of 0, the referencing named annotation style has control over the final text height.

If you don't need a text style anymore, remove it from a drawing with the Delete method of the AcadTextStyle object and not the Delete method of the AcadTextStyles collection object. The PurgeAll method of an AcadDocument or ThisDrawing object can also be used to remove all unused text styles from a drawing. I discussed the PurgeAll method in Chapter 4.

The following sample code statements set the font of the text style assigned to the *oStyle* variable, enable boldface, and set the oblique angle to 10:

```
Dim sFont As String
Dim bBold As Boolean, bItalic As Boolean
Dim nCharSet As Long
Dim nPitchandFamily As Long

' Sets the font, enables boldface, and assigns an
' oblique angle to the style based on the active style
ThisDrawing.ActiveTextStyle.GetFont sFont, bBold, _
   bItalic, nCharSet, nPitchandFamily

oStyle.SetFont "Arial", True, False, nCharSet, nPitchandFamily
oStyle.ObliqueAngle = 10
```

Assigning a Text Style

A text style can be assigned to an object directly or inherited by the active text style of the drawing. You assign a text style to an AcadText or AcadMText object with the StyleName property. The StyleName property returns or accepts a string that represents the name of current or the text style to be assigned. When a new text object is created, the text style applied is inherited from the ActiveTextStyle property of the AcadDocument or ThisDrawing object. The ActiveTextStyle property returns and expects an AcadTextStyle object.

NOTE As an alternative to the `ActiveTextStyle` property, you can use the `textstyle` system variable. The `textstyle` system variable accepts a string that represents the name of the text style to be inherited by each newly created text object.

The following code statements assign the text style named `General` to the `ActiveTextStyle` property:

```
' Sets the General text style as the active text style
Dim oStyle As AcadTextStyle
Set oStyle = ThisDrawing.TextStyles("GENERAL")
ThisDrawing.ActiveTextStyle = oStyle
```

Dimensioning Objects

Dimensions are annotation objects that show a measurement value in a drawing. The value in which a dimension displays depends on the type of dimension object created. A dimension can measure the linear distance between two points, the radial value of a circle or an arc, the X or Y value of a coordinate, the angle between two vectors, or the length of an angle. Similar to text objects, the appearance of a dimension is controlled by a dimension style. Dimension objects are graphical objects just like lines and circles, so they inherit properties and methods from `AcadEntity`. Dimensions also inherit properties from a class named `AcadDimension`.

Creating Dimensions

Nine types of dimensions can be created with the AutoCAD Object library and VBA. When you want to add a dimension object to a drawing, use one of the functions that begin with the name `AddDim`. The functions used to add a dimension object can be accessed from an `AcadModelSPace`, `AcadPaperSpace`, or `AcadBlock` collection object. Table 6.2 lists the functions that can be used to add a new dimension object.

TABLE 6.2: Functions used to create new dimensions

FUNCTION	DESCRIPTION
`AddDim3PointAngular`	Adds an angular dimension based on three points; same as that created with the `dimangular` command.
`AddDimAligned`	Adds a linear dimension that is parallel to the two points specified; same as the `dimaligned` command.
`AddDimAngular`	Adds an angular dimension based on two vectors; same as that created with the `dimangular` command.
`AddDimArc`	Adds an arc length dimension based on the center of an arc and two points along the arc; same as that created with the `dimarc` command.
`AddDimDiametric`	Adds a diametric dimension that reflects the diameter of a circle or an arc; same as that created with the `dimdiameter` command.

Function	Description
AddDimOrdinate	Adds an ordinate dimension that displays the X or Y value of a coordinate; same as that created with the dimordinate command.
AddDimRadial	Adds a radial dimension that reflects the radius of a circle or an arc; same as that created with the dimradius command.
AddDimRadialLarge	Adds a radial dimension with a jogged line that indicates the radius of a circle or arc, but the dimension doesn't start at the center of the object dimensioned; same as that created with the dimjogged command.
AddDimRotated	Adds a linear dimension that measures the distance between two points, but the dimension line of the dimension is rotated at a specified value; same as that created with the dimrotated command.

For specifics on the arguments that are required to add a dimension object, see the AutoCAD Help system or the Object Browser in the VBA Editor.

The following code statements add two circles, add a linear dimension between the center points of the two circles with the `AddDimRotated` function, and finally, add a diameter dimension to one of the circles with the `AddDiametric` function (see Figure 6.4):

```
' Defines the center point of the circles
Dim dCenPt1(2) As Double, dCenPt2(2) As Double
dCenPt1(0) = 2.5: dCenPt1(1) = 1: dCenPt1(2) = 0
dCenPt2(0) = 5.5: dCenPt2(1) = 2: dCenPt2(2) = 0

' Adds the two circles
ThisDrawing.ModelSpace.AddCircle dCenPt1, 0.5
ThisDrawing.ModelSpace.AddCircle dCenPt2, 0.5

' Adds the linear dimension
Dim dDimPlace(2) As Double
dDimPlace(0) = dCenPt2(0) - dCenPt1(0)
dDimPlace(1) = dCenPt2(1) + 1: dDimPlace(2) = 0

Dim oDimRot As AcadDimRotated
Set oDimRot = ThisDrawing.ModelSpace.AddDimRotated( _
              dCenPt1, dCenPt2, dDimPlace, 0)

' Adds the diametric dimension
Dim vDimChordPt1 As Variant
vDimChordPt1 = ThisDrawing.Utility.PolarPoint( _
            dCenPt1, -0.7854, 0.5)

Dim vDimChordPt2 As Variant
```

```
    vDimChordPt2 = ThisDrawing.Utility.PolarPoint( _
                dCenPt1, 0.7854 * 3, 0.5)

    Dim oDimDia As AcadDimDiametric
    Set oDimDia = ThisDrawing.ModelSpace.AddDimDiametric( _
                  vDimChordPt2, vDimChordPt1, 1)
```

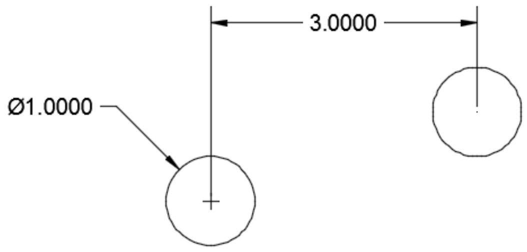


Figure 6.4
Aligned and diametric dimensions showing the measurement values of two circles

After you create a dimension object, you can modify its properties. However, based on the properties that you modify, a dimension override might be applied. For example, if you change the value of the `DimensionLineColor` property and later make a change to the dimension style applied to the dimension, the color of the dimension line will not be updated unless you remove the override from the dimension.

NOTE Dimensions created with the AutoCAD Object library are not associative; the dimension isn't updated if the objects that the dimension measures are changed. If you want to create associative dimensions, consider using the `SendCommand` or `PostCommand` method with the appropriate command sequence.

Formatting Dimensions with Styles

Dimension styles are stored in and accessed from the `AcadDimStyles` collection object. Each dimension style in a drawing is represented by an `AcadDimStyle` object. A new dimension style can be added to a drawing with the `Add` method of the collection object. The `Add` method expects a string that contains the name of the new dimension type to be created and returns an `AcadDimStyle` object. The `Item` method of the `AcadDimStyles` collection object is used to get an existing dimension style in the drawing; if the dimension style doesn't exist, an error is generated. I discuss how to handle errors in Chapter 13.

The `Item` method accepts a string that represents the name of the dimension style you want to work with or an integer value. The integer value represents the index of the dimension style in the `AcadDimStyles` collection object you want to return. The index of the first dimension style in the drawing starts with `0`, and the highest index is one less than the number of dimension styles in the `AcadDimStyles` collection object returned by the `Count` property. If you want to step through all the dimension styles in the drawing, you can use a `For` statement.

The following sample code statements check for the existence of a dimension style named `Arch24`; if the dimension style doesn't exist, it is created:

```
On Error Resume Next

' Gets the DimStyles collection
Dim oStyles As AcadDimStyles
Set oStyles = ThisDrawing.DimStyles

' Gets the dimension style named Arch24
Dim oStyle As AcadDimStyle
Set oStyle = oStyles("Arch24")

' If an error is returned, create the dimension style
If Err Then
  Err.Clear

  ' Creates a new dimension style
  Set oStyle = oStyles.Add("Arch24")

End If
```

NOTE Although the `Add` method won't return an error if a text style with the same name already exists, I recommend using the `Item` method of the `AcadDimStyles` collection object to check to see whether a dimension style already exists.

After you create or decide to modify an `AcadDimStyle` object, how to go about modifying the dimension style might not be immediately obvious. From the AutoCAD user interface, you commonly would use the Dimension Style Manager (displayed with the `ddim` command), but at the Command prompt, you could use the `-dimstyle` command.

Although you could use the `-dimstyle` command, the workflow with VBA is to modify the values of dimension-related system variables with the `SetVariable` method of an `AcadDocument` or a `ThisDrawing` object, and then use the `CopyFrom` method of the `AcadDimStyle` object to copy the values of the dimension system variables to the dimension style. Modifying the dimension system variables of a drawing will result in the creation of drawing-level dimension overrides. When creating a new dimension variable, I recommend storing the name of the current dimension style so it can be restored after you modify your dimension style.

Now, there is a problem that isn't easy to resolve: the preservation of drawing-level dimension overrides when modifying an existing dimension style. The reason is that when a dimension style is set as active, the previous drawing-level dimension variable overrides are lost. It is always best to restore the previous state of a drawing if you don't want to affect the current settings for the user.

The only way to preserve drawing-level dimension variable overrides is to create an array containing the current value of all dimension variables and then restore the values after the previous style has been set as active. An example of storing and restoring system variables for a number of system variables is shown in the "Setting the Values of Drafting-Related System Variables and Preferences" section of Chapter 3, "Interacting with the Application and Documents Objects."

Here are code statements that demonstrate how to change the values of the `dimblk` and `dimscale` dimension system variables, copy the values of the dimension variables of the drawing to the dimension style named Arch24, and then restore the previous dimension style and dimension values:

```
' Store the current dimension style
Dim oCurDimStyle As AcadDimStyle
Set oCurDimStyle = ThisDrawing.ActiveDimStyle

' Store current values to override
Dim vValues(1) As Variant
vValues(0) = ThisDrawing.GetVariable("DIMBLK")
vValues(1) = ThisDrawing.GetVariable("DIMSCALE")

' Change the DIMBLK and DIMSCALE system variable for the drawing
ThisDrawing.SetVariable "DIMBLK", "ARCHTICK"
ThisDrawing.SetVariable "DIMSCALE", 24#

' Create the new dimension style and copy the variable values from the drawing
oStyle.CopyFrom ThisDrawing

' Restore the previous style
Set ThisDrawing.ActiveDimStyle = oCurDimStyle

' Restore the values of the overridden variables
If vValues(0) = "" Then
  ThisDrawing.SetVariable "DIMBLK", "."
Else
  ThisDrawing.SetVariable "DIMBLK", vValues(0)
End If

ThisDrawing.SetVariable "DIMSCALE", vValues(1)
```

If you don't need a dimension style anymore, remove it from a drawing with the `Delete` method of the `AcadDimStyle` object and not the `Delete` method of the `AcadDimStyles` collection object. The `PurgeAll` method of an `AcadDocument` or `ThisDrawing` object can also be used to remove all unused dimension styles from a drawing. I discussed the `PurgeAll` method in Chapter 4.

Assigning a Dimension Style

You can change the dimension style of a dimension object after it has been added to a drawing with the `StyleName` property. The `StyleName` property returns or accepts a string that represents the name of the current or dimension style to be assigned. When a new dimension object is created, the dimension style applied is inherited from the `ActiveDimStyle` property of the `AcadDocument` or `ThisDrawing` object. The `ActiveDimStyle` property returns and expects an `AcadDimStyle` object.

NOTE Unlike other `Active*` properties, the `ActiveDimStyle` property doesn't have a system variable alternative that can be used to set the default dimension style for new dimension objects. The `dimstyle` system variable can be used to get the name of the current dimension style.

The following code statements assign the dimension style named `Arch24` to the `ActiveDimStyle` property:

```
' Sets the Arch24 text style as the active dimension style
Dim oStyle As AcadDimStyle
Set oStyle = ThisDrawing.DimStyles("ARCH24")
ThisDrawing.ActiveDimStyle = oStyle
```

Creating and Modifying Geometric Tolerances

Geometric tolerances, also referred to as *control frames*, are used to display acceptable deviations of a form, location, or other measurements in mechanical designs. A geometric tolerance is represented by an `AcadTolerance` object. Similar to `AcadMText` objects, `AcadTolerance` objects accept text strings with control codes in them to define the appearance of the final object that is displayed in the drawing. The control codes that an `AcadTolerance` object accepts define the symbols, tolerance, and datum values that are displayed in the geometric tolerance object. I recommend using the AutoCAD tolerance and list commands to learn the control codes and text sequences that go into defining a geometric tolerance object.

The following is an example of the output displayed by the `list` command for a geometric tolerance object that contains a Parallelism symbol, with a tolerance value of `0.00125` and a datum value of `B` (see Figure 6.5).

```
Text
  {\Fgdt;f}%%v{\Fgdt;n}.00125%%v%%vB%%v%%v
```

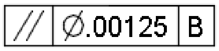


FIGURE 6.5 Geometric tolerance object created with the `AddTolerance` function

To create a geometric tolerance value, use the `AddTolerance` function of an `AcadModelSpace`, `AcadPaperSpace`, or `AcadBlock` collection object. The geometric tolerance object shown in Figure 6.5 can be created with the following code statements:

```
' Defines the insertion point and direction vector
' for the Tolerance object
Dim dInsPt(2) As Double, dDirVec(2) As Double
dInsPt(0) = 2.5: dInsPt(1) = 2.5: dInsPt(2) = 0
dDirVec(0) = 1: dDirVec(1) = 0: dDirVec(2) = 0

' Creates a new Tolerance object
Dim oTol As AcadTolerance
Set oTol = ThisDrawing.ModelSpace.AddTolerance( _
    "{\Fgdt;f}%%v{\Fgdt;n}.00125%%v%%vB%%v%%v", dInsPt, dDirVec)
```

The text string, insertion point, and direction among other characteristics of a geometric object can be queried or modified using the properties and methods of the `AcadTolerance` object. Like AcadDimension objects, an AcadTolerance object inherits the way it looks by the dimension style it is assigned. When initially created, the geometric tolerance object is assigned the dimension style that is assigned to the ActiveDimStyle property, and the StyleName property of an AcadTolerance object can be used to assign the object a specific dimension style.

If you need to use geometric tolerance objects in your drawings, see the AutoCAD Help system or Object Browser in the VBA Editor for more information.

Adding Leaders

Leaders, also known as *callouts*, are used to bring attention to a feature in a drawing. A leader starts with an arrowhead that is connected to multiple straight segments or a spline. The end of a leader often includes an attachment: a text object that contains a label or descriptive text. An attachment could also be a geometric tolerance object or block reference. AutoCAD supports two types of leaders: multileader and legacy.

Multileaders are leaders that can be made up of multiple leader lines and one or more attachments. The attachment and leader lines behave as a single object with multileaders. Legacy leaders don't provide as much flexibility as multileaders. Leader lines and the attachment of a legacy leader can be connected to or separate from the leader object.

Working with Multileaders

Multileaders were introduced in AutoCAD 2008 to improve the workflow when working with leaders. A multileader object is represented by the AcadMLeader object in a drawing file. Their initial appearance is controlled by a multileader style. The methods and properties of an AcadMLeader object allow you to add and modify leader lines and the content of a multileader object.

In addition to modifying a multileader object as a whole, you can modify the appearance of each leader line attached to the multileader object. Along with methods and properties specific to the AcadMLeader object, an AcadMLeader object inherits properties and methods from an AcadEntity.

Placing and Modifying Multileaders

A multileader object is created with the AddMLeader function. The AddMLeader method is available from an AcadModelSpace, AcadPaperSpace, or AcadBlock collection object and returns an AcadMLeader object. When you create a leader with the AddMLeader function, you specify the vertices of the initial leader line for the multileader. The AddMLeader function also returns an index for the leader line. which is represented by an AcadMLeaderLeader.

When a multileader is added to a drawing, its appearance is inherited by the active multileader style. I explain how to define and manage multileader styles in the next section, "Defining Multileader Styles." You will learn to apply a named multileader style in the "Assigning a Multileader Style" section.

The following code statements add a multileader with two leader lines and an attachment object of MText (see Figure 6.6):

```
' Defines the points of the first leader
Dim dLeader1Pts(0 To 5) As Double
dLeader1Pts(0) = 0.1326: dLeader1Pts(1) = 0.1326: dLeader1Pts(2) = 0
```

```
dLeader1Pts(3) = 1.1246: dLeader1Pts(4) = 2.1246: dLeader1Pts(5) = 0

' Defines the points of the second leader
Dim dLeader2Pts(0 To 5) As Double
dLeader2Pts(0) = 0.1847: dLeader2Pts(1) = 1.7826: dLeader2Pts(2) = 0
dLeader2Pts(3) = 1.1246: dLeader2Pts(4) = 2.1246: dLeader2Pts(5) = 0

' Adds the new multileader object
Dim lLeaderIdx As Long
Dim oMLeader As AcadMLeader
Set oMLeader = ThisDrawing.ModelSpace.AddMLeader(dLeader1Pts, lLeaderIdx)

' Adds the second leader line
oMLeader.AddLeaderLine lLeaderIdx, dLeader2Pts

' Attaches the MText object
oMLeader.ContentType = acMTextContent
oMLeader.TextString = "3/16""R"
```

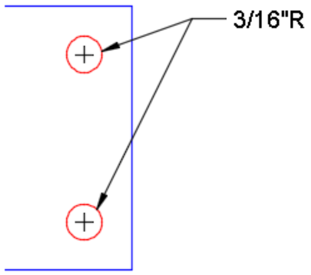


Figure 6.6
Multileader with two leader lines

After placing a multileader object, you can refine the leader lines, content, and appearance of the object using its methods and properties. However, depending on the properties that you modify, a style override might be applied. For example, if you change the value of the `ArrowheadBlock` property and later make a change to the multileader style applied to the object, the arrowhead of the leader lines will not be updated unless you remove the Xdata attached to the multileader that represents the data associated with the override. I explain more about Xdata in Chapter 9, "Storing and Retrieving Custom Data."

Defining Multileader Styles

Multileader styles are not accessed directly through a collection object like `AcadTextStyles` for text styles and `AcadDimStyles` for dimension styles. Named multileader styles stored in a drawing are stored in the ACAD_MLEADERSTYLE dictionary, which is accessed from the `AcadDictionaries` collection object. Each multileader style in the ACAD_MLEADERSTYLE dictionary is represented by an `AcadMLeaderStyle` object.

Use the `AddObject` function to create a new multileader style and the `GetObject` function to get an existing object that is in the dictionary. When using the `AddObject` function, you must specify two strings: the first is the name of the style you want to create and the second is the class name of AcDbMLeaderStyle. You can learn more about working with dictionaries in Chapter 9.

The following code statements create a multileader style named Callouts if it doesn't already exist:

```
On Error Resume Next

' Gets the multileader styles dictionary
Dim oDict As AcadDictionary
Set oDict = ThisDrawing.Dictionaries.Item("ACAD_MLEADERSTYLE")

' If no error, continue
If Not oDict Is Nothing Then
  ' Gets the multileader style named Callouts
  Dim oMLStyle As AcadMLeaderStyle
  Set oMLStyle = oDict.GetObject("Callouts")

  ' If an error is returned, create the multileader style
  If Err Then
    Err.Clear

    ' Creates a new dimension style
    Set oStyle = oDict.AddObject("Callouts", "AcDbMLeaderStyle")
  End If

  ' Defines the landing settings for the multileader style
  oMLStyle.EnableLanding = True
  oMLStyle.LandingGap = 0.1
End If
```

A multileader style that is no longer needed can be removed with the Remove method of the AcadDictionary object. For more information on the properties and methods of the AcadMLeaderStyle object, see the AutoCAD Help system or the Object Browser in the VBA Editor.

Assigning a Multileader Style

The active multileader style is assigned to a multileader when it is first added to a drawing, but the style assigned can be changed using the StyleName property. The StyleName property returns or accepts a string that represents the name of the current or multileader style to be assigned. When a new multileader object is created, the multileader style applied is inherited from the cmleaderstyle system variable of the drawing. You can use the SetVariable and GetVariable methods of an AcadDocument or a ThisDrawing object.

The following code statement assigns the multileader style named Callouts to the cmleaderstyle system variable:

```
' Sets the Callouts multileader style active
ThisDrawing.SetVariable "cmleaderstyle", "callouts"
```

Creating and Modifying Legacy Leaders

Legacy leader objects are represented by an `AcadLeader` object and are added to a drawing with the `AddLeader` function. The `AddLeader` method is available from an `AcadModelSpace`, `AcadPaperSpace`, or `AcadBlock` collection object. When you create a leader with the `AddLeader` function, you can choose to add an attachment object or no attachment. The types of objects you can attach to an `AcadLeader` object are `AcadMText`, `AcadTolerance`, and `AcadBlockReference`. If you don't want to add an attachment to a leader object, pass the value of `Nothing` to the `AddLeader` function instead of the object that represents the attachment object.

Unlike multileader objects, legacy leader objects inherit their format and appearance from a dimension style. Use the `StyleName` property of the `AcadLeader` object to assign a dimension style to the leader. The properties of the leader object can also be used to create an override.

The leader object shown in Figure 6.7 can be created with the following code statements:

```
' Defines the points of the leader line
Dim points(0 To 8) As Double
points(0) = 0: points(1) = 0: points(2) = 0
points(3) = 0.717: points(4) = 1.0239: points(5) = 0
points(6) = 1.217: points(7) = 1.0239: points(8) = 0

' Defines the insertion point and height for the text object
Dim dInsPt(2) As Double, dHeight As Double
dInsPt(0) = points(6): dInsPt(1) = points(7): dInsPt(2) = points(8)

' Creates a new text object
Dim oMText As AcadMText
Set oMText = ThisDrawing.ModelSpace.AddMText(dInsPt, 4#, _
    "TYP (4) Drill Holes")

' Sets the justification of the text to middle left
oMText.AttachmentPoint = acAttachmentPointMiddleLeft

' Moves the alignment point of the justified text
' to the original insertion point
oMText.InsertionPoint = dInsPt

Dim annotationObject As AcadObject
Set annotationObject = oMText

' Creates the leader object in model space
Dim leaderObj As AcadLeader
Set leaderObj = ThisDrawing.ModelSpace.AddLeader(points, _
                    annotationObject, acLineWithArrow)
```

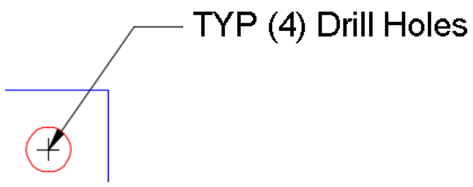


Figure 6.7
Legacy leader created with the `AddLeader` function

For more information on legacy leaders, search on `AcadLeader` in the AutoCAD Help system.

Organizing Data with Tables

Data in a drawing can often be presented in a tabular form with a table. Tables can be helpful in creating schedules or a bill of materials (BOM), which provides a quantitative listing of the objects in a drawing. Tables were introduced in AutoCAD 2005 to simplify the process of creating tables, which commonly had been made up of lines and single-line text objects. A table object is represented by the `AcadTable` object and the initial appearance is controlled by a table style.

The methods and properties of an `AcadTable` object allow you to add and modify the content of a table object. Just like other graphical objects, the `AcadTable` object inherits some of its properties and methods from an `AcadEntity` object.

Inserting and Modifying a Table

A table object is represented by the `AcadTable` object. The `AddTable` function allows you to create a table object based on an insertion point, number of rows and columns, as well as a row height and column width. The `AddTable` method is available from an `AcadModelSpace`, `AcadPaperSpace`, or `AcadBlock` collection object and returns an `AcadTable` object.

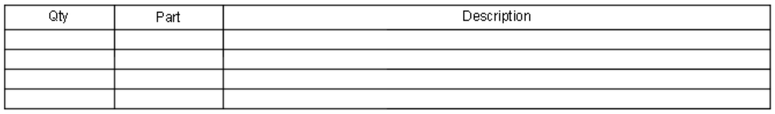
The appearance of a table is defined by a table style and cell styles. The default table style assigned to a new table is based on the active table style of a drawing. I explain how to define and manage table styles in the "Formatting Tables" section later in this chapter. You can learn to apply a named table style in the "Assigning a Table Style" section.

The following code statements add a table with five rows and three columns, and add labels to the header rows (see Figure 6.8):

```
' Defines the insertion point of the table
Dim dInsPt(2) As Double
dInsPt(0) = 5: dInsPt(1) = 2.5: dInsPt(2) = 0

' Adds the new table object
Dim oTable As AcadTable
Set oTable = ThisDrawing.ModelSpace.AddTable(dInsPt, 5, 3, 0.25, 2)

' Supresses the Title row and unmerge the cells in the first row
oTable.TitleSuppressed = True
oTable.UnmergeCells 0, 0, 0, 2
```

```
' Sets the values of the header row
oTable.SetCellValue 0, 0, "Qty"
oTable.SetCellValue 0, 1, "Part"
oTable.SetCellValue 0, 2, "Description"

' Sets the width of the third column
oTable.SetColumnWidth 2, 10
```

FIGURE 6.8
Empty BOM table



Qty	Part	Description

Due to the complexities of tables, it isn't practical to cover everything that is possible. You can merge cells, add block references to a cell, and control the formatting of individual cells with the AutoCAD Object library and VBA. If you need to work with tables, I recommend referring to the AutoCAD Help system.

Formatting Tables

Table styles—like multileader styles—are not accessed directly through a collection object like `AcadTextStyles` for text styles and `AcadDimStyles` for dimension styles. Table styles are stored in the ACAD_TABLESTYLE dictionary, which is accessed from the `AcadDictionaries` collection object. Each table style in the ACAD_TABLESTYLE dictionary is represented by an `AcadTableStyle` object.

New table styles are created with the `AddObject` function and the `GetObject` function is used to obtain an existing table style in a drawing. When using the `AddObject` function, you must specify two strings: the first is the name of the style you want to create and the second is the class name of `AcDbTableStyle`. You can learn more about working with dictionaries in Chapter 9.

The following code statements create a table style named `BOM` if it doesn't already exist:

```
On Error Resume Next

' Gets the table styles dictionary
Dim oDict As AcadDictionary
Set oDict = ThisDrawing.dictionaries.Item("ACAD_TABLESTYLE")

' If no error, continue
If Not oDict Is Nothing Then
  ' Gets the table style named BOM
  Dim oTblStyle As AcadTableStyle
  Set oTblStyle = oDict.GetObject("BOM")

  ' If an error is returned, create the multileader style
  If Err Then
    Err.Clear
    ' Creates a new table style
    Set oTblStyle = oDict.AddObject("BOM", "AcDbTableStyle")
  End If

  ' Supresses the title row and displays the header row of the table style
  oTblStyle.TitleSuppressed = True
  oTblStyle.HeaderSuppressed = False

  ' Creates a new cell style
  oTblStyle.CreateCellStyle "BOM_Header"
  oTblStyle.SetCellClass "BOM_Header", 1

  ' Sets the background color of the new cell style
  Dim oClr As AcadAcCmColor
  Set oClr = oTblStyle.GetBackgroundColor2("BOM_Header")
  oClr.ColorMethod = acColorMethodByACI
  oClr.ColorIndex = 9
  oTblStyle.SetBackgroundColor2 "BOM_Header", oClr

  ' Sets the color of the text for the cell style
  oClr.ColorIndex = acBlue
  oTblStyle.SetColor2 "BOM_Header", oClr
End If
```

A table style that is no longer needed can be removed with the `Remove` method of the `AcadDictionary` object. For more information on the properties and methods of the `AcadTableStyle` object, see the AutoCAD Help system or the Object Browser in the VBA Editor.

Assigning a Table Style

You can change the style of a table once it has been added to a drawing with the `StyleName` property. The `StyleName` property returns or accepts a string that represents the name of the current or table style to be assigned. When a new table object is created, the style applied is inherited from the `ctablestyle` system variable. You can use the `SetVariable` and `GetVariable` methods of an `AcadDocument` or a `ThisDrawing` object.

The following code statement assigns the table style named BOM to the `ctablestyle` system variable:

```
' Sets the BOM style active
ThisDrawing.SetVariable "ctablestyle", "BOM"
```

Creating Fields

Fields are used to add dynamic values to a text object based on the current value of an object's property, a drawing file property, date, system variable, table cell, and many other types of values stored in a drawing. A field can be added to a stand-alone text object, dimension, table cell, and even block attributes. Fields are implemented with the use of control codes. Typically, a field is added to a drawing using the Field dialog box displayed with the `field` command. In the lower-left corner of the Field dialog box is an area labeled Field Expression. The Field Expression area displays the text that you can assign to the `TextString` property of an annotation object or pass to the `SetCellValue` method of an `AcadTable` object to assign a value to a table cell. For example, the following is an example of the field expression used to add today's date to the drawing in an `MText` object with the `MM/dd/yyyy` format:

```
%<\AcVar Date \f "MM/dd/yyyy">%
```

To create a new `MText` object with the example field expression, the VBA code statements might look like the following:

```
' Defines the insertion point and width for the text object
Dim dInsPt(2) As Double, dWidth As Double
dInsPt(0) = 0: dInsPt(1) = 0: dInsPt(2) = 0
dWidth = 2.5

' Creates a new MText object with a field
Dim oMText As AcadMText
Set oMText = ThisDrawing.ModelSpace.AddMText(dInsPt, dWidth, _
    "%<\AcVar Date \f ""MM/dd/yyyy"">%")
```

NOTE The `fieldeval` and `fielddisplay` system variables affect when fields are evaluated and if fields are displayed with a gray background in the drawing. For more information on these system variables, see the AutoCAD Help system.

Exercise: Adding a Label to the Plate

In this section, you will continue to build on the `DrawPlate` project that was introduced in Chapter 4. Here is the key concept I cover in this exercise:

Creating an MText Object Simple and complex text strings can be added to a drawing with an `MText` object. A single-line text object can also be used to add descriptive text or a label to a drawing.

NOTE The steps in this exercise depend on the completion of the steps in the "Exercise: Getting Input from the User to Draw the Plate" section of Chapter 5, "Interacting with the User and Controlling the Current View." If you didn't complete the steps, do so now or start with the `ch06_drawplate.dvb` sample file available for download from `www.sybex.com/go/autocadcustomization`. Place the sample file in the `MyCustomFiles` folder within the `Documents` (or `My Documents`) folder, or the location where you are storing the DVB files. Also, remove `ch06_` from the filename before you begin working.

Revising the *CLI_DrawPlate* Function

These changes to the `CLI_DrawPlate` function add an `MText` object to display a basic label for the plate drawn. In the following steps you will update code statements in the `CLI_DrawPlate` function of the `drawplate.dvb` project file:

1. Load the `drawplate.dvb` file into the AutoCAD drawing environment and display the VBA Editor.
2. In the VBA Editor, in the Project Explorer, double-click the `basDrawPlate` component.
3. In the code editor window, scroll to the bottom of the `CLI_DrawPlate` function, locate the following code statements, and add the code statements shown in boldface:

```
    ' Calculate and place the circle in the upper-left
      ' corner of the rectangle.
      dAng = myUtilities.Atn2(dPtList(6) - dPtList(0), _
                              dPtList(7) - dPtList(1))
      cenPt4 = ThisDrawing.Utility.PolarPoint(cenPt1, dAng, dDist - 1)
      myUtilities.CreateCircle cenPt4, 0.1875

      ' Get the insertion point for the text label
      Dim insPt As Variant
      insPt = Null

      insPt = ThisDrawing.Utility.GetPoint(, _
              removeCmdPrompt & "Specify label insertion point " & _
                                "<or press Enter to cancel placement>: ")

      ' If a point was specified, placed the label
      If IsNull(insPt) = False Then
        ' Define the label to add
        Dim sTextVal As String
        sTextVal = "Plate Size: " & _
                   Format(ThisDrawing.Utility. _
                     RealToString(width, acDecimal, 4), "0.0###") & _
                   "x" & _
                   Format(ThisDrawing.Utility. _
                     RealToString(height, acDecimal, 4), "0.0###")

        ' Create label
        Set oLyr = myUtilities.CreateLayer("Label", acWhite)
        ThisDrawing.ActiveLayer = oLyr

        myUtilities.CreateText insPt, acAttachmentPointMiddleCenter, _
                               0.5, 0#, sTextVal
      End If
    End If
  Loop Until IsNull(basePt) = True And sKeyword = ""

  ' Restore the saved system variable values
  myUtilities.SetSysvars sysvarNames, sysvarVals
```

4. Click File ➢ Save.

Revising the *Utilities* Class

These changes to the `Utilities` class introduce a new function named `CreateText`. The `CreateText` function consolidates the creation of an `MText` object and the setting of specific properties and returns an `AcadMText` object. In the following steps you will add the constant value and two functions to the `clsUtilities` class module:

1. In the VBA Editor, in the Project Explorer, double-click the `clsUtilities` component.
2. Scroll to the bottom of the code editor window and click to the right of the last code statement. Press Enter twice.
3. Type the following code statements; the comments are here for your information and don't need to be typed:

```
' CreateText function draws a MText object.
' Function expects an insertion point, attachment style,
' text height and rotation, and a string.
Public Function CreateText(insPoint As Variant, _
                           attachmentPt As AcAttachmentPoint, _
                           textHeight As Double, _
                           textRotation As Double, _
                           textString As String) As AcadMText
  Set CreateText = ThisDrawing.ActiveLayout.Block. _
                   AddMText(insPoint, 0, textString)

  ' Sets the text height, attachment point, and rotation of the MText object
  CreateText.height = textHeight
  CreateText.AttachmentPoint = attachmentPt
  CreateText.insertionPoint = insPoint
  CreateText.rotation = textRotation
End Function
```

4. Click File ➢ Save.
5. Export the `clsUtilities` class model from the `drawplate.dvb` file to a file named `clsUtilities.cls` in the `MyCustomFiles` folder, as explained in Chapter 5.

Using the Revised *drawplate* Function

Now that the `drawplate.dvb` project file has been revised, you can test the changes that have been made. The following steps explain how to use the revised `drawplate` function:

1. Switch to AutoCAD by clicking on its icon in the Windows taskbar or click View ➢ AutoCAD from the menu bar in the Visual Basic Editor.
2. In AutoCAD, at the Command prompt, type **`vbarun`** and press Enter.
3. When the Macros dialog box opens, select the `DrawPlate.dvb!basDrawPlate.CLI_DrawPlate` macro from the list and click Run.

4. At the `Specify base point for the plate or [Width/Height]:` prompt, pick a point in the drawing area to draw the plate and holes based on the width and height values specified.
5. At the `Specify label insertion point <or press Enter to cancel placement>:` prompt, pick a point below the plate to place the label.
6. Press Enter to exit the macro when you are done.

Chapter 7

Working with Blocks and External References

Most designs created with the AutoCAD® program start off with simple geometric objects, such as lines, circles, and arcs. The geometric objects are used to represent holes, bolts, motors, and even the outside of a building. As a design grows in complexity, elements often are repeated many times. For example, you might use several lines and circles to represent a bolt head or a desk with a grommet.

AutoCAD allows you to reuse geometry by creating what is known as a *block*. A block is a named grouping of objects that can be inserted in a drawing multiple times. Each insertion creates a *block reference* that displays the objects stored in the block at the insertion point. If the block is changed, each block reference based on that block is updated.

Blocks aren't the only method for reusing geometry or other data in a drawing. A drawing file can also include external references *(xrefs)* to geometry stored in another drawing file. External references can include blocks, raster images, and other documents. When you reference another document, such as a PDF or DWF file, it is known as an *underlay*. In this chapter, I explain how to use VBA to work with blocks and external referenced files.

Managing Block Definitions

Blocks make it possible to logically group basic geometry together with a unique name and then create instances of that geometry within a drawing. Blocks are implemented as two separate objects: block definitions and block references. Block definitions are nongraphical objects that are stored in the `AcadBlocks` collection object. Each block definition is represented by an `AcadBlock` object, which contains the geometry and attribute definitions that define how the block should appear and behave when it is inserted into the drawing area. A block definition can contain either static or dynamic properties.

Figure 7.1 shows the relationship between a block definition and a block reference and how the attributes of the block are used to bring the geometry into model space.

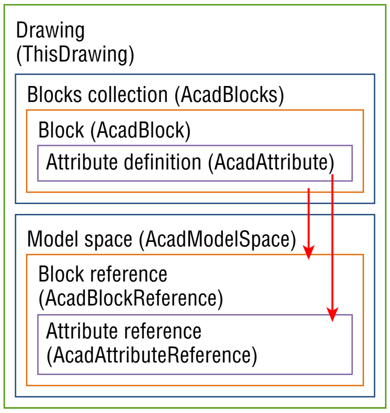


Figure 7.1
Block-definition-to-block-reference relationship

You can think of a block definition much like a cookie recipe. The recipe lists the ingredients (which determines how the cookie should taste) and provides instructions for combining those ingredients and baking the dough. What the recipe doesn't control is how much dough is placed on any particular spot on the cookie sheet before baking. The exact placement and amount of the cookie dough on the tray is determined by the baker. Similarly, an end user uses a block reference in a drawing to determine the exact placement, size, and number of geometries to be displayed. I explain how to insert and work with block references in the "Inserting and Working with Block References" section later in this chapter.

Creating a Block Definition

A new block definition can be added to a drawing using the `Add` function of the `AcadBlocks` collection object. The `Add` function expects two argument values and returns an `AcadBlock` object. Before adding a new block definition, you should use the `Item` method with an error handler to check to see if a block definition with the specific name you want to use already exists in the drawing. The `Item` method can also be used to get a specific block definition in the `AcadBlocks` collection object and, as with other collections, a `For` statement can be used to step through the block definitions in a drawing.

The following shows the syntax of the `Add` function:

```
retVal = object.Add(origin, blockName)
```

Its arguments are as follows:

`retVal` The *`retVal`* argument represents the new `AcadBlock` collection object returned by the `Add` function.

`object` The *`object`* argument specifies the `AcadBlocks` collection object that is used to add a new block definition.

`origin` The *`origin`* argument is an array of three doubles that defines the origin of the new block definition; think insertion point for the block reference.

`blockName` The *`blockName`* argument is a string that specifies the unique name to be assigned to the new block definition.

The following code statements add a new block definition named RoomNum:

```
' Defines the origin of the block
Dim dOrigin(2) As Double
dOrigin(0) = 5: dOrigin(1) = 2.5: dOrigin(2) = 0

Dim oBlkDef As AcadBlock
Set oBlkDef = ThisDrawing.Blocks.Add(dOrigin, "RoomNum")
```

Here is an example that checks for the existence of a block definition named RoomNum:

```
On Error Resume Next

Dim oBlkDef As AcadBlock
Set oBlkDef = ThisDrawing.Blocks("RoomNum")

If Err Then
  MsgBox "Block definition doesn't exist."
Else
  MsgBox "Block definition exists."
End If
```

After an AcadBlock object has been obtained using the Item or Add method of the AcadBlocks collection object, you can step through the objects of the block using a For statement or the Item method of the AcadBlock object. You can use the same functions to add new objects to a block definition as you would to add objects to model space or paper space. I explained how to add objects to and step through the objects of model space in Chapter 4, "Creating and Modifying Drawing Objects."

The following code statements add a closed lightweight polyline to the block definition named RoomNum:

```
On Error Resume Next

' Gets the RoomNum block definition
Dim oBlkDef As AcadBlock
Set oBlkDef = ThisDrawing.Blocks("RoomNum")

' If the block doesn't exist, an error is generated
If Err Then
  ' Defines the origin of the block
  Dim dOrigin(2) As Double
  dOrigin(0) = 5: dOrigin(1) = 2.5: dOrigin(2) = 0

  ' Adds the block definition
  Set oBlkDef = ThisDrawing.Blocks.Add(dOrigin, "RoomNum")

  ' Defines the vertex points for the
  ' lightweight polyline
  Dim dPts(7) As Double
  dPts(0) = 0:  dPts(1) = 0
```

```
    dPts(2) = 10: dPts(3) = 0
    dPts(4) = 10: dPts(5) = 5
    dPts(6) = 0:  dPts(7) = 5

    ' Adds a new lightweight polyline to the block
    Dim oLWPoly As AcadLWPolyline
    Set oLWPoly = oBlkDef.AddLightWeightPolyline(dPts)

    ' Closes the lightweight polyline
    oLWPoly.Closed = True

    ' Sets the layer of the lightweight polyline to 0
    oLWPoly.Layer = "0"
  End If
```

NOTE I recommend placing objects in a block definition on layer 0 when the object should inherit its properties from the layer that the block reference is inserted onto. The appearance of objects in a block definition can be controlled `ByBlock` or `ByLayer` when you insert a block definition as a block reference. For information on the `ByBlock` and `ByLayer` values, see the AutoCAD Help system.

When a block definition is no longer needed, it can be removed from the drawing using the `Delete` method of an `AcadBlock` object. A block definition can't be deleted if a block reference associated with the block definition is inserted into the drawing.

NOTE Dynamic blocks—block definitions with dynamic properties—can't be created with the AutoCAD Object library. However, you can modify the dynamic properties of a block reference using the AutoCAD Object library. I explain how to work with blocks that contain dynamic properties in the "Working with Dynamic Properties" section later in this chapter.

Adding Attribute Definitions

A block definition can contain what is known as an *attribute*. An attribute is similar to a text object, except its value can be changed after a block reference has been inserted into a drawing. Attributes allow you to store string values and then extract their values later. There are two types of attributes that are part of the block creation and insertion process: attribute definitions and attribute references. Attribute definitions can be added to a block definition, and attribute references are part of each block reference inserted into a drawing that is associated with a block definition that has one or more attribute definitions.

The `AddAttribute` function is used to add an attribute definition to a block definition and returns an `AcadAttribute` object. The following shows the syntax of the `AddAttribute` function:

```
retVal = object.AddAttribute(height, mode, prompt, insertionPoint, tag, value)
```

Its arguments are as follows:

`retVal` The *`retVal`* argument represents the new `AcadAttribute` object returned by the `AddAttribute` function.

`object` The *`object`* argument specifies the `AcadBlock` object to add the attribute definition.

height The *height* argument is a double that represents the height of the attribute.

mode The *mode* argument is an integer that represents the behavior of the attribute reference added to a block reference when the block is inserted into the drawing. Instead of using an integer value, I recommend that you use the constant values of the AcAttributeMode enumerator. Table 7.1 lists each of the constant values of the AcAttributeMode enumerator. You can specify more than one constant by separating each constant with a plus symbol, such as acAttributeModeInvisible + acAttributeModeNormal.

prompt The *prompt* argument is a string that represents the text that provides a hint for the value that's expected when the block reference is inserted.

insertionPoint The *insertionPoint* argument is an array of three doubles that defines the insertion point of the attribute definition.

tag The *tag* argument is a string that represents the text that's displayed in the drawing if the block reference is exploded after being inserted and the value used to extract the attribute's value from a block reference. A tag cannot contain spaces.

value The *value* argument is a string that represents the default value of the attribute when the block reference is inserted.

Table 7.1 lists the constant values of the AcAttributeMode enumerator that can be passed to the mode argument of the AddAttribute function or assigned to the Mode property of an AcadAttribute object.

Table 7.1: Constant values of the AcAttributeMode enumerator

Constant	Description
acAttributeModeConstant	Indicates the value of the attribute can't be changed.
acAttributeModeInvisible	Attribute is marked as invisible. The attmode system variable controls the display of all invisible attributes.
acAttributeModeLockPosition	Position of the attribute can't be adjusted using grip editing.
acAttributeModeMultipleLine	Attribute supports multiple lines of text instead of the standard single line of text.
acAttributeModeNormal	Default display behavior of the attribute is maintained when the block is inserted using the insert command.
acAttributeModePreset	Value of the attribute is preset. When the block is inserted using the insert command, the user isn't prompted to enter a value for the attribute.
acAttributeModeVerify	User is prompted to verify the value they provide for the attribute when inserting the block reference with the insert command.

The following code statements add an attribute definition to the block definition assigned to the *oBlkDef* variable:

```
' Defines the insertion point of the attribute definition
Dim dInsPt(2) As Double
dInsPt(0) = 5: dInsPt(1) = 2.5: dInsPt(2) = 0

' Adds the attribute definition to the block
Dim oAttDef As AcadAttribute
Set oAttDef = oBlkDef.AddAttribute(2.5, acAttributeModeNormal, _
                                   "Room#", dInsPt, "Room#", "101")

' Sets the alignment for the attribute's text
oAttDef.Alignment = acAlignmentMiddleCenter
oAttDef.TextAlignmentPoint = dInsPt
```

After adding an attribute definition to a block definition, you can modify its appearance and behavior using the properties and methods of the `AcadAttribute` object. The properties and methods of an `AcadAttribute` object are similar to those of an `AcadText` object. I discussed the `AcadText` object in Chapter 6, "Annotating Objects."

Table 7.2 lists the properties of the `AcadAttribute` object that are unique to the object and are different from those of an `AcadText` object.

TABLE 7.2: Properties related to an `AcadAttribute` object

PROPERTY	DESCRIPTION
`Constant`	Returns `True` if the attribute is set to the constant mode.
`Invisible`	Returns `True` if the attribute should be invisible when the block reference is inserted.
`LockPosition`	Returns `True` if the attribute can't be modified using grip editing when the block reference is inserted.
`MTextAttribute`	Returns `True` if the attribute should be multiline instead of single-line text.
`MTextAttributeContent`	Specifies the content for the multiline text when the `MTextAttribute` property is `True.`
`MTextBoundaryWidth`	Specifies the width of the multiline text when the `MTextAttribute` property is `True.`

Property	**Description**
`MTextDrawingDirection`	Specifies the direction in which the text should be drawn when the `MTextAttribute` property is `True`.
`Preset`	Returns `True` if the user shouldn't be prompted to enter a value for the attribute when the block is inserted.
`PromptString`	Specifies the prompt string that is displayed for the attribute when the block is inserted.
`TagString`	Specifies the tag for the attribute that is displayed in the drawing if the block reference is exploded after being inserted. The tag can also be useful when trying to identify which attribute's value to extract when generating a BOM.
`TextString`	Specifies the default text for the attribute to use when the block is inserted.
`Verify`	Returns `True` if the user should be prompted to verify the value of the attribute when the block is inserted.

NOTE If you change the `MTextAttributeContent`, `MTextBoundaryWidth`, or `MTextDrawingDirection` property, you must execute the `UpdateMTextAttribute` method of the `AcadAttribute` object. The `UpdateMTextAttribute` method updates the display of the multiline attribute in the drawing.

Modifying and Redefining a Block Definition

You can add new objects or modify existing objects of a block definition, much like you can in model space or paper space. The `Item` method of the `AcadBlock` object can be used to get a specific object or a `For` statement to step through all objects in a block definition.

In addition to modifying the objects in a block definition, the origin—the insertion point of a block—can be modified using the `Origin` property. The `Origin` property of an `AcadBlock` object allows you to get or set the origin for the block definition. The origin of a block definition is defined as a double array with three elements.

Besides modifying the objects and origin of a block, you can specify whether a block reference created from a block can be exploded, set the units that control the scaling of a block, and make other changes. Table 7.3 lists the properties that control the behavior of and provide information about an `AcadBlock` object.

Table 7.3: Properties related to an `AcadBlock` object

Property	Description
`BlockScaling`	Specifies if the block can only be scaled uniformly (`acUniform`) or the block can be assigned a different scale factor for each axis (`acAny`).
`Comments`	Specifies a string that describes the block definition.
`Count`	Returns an integer value that contains the number of block references that have been inserted into the drawing based on the block definition.
`Explodable`	Returns `True` if the block reference inserted into the drawing can be exploded.
`Name`	Specifies a string that contains the name of the block definition.
`Units`	Specifies the units of measurement for the block. A constant value from the `AcInsertUnits` enumerator is returned by or can be assigned to this property. See the Object Browser in the VBA Editor for a list of possible values. The units specified affect the insertion scale of the block.

After a block definition has been updated, you should use the `Regen` method of the `AcadApplication` object to regenerate the display of the drawing. You can also update the display of an individual block reference using the `Update` method. I explained the `Regen` and `Update` methods in Chapter 4. If you add or remove an `AcadAttribute` object in a block definition, the block references inserted into model space or paper space aren't updated to reflect the change unless the attribute being changed is defined as a constant.

To reflect the changes in the attributes between a block definition and the block references inserted into a drawing, you will need to do the following:

1. Insert a new block reference into the drawing.
2. Update the attribute values of the new block reference with those from the existing block reference.
3. Remove the old block reference.

I discuss more about working with block references in the "Inserting and Working with Block References" section later in this chapter.

Determining the Type of Block Definition

Block definitions stored in the `AcadBlocks` collection object of a drawing aren't only used to insert block references. Model space and paper space are also block definitions with special names along with external references (xrefs) and layouts. You can determine a block definition's type using the properties in Table 7.4. I discuss xrefs in the "Working with Xrefs" section later in this chapter and layouts in Chapter 8, "Outputting Drawings."

Table 7.4: Properties used to determine a block definition's type

Property	**Description**
`IsDynamicBlock`	Returns `True` if the block definition contains dynamic properties.
`IsLayout`	Returns `True` if the block definition is for a layout. You can use the `Layout` property of an `AcadBlock`, `AcadModelSpace`, or `AcadPaperSpace` object to get the object's associated `AcadLayout` object.
`IsXref`	Returns `True` if the block definition is for an external reference.

Inserting and Working with Block References

A block reference is an instance—not a copy—of the geometry from a block definition; the geometry only exists as part of the block definition, with the exception of attributes. Attribute definitions that are part of a block definition are added to a block reference as attribute references unless the attribute definition is defined as a constant attribute. Constant attributes are part of the geometry that is inherited from a block definition and are not part of the block reference.

Inserting a Block Reference

The `InsertBlock` function is used to insert a reference of a block definition in model space, paper space, or another block definition and expects seven argument values that define the block definition you want to insert, as well as the placement and size of the block reference. After a block reference has been inserted, an `AcadBlockReference` object is returned.

The following shows the syntax of the `InsertBlock` function:

```
retVal = object.InsertBlock(insertionPoint, blockName, xScale, yScale,
                            zScale, rotation [, password])
```

Its arguments are as follows:

retVal The *retVal* argument represents the new `AcadBlockReference` object returned by the `InsertBlock` function.

object The *object* argument specifies the `AcadBlock`, `AcadModelSpace`, or `AcadPaperSpace` object where the block reference is to be inserted.

insertionPoint The *insertionPoint* argument is an array of doubles that represents the insertion point of the block reference.

blockName If you wish to insert the reference into a block definition, use the *blockName* argument (a string) to specify the name of that block definition. The block must already be defined in the drawing before the insertion can be executed. If you wish to insert a DWG file as a reference into a drawing, specify the full path of a DWG file. When a DWG file is specified, an AcadBlock object based on the objects in model space of the DWG file specified is created, and then the block reference is inserted.

NOTE An error is generated if the block definition being inserted doesn't already exist in the drawing. You can catch the error and use the Add method to create the block definition or specify a DWG file to insert that might contain the objects for the block you want to use.

For example, the following inserts a block named grid.dwg from the location c:\symbols:

```
Set oBlkRef = ThisDrawing.ModelSpace.InsertBlock( _
                insPt, "c:\symbols\grid.dwg", 1, 1, 1, 0)
```

xScale*, *yScale*, and *zScale The *xScale*, *yScale*, and *zScale* arguments are doubles that represent the scale factors of the block reference.

rotation The *rotation* argument is a double that represents the rotation angle of the block reference. The rotation angle must be expressed in radians.

password The *password* argument is an optional string that represents the password assigned to restrict the drawing file from being opened or inserted into by unapproved users. This argument is only required if you are inserting a block based on a DWG file that is password protected.

The following code statements insert a block reference based on a block named RoomNum at 15,27. (Remember, the block you name in code like this must already be defined in your drawing before the code can be executed. Be sure to use an error handler to add the block if it doesn't exist.)

```
' Defines the insertion point
Dim insPt(2) As Double
insPt(0) = 15: insPt(1) = 27: insPt(2) = 0

' Defines the name of the block
Dim blkName As String
blkName = "RoomNum"

' Inserts the block reference
Dim oBlkRef As AcadBlockReference
Set oBlkRef = ThisDrawing.ModelSpace.InsertBlock( _
                 insPt, blkName, 1, 1, 1, 0)
```

Modifying a Block Reference

Once a block reference, an AcadBlockReference object, is inserted into a drawing, you can modify it using the methods and properties inherited from the AcadEntity object and those specific to the AcadBlockReference object. I explained how to use the methods and properties of the AcadEntity object in Chapter 4. Table 7.5 lists the properties that are used to change the placement, rotation, and scale of a block reference.

Table 7.5: Properties used to affect a block reference

Property	**Description**
InsertionPoint	Specifies the insertion point of the block reference in the drawing and is an array of doubles
InsUnits	Returns a string that represents the insertion units saved with the block
InsUnitsFactor	Returns the insertion factor that is based on the insertion units of the block and those of the drawing
Rotation	Specifies the rotation of the block reference
XEffectiveScaleFactor	Specifies the effective scale factor along the X-axis for the block reference
XScaleFactor	Specifies the scale factor along the X-axis for the block reference
YEffectiveScaleFactor	Specifies the effective scale factor along the Y-axis for the block reference
YScaleFactor	Specifies the scale factor along the Y-axis for the block reference
ZEffectiveScaleFactor	Specifies the effective scale factor along the Z-axis for the block reference
ZScaleFactor	Specifies the scale factor along the Z-axis for the block reference

Block references also support the ability to be exploded. The Explode method is used to explode a block reference and it returns an array of the objects added to the drawing as a result of exploding the block reference. The objects in the array are copies of the objects from the block definition associated with the block reference. When the Explode method is executed, the block reference isn't removed from the drawing. You must decide what to do with the block reference. Typically, the block reference is removed using the Delete method, while the objects from the Explode method are kept.

The following code statements explode the first block reference located in model space and then list the block definition name and the objects that make up the block definition:

```
Sub ExplodeFirstBlkRef()
  Dim oBlkRef As AcadBlockReference
  Dim oObj As Object

  ' Step through model space
  For Each oObj In ThisDrawing.ModelSpace
    ' If a block reference is found, explode it
    If TypeOf oObj Is AcadBlockReference Then
      Set oBlkRef = oObj

      ' Explode the block reference
      Dim vObjArray As Variant
      vObjArray = oBlkRef.Explode

      ' List the objects that were added
      ThisDrawing.Utility.Prompt vbLf & "Block exploded: " & _
                                 oBlkRef.Name & vbLf

      ThisDrawing.Utility.Prompt vbLf & "Objects added: " & _
                                 vbLf

      ' Remove the block reference
      oBlkRef.Delete

      Dim oAcadObj As AcadObject
      Dim oObjFromBlkRef As Variant

      For Each oObjFromBlkRef In vObjArray
        Set oAcadObj = oObjFromBlkRef

        ThisDrawing.Utility.Prompt "  " & oAcadObj.ObjectName & _
                                   vbLf
      Next oObjFromBlkRef

      ' Exit the For statement since we are interested
      ' in the first block reference only
      Exit For
    End If
  Next oObj
End Sub
```

Here is an example of the output from the previous example code:

```
Block exploded: 2x4x8
Objects added:
  AcDbPolyline
  AcDbLine
  AcDbLine
```

Accessing the Attributes of a Block

When a block reference is first inserted into a drawing, the default values of all attributes are used. The value of each nonconstant attribute of a block reference can be changed. Before you access the attributes of a block reference, you should make sure the block reference has attributes. The `HasAttributes` property of an `AcadBlockReference` object returns `True` if a block reference has attributes, either constant or nonconstant.

The `GetAttributes` and `GetConstantAttributes` functions of an `AcadBlockReference` object are used to access the attributes of a block reference. Neither function accepts any arguments. The `GetAttributes` function returns an array of `AcadAttributeReference` objects that aren't defined as constant attributes attached to a block reference, whereas the `GetConstantAttributes` function returns an array of `AcadAttribute` objects.

Listing 7.1 is a custom procedure that demonstrates how to get both the attributes and constant attributes attached to a block reference.

LISTING 7.1: Lists attribute tags and values of a block reference

```
Sub ListBlockAtts()
  ' Prompt the user to select a block reference
  Dim oObj As Object
  Dim vPtPicked As Variant
  ThisDrawing.Utility.GetEntity oObj, vPtPicked, vbLf & _
                              "Select a block reference: "

  ' Check to see if the entity selected is a
  ' block reference
  If TypeOf oObj Is AcadBlockReference Then
    Dim oBlkRef As AcadBlockReference
    Set oBlkRef = oObj

    ' Output information about the block
    ThisDrawing.Utility.Prompt vbLf & "*Block Reference*" & _
                               vbLf & "  Block name: " & _
                               oBlkRef.Name & vbLf

    ' Check to see if the block reference has attributes
    If oBlkRef.HasAttributes = True Then
      Dim oAttRef As AcadAttributeReference
      Dim oAttDef As AcadAttribute
      Dim vObj As Variant

      ' Gets the nonconstant attributes
      ThisDrawing.Utility.Prompt vbLf & "*Nonconstant Attributes*"

      Dim vArAtts As Variant
```

```
        vArAtts = oBlkRef.GetAttributes

        ' Steps through the nonconstant attributes
        If UBound(vArAtts) > -1 Then
          For Each vObj In vArAtts
            Set oAttRef = vObj

            ' Outputs the tag and text of the attribute
            ThisDrawing.Utility.Prompt vbLf & "  Tag: " & _
                                       oAttRef.TagString & _
                                       vbLf & "  Value: " & _
                                       oAttRef.TextString
          Next vObj
        Else
          ThisDrawing.Utility.Prompt vbLf & "  None"
        End If

        ' Gets the nonconstant attributes
        ThisDrawing.Utility.Prompt vbLf & "*Constant Attributes*"

        ' Gets the constant attributes
        vArAtts = oBlkRef.GetConstantAttributes

        ' Steps through the constant attributes
        If UBound(vArAtts) > -1 Then
          For Each vObj In vArAtts
            Set oAttDef = vObj

            ' Outputs the tag and text of the attribute
            ThisDrawing.Utility.Prompt vbLf & "  Tag: " & _
                                       oAttDef.TagString & _
                                       vbLf & "  Value: " & _
                                       oAttDef.TextString
          Next vObj
        Else
          ThisDrawing.Utility.Prompt vbLf & "  None"
        End If

        ThisDrawing.Utility.Prompt vbLf
      End If
    End If
  End Sub
```

Here is an example of the output generated by the custom `ListBlockAtts` procedure from Listing 7.1:

```
*Block Reference*
  Block name: RoomNumber
*Nonconstant Attributes*
```

```
  Tag: ROOM#
  Value: 101
*Constant Attributes*
  None
```

In addition to listing the values of the attributes attached to a block reference, you can modify the appearance and placement of the attribute references returned by the `GetAttributes` function. If you make changes to an attribute reference, make sure to execute the `Update` method and regenerate the display of the object. The `AcadAttributeReference` and `AcadAttribute` objects are nearly identical. However, the `AcadAttributeReference` object doesn't support the `Mode`, `Preset`, `PromptString`, or `Verify` property.

Working with Dynamic Properties

Most block references display a single set of geometry, meaning that the objects that are included in the block definition are the only ones that can be shown in the drawing. Starting with AutoCAD 2006, block definitions were extended to support what are known as *dynamic properties*. Block definitions with dynamic properties are known as *dynamic blocks*. You can't create dynamic blocks with the AutoCAD Object library, but you can modify the custom properties of a dynamic block after it is inserted into a drawing. For information on how to create a dynamic block, see the topic "About Dynamic Blocks" in the AutoCAD Help system.

The `IsDynamicBlock` property of the `AcadBlockReference` object can be used to determine whether a block reference has dynamic properties. When the `IsDynamicBlock` property returns `True`, the block reference has dynamic properties that can be queried and modified.

Once you have verified that a block reference has dynamic properties, you use the `GetDynamicBlockProperties` function to get an array of `AcadDynamicBlockReferenceProperty` objects. The `Value` property of an `AcadDynamicBlockReferenceProperty` object is used to get and set the value of a dynamic property, whereas the `PropertyName` property returns a string that represents the name of the dynamic property.

Listing 7.2 is a custom procedure that demonstrates how to get the custom properties and their values of a block reference named `Door - Imperial`. You can insert the `Door - Imperial` block reference using the block tool on the Architectural tab of the Tool Palettes window (displayed using the `toolpalettes` command).

Listing 7.2: Listing custom properties and values of a block reference

```
Sub ListCustomProperties()
  ' Prompt the user to select a block reference
  Dim oObj As Object
  Dim vPtPicked As Variant
  ThisDrawing.Utility.GetEntity oObj, vPtPicked, vbLf & _
                              "Select a block reference: "

  ' Check to see if the entity selected is a
  ' block reference
  If TypeOf oObj Is AcadBlockReference Then
    Dim oBlkRef As AcadBlockReference
    Set oBlkRef = oObj
```

```
' Output information about the block
ThisDrawing.Utility.Prompt vbLf & "*Block Reference*" & _
                           vbLf & "  Block name: " & _
                           oBlkRef.Name & vbLf

' Check to see if the block reference has dynamic properties
If oBlkRef.IsDynamicBlock = True Then
  Dim oDynProp As AcadDynamicBlockReferenceProperty
  Dim vObj As Variant

  ' Gets the block reference's dynamic properties
  ThisDrawing.Utility.Prompt vbLf & "*Dynamic Properties*"

  Dim vDynProps As Variant
  vDynProps = oBlkRef.GetDynamicBlockProperties

  ' Steps through the dynamic properties
  If UBound(vDynProps) > -1 Then
    For Each vObj In vDynProps
      Set oDynProp = vObj

      ' Outputs the property name and value
      Dim sValue As String

      If IsArray(oDynProp.Value) = False Then
        sValue = CStr(oDynProp.Value)
      Else

        For Each vVal In oDynProp.Value
          If sValue <> "" Then sValue = sValue & ","

          sValue = sValue & CStr(vVal)
        Next vVal
      End If

      ThisDrawing.Utility.Prompt vbLf & "  Property Name: " & _
                                 oDynProp.PropertyName & _
                                 vbLf & "  Value: " & _
                                 sValue

      sValue = ""
    Next vObj
  Else
    ThisDrawing.Utility.Prompt vbLf & "  None"
  End If
```

```
        ThisDrawing.Utility.Prompt vbLf
      End If
    End If
  End Sub
```

Here is an example of the output generated by the custom `ListCustomProperties` procedure from Listing 7.2:

```
*Block Reference*
  Block name: *U3
*Dynamic Properties*
  Property Name: Door Size
  Value: 40
  Property Name: Origin
  Value: 0,0
  Property Name: Wall Thickness
  Value: 6
  Property Name: Origin
  Value: 0,0
  Property Name: Hinge
  Value: 0
  Property Name: Swing
  Value: 0
  Property Name: Opening Angle
  Value: Open 30º
```

When a user manipulates a grip associated with a dynamic property, onscreen it looks like the user is manipulating the block reference through a stretching, arraying, moving, or other action. The action that is performed by the user results in the creation of a new anonymous block definition. An anonymous block is a block that can't be inserted into a drawing but that is used as a way to let AutoCAD create and manage unique blocks.

NOTE The name of a block reference is typically obtained using the `Name` property, but with dynamic blocks the `Name` property might return an anonymous block name such as `*U8`. An anonymous block name is created as a result of manipulating one of the grips associated with a dynamic property on a block reference. To get the original name of the block definition for a dynamic block, you use the `EffectiveName` property.

You can convert a dynamic block to an anonymous block without dynamic properties using the `ConvertToAnonymousBlock` method or a new block definition using the `ConvertToStaticBlock` method. The `ConvertToStaticBlock` method expects a string that represents the name of the new block definition.

The appearance and custom properties of a dynamic block can be reset to their default values. To reset the appearance and custom properties of a dynamic block, you use the

ResetBlock method of an AcadBlockReference object. The ResetBlock method doesn't accept any argument values and doesn't return a value.

Managing External References

AutoCAD allows you to create what is known as an *external reference*. An external reference is a reference to a file that is stored outside of a drawing file. The contents in an external file can be another drawing, a raster or vector image, or even a file that supports Object Linking and Embedding (OLE). OLE allows you to embed, among other things, a Word document or an Excel spreadsheet into a drawing file. In addition to referencing a file, you can import objects into a drawing using OLE. An OLE object is represented by the AcadOle object in the AutoCAD Object library. You can modify, but not create, an OLE object with the AutoCAD Object library. I discuss how to import objects or a file in Chapter 8.

You can see which files are externally referenced to a file by accessing the items of the AcadFileDependencies collection object. Each file that a drawing is dependent on to correctly display objects is part of the AcadFileDependencies collection object. I mention the AcadFileDependencies collection object in the "Listing File Dependencies" section later in this chapter.

Working with Xrefs

An external drawing file referenced into a drawing is known as an *xref*. Xrefs are similar to blocks because they allow for the reuse of geometry in any drawing with one distinct difference. The difference that sets blocks and xrefs apart is that any changes made to the objects in the external drawing file are reflected in any drawings that reference the file. Xrefs are frequently used in architectural and civil engineering drawings to reference a floor plan or survey drawing. An xref is represented by an AcadExternalReference object and is similar to an AcadBlockReference object except in the way that the object can be modified and managed.

Attaching an Xref

An xref is attached to a drawing, not inserted like a block or added like other graphical objects. The AttachExternalReference function returns an AcadExternalReference object and expects nine argument values that define which file to attach, as well as the placement and size of the xref. When an xref is attached to a drawing, an AcadBlock object is created. The AcadBlock object contains the geometry that is in the referenced drawing file, but objects can't be added or modified in that AcadBlock object. Figure 7.2 shows the flow of data that takes place when a drawing file is attached to a drawing and an xref is placed in model space.

The following shows the syntax of the AttachExternalReference function:

```
retVal = object.AttachExternalReference(fileName, xrefName, insertionPoint,
                                        xScale, yScale, zScale, rotation,
                                        overlay [, password])
```

Its arguments are as follows:

retVal The *retVal* argument represents the new AcadExternalReference object returned by the AttachExternalReference function.

object The *object* argument specifies the AcadBlock, AcadModelSpace, or AcadPaperSpace object where you wish to attach the xref.

fileName The *fileName* argument is a string that represents the name of the external DWG file you want to reference.

xrefName The *xrefName* argument is a string that represents the name you want to assign to the AcadBlock object that is added to the drawing.

insertionPoint The *insertionPoint* argument is an array of doubles that represents the insertion point of the xref.

xScale*, *yScale*, and *zScale The *xScale*, *yScale*, and *zScale* arguments are doubles that represent the scale factors of the xref.

rotation The *rotation* argument is a double that represent the rotation angle of the xref. The rotation angle must be expressed in radians.

overlay The *overlay* argument is a Boolean that represents the reference type for the xref. There are two reference types: attachment and overlay. The reference types don't affect the current drawing unless the drawing is referenced into another drawing. An attachment reference type allows the xref to be displayed in other drawings that reference the drawing that contains the xref, whereas an overlay reference restricts the xref to be displayed only in the drawing to which it is attached. Use a value of True to specify an overlay reference type.

password The *password* argument is an optional string that represents the password assigned to restrict the drawing file from being opened or referenced by unapproved users.

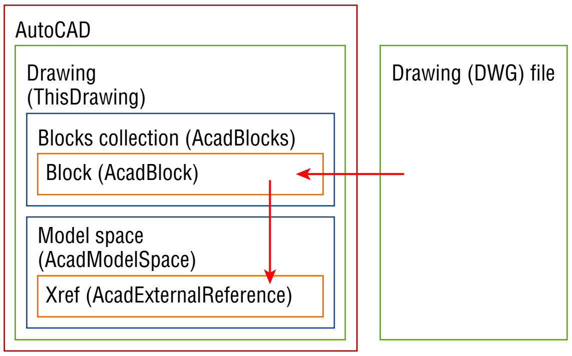

AutoCAD
Drawing (ThisDrawing)
Blocks collection (AcadBlocks)
Block (AcadBlock)
Model space (AcadModelSpace)
Xref (AcadExternalReference)
Drawing (DWG) file

FIGURE 7.2
Xref attachment flow

The following code statements add an xref based on the Ch07_Building_Plan.dwg file at 0,0 and set the reference type to attachment:

```
' Defines the insertion point
Dim insPt(2) As Double
insPt(0) = 0: insPt(1) = 0: insPt(2) = 0

' Defines the path to the drawing file
```

```
Dim dwgName As String
dwgName = ThisDrawing.GetVariable("MyDocumentsPrefix") & _
            "\MyCustomFiles\Ch07_Building_Plan.dwg"

' Adds the xref
Dim oXref As AcadExternalReference
Set oXref = ThisDrawing.ModelSpace.AttachExternalReference( _
               dwgName, "Building_Plan", insPt, 1, 1, 1, 0, False)
```

TIP The objects of all attached xrefs can be faded using the `xdwgfadectl` and `xfadectl` system variables. Use the `SetVariable` method of an `AcadDocument` or `ThisDrawing` object to change the values of the system variables, or the `GetVariable` function to get their current values.

Getting Information About and Modifying an Xref

Once an xref has been attached to a drawing, you can access information about the instance of the xref. As I previously mentioned, an xref is similar to a block reference that has been inserted into a drawing, and even the `AcadExternalReference` and `AcadBlockReference` objects share many of the same properties and methods. For information on how to access information about and modify a block reference, see the "Modifying a Block Reference" section earlier in this chapter.

Although an `AcadExternalReference` and `AcadBlockReference` object have much in common, there are a few differences:

- Xrefs do not support attributes.
- Xrefs do not support dynamic block properties.
- Xrefs can't be exploded unless they have been bound to the drawing first.
- The path to the external file can be modified.
- The objects in the external referenced file can be accessed.

Although there are a number of differences between xrefs and block references, in the AutoCAD Object library the `AcadExternalReference` and `AcadBlockReference` objects are similar. The only difference between the two object types is the `Path` property. The `Path` property of an `AcadExternalReference` object can be used to get and specify the path that AutoCAD should look in for the externally referenced drawing file. I show an example of using the `Path` property in the next section.

For each externally referenced file that is attached to a drawing, AutoCAD creates an in-memory database for that file. The database is represented by an `AcadDatabase` object and contains access to the nongraphical and graphical objects stored in the externally referenced file. The database of an xref can be accessed with the `XRefDatabase` property of an `AcadBlock` object.

Objects in the database of an xref returned by the `XRefDatabase` property can't be directly modified. However, it is possible to open the externally referenced drawing file into memory with the AutoCAD/ObjectDBX Common Type library. After a drawing file is opened in memory with the AutoCAD/ObjectDBX Common Type library, the objects in the file can then be modified. Once changes have been made to the drawing, you use the Reload method of an AcadBlock object in the drawing to which the xref is attached to update its display. I mention the Reload method in the next section and how to reference other object libraries in Chapter 12, "Communicating with Other Applications."

Changing the Layers of an Xref

Although you can't make changes to the geometry of an AcadBlock object that references an external file, you can affect the layers of an xref. To change a layer in an xref, set the visretain system variable to 1 with the SetVariable method of an AcadDocument or ThisDrawing object. After the visretain system variable has been enabled, you can use the XRefDatabase property of the AcadBlock object and access its AcadLayers collection, which contains the layers used by the objects of the xref. Any changes made to the layers are maintained in the drawing file that contains the xref and not the externally referenced file.

When locating an item in a collection object of the xref database, you must add the name of the xref with a pipe symbol as a prefix to the item's name. For example, to get the Surfaces layer in the xref named Building_Plan, you use the value Building_Plan|Surfaces.

The following code statements change the color of the layer named Surfaces to yellow:

```
' Enable the visretain system variable
ThisDrawing.SetVariable "visretain", 1

' Defines the name of the xref to locate
Dim sXrefName As String
sXrefName = "Building_Plan"

' Gets the name of the block
Dim oBlkDef As AcadBlock
Set oBlkDef = ThisDrawing.Blocks(sXrefName)

' Change the Surface layer in the xref to yellow
oBlkDef.XRefDatabase.Layers(sXrefName & "|" _
                            & "Surfaces").color = acYellow
```

Managing an Attached Xref

The management of the reference between an external drawing file and an xref is handled through an AcadBlock object. The name of the AcadBlock object used by an xref can be obtained with the Name property of the AcadExternalReference object. Once the name of the block has been obtained, you can use the Item method of the AcadBlocks collection object to get the AcadBlock object associated with the xref.

In addition to using the Item method, you can use a For statement to step through all the blocks in a drawing and see which ones are associated with an external file. While stepping through the AcadBlocks collection object, the IsXref property of the AcadBlock object returns True if the block represents an external referenced file.

The following code statements get the AcadBlock object for a block named Building_Plan and then use the IsXref property to see if it is an xref. If the block is an xref, a message box with the path to the external referenced file is displayed.

```
' Defines the name of the xref to locate
Dim sXrefName As String
sXrefName = "Building_Plan"

' Gets the name of the block
Dim oBlkDef As AcadBlock
Set oBlkDef = ThisDrawing.Blocks(sXrefName)

' Check to see if the block is an xref
If oBlkDef.IsXRef Then
  ' Display information about the xref
  MsgBox "Block name: " & sXrefName & _
         vbLf & "Path: " & oBlkDef.Path
End If
```

The Path property shown in the previous sample code is used to get the current path to the external referenced file, but it can also be used to update the default location for the externally referenced drawing file. After an AcadBlock object containing an external reference has been obtained, you can then manage the reference between the drawing and the external referenced file stored outside of AutoCAD. Table 7.6 lists the four functions that can be used to manage an xref.

Table 7.6: Methods used to manage an xref

Method	Description
Bind	Removes the reference to the external file, and all xrefs attached to the drawing are converted to blocks and stored as part of the drawing. Changes made to the external file no longer affect the objects in the drawing. The method expects a Boolean value; use True if you do not want to add a prefix to the symbol names that are created from the external reference or use False to add a prefix to the symbol name. Use a value of False to maintain the appearance of objects in the xref. Specifying a value of True indicates that the objects from the xref being merged will use the nongraphical objects defined in the drawing to which the xref is attached. If the nongraphical objects don't exist in the drawing to which the xref is attached and True is specified, the nongraphical object is copied from the xref's database.
Detach	Removes the reference to the external referenced file, and all xrefs attached to the drawing are removed. This method doesn't accept any arguments.
Reload	Updates the geometry in the drawing by reading the objects from the external referenced file. This method doesn't accept any arguments.
Unload	Maintains the reference to the external referenced file, and all xrefs remain in the drawing. The file isn't loaded into the drawing, which results in the objects contained in the file not being displayed. This method doesn't accept any arguments.

The following code statements reload the external reference named `Building_Plan`:

```
' Defines the name of the xref to locate
Dim sXrefName As String
sXrefName = "Building_Plan"

' Gets the name of the block
Dim oBlkDef As AcadBlock
Set oBlkDef = ThisDrawing.Blocks(sXrefName)

' Reload the xref
oBlkDef.Reload
```

Attaching and Modifying Raster Images

A raster image stored in an external file can be attached to a drawing. You might want to reference an external image file to place a company logo on a title block, display a watermark, or reference a topography map. An image file that has been added to a drawing is represented by an `AcadRasterImage` object.

NOTE Before attaching an image to a drawing file, keep in mind that large image files can increase the amount of time it takes to open a drawing and even change the display of a drawing.

A raster image can be added to model space or paper space using the `AddRaster` function. The `AddRaster` function returns an `AcadRasterImage` object and expects four argument values that specify the image file you want to add and then the placement and size of the image.

The following shows the syntax of the `AddRaster` function:

```
retVal = object.AddRaster(fileName, insertionPoint, scaleFactor, rotation)
```

Its arguments are as follows:

retVal The *retVal* argument represents the new `AcadRasterImage` object returned by the `AddRaster` function.

object The *object* argument specifies the `AcadBlock`, `AcadModelSpace`, or `AcadPaperSpace` object and indicates where you want to add the raster image.

fileName The *fileName* argument is a string that represents the name of the image file.

insertionPoint The *insertionPoint* argument is an array of doubles that represents the insertion point of the raster image.

scaleFactor The *scaleFactor* argument is a double that represents the scale factor of the raster image.

rotation The *rotation* argument is a double that represents the rotation angle of the raster image. The rotation angle must be expressed in radians.

The following code statements add a raster image based on the `acp_logo.png` filename to `5,5` and set the background of the image to transparent:

```
' Defines the insertion point
Dim insPt(2) As Double
insPt(0) = 5: insPt(1) = 5: insPt(2) = 0
```

```
' Defines the path to the image
Dim imageName As String
imageName = ThisDrawing.GetVariable("MyDocumentsPrefix") & _
            "\MyCustomFiles\acp_logo.png"

' Adds the raster image
Dim oRaster As AcadRasterImage
Set oRaster = ThisDrawing.ModelSpace. _
                 AddRaster(imageName, insPt, 1, 0)

' Sets the background of the image to transparent
oRaster.Transparency = True
```

After a raster image has been added to a drawing, you can control its appearance using the object properties and methods. A raster image supports the same general properties and methods as all graphical objects in a drawing, along with additional object-specific properties. Table 7.7 lists the properties that are specific to a raster image.

Table 7.7: Raster image–related properties and methods

Property/Method	Description
`Brightness`	Specifies the brightness applied to the raster image; the valid range is 0 to 100.
`ClippingEnabled`	Returns `True` if the raster image is clipped.
`ClipBoundary`	Specifies the clipping boundary of the raster image. The `ClipBoundary` method expects an array of doubles that form a closed region. The array must contain a minimum of six elements; each element pairing specifies a 2D coordinate value.
`Contrast`	Specifies the contrast applied to the raster image; the valid range is 0 to 100.
`Fade`	Specifies the fade value applied to the raster image; the valid range is 0 to 100. The greater the value, the more transparent the object.
`Height`	Returns the height, in pixels, for the raster image. This property is read-only.
`ImageFile`	Specifies the full path to the external file for the raster image.
`ImageHeight`	Specifies the height, in pixels, for the raster image.
`ImageVisibility`	Returns `True` if the raster image is visible.
`ImageWidth`	Specifies the width, in pixels, for the raster image.

Property/Method	Description
`Name`	Specifies the name of the raster image.
`Origin`	Specifies the insertion point of the raster image in the drawing and is an array of doubles.
`Rotation`	Specifies the rotation of the raster image.
`ScaleFactor`	Specifies the scale factor applied to the raster image.
`ShowRotation`	Returns `True` if the raster image is displayed at its specified rotation.
`Transparency`	Returns `True` if the background of the raster image is displayed as transparent.
`Width`	Returns the width, in pixels, for the underlay. This property is read-only.

Masking Objects with Wipeouts

A wipeout object is used to mask or hide other objects in a drawing. For example, you can place a wipeout behind the text or extension line of a dimension to make the text easier to read and make it easier to identify the objects that are dimensioned. An `AcadWipeout` object is used to represent a wipeout object that was created in a drawing.

There is no method to create a new wipeout, but you can use the wipeout command with the `SendCommand` or `PostCommand` method. The properties of an `AcadWipeout` object are the same as an `AcadRasterImage` object. I explained how to work with raster images in the "Attaching and Modifying Raster Images" section earlier in this chapter.

Working with Underlays

Underlays consist of geometry and annotation that is referenced into a drawing file from a drawing web (DWF/DWFx) file, MicroStation design (DGN) file, or an Adobe portable document (PDF) file. The geometry in an underlay is less accurate than that of a drawing because of the source applications that created the objects. Even though an underlay is less accurate, the accuracy might be enough for many designs created in the architectural and civil engineering industries.

When an underlay is attached to a drawing, its objects can be controlled using the layer information embedded in the underlay. As users create new objects in a drawing, they can use object snaps to snap to geometry that is part of an underlay.

The AutoCAD Object library doesn't provide support for attaching or detaching an underlay, but it does provide some support for querying and modifying an underlay that has already been attached to a drawing. If you want to attach an underlay, you can use the `-dgnattach`, `-dwfattach`, or `-pdfattach` commands with the `SendCommand` or `PostCommand` method.

The following objects represent the underlays that can be attached to a drawing:

- `AcadDgnUnderlay`—DGN underlay
- `AcadDwfUnderlay`—DWF/DWFx underlay
- `AcadPdfUnderlay`—PDF underlay

The following code statements demonstrate how the `ObjectName` property can be used to determine the type of an underlay object. The first two code statements get the first object in model space and expect the object to be an underlay.

```
Dim oEnt As AcadEntity
Set oEnt = ThisDrawing.ModelSpace(0)

Select Case oEnt.ObjectName
  Case "AcDbDgnReference"
    MsgBox "Underlay is a DGN file."
  Case "AcDbDwfReference"
    MsgBox "Underlay is a DWF file."
  Case "AcDbPdfReference"
    MsgBox "Underlay is a PDF file."
End Select
```

An underlay shares many properties in common with an `AcadRaster` object. The following properties are shared between underlays and raster images:

- `ClippingEnabled`
- `Contrast`
- `Fade`
- `Height`
- `Rotation`
- `ScaleFactor`
- `Width`

Table 7.8 lists the properties specific to an underlay. These properties can be used to control the display of the object and get information about the referenced file.

Table 30.8: Underlay-related properties

Property	**Description**
`AdjustForBackground`	Returns `True` if the colors in the underlay are adjusted for the current background color of the viewport.
`File`	Specifies the full path to the external file that contains the objects for the underlay.
`ItemName`	Specifies the sheet or design model name in the underlay file you want to display. A sheet or design model is one of the pages or designs stored in the underlay file. For example, a PDF file can contain several pages, and you use `ItemName` to specify which page you want to display.

Property	**Description**
`Monochrome`	Returns `True` if the colors of the underlay are displayed as monochromatic.
`Position`	Specifies the insertion point of the underlay in the drawing and is an array of doubles.
`UnderlayLayerOverrideApplied`	Specifies whether layer overrides are applied to the underlay; a constant value of `acNoOverrides` means no overrides are applied, whereas `acApplied` indicates overrides are applied.
`UnderlayName`	Specifies the name of the underlay file.
`UnderlayVisibility`	Returns `True` if the objects in the underlay should be visible.

Listing File Dependencies

A drawing file relies on a number of support files to display objects accurately. These support files might be font files, plot styles, external referenced files, and much more. You can use the `AcadFileDependencies` collection object to access a listing of the files that need to be included when sharing your files with a subcontractor or archiving your designs. Each dependency in the `AcadFileDependencies` collection object is represented by an `AcadFileDependency` object.

Although it is possible to directly add new entries for file dependencies to a drawing, I recommend letting the AutoCAD application and AutoCAD Object library do the work for you. Incorrectly defining a file dependency could have unexpected results on a drawing; objects might not display correctly or at all. Methods such as `AttachExternalReference` and `AddRaster` will add the appropriate file dependency entries to a drawing.

If you want, you can use the `CreateEntry`, `RemoveEntry`, and `UpdateEntry` methods to manage the file dependencies of a drawing. See the AutoCAD Help system for information on the methods used to manage file dependencies. In most cases, you will simply want to query the file dependencies of a drawing to learn which files might be missing and help the user locate them if possible. Use the `Item` method or a `For` statement to step through the file dependencies of the `AcadFileDependencies` collection object.

The following code statements display information about each file dependency at the Command prompt:

```
Sub ListDependencies()
  Dim oFileDep As AcadFileDependency

  For Each oFileDep In ThisDrawing.FileDependencies
    ThisDrawing.Utility.Prompt _
           vbLf & "Affects graphics: " & CStr(oFileDep.AffectsGraphics) & _
           vbLf & "Feature: " & oFileDep.Feature & _
           vbLf & "File name: " & oFileDep.FileName & _
           vbLf & "File size (Bytes): " & CStr(oFileDep.FileSize) & _
           vbLf & "Found path: " & oFileDep.FoundPath & _
           vbLf
  Next oFileDep
End Sub
```

NOTE A drawing file must have been saved once before you access its file dependency entries with the `AcadFileDependencies` collection object. Using the file path returned by the `FileName` and `FoundPath` properties of an `AcadFileDependency` object, you can use the `FileSystemObject` object to get more information about the referenced file. I explain how to use the `FileSystemObject` object in Chapter 12.

Here is an example of the output produced for a file dependency:

```
Affects graphics: True
Feature: Acad:Text
File name: arial.ttf
File size (Bytes): 895200
Found path: C:\WINDOWS\FONTS\
```

Exercise: Creating and Querying Blocks

In this section, you will create several new procedures that create and insert room label blocks into a drawing, move the blocks to specific layers based on their names, and extract the attributes of the blocks to produce a bill of materials (BOM). Room labels and blocks with attributes are often used in architectural drawings, but the same concepts can be applied to callouts and parts in mechanical drawings.

As you insert a room label block with the custom program, a counter increments by 1 so you can place the next room label without needing to manually enter a new value. The last calculated value is stored in a custom dictionary so it can be retrieved the next time the program is started. The key concepts I cover in this exercise are:

Creating and Modifying Block Definitions Block definitions are used to store a grouping of graphical objects that can be inserted into a drawing. Inserting a block definition creates a block reference that creates an instance of the objects defined in a block definition and not a copy of the objects.

Modify and Extracting Attributes The attributes attached to a block reference can be modified to hold different values per block reference, and those values can be extracted to a database or even a table within the drawing. Attribute values can represent project information, part numbers and descriptions of the parts required to assemble a new project, and so on.

NOTE The steps in this exercise depend on the completion of the steps in the "Exercise: Creating, Querying, and Modifying Objects" section of Chapter 4. If you didn't complete the steps, do so now or start with the `ch07_clsUtilities.cls` sample file available for download from `www.sybex.com/go/autocadcustomization`. Place these sample files in the `MyCustomFiles` folder under the `Documents` (or `My Documents`) folder, or the location you are using to store the DVB files. Also, remove the `ch07_` prefix from the name of the CLS file. You will also be working with the `ch07_building_plan.dwg` from this chapter's sample files.

Creating the *RoomLabel* Project

The RoomLabel project will contain functions and a main procedure that allow you to create and insert a room label block based on end-user input. The number applied to the block is incremented each time the block is placed in the current drawing. The following steps explain how to create a project named RoomLabel and to save it to a file named roomlabel.dvb:

1. Create a new VBA project with the name RoomLabel. Make sure to change the default project name (ACADProject) to RoomLabel in the VBA Editor.
2. In the VBA Editor, in the Project Explorer, right-click the new project and choose Import File.
3. When the Import File dialog box opens, browse to and select the clsUtilities.cls file in the MyCustomFiles folder. Click Open.

   The clsUtilities.cls file contains the utility procedures that you created as part of the DrawPlate project.
4. In the Project Explorer, right-click the new project and choose Insert ➢ Module. Change the default name of the new module to **basRoomLabel**.
5. On the menu bar, click File ➢ Save.

Creating the *RoomLabel* Block Definition

Creating separate drawing files that your custom programs depend on has advantages and disadvantages. One advantage of creating a separate drawing file is that you can use the AutoCAD user interface to create the block file. However, AutoCAD must be able to locate the drawing file so that the custom program can use the file. If AutoCAD can't locate the file, the custom program will have problems. Creating a block definition through code allows you to avoid the need of maintaining separate files for your blocks, thus making it easier to share a custom application with your clients or subcontractors. A disadvantage of using code to create your blocks is the time it takes to write the code for all your blocks and then having to maintain the code once it has been written.

In these steps, you create a custom function named roomlabel_createblkdef that will be used to create the block definition for the room label block if it doesn't already exist in the drawing.

1. In the Project Explorer, double-click the basRoomLabel component.
2. In the text editor area of the basRoomLabel component, type the following. (The comments are here for your information and don't need to be typed.)

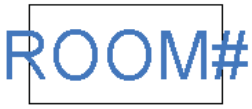
```
Private myUtilities As New clsUtilities

' Constant for the removal of the "Command: " prompt msg
Const removeCmdPrompt As String = vbBack & vbBack & vbBack & _
                                  vbBack & vbBack & vbBack & _
                                  vbBack & vbBack & vbBack & vbLf

Private g_nLastNumber As Integer
Private g_sLastPrefix As String
```

```
' Creates the block definition roomlabel
Private Sub RoomLabel_CreateBlkDef()
  On Error Resume Next

  ' Check for the existence of the roomlabel block definition
  Dim oBlkDef As AcadBlock
  Set oBlkDef = ThisDrawing.Blocks("roomlabel")

  ' If an error was generated, create the block definition
  If Err Then
    Err.Clear

    ' Define the block's origin
    Dim dInsPt(2) As Double
    dInsPt(0) = 18: dInsPt(1) = 9: dInsPt(2) = 0

    ' Create the block definition
    Set oBlkDef = ThisDrawing.Blocks.Add(dInsPt, "roomlabel")

    ' Add a rectangle to the block
    Dim dPtList(7) As Double
    dPtList(0) = 0:  dPtList(1) = 0
    dPtList(2) = 36: dPtList(3) = 0
    dPtList(4) = 36: dPtList(5) = 18
    dPtList(6) = 0:  dPtList(7) = 18

    Dim oLWPline As AcadLWPolyline
    Set oLWPline = oBlkDef.AddLightWeightPolyline(dPtList)
    oLWPline.Closed = True

    ' Add the attribute definition to the block
    Dim oAttDef As AcadAttribute
    Set oAttDef = oBlkDef.AddAttribute(9, acAttributeModeLockPosition, _
                                   "ROOM#", dInsPt, "ROOM#", "L000")
    oAttDef.Layer = "Plan_RoomLabel_Anno"

    ' Set the alignment of the attribute
    oAttDef.Alignment = acAlignmentMiddleCenter
    oAttDef.TextAlignmentPoint = dInsPt
  End If
End Sub
```

3. Click File ➢ Save.

Figure 7.3 shows the block definition that is created by this procedure. To see the contents of the block definition, use the `bedit` command and select the `RoomLabel` block. As an alternative, you can insert the `RoomLabel` block into the drawing and explode it.

FIGURE 7.3
`RoomLabel` block definition



Inserting a Block Reference Based on the *RoomLabel* Block Definition

Once you've created the block definition and added it to the `AcadBlocks` collection object, you can insert it into the drawing by using the `insert` command or the `InsertBlock` function in the AutoCAD Object library.

In these steps, you create two custom functions named `changeattvalue` and `roomlabel_insertblkref`. The `changeattvalue` function allows you to revise the insertion point and value of an attribute reference attached to a block reference based on the attribute's tag. The `roomlabel_insertblkref` function creates a block reference based on the `RoomLabel` block definition that was created with the `roomlabel_createblkdef` function.

1. In the text editor area of the `basRoomLabel` component, scroll to the bottom of the last procedure and press Enter a few times. Then, type the following. (The comments are here for your information and don't need to be typed.)

```
' Changes the value of an attribute reference in a block reference
Private Sub ChangeAttValue(oBlkRef As AcadBlockReference, _
                           vInsPt As Variant, sAttTag As String, _
                           sNewValue As String)

  ' Check to see if the block reference has attribute references
  If oBlkRef.HasAttributes Then
    ' Get the attributes of the block reference
    Dim vAtts As Variant
    vAtts = oBlkRef.GetAttributes

    Dim nCnt As Integer


    ' Step through the attributes in the block reference
    Dim oAttRef As AcadAttributeReference
    For nCnt = 0 To UBound(vAtts)
      Set oAttRef = vAtts(nCnt)

      ' Compare the attributes tag with the tag
      ' passed to the function
      If UCase(oAttRef.TagString) = UCase(sAttTag) Then
        oAttRef.InsertionPoint = vInsPt
        oAttRef.TextAlignmentPoint = vInsPt
        oAttRef.textString = sNewValue

        ' Exit the For statement
        Exit For
      End If
    Next
  End If
End Sub

' Creates the block definition roomlabel
Private Sub RoomLabel_InsertBlkRef(vInsPt As Variant, _
                                   sLabelValue As String)

  ' Add the layer Plan_RoomLabel_Anno
  myUtilities.CreateLayer "Plan_RoomLabel_Anno", 150

  ' Create the "roomlabel" block definition
  RoomLabel_CreateBlkDef

  ' Insert the block into model space
  Dim oBlkRef As AcadBlockReference
  Set oBlkRef = ThisDrawing.ModelSpace. _
              InsertBlock(vInsPt, "roomlabel", _
              1, 1, 1, 0)

  ' Changes the attribute value of the "ROOM#"
  ChangeAttValue oBlkRef, vInsPt, "ROOM#", sLabelValue
End Sub
```

2. Click File ➢ Save.

Prompting the User for an Insertion Point and a Room Number

Now that you have defined the functions to create the block definition and inserted the block reference into a drawing, the last function creates the main procedure that will prompt the user for input. The `roomlabel` procedure will allow the user to specify a point in the drawing, provide a new room number, or provide a new prefix. The `roomlabel` procedure uses the default number of `101` and prefix of `L`. As you use the `roomlabel` procedure, it increments the counter by 1 so that you can continue placing room labels.

In these steps, you create the custom procedure named `roomlabel` that uses all of the functions that you defined in this exercise to place a `RoomLabel` block each time you specify a point in the drawing.

1. In the text editor area of the `basRoomLabel` component, scroll to the bottom of the last procedure and press Enter a few times. Then, type the following. (The comments are here for your information and don't need to be typed.)

```
' Prompts the user for an insertion point and room number
Public Sub RoomLabel()
  On Error Resume Next

  ' Set the default values
```

```
Dim nLastNumber As Integer, sLastPrefix As String
If g_nLastNumber <> 0 Then
  nLastNumber = g_nLastNumber
  sLastPrefix = g_sLastPrefix
Else
  nLastNumber = 101
  sLastPrefix = "L"
End If

' Display current values
ThisDrawing.Utility.Prompt removeCmdPrompt & _
                           "Prefix: " & sLastPrefix & _
                           vbTab & "Number: " & CStr(nLastNumber)

Dim basePt As Variant

' Continue to ask for input until a point is provided
Do
  Dim sKeyword As String
  sKeyword = ""
  basePt = Null

  ' Setup default keywords
  ThisDrawing.Utility.InitializeUserInput 0, "Number Prefix"

  ' Prompt for a base point, number, or prefix value
  basePt = ThisDrawing.Utility.GetPoint(, _
           removeCmdPrompt & "Specify point for room label (" & _
           sLastPrefix & CStr(nLastNumber) & _
           ") or change [Number/Prefix]: ")

  ' If an error occurs, the user entered a keyword or pressed Enter
  If Err Then
    Err.Clear

    sKeyword = ThisDrawing.Utility.GetInput

    Select Case sKeyword
      Case "Number"
        nLastNumber = ThisDrawing.Utility. _
                GetInteger(removeCmdPrompt & _
                           "Enter new room number <" & _
                           CStr(nLastNumber) & ">: ")
      Case "Prefix"
        sLastPrefix = ThisDrawing.Utility. _
                GetString(False, removeCmdPrompt & _
                          "Enter new room number prefix <" & _
```

```
                                  sLastPrefix & ">: ")
      End Select
    End If

    ' If a base point was specified, then insert a block reference
    If IsNull(basePt) = False Then
      RoomLabel_InsertBlkRef basePt, sLastPrefix & CStr(nLastNumber)

      ' Increment number by 1
      nLastNumber = nLastNumber + 1
    End If
  Loop Until IsNull(basePt) = True And sKeyword = ""

  ' Store the latest values in the global variables
  g_nLastNumber = nLastNumber
  g_sLastPrefix = sLastPrefix
End Sub
```

2. Click File ➢ Save.

Adding Room Labels to a Drawing

The `roomlabel.dvb` file contains the main `roomlabel` procedure and some helper functions defined in the `clsUtilities.cls` file to define new layers.

NOTE The following steps require a drawing file named `ch07_building_plan.dwg`. If you didn't download the sample files previously, download them now from `www.sybex.com/go/autocadcustomization`. Place these sample files in the `MyCustomFiles` folder under the `Documents` (or `My Documents`) folder.

The following steps explain how to use the `roomlabel` procedure that is in the `roomlabel.lsp` file:

1. Open `Ch07_Building_Plan.dwg`. Figure 7.4 shows the plan drawing of the office building.

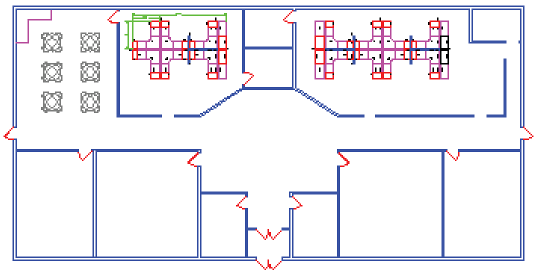


Figure 7.4
Plan view of the office building

2. At the Command prompt, type **vbarun** and press Enter.

3. When the Macros dialog box opens, select the `RoomLabel.dvb!basRoomLabel.RoomLabel` macro from the list and click Run.
4. At the `Specify point for room label (L101) or change [Number/Prefix]:` prompt, specify a point inside the room in the lower-left corner of the building.

   The room label definition block and `Plan_RoomLabel_Anno` layer are created the first time the `roomlabel` procedure is used. The `RoomLabel` block definition should look like Figure 7.5 when inserted into the drawing.

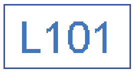


FIGURE 7.5 Inserted `RoomLabel` block

5. At the `Specify point for room label (L101) or change [Number/Prefix]:` prompt, type **`n`** and press Enter.
6. At the `Enter new room number <102>:` prompt, type **`105`** and press Enter.
7. At the `Specify point for room label (L105) or change [Number/Prefix]:` prompt, type **`p`** and press Enter.
8. At the `Enter new room number prefix <L>:` prompt, type **`R`** and press Enter.
9. At the `Specify point for room label (R105) or change [Number/Prefix]:` prompt, specify a point in the large open area in the middle of the building.
10. Press Enter to end `roomlabel`.
11. Close and discard the changes to the drawing file.

Creating the *FurnTools* Project

The `FurnTools` project will contain several functions and main procedures that modify the properties and extract the attribute values of block references that have been inserted into a drawing. The following steps explain how to create a project named `FurnTools` and save it to a file named `furntools.dvb`:

1. Create a new VBA project with the name `FurnTools`. Make sure to also change the default project name (`ACADProject`) to `FurnTools` in the VBA Editor.
2. In the VBA Editor, in the Project Explorer, right-click the new project and choose Import File.
3. When the Import File dialog box opens, browse to and select the `clsUtilities.cls` file in the `MyCustomFiles` folder. Click Open.

   The `clsUtilities.cls` file contains the utility procedures that you created as part of the `DrawPlate` project.
4. In the Project Explorer, right-click the new project and choose Insert ➢ Module. Change the name of the new module to **`basFurnTools`**.

5. On the menu bar, click File ➤ Save.

Moving Objects to Correct Layers

Not everyone will agree on the naming conventions, plot styles, and other various aspects of layers, but there are two things drafters can agree on when it comes to layers:

- Objects should inherit their properties, for the most part, from the objects in which they are placed.
- Objects should only be placed on layer 0 when creating blocks.

Although I would like to think that all of the drawings I have ever created are perfect, I know they aren't. Rushed deadlines, changing project parameters, and other distractions impede perfection. Objects may have been placed on the wrong layer, or maybe it wasn't my fault and standards simply changed during the course of a project. With VBA and the AutoCAD Object library, you can identify potential problems in a drawing and let the user know about them so they can be fixed. You might even be able to fix the problems automatically without user input.

In these steps, you will create a custom procedure named `furnlayers` that will be used to identify objects by type and value to ensure they are placed on the correct layer. This is achieved by using selection sets and entity data lists, along with looping and conditional statements.

1. In the Project Explorer, double-click the `basFurnTools` component.
2. In the text editor area of the `basFurnTools` component, type the following. (The comments are here for your information and don't need to be typed.)

```
Private myUtilities As New clsUtilities

' Constants for PI
Const PI As Double = 3.14159265358979

' Moves objects to the correct layers based on a set of established rules
Sub FurnLayers()
  On Error Resume Next

  ' Get the blocks to extract
  Dim oSSFurn As AcadSelectionSet
  Set oSSFurn = ThisDrawing.SelectionSets.Add("SSFurn")

  ' If an error is generated, selection set already exists
  If Err Then
    Err.Clear

    Set oSSFurn = ThisDrawing.SelectionSets("SSFurn")
  End If

  ' Define the selection set filter to select only blocks
  Dim nDXFCodes(3) As Integer, nValue(3) As Variant
  nDXFCodes(0) = -4: nValue(0) = "<OR":
```

```
nDXFCodes(1) = 0: nValue(1) = "INSERT"
nDXFCodes(2) = 0: nValue(2) = "DIMENSION"
nDXFCodes(3) = -4: nValue(3) = "OR>"

Dim vDXFCodes As Variant, vValues As Variant
vDXFCodes = nDXFCodes
vValues = nValue

' Allow the user to select objects in the drawing
oSSFurn.SelectOnScreen vDXFCodes, vValues

' Proceed if oSSFurn is greater than 0
If oSSFurn.Count > 0 Then

  ' Step through each object in the selection set
  Dim oEnt As AcadEntity
  For Each oEnt In oSSFurn
    ' Check to see if the object is a block reference
    If oEnt.ObjectName = "AcDbBlockReference" Then
      Dim oBlkRef As AcadBlockReference
      Set oBlkRef = oEnt

      ' Get the name of the block, use EffectiveName because
      ' the block could be dynamic
      Dim sBlkName As String
      sBlkName = oBlkRef.EffectiveName

      ' If the block name starts with RD or CD,
      ' then place it on the surfaces layer
      If sBlkName Like "RD*" Or _
         sBlkName Like "CD*" Then
        oBlkRef.Layer = "Surfaces"

      ' If the block name starts with PNL, PE, and PX,
      ' then place it on the panels layer
      ElseIf sBlkName Like "PNL*" Or _
             sBlkName Like "PE*" Or _
             sBlkName Like "PX*" Then
        oBlkRef.Layer = "Panels"

      ' If the block name starts with SF,
      ' then place it on the panels layer
      ElseIf sBlkName Like "SF*" Or _
             sBlkName Like "FF*" Then
        oBlkRef.Layer = "Storage"
      End If
    ElseIf oEnt.ObjectName Like "AcDb*Dim*" Then
```

```
        oEnt.Layer = "Dimensions"
      End If
    Next oEnt

    ' Remove the selection set
    oSSFurn.Delete
  End If
End Sub
```

3. Click File ➢ Save.

Creating a Basic Block Attribute Extraction Program

The designs you create take time and often are a source of income or savings for your company. Based on the types of objects in a drawing, you can step through a drawing and get attribute information from blocks or even geometric values such as lengths and radii of circles. You can use the objects in a drawing to estimate the potential cost of a project or even provide information to manufacturing.

In these steps, you create four custom functions named `ExtAttsFurnBOM`, `SortArray`, `TableFurnBOM`, and `RowValuesFurnBOM`. The `ExtAttsFurnBOM` function extracts the values of the attributes in the selected blocks and then uses the `SortArray` function to sort the attribute values before quantifying them. The `TableFurnBOM` and `RowValuesFurnBOM` functions are used to create a grid of lines containing the extracted values.

1. In the text editor area of the `basFurnTools` component, scroll to the bottom of the last procedure and press Enter a few times. Then, type the following. (The comments are here for your information and don't need to be typed.)

```
' ExtAttsFurnBOM - Extracts, sorts, and quantifies the attribute information
Private Function ExtAttsFurnBOM(oSSFurn As AcadSelectionSet) As Variant
  Dim sList() As String

  Dim sPart As String, sLabel As String

  ' Step through each block in the selection set
  Dim oBlkRef As AcadBlockReference
  Dim nListCnt As Integer
  nListCnt = 0

  For Each oBlkRef In oSSFurn
    ' Step through the objects that appear after
    ' the block reference, looking for attributes
    Dim vAtts As Variant
    vAtts = oBlkRef.GetAttributes

    ' Check to see if the block has attributes
    If oBlkRef.HasAttributes = True Then
      ' Get the attributes of the block reference
      Dim vAttRefs As Variant
      vAttRefs = oBlkRef.GetAttributes

      Dim oAttRef As AcadAttributeReference
      Dim nAttCnt As Integer

      For nAttCnt = LBound(vAttRefs) To UBound(vAttRefs)
        Set oAttRef = vAttRefs(nAttCnt)

        If UCase(oAttRef.TagString) = "PART" Then
          sPart = oAttRef.textString
        ElseIf UCase(oAttRef.TagString) = "LABEL" Then
          sLabel = oAttRef.textString
        End If
      Next
    End If

    ' Resize the array
    ReDim Preserve sList(nListCnt)

    ' Add the part and label values to the array
    sList(nListCnt) = sLabel & vbTab & sPart

    ' Increment the counter
    nListCnt = nListCnt + 1
  Next oBlkRef

  ' Sort the array of parts and labels
  Dim vFurnListSorted As Variant
  vFurnListSorted = SortArray(sList)

  ' Quantify the list of parts and labels
  ' Step through each value in the sorted array
  Dim sFurnList() As String
  Dim vCurVal As Variant, sPreVal As String
  Dim sItems As Variant
  nCnt = 0: nListCnt = 0

  For Each vCurVal In vFurnListSorted
    ' Check to see if the previous value is the same as the current value
    If CStr(vCurVal) = sPreVal Or sPreVal = "" Then
      ' Increment the counter by 1
      nCnt = nCnt + 1

      ' Values weren't the same, so record the quantity
    Else
```

```
      ' Split the values of the item
      sItems = Split(sPreVal, vbTab)

      ' Resize the array
      ReDim Preserve sFurnList(nListCnt)

      ' Add the part and label values to the array
      sFurnList(nListCnt) = CStr(nCnt) & vbTab & sItems(0) & vbTab & sItems(1)

      ' Increment the array counter
      nListCnt = nListCnt + 1

      ' Reset the counter
      nCnt = 1
    End If

    sPreVal = CStr(vCurVal)
  Next vCurVal

  ' Append the last item
  ' Split the values of the item
  sItems = Split(sPreVal, vbTab)

  ' Resize the array
  ReDim Preserve sFurnList(nListCnt)

  ' Add the part and label values to the array
  sFurnList(nListCnt) = CStr(nCnt) & vbTab & sItems(0) & vbTab & sItems(1)

  ' Return the sorted and quantified array
  ExtAttsFurnBOM = sFurnList
End Function

' Performs a basic sort on the string values in an array,
' and returns the newly sorted array.
Private Function SortArray(vArray As Variant) As Variant
  Dim nFIdx As Integer, nLIdx As Integer
  nFIdx = LBound(vArray): nLIdx = UBound(vArray)

  Dim nOuterCnt As Integer, nInnerCnt As Integer
  Dim sTemp As String

  For nOuterCnt = nFIdx To nLIdx - 1
    For nInnerCnt = nOuterCnt + 1 To nLIdx
      If vArray(nOuterCnt) > vArray(nInnerCnt) Then
        sTemp = vArray(nInnerCnt)
        vArray(nInnerCnt) = vArray(nOuterCnt)

        vArray(nOuterCnt) = sTemp
      End If
    Next nInnerCnt
  Next nOuterCnt

  SortArray = vArray
End Function

' Create the bill of materials table/grid
Private Sub TableFurnBOM(vQtyList As Variant, dInsPt() As Double)
  ' Define the sizes of the table and grid
  Dim dColWidths(3) As Double
  dColWidths(0) = 0: dColWidths(1) = 15
  dColWidths(2) = 45: dColWidths(3) = 50

  Dim dTableWidth As Double, dTableHeight As Double
  dTableWidth = 0: dTableHeight = 0

  Dim nRow As Integer
  nRow = 1

  Dim dRowHeight As Double, dTextHeight As Double
  dRowHeight = 4: dTextHeight = dRowHeight - 1

  ' Get the table width by adding all column widths
  Dim vColWidth As Variant
  For Each vColWidth In dColWidths
    dTableWidth = dTableWidth + CDbl(vColWidth)
  Next vColWidth

  ' Define the standard table headers
  Dim sHeaders(2) As String
  sHeaders(0) = "QTY": sHeaders(1) = "LABELS": sHeaders(2) = "PARTS"

  ' Create the top of the table
  Dim vInsPtRight As Variant
  vInsPtRight = ThisDrawing.Utility.PolarPoint( _
                          dInsPt, 0, dTableWidth)
  Dim oLine As AcadLine
  Set oLine = ThisDrawing.ModelSpace.AddLine(dInsPt, vInsPtRight)

  ' Get the bottom of the header row
  Dim vBottomRow As Variant
  vBottomRow = ThisDrawing.Utility.PolarPoint( _
                         dInsPt, ((PI / 2) * -1), dRowHeight)
  ' Add headers to the table
  RowValuesFurnBOM sHeaders, vBottomRow, dColWidths, dTextHeight

  ' Step through each item in the list
  Dim vItem As Variant
  For Each vItem In vQtyList
    nRow = nRow + 1

    vBottomRow = ThisDrawing.Utility.PolarPoint( _
                         dInsPt, ((PI / 2) * -1), dRowHeight * nRow)

    RowValuesFurnBOM Split(vItem, vbTab), vBottomRow, dColWidths, dTextHeight
  Next vItem

  ' Create the vertical lines for each column
  dColWidthTotal = 0

  For Each vColWidth In dColWidths
    ' Calculate the placement of each vertical line (left to right)
    dColWidthTotal = CDbl(vColWidth) + dColWidthTotal

    Dim vColBasePt As Variant
    vColBasePt = ThisDrawing.Utility.PolarPoint( _
                         dInsPt, 0, dColWidthTotal)

    Dim vColBottomPt As Variant
    vColBottomPt = ThisDrawing.Utility.PolarPoint( _
                           vColBasePt, ((PI / 2) * -1), _
                           myUtilities.Calc2DDistance(dInsPt(0), _
                                                      dInsPt(1), _
                                                      vBottomRow(0), _
                                                      vBottomRow(1)))

    ' Draw the vertical line
    Set oLine = ThisDrawing.ModelSpace.AddLine(vColBasePt, vColBottomPt)

  Next vColWidth
End Sub

' Create a row and populate the data for the table
Private Sub RowValuesFurnBOM(vItems As Variant, _
                             vBottomRow As Variant, _
                             vColWidths As Variant, _
                             dTextHeight As Double)

  ' Calculate the insertion point for the header text
  Dim dRowText(2) As Double
  dRowText(0) = 0.5 + vBottomRow(0)
  dRowText(1) = 0.5 + vBottomRow(1)
  dRowText(2) = vBottomRow(2)

  Dim dTableWidth As Double
  dTableWidth = 0

  ' Get the table width by adding all column widths
  Dim vColWidth As Variant
  For Each vColWidth In vColWidths
    dTableWidth = dTableWidth + CDbl(vColWidth)
  Next vColWidth

  ' Lay out the text in each row
  Dim nCol As Integer, dColWidthTotal As Double
  nCol = 0: dColWidthTotal = 0

  Dim vItem As Variant
  For Each vItem In vItems
    ' Calculate the placement of each text object (left to right)
    dColWidthTotal = dColWidthTotal + vColWidths(nCol)

    Dim vInsTextCol As Variant
    vInsTextCol = ThisDrawing.Utility.PolarPoint( _
                      dRowText, 0, dColWidthTotal)

    ' Draw the single-line text object
    Dim oText As AcadText
    Set oText = ThisDrawing.ModelSpace.AddText(CStr(vItem), _
                              vInsTextCol, dTextHeight)

    ' Create the row line
    Dim vBottomRowRight As Variant
    vBottomRowRight = ThisDrawing.Utility.PolarPoint( _
                        vBottomRow, 0, dTableWidth)
    Dim oLine As AcadLine
    Set oLine = ThisDrawing.ModelSpace.AddLine(vBottomRow, vBottomRowRight)

    ' Increment the counter
    nCol = nCol + 1
  Next vItem
End Sub

' Extracts, aggregates, and counts attributes from the furniture blocks
Sub FurnBOM()
  On Error Resume Next
```

```
' Get the blocks to extract
Dim oSSFurn As AcadSelectionSet
Set oSSFurn = ThisDrawing.SelectionSets.Add("SSFurn")

' If an error is generated, selection set already exists
If Err Then
  Err.Clear

  Set oSSFurn = ThisDrawing.SelectionSets("SSFurn")
End If

' Define the selection set filter to select only blocks
Dim nDXFCodes(0) As Integer, nValue(0) As Variant
nDXFCodes(0) = 0
nValue(0) = "INSERT"

Dim vDXFCodes As Variant, vValues As Variant
vDXFCodes = nDXFCodes
vValues = nValue

' Allow the user to select objects in the drawing
oSSFurn.SelectOnScreen vDXFCodes, vValues

' Use the ExtAttsFurnBOM to extract and quantify the attributes in the blocks
' If a selection set was created, then look for attributes
If oSSFurn.Count > 0 Then
  ' Extract and quantify the parts in the drawing
  Dim vAttList As Variant
  vAttList = ExtAttsFurnBOM(oSSFurn)

  ' Create the layer named BOM and set it current
  Dim oLayer As AcadLayer
  Set oLayer = myUtilities.CreateLayer("BOM", 8)
  Set ThisDrawing.ActiveLayer = oLayer.Name

  ' Prompt the user for the point to create the BOM
  Dim vInsPt As Variant
  vInsPt = ThisDrawing.Utility.GetPoint(, vbLf & _
               "Specify upper-left corner of BOM: ")

  ' Start the function that creates the table grid
  Dim dInsPt(2) As Double
  dInsPt(0) = vInsPt(0): dInsPt(1) = vInsPt(1): dInsPt(2) = vInsPt(2)

  TableFurnBOM vAttList, dInsPt
```

```
    ' Remove the selection set
    oSSFurn.Delete
  End If
End Sub
```

2. Click File ➤ Save.

Using the Procedures of the *FurnTools* Project

The procedures you added to FurnTools project leverage some of the functions defined in clsUtilities.cls. These tools allow you to change the layers of objects in a drawing and extract information from the objects in a drawing as well. More specifically, they allow you to work with blocks that represent an office furniture layout.

Although you might be working in a civil engineering– or mechanical design–related field, these concepts can and do apply to the work you do—just in different ways. Instead of extracting information from a furniture block, you could get and set information in a title block, a callout, or even an elevation marker. Making sure hatching is placed on the correct layers along with dimensions can improve the quality of output for the designs your company creates.

NOTE The following steps require a drawing file named ch07_building_plan.dwg. If you didn't download the sample files previously, download them now from www.sybex.com/go/autocadcustomization. Place these sample files in the MyCustomFiles folder under the Documents (or My Documents) folder.

The following steps explain how to use the FurnLayers procedure:

1. Open ch07_building_plan.dwg.
2. At the Command prompt, type **vbarun** and press Enter.
3. When the Macros dialog box opens, select the FurnTools.dvb!basFurnTools.FurnLayers macro from the list and click Run.
4. At the Select objects: prompt, select all the objects in the drawing and press Enter.

   The objects in the drawing are placed on the correct layers, and this can be seen as the objects were all previously placed on layer 0 and had a color of white (or black based on the background color of the drawing area).

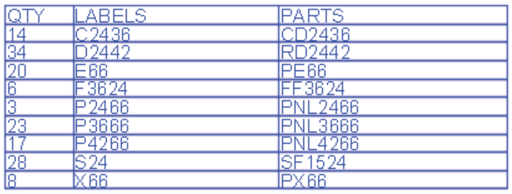
The following steps explain how to use the FurnBom procedure:

1. At the Command prompt, type **vbarun** and press Enter.
2. When the Macros dialog box opens, select the FurnTools.dvb!basFurnTools.FurnBOM macro from the list and click Run.
3. At the Select objects: prompt, select all the objects in the drawing. Don't press Enter yet.

   Notice that the dimension objects aren't highlighted. As a result of the selection set filter being applied with the SelectOnScreen function, the SelectOnScreen function only allows block references (insert object types) to be selected.

4. Press Enter to end the object selection.
5. At the `Specify upper-left corner of BOM:` prompt, specify a point to the right of the furniture layout in the drawing.

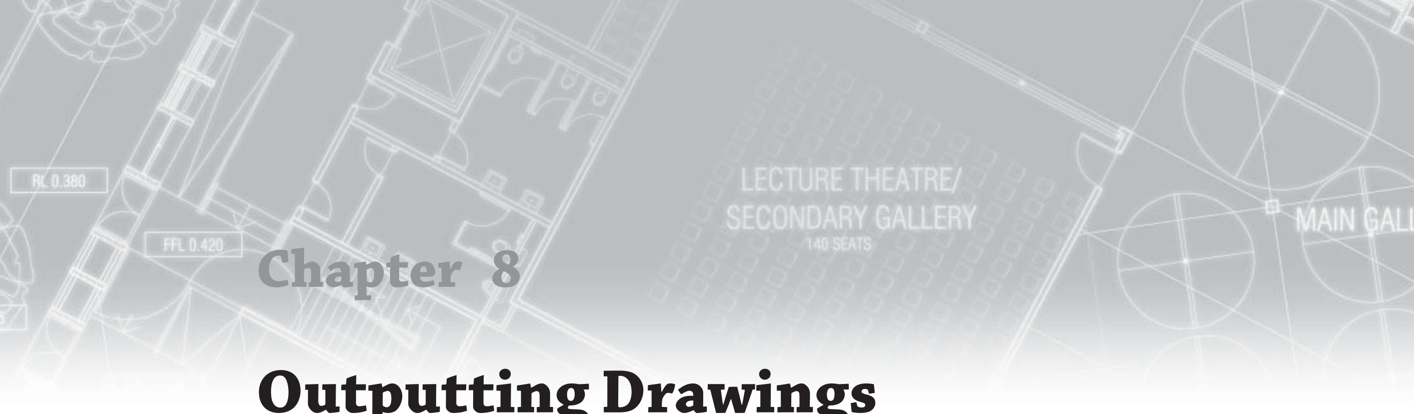
   The bill of materials that represents the furniture blocks is placed in a table grid, as shown Figure 7.6.



QTY	LABELS	PARTS
14	C2436	CD2436
34	D2442	RD2442
20	E66	PE66
6	F3624	FF3624
3	P2466	PNL2466
23	P3666	PNL3666
17	P4266	PNL4266
28	S24	SF1524
8	X66	PX66

FIGURE 7.6
Bill of materials generated from the office furniture layout

6. Close and discard the changes to the drawing file.



Chapter 8

Outputting Drawings

Autodesk® AutoCAD® drawing files are the living documents that an engineer or a drafter creates to communicate the product or building being designed. Typically before a design is brought from the digital to the physical world, it goes through a series of reviews and then final sign-off with the customer. As part of the review and sign-off process, it is common practice to output a drawing file to an electronic file or hardcopy—known as paper.

Plotting is the most common way of outputting a drawing, whether to an electronic file or to a hardcopy. You indicate which layouts to plot and plot settings to use. Plot settings can be assigned directly to an individual layout or assigned to multiple layouts using a page setup. A layout typically contains one or more viewports that display objects from model space and a title block that provides information about the objects in the viewport, such as project location and recent revisions.

In addition to plotting a layout, you can use the information contained in a drawing with another application by exporting or importing other file types. For example, you might use an external structural analysis or sun study application to complete some of the tasks you perform or even insert an image of the drawing into a presentation.

Creating and Managing Layouts

A layout—also known as *paper space*—is used to organize and control which objects should be output to an electronic file or hardcopy, and how. Layouts are the digital equivalent of a physical sheet of paper in which objects from model space are displayed using floating viewports. A floating viewport allows you to specify which area of model space to display and at which scale to display it on a layout. Each floating viewport can display a separate area of model space and have a different scale. I explain how to create and modify floating viewports in the "Displaying Model Space Objects with Viewports" section later in this chapter.

Floating viewports aren't the only objects typically found on a layout. A layout commonly has a title block that provides information about the objects being plotted and which project they are associated with. Dimensions and notes can also be found on a layout. In addition to the graphical objects I mentioned, each layout contains a set of plot settings. The plot settings are used to control how the objects on the layout are output, and which device and paper size should be used when outputting the layout. I discuss plot settings in the "Controlling the Output of a Layout" section later in this chapter.

From the perspective of the AutoCAD Object library, a layout is represented by an `AcadLayout` object that is stored in the `AcadLayouts` collection object. You access the `AcadLayouts` collection object with the `Layouts` property of the `AcadDocument` or `ThisDrawing` object. Unlike most objects in the AutoCAD Object library, a layout is a container object made up of two different object types: `AcadBlock` and `AcadPlotConfiguration`.

Creating a Layout

A new layout can be added to a drawing using the `Add` method of the `AcadLayouts` collection object. The `Add` method expects a string that represents the name of the new layout to add and returns an `AcadLayout` object of the new layout added. Each layout must have a unique name. You can use the `Item` method with an error handler to check to see if a layout with a specific name already exists in the drawing or to get a specific layout in the `AcadLayouts` collection object. Just like other collections, a `For` statement can be used to step through the layouts in a drawing.

The following code statements add a new layout named `Demolition`:

```
Dim oLayout As AcadLayout
Set oLayout = ThisDrawing.Layouts.Add("Demolition")
```

Here are example code statements that check for the existence of a layout named Demolition:

```
On Error Resume Next

Dim oLayout As AcadLayout
Set oLayout = ThisDrawing.Layouts("Demolition")

If Err Then
  MsgBox "Layout isn't in the drawing."
Else
  MsgBox "Layout was found."
End If
```

When a layout is no longer needed, it can be removed from the drawing using the `Delete` method of an `AcadLayout` object.

NOTE After you create a new layout, you can copy the plot settings from an existing layout to another layout with the `CopyFrom` method. I explain the `CopyFrom` method in the "Creating and Managing Named Page Setups" section.

Working with a Layout

Once a layout has been created, it can be set as current or objects can be added to it, like adding objects to model space. The `ActiveLayout` property of the `AcadDocument` object is used to set a layout as current. The `ActiveLayout` property expects an object of the `AcadLayout` type, which can be obtained by the `Add` or `Item` method of the `AcadLayouts` collection object. You can also use the `ActiveLayout` property to get the `AcadLayout` object of the current layout.

The following code statement sets a layout named `Demolition` as current:

```
ThisDrawing.ActiveLayout = ThisDrawing.Layouts("Demolition")
```

NOTE When a layout is set as current using the `ActiveLayout` property or the layout tabs along the bottom of a drawing window in the AutoCAD user interface, an event named `LayoutSwitched` is triggered, if it has been defined. You can use this event to control the display of objects on a layout after it is set as current or to change the values of system variables as needed. The LayoutSwitched event is a member of the AcadDocument object. I explain how to use events for the AcadDocument object in Chapter 10, "Modifying the Application and Working with Events."

The Block property of an AcadLayout object returns the AcadBlock object, which contains the graphical objects on a layout. I explained how to work with blocks in Chapter 7, "Working with Blocks and External References." In addition to using the Block property to get the objects on a layout, the PaperSpace property of an AcadDocument or a ThisDrawing object can be used to access the objects on a layout. The PaperSpace property returns an AcadPaperSpace object, which is the only way to add a floating viewport to a layout. A floating viewport displays the objects in model space on a layout. I explain how to add and modify floating viewports in the "Displaying Model Space Objects with Viewports" section.

The following code statements insert a drawing file named c-tblk.dwg onto the layout named Demolition:

```
' Gets the layout named Demolition
Dim oLayout As AcadLayout
Set oLayout = ThisDrawing.Layouts("Demolition")

' Defines the title block name and location
Dim sTitleBlk As String
sTitleBlk = ThisDrawing.GetVariable("MyDocumentsPrefix") & _
            "\c-tblk.dwg"

' Inserts the drawing at 0,0,0
Dim dInsPt(2) As Double
dInsPt(0) = 0: dInsPt(1) = 0: dInsPt(2) = 0
oLayout.Block.InsertBlock dInsPt, sTitleBlk, 1, 1, 1, 0
```

Controlling the Display of Layout Tabs

The layouts of an open drawing are typically accessed by the user with the tabs displayed along the bottom of a drawing window. The display of the layout tabs can be toggled with the DisplayLayoutTabs property of the AcadPreferencesDisplay object. I explained how to access the preferences of the AutoCAD application and an open drawing in Chapter 3, "Interacting with the Application and Documents Objects."

Along with controlling the display of the layout tabs, you can control the order in which layouts appear. The TabOrder property of the AcadLayout object can be used to get or set the order of a layout. The leftmost tab has an order of 0 and is always the Model tab. The order of the Model tab can't be changed, but that of a named layout can. You assign the TabOrder property an integer value that specifies the new location of the tab; the order of all other layouts is automatically updated.

Displaying Model Space Objects with Viewports

The objects added to a layout fall into one of two categories: annotation or viewports. Annotation can be in the form of general notes, dimensions, tables, and even a title block.

You place annotation on a layout to help communicate your design. Although a title block isn't typically thought of as annotation, in the general sense any object that isn't part of the actual design in a drawing is annotation.

Viewports are windows into model space that control the objects to be displayed on a layout; not only do they control the display of objects; viewports also control the scale at which the objects are displayed. The viewport on a layout, other than the Model tab, is represented by the `AcadPViewport` object and shouldn't be confused with the `AcadViewport` object, which represents a tiled viewport in model space. You learned about tiled viewports in Chapter 5, "Interacting with the User and Controlling the Current View."

Adding a Floating Viewport

A floating viewport can be added to a layout using the `AddPViewport` function. The `AddPViewport` function returns an `AcadPViewport` object and expects three argument values that define the placement and size of the floating viewport. Once the floating viewport has been created, you then define the area of model space that should be displayed and at which scale. I explain how to modify a viewport in the "Modifying a Floating Viewport" section later in this chapter.

The following shows the syntax of the `AddPViewport` function:

```
retVal = object.AddPViewport(centerPoint, width, height)
```

Its arguments are as follows:

`object` The *`object`* argument represents the `AcadPaperSpace` object returned by the `PaperSpace` property of an `AcadDocument` object.

`centerPoint` The *`centerPoint`* argument is an array of doubles that represents the center of the viewport on the layout.

`Width`* and *`height` The *`width`* and *`height`* arguments are doubles that represent the width and height of the viewport.

The following code statements create a new viewport that is 200 units wide by 190 units high and centered at 102,97.5:

```
' Get the active paper space block
Dim oPSpace As AcadPaperSpace
Set oPSpace = ThisDrawing.PaperSpace

' Define the center point of the viewport
Dim dCenPt(2) As Double
dCenPt(0) = 102: dCenPt(1) = 97.5: dCenPt(2) = 0

' Add the viewport to the layout
Dim oPVport As AcadPViewport
Set oPVport = oPSpace.AddPViewport(dCenPt, 200, 190)
```

NOTE Viewports added with the `AddPViewport` function are off by default. When a viewport is turned off, the objects in model space aren't displayed. You use the `Display` method of an `AcadPViewport` object to turn on the display of objects in model space. The `Display` method accepts a single Boolean value: `True` to turn on a viewport or `False` to turn it off. The `ViewportOn` property of an `AcadPViewport` object can be used to determine the current display state of a viewport. The following code statement turns on a viewport:

```
object.Display True
```

Setting a Viewport as Current

The `ActivePViewport` property of an `AcadDocument` object is used to determine which viewport is current or to set a viewport as current. Before you set a viewport as current, model space must be active. Model space can be set as active using the `MSpace` property of an `AcadDocument` object. When the `MSpace` property is `True`, objects in model space can be edited. The changes made through a viewport on a layout are the same as if they were made on the Model tab or in model space directly.

The following code statements make the viewport object assigned to the *`oVPort`* variable active and enable model space:

```
ThisDrawing.ActivePViewport = oVPort
ThisDrawing.MSpace = True
```

As an alternative to the `MSpace` property, you can use the `ActiveSpace` property to determine which space is active and to switch between model space and paper space while a layout is active. The `MSpace` property can't be used from the Model tab, but the `ActiveSpace` property can be. You can use `ActiveSpace` to switch to paper space if paper space isn't currently active. `ActiveSpace` behaves similar to the `tilemode` system variable.

The following code statement sets paper space active on a layout or switches from the Model tab to the most recently used named layout:

```
ThisDrawing.ActiveSpace = acPaperSpace
```

Modifying a Floating Viewport

Once a viewport has been added to a layout or a reference to an `AcadPViewport` object has been obtained, you can modify its properties. The properties of a viewport allow you to do the following:

- Control which area of model space is visible and the scale at which objects are displayed
- Specify the general visibility settings for the objects in model space
- Determine whether a viewport represents a view in a sheet set

Specifying the Display Settings for a Viewport

The objects that are displayed in a viewport vary based on the features of a design being communicated. For example, in some drawings you might want to display all objects, whereas in others you might only want to show a small area to be detailed.

Table 8.1 lists the properties of an `AcadPViewport` object that control the display of objects from model space objects. The following code statements change the arc smoothness and display locking status for a floating viewport assigned to the *`oPVport`*:

```
oPVport.ArcSmoothness = 1500
oPVport.DisplayLocked = True
```

Table 8.1: Display-related properties of an `AcadPViewport` object

Property	Description
`ArcSmoothness`	Specifies the smoothness for curved model space objects. Enter a value from 1 to 20,000.
`Clipped`	Returns `True` if the viewport is clipped. There is no method for clipping a viewport; you must use the `vpclip` command.
`Direction`	Specifies the view direction of the model space objects. View direction is expressed as an array of three double values.
`DisplayLocked`	Specifies the lock state of the viewport's display. When the display is locked, the user isn't able to change the viewport's view.
`LayerPropertyOverrides`	Returns `True` if the viewport has layer overrides applied. There is no method for applying or modifying layer overrides; you must use the `-vport` command.
`LensLength`	Specifies the lens length applied to the viewport when perspective viewing is enabled; use the perspective system variable to enable perspective view in the current viewport.
`ModelView`	Specifies the `AcadView` object that defines the area of model space that should be displayed in the viewport. After assigning an `AcadView` object, you must execute the `SyncModelView` method to update the view in the viewport to match the `AcadView` object.
`ShadePlot`	Specifies the visual style that should be applied to the model space objects displayed in the viewport.
`Target`	Specifies the target point of the current view in the viewport.
`TwistAngle`	Specifies the twist angle to be applied to the current view in the viewport.

You can learn more about the properties listed in Table 8.1 from the AutoCAD Help system.

Scaling Objects in a Viewport

In addition to defining which area of model space to display in a viewport, the scale at which the objects are displayed is critical to outputting a design. When you produce a drawing, the objects displayed are commonly output at a specific scale so the recipient of the hardcopy can do measurements in the field. The `StandardScale` and `CustomScale` properties allow you to set the scale for the objects in model space.

The `StandardScale` property lets you specify a standard scale value from a set of constant values. The constant values include many standard plot scales used in Imperial and metric drawings in addition to the values that fit all objects in model space to the viewport or that use a custom scale. When using the `acVpScaleToFit` constant value, the extents of all the objects in model space are displayed and a custom scale is applied to the viewport.

An example of a constant value that represents a standard scale is `acVp1_4in_1ft`, which assigns a scale of ¼″ = 1′-0″ to a viewport. You can get a full list of the constant values that are supported by searching on `AcViewportScale` in the Object Browser of the VBA Editor.

Unlike the `StandardScale` property, the `CustomScale` property accepts and returns a double value that defines the scale factor for the objects displayed in a viewport. For example, the scale factor of ¼″ = 1′-0″ is 0.02083, which is calculated by dividing 0.25 by 12. To use a custom scale, the `StandardScale` property must be assigned the constant value of `acVpCustomScale`.

The following code statement sets the scale factor of ½″ = 1′-0″ using the constant value `acVp1_2in_1ft` to a viewport assigned to the variable `oVPort`:

```
oVPort.StandardScale = acVp1_2in_1ft
```

Establishing the Settings for Drafting Aids in a Viewport

Although viewports are commonly used to display objects from model space, an end user can work in model space from a viewport. Double-clicking over a viewport enters model space from that viewport. When a viewport is activated, many of the drafting aids available in model space are also available for use from within a viewport. Each viewport stores the current state and settings for several drafting aids.

Table 8.2 lists the properties of an `AcadPViewport` object that enable and control drafting aids related to model space objects in a viewport. The following code statements disable the grid and snap modes for a floating viewport assigned to the *`oPVport`*:

```
oPVport.Grid = False
oPVport.SnapOn = False
```

Table 8.2: Drafting aids–related properties of an `AcadPViewport` object

Property	Description
`GridOn`	Displays the grid in a viewport
`SnapBasePoint`	Specifies the base point of snap mode in a viewport; the base point is an array of three double values
`SnapOn`	Enables snap mode in a viewport
`SnapRotationAngle`	Specifies the rotation angle of snap mode in a viewport
`UCSIconAtOrigin`	Displays the user coordinate system (UCS) icon at the origin of the drawing when the origin is visible in a viewport
`UCSIconOn`	Displays the UCS icon in a viewport
`UCSPerViewport`	Enables the ability to change the orientation of the UCS icon in a viewport

The GridOn and SnapOn properties are used to enable or disable the use of the grid or snap modes but don't affect the grid or snap spacing. The GetGridSpacing and GetSnapSpacing methods return the current X and Y spacing values of the grid and snap modes of a viewport. Typically, the grid and snap spacing values are the same, but they don't need to be. To change the current spacing values of the grid and snap modes, use the SetGridSpacing and SetSnapSpacing methods. The SetGridSpacing and SetSnapSpacing methods accept two double values: the first sets the X spacing value of the grid or snap mode, and the second sets the Y spacing value.

You can learn more about the properties listed in Table 8.2 from the AutoCAD Help system.

Getting Information about a Sheet View

A *sheet view* is a viewport with some additional information. The drawings are part of a sheet set and contain layouts with named views in model space. The AutoCAD Object library doesn't allow you to create or manage sheet views created with the Sheet Set Manager.

NOTE You can use the Sheet Set Object library to create and manage sheet views. I explain how to reference other libraries in Chapter 12, "Communicating with Other Applications."

Table 8.3 lists the properties of an AcadPViewport object that allow you to obtain information about a sheet view.

Table 8.3: Sheet view–related properties of an AcadPViewport object

Property	Description
HasSheetView	Returns True if the viewport is associated with a sheet view
LabelBlockId	Specifies the object ID of the AcadBlock object that is used as the label block for the viewport
SheetView	Specifies the AcadView object that represents the sheet view associated with the viewport

You can learn more about the properties listed in Table 8.3 from the AutoCAD Help system.

Controlling the Output of a Layout

In addition to organizing graphical objects for output, a layout also includes a set of properties known as *plot settings*. The plot settings of a layout specify the device, paper size, scale, and orientation for output. Other settings control the output of a layout to an electronic file or hardcopy. Plot settings can be stored in what is known as a *plot configuration*. In the AutoCAD user interface, a plot configuration is referred to as a *page setup*.

A plot configuration or page setup allows you to apply the same plot settings to multiple layouts. The AcadPlotConfiguration object represents one of the page setups stored in a drawing. All page setups in a drawing are accessed from the `AcadPlotConfigurations` collection object. You obtain the `AcadPlotConfigurations` collection object of a drawing by using the `PlotConfigurations` property of an `AcadDocument` object.

Creating and Managing Named Page Setups

The `Add` function of the `AcadPlotConfigurations` collection object creates a new page setup and returns an `AcadPlotConfiguration` object. When adding a new page setup with the `Add` function, you must provide a string that contains a unique name for the page setup. The `Add` function also accepts a second optional argument of the Boolean data type that specifies the model type for the page setup.

A Boolean value of `True` creates a page setup that can only be applied to the Model tab or `False` for a page setup that can be applied to a named layout. If no model type is specified, the model type is determined by the active layout tab. When working with an existing page setup, you can check its model type by using the `ModelType` property of an `AcadPlotConfiguration` object.

The following code statements create a new page setup named `CheckPlot` that can be applied to a named layout:

```
Dim oPltConfig As AcadPlotConfiguration
Set oPltConfig = ThisDrawing.PlotConfigurations. _
                    Add("CheckPlot", False)
```

If you want to modify an existing page setup, use the `Item` method of the `AcadPlotConfigurations` collection object or a `For` statement to obtain an `AcadPlotConfiguration` object. Once an `AcadPlotConfiguration` object has been obtained, you can then modify the individual properties of the page setup or copy the properties of another page setup. The `CopyFrom` method of an `AcadPlotConfiguration` object copies the plot settings from one page setup to another. It accepts a single argument that specifies an `AcadPlotConfiguration` object type.

TIP The `CopyFrom` method can be used to apply plot settings between two `AcadLayout` or `AcadPlotConfiguration` objects. The objects don't need to be of the same type, so you can copy the plot settings between a layout and a page setup.

When a page setup is no longer needed, it can be removed from the drawing using the `Delete` method of an `AcadPlotConfiguration` object. Even if an `AcadPlotConfiguration` object was applied to a layout, it can be removed from the drawing, as the plot settings of a plot configuration are copied to a layout and not referenced by a layout. This is unlike other named objects. For example, a dimension style is referenced to a dimension object.

Specifying an Output Device and a Paper Size

Plot settings contain two main properties that control the device and paper size to use when outputting a layout. The name of the device to use when outputting a layout is specified by the `ConfigName` property of an `AcadPlotConfiguration` object. A device name can be a system printer configured in Windows or a PC3 file that is created and managed with the AutoCAD Plotter Manager (started with the `plottermanager` command).

The devices you can assign to the plot settings of a layout or page setup are the same ones listed in the Printer/Plotter drop-down list of the Plot and Page Setup dialog boxes. Although you can use the Printer/Plotter drop-down list to get an idea of which devices are available to your programs, the actual name of a device might be different. Use the `GetPlotDeviceNames` function to obtain an array of the names for the available output devices that can be assigned to the `ConfigName` property.

NOTE Before you can use the `GetPlotDeviceNames` function, you must execute the `RefreshPlotDeviceInfo` method. The `RefreshPlotDeviceInfo` method updates the information on the available devices. You must also execute the `RefreshPlotDeviceInfo` method before using the `GetCanonicalMediaNames` function.

The paper size—physical or virtual—used to output a layout is known as a *canonical media name*. A canonical media name is a unique string used to identify a paper size supported by the device assigned to the `ConfigName` property. You specify a canonical media name for the plot settings with the `CanonicalMediaName` property of an `AcadPlotConfiguration` object. The `GetCanonicalMediaNames` function is used to obtain an array of all canonical media names of the device specified by the `ConfigName` property.

In addition to a canonical media name, each paper size has a locale media name. The locale media name is the name of a paper size that is displayed in the Paper Size drop-down list of the Plot and Page Setup dialog boxes. You can get the locale media name of a paper size by passing a canonical media name to the `GetLocaleMediaName` function.

The paper sizes that a device supports have a fixed size and margin, you can get the dimensions of the paper size assigned to the `CanonicalMediaName` property with the `GetPaperSize` method. The `GetPaperSize` method can be used to return two double values that represent the width and height of the paper size, whereas the `GetPaperMargins` method returns two arrays of two double values that represent the number of millimeters from the lower-left and upper-right corners of the paper.

The following code statements display a message box with the paper size and plottable area assigned to the current layout. The plottable area is calculated by the subtracting the margin from the paper size.

```
Dim vLowerLeft As Variant, vUpperRight As Variant
Dim oLayout As AcadLayout
Dim dHeight As Double, dWidth As Double

' Gets the active layout
Set oLayout = ThisDrawing.ActiveLayout

' Gets the margin and paper size
oLayout.GetPaperMargins vLowerLeft, vUpperRight
oLayout.GetPaperSize dWidth, dHeight

MsgBox "Layout paper size: " & CStr(dWidth) & " x " & _
                               CStr(dHeight) & vbLf & _
       "Plottable area: " & _
            CStr(dWidth - vUpperRight(0) - vLowerLeft(0)) & _
            " x " & CStr(dHeight - vUpperRight(1) - vLowerLeft(1))
```

TIP The canonical media name specified indicates the orientation of the paper to output the plot, but the PlotRotation property of an AcadPlotConfiguration object can be used to rotate the paper in 90-degree increments. The PlotRotation property expects a constant value of ac0degrees, ac90degrees, ac180degrees, or ac270degrees.

The following code statements assign the DWF6 ePlot.pc3 device and a paper size of ANSI B to a page setup assigned to the *oPltConfig* variable:

```
' Set the plot device to DWF6 ePlot
oPltConfig.ConfigName = "DWF6 ePlot.pc3"

' Set the paper size to ANSI B
oPltConfig.CanonicalMediaName = "ANSI_B_(17.00_x_11.00_Inches)"
```

The following code statements list all available devices, and the canonical and locale media names of the first paper size of the DWF6 ePlot.pc3 device:

```
Sub ListDevicesAndPaperSizes()
  Dim oPltConfig As AcadPlotConfiguration
  Set oPltConfig = ThisDrawing.PlotConfigurations.Add("Check")

  ' Display introduction text
  ThisDrawing.Utility.Prompt vbLf + "Available devices:"

  ' Update device and paper size information
  oPltConfig.RefreshPlotDeviceInfo

  ' Get the available plot devices
  Dim vDevices As Variant
  vDevices = oPltConfig.GetPlotDeviceNames

  ' Output the names of each device
  For Each sDeviceName In vDevices
    ThisDrawing.Utility.Prompt vbLf + "  " + sDeviceName
  Next sDeviceName

  ' Get the first canonical media name of the DWF6 ePlot device
  oPltConfig.ConfigName = "DWF6 ePlot.pc3"

  Dim vMediaNames As Variant
  vMediaNames = oPltConfig.GetCanonicalMediaNames

  ' Display first canonical media name
  ThisDrawing.Utility.Prompt vbLf + "Canonical media name: " + _
                             vMediaNames(0)

  ' Display first locale media name
  ThisDrawing.Utility.Prompt vbLf + "Locale media name: " + _
                             oPltConfig.GetLocaleMediaName(vMediaNames(0)) + _
                             vbLf
End Sub
```

Here is an example of the output that is created by the sample code:

```
Available devices:
  None
  Snagit 11
  Send To OneNote 2013
  Microsoft XPS Document Writer
  HP ePrint
  Fax
  Default Windows System Printer.pc3
  DWF6 ePlot.pc3
  DWFx ePlot (XPS Compatible).pc3
  DWG To PDF.pc3
  PublishToWeb JPG.pc3
  PublishToWeb PNG.pc3
Canonical media name: ISO_full_bleed_B5_(250.00_x_176.00_MM)
Locale media name: ISO full bleed B5 (250.00 x 176.00 MM)
```

NOTE The `PaperUnits` property controls the units used to represent the scale factor and plot offset settings of a layout or page setup. Use the constant value of `acInches` to set inches as the paper units, `acMillimeters` for millimeters, or `acPixels` for pixels. Pixels are typically used when outputting to a raster image.

Setting a Plot Style as Current

Plot styles are used to control the color, linetype, lineweight, screening, and many other settings that affect the way graphical objects are output. A plot style can be one of two types: color-dependent or named. Color-dependent plot styles are stored in CTB files, and named plot styles are stored in STB files. Plot style files are created and managed with the AutoCAD Plot Style Manager (displayed with the `stylesmanager` command). The name of the plot style to use when outputting a layout is specified by the `StyleSheet` property of an `AcadPlotConfiguration` object.

A drawing file can support only one plot style type at a time, either color-dependent or named. The `pstylemode` system variable indicates whether a drawing is configured to use color-dependent or named plot styles. When the `pstylemode` system variable returns a value of 1, the drawing is configured to use color-dependent plot styles and CTB files. A value of 0 indicates that a drawing can use named plot styles and STB files. Assigning a plot style of the wrong type to a layout causes an error.

The plot styles you can assign to the plot settings of a layout or page setup are the same ones displayed in the Plot Style Table drop-down list of the Plot and Page Setup dialog boxes. The `GetPlotStyleTableNames` function is used to obtain an array of the names for the available plot styles that can be assigned to the `StyleSheet` property.

TIP Before you can use the `GetPlotStyleTableNames` function, you must execute the `RefreshPlotDeviceInfo` method. The `RefreshPlotDeviceInfo` method updates the information on the available plot styles.

The following code statements assign the monochrome.ctb or monochrome.stb file to a plot configuration assigned to a variable named *oPltConfig*:

```
' If pstylemode = 0, then drawing is using named plot styles
' Assign the correct monochrome plot style
If ThisDrawing.GetVariable("pstylemode") = 0 Then
  oPltConfig.StyleSheet = "monochrome.stb"
Else
  oPltConfig.StyleSheet = "monochrome.ctb"
End If
```

The following code statements list all available plot styles:

```
Sub ListPlotStyles()
  Dim oPltConfig As AcadPlotConfiguration
  Set oPltConfig = ThisDrawing.PlotConfigurations.Add("CheckPlot")

  ' Display introduction text
  ThisDrawing.Utility.Prompt vbLf + "Available plot styles:"

  ' Update plot style information
  oPltConfig.RefreshPlotDeviceInfo

  ' Get the available plot styles
  Dim vPStyles As Variant
  vPStyles = oPltConfig.GetPlotStyleTableNames

  ' Output the name of each plot style
  For Each sPSName In vPStyles
    ThisDrawing.Utility.Prompt vbLf + "  " + sPSName
  Next sPSName

  ThisDrawing.Utility.Prompt vbLf
End Sub
```

Here is an example of the output that is created by the sample code:

```
Available plot styles:
  acad.stb
  Autodesk-Color.stb
  Autodesk-MONO.stb
  monochrome.stb
  acad.ctb
  DWF Virtual Pens.ctb
  Fill Patterns.ctb
  Grayscale.ctb
  monochrome.ctb
  Screening 100%.ctb
  Screening 25%.ctb
  Screening 50%.ctb
  Screening 75%.ctb
```

Even when a plot style has been assigned to a layout, the plot style isn't used when outputting a layout unless the `PlotWithPlotStyles` property is set to `True`. Although a plot style can be assigned to a layout or page setup, it can also be used to affect the appearance of graphical objects onscreen. The `ShowPlotStyles` property must be set to `True` before the plot style assigned to a layout affects objects onscreen.

Defining the Area to Output

When plotting a layout, you typically specify the entire layout and not an area within the layout. However, when plotting from the Model tab it is common to plot a small area or the extents of all objects. The `PlotType` property of an `AcadPlotConfiguration` object specifies what should be plotted. The values of the `PlotType` property are the same as those in the What To Plot drop-down list in the Plot Area section of the Plot or Page Setup dialog boxes.

Table 8.4 lists the constant values of the `AcPlotType` enumerator that can be assigned to or returned by the `PlotType` property. The following code statements set the plot type to the extents of the layout:

```
oPltConfig.PlotType = acExtents
```

Table 8.4: Constant values of the `AcPlotType` enumerator

Constant	Description
`acDisplay`	Plot area matches what is shown onscreen.
`acExtents`	The extents of the drawing objects on the layout define the area to plot.
`acLayout`	Margins of the active named layout are used to define the area to plot. Applies only to named layouts.
`acLimits`	Drawing limits of the Model tab define the area to plot. The limits of model space are set with the `limits` command. Applies only to the Model tab.
`acView`	Defines the area to plot with a named view. The named view to plot is set with the `ViewToPlot` property. The `ViewToPlot` property accepts and returns an `AcadView` object. I discussed how to create named views and the `AcadView` object in Chapter 5, "Interacting with the User and Controlling the Current View."
`acWindow`	Two points are used to define a window that sets the area to plot. The `GetWindowToPlot` and `SetWindowToPlot` methods get and set the corners of the window to plot, respectively. Each corner of the window is represented by an array of two double values.

The `PlotOrigin` property of an `AcadPlotConfiguration` object specifies the lower-left corner of the area to plot. The value assigned to or returned by the `PlotOrigin` property is an array of two double values. A plot origin of `0,0` is the most common value. Adjusting the origin shifts the geometry in the output. For example, to shift the geometry 2 units to the right and 1 unit up, use an origin of 2,1. If you are plotting a view or a window, you might want to center the plot on the paper. To center a plot, set the `CenterPlot` property of an `AcadPlotConfiguration` object to `True`.

Changing Other Related Output Settings

Based on the area being plotted, you might want to use lineweights or generate a hidden line view of 3D objects. Table 8.5 lists additional properties you might need to specify when configuring plot settings. The following code statements disable the plotting of viewport borders and enable the plotting of lineweights for the plot configuration assigned to *oPltConfig*:

```
oPltConfig.PlotViewportBorders = False
oPltConfig.PlotWithLineweights = True
```

Table 8.5: Additional plot settings of an `AcadPlotConfiguration` object

Constant	Description
`PlotHidden`	Specifies whether the hidden line view is applied to the objects being plotted.
`PlotViewportBorders`	Specifies whether the border of the viewports on a named layout should be plotted.
`PlotViewportsFirst`	Specifies whether viewports on a named layout should be plotted first.
`PlotWithLineweights`	Specifies whether lineweights assigned to an object are used to affect the appearance of the objects in the drawing when plotted.
`ScaleLineweights`	Specifies whether the lineweights on a layout are scaled.
`SetCustomScale`	Specifies the custom plot scale to apply to the objects being plotted. Set the `UseStandardScale` property to `False` when using a custom scale.
`StandardScale`	Specifies the standard plot scale to apply to the objects being plotted. Set the `UseStandardScale` property to `True` when using a standard scale.
`UseStandardScale`	Specifies whether a custom or standard scale should be used when plotting the specified area.

You can learn more about the properties listed in Table 8.5 from the AutoCAD Help system.

Plotting and Previewing a Layout

Now that you have organized objects on a layout, displayed objects from model space in a floating viewport, and specified the plot settings to use when outputting a layout, you are ready to plot a layout. The `AcadPlot` object, which is obtained using the `Plot` property of the `AcadDocument` object, contains the methods for plotting a layout. Before you can plot, you must add the names of the layouts to be plotted to an array of string values. Once you've defined the array, you pass it to the `SetLayoutsToPlot` method, which lets AutoCAD know the layouts to be plotted next.

Plot Today, Print Tomorrow

Plotting is often associated with large output devices, but it can also mean to print using a system printer—a small inkjet or laser printer that uses letter, A4, and even legal- or tabloid-size paper. The GetPlotDeviceNames function returns not only the plotters configured for AutoCAD, but also the system printers available. As with plotters, you assign a system printer to a layout or plot configuration with the ConfigName property. The media sizes supported by the system printer can be retrieved using the GetCanonicalMediaNames function and specifying the media size to use with the CanonicalMediaName property.

For example, if a printer named First Floor Copy Room Printer is configured in Windows and you use it to print your Microsoft Word documents, you can assign the same name to the ConfigName property of the layout or plot configuration in an AutoCAD drawing. The values of Letter and A4 represent two of the possible paper sizes that can be assigned to the CanonicalMediaName property.

As a programmer, you'll find that configuring a layout or plot configuration can be challenging. After all, you don't have control over which devices or media sizes a user has access to. The best solution is to prompt the user to select the device and media size your program should use via a user form. I explain how to work with user forms and controls in Chapter 11, "Creating and Displaying User Forms." After the user provides you with the device and media size to use, you can store the values in the Windows Registry and retrieve the values when needed. Using this approach, your program can be adapted for use in different environments.

The following code statements create an array of two layout names and set them to be plotted:

```
' Assign the names of layouts to plot
Dim sLayoutNames(1) As String
sLayoutNames(0) = "Layout1"
sLayoutNames(1) = "Layout2"

' Set the layouts to plot
ThisDrawing.Plot.SetLayoutsToPlot sLayoutNames
```

After the layouts to be plotted have been specified with the `SetLayoutsToPlot` method, they can be plotted to a hardcopy using the `PlotToDevice` method or to an electronic file using the `PlotToFile` method. When you're using the `PlotToDevice` or `PlotToFile` method, each layout specified can be plotted using its own plot settings, or the plot settings of each layout can be overridden with the settings of an `AcadPlotConfiguration` object. The `PlotToDevice` and `PlotToFile` methods return a Boolean value of `True` if all layouts are successfully plotted; otherwise, `False` is returned.

TIP The `BeginPlot` and `EndPlot` events can be used to monitor the start and end of the plot process. These events are members of the `AcadDocument` object. I explain how to use events for the `AcadDocument` object in Chapter 10.

The following shows the syntax of the `PlotToDevice` and `PlotToFile` methods:

```
object.PlotToDevice [plotConfig]
object.PlotToFile fileName [, plotConfig]
```

Their arguments are as follows:

object The *object* argument represents the variable assigned the AcadPlot object that you will be working with.

fileName The *fileName* argument is a string that specifies the full path of the file to be created.

plotConfig The *plotConfig* argument is optional and of the AcadPlotConfiguration object type. The AcadPlotConfiguration object overrides the plot settings of the layouts specified by the SetLayoutsToPlot method.

The PlotToDevice and PlotToFile methods are affected by the backgroundplot system variable. When the backgroundplot system variable is set to 2 or 3, plotting occurs in the background; the plotting can take longer, but the VBA program completes sooner.

NOTE When you're plotting a layout with the PlotToDevice and PlotToFile methods, any errors that are generated while plotting are displayed in message boxes. The QuietErrorMode property of the AcadPlot object can be used to disable the error message displays during plotting; a plot log is generated instead. Set the QuietErrorMode property to True to log plot errors and disable the error message boxes when plotting.

The following code statements plot the layouts specified by the SetLayoutsToPlot method and each layout's plot settings:

```
' Plot the layouts quietly
ThisDrawing.Plot.QuietErrorMode = True

If ThisDrawing.Plot.PlotToDevice Then
  MsgBox "Layouts successfully plotted."
End If
```

As an alternative to immediately plotting a layout, you can display a layout in the Preview window and let the user decide whether to plot the layout based on the preview. The DisplayPlotPreview method displays the current layout in the Preview window, which is the same as the one opened with the Preview button in the Plot dialog box or when the preview command is executed. The execution of the VBA macro is suspended until the Preview window is dismissed.

NOTE An error is generated if you call the SetLayoutsToPlot method before calling the DisplayPlotPreview method.

Exporting and Importing File Formats

Although the objects on a layout can be plotted to an electronic file with a configured device, you can also export the objects of a drawing to a supported file format. An exported file can be used in a presentation, imported into an analysis software package, or even used to print a prototype in a 3D printer. The Export method of the AcadDocument object allows you to export specified objects from a drawing. Exporting objects from a drawing requires you to specify the name and location of the file, a file extension, and the objects to export.

The filename and location you pass to the `Export` method can't include a file extension; the file type is determined by the file extension specified. The file extensions `.wmf`, `.sat`, `.eps`, `.dxf`, and `.bmp` are supported. The graphical objects you want to export must be passed to the `Export` method using an `AcadSelectionSet` object. As an alternative, you can allow the user to select which objects to export by passing an `AcadSelectionSet` object with no objects or specify a value of `Nothing` to export all objects in a drawing.

A previously exported or supported file created by another application can be imported into a drawing with the `Import` method, which is a member of the `AcadDocument` object. The `Import` method requires you to specify the full filename and location of the file you want to import, as well as an insertion point and scale factor to control the placement and size of the imported objects.

The following code statements export all objects in a drawing to a DXF file and then import them back into the current drawing at half of their original scale:

```
' Export objects to a DXF file
Dim sDXFFile As String, sFileExt As String
sDXFFile = ThisDrawing.GetVariable("MyDocumentsPrefix") & _
           "\ACP_Sample"
sFileExt = "DXF"

ThisDrawing.Export sDXFFile, sFileExt, Nothing

' Import a DXF file
Dim dInsPt(2) As Double
dInsPt(0) = 0: dInsPt(1) = 0: dInsPt(2) = 0

ThisDrawing.Import sDXFFile & "." & sFileExt, dInsPt, 0.5
```

Exercise: Adding a Layout to Create a Check Plot

As part of the design process, many companies create what is known as a *check plot*. A check plot is a hardcopy of a layout that is used by an engineer to review a design that was created in AutoCAD. During the review, comments and markups are handwritten on the hardcopy and then passed back to the drafter for corrections. Over time, the review process has been slowly moving from an analog process (hardcopy) to being digitally done on a workstation.

In this section, you will continue to work with the `DrawingSetup` project that you created in Chapter 3. As part of the existing project, you will create several new procedures that create and configure an output device so a check plot can be output. The key concepts that are covered in this exercise are as follows:

Creating and Working with a Layout Layouts allow you to organize objects in a drawing for output. Once a layout has been created, annotation and viewports can be added to help communicate a design.

Configuring the Plot Settings of a Layout Before a layout can be output, you must specify the device and paper size you want to use, among other settings that control the appearance of the objects on the layout.

Adding and Modifying Viewports Viewports are used to control which objects from model space you want to display as part of a layout. Each viewport can be assigned a different scale to control the size at which the objects from model space are displayed.

Plotting a Layout Plotting a layout allows you to output a design to hardcopy or an electronic file to share with others.

NOTE The steps in this exercise depend on the completion of the steps in the "Exercise: Setting Up a Project" section of Chapter 3. If you didn't complete the steps, do so now or start with the `ch08_drawingsetup.dvb` sample file available for download from `www.sybex.com/go/autocadcustomization`. Place this sample file in the `MyCustomFiles` folder under the `Documents` (or `My Documents`) folder, or the location you are using to store the DVB files. After the files are saved to the location you are using to store DVB files, remove `ch08_` from the filename. You will also need the sample files `ch08_building_plan.dwg`, `ch08_clsUtilities.cls`, and `b-tblk.dwg` for this exercise.

Creating the Layout

A layout is used to organize objects from model space and the annotation required to communicate the design within viewports. Depending on the type of drawings you work with, there can be benefits to creating layouts dynamically as they are needed instead of manually adding them to your drawings. The following steps explain how to create a procedure named `AddCheckPlotLayout` to the `drawingsetup.dvb` project:

1. Open the VBA Editor and load the `drawingsetup.dvb` file.
2. In the VBA Editor Project Explorer, double-click the code module named `basDrawingSetup`.
3. When the code editor window opens, scroll to the bottom and click after the last `End Sub` statement. Press Enter twice.
4. Type the following; the comments are included for your information and don't need to be typed:

```
' Adds a new layout based on the name passed to the function
Private Function AddLayout(sLayoutName As String) As AcadLayout
  On Error Resume Next

  ' Get the layout
  Set AddLayout = ThisDrawing.Layouts(sLayoutName)

  ' If an error is generated, the layout doesn't exist
  If Err Then
    Err.Clear

    ' Add the layout
    Set AddLayout = ThisDrawing.Layouts.Add(sLayoutName)
  End If
End Function
```

5. On the menu bar, click File ➢ Save.

Adding and Modifying a Plot Configuration

Plot settings control how a layout is output to a device (printer, plotter, or file). You can modify the plot settings of a layout directly or create a named plot configuration and then copy those plot settings to a layout.

The following steps define a procedure named `AddPlotConfig`, which is a helper function used to create a plot configuration based on a device and media size. You will use this function later to create a new plot configuration or return the plot configuration if it already exists in the drawing. The function returns an `AcadPlotConfiguration` object that represents the new plot configuration.

1. In the code editor window, scroll to the bottom and click after the last `End Function` statement. Press Enter twice.

2. Type the following; the comments are here for your information and don't need to be typed:

```
' Adds a plot configuration based on the name and values
' passed to the function
Private Function AddPlotConfig(sPltConfigName As String, _
                               sDeviceName As String, _
                               sMediaName As String, _
                               sPlotStyleName As String, _
                               bModelType As Boolean, _
                               nPlotType As AcPlotType, _
                               nPlotRotation As AcPlotRotation, _
                              ) As AcadPlotConfiguration
  On Error Resume Next

  ' Get the plot configuration
  Set AddPlotConfig = ThisDrawing. _
                        PlotConfigurations(sPltConfigName)

  ' If an error is generated, the plot configuration doesn't exist
  If Err Then
    Err.Clear

    ' Add the plot configuration
    Set AddPlotConfig = ThisDrawing. _
                          PlotConfigurations. _
                          Add(sPltConfigName, bModelType)

    ' Assign a device name
    AddPlotConfig.ConfigName = sDeviceName

    ' Assign a media name
    AddPlotConfig.CanonicalMediaName = sMediaName

    ' Assign a plot style name
    AddPlotConfig.StyleSheet = sPlotStyleName

    ' Assign the layout plot type
    AddPlotConfig.PlotType = nPlotType
    ' Assign the plot rotation
    AddPlotConfig.PlotRotation = nPlotRotation
  End If
End Function
```

3. On the menu bar, click File ➢ Save.

Inserting a Title Block

Title blocks are a form of annotation that is used to help identify and communicate the project with which the drawing is associated. Depending on your design, a title block might display the location of a building, the model number of a new part to be manufactured, revision history, and much more. In the exercises in Chapter 3, you inserted the title block `b-tblk.dwg` into a drawing using the `insert` command with the `SendCommand` method, but as I explained earlier in the book, using commands for this kind of operation has drawbacks.

In the next steps, you will create a new procedure named `AddBlkReference` that will insert a title block onto a specified layout with a known location, rotation, and scale. The procedure will then be used later to insert that same block.

1. In the code editor window, scroll to the bottom and click after the last `End Function` statement. Press Enter twice.
2. Type the following; the comments are here for your information and don't need to be typed:

```
' Insert a block onto a specified layout
Private Function AddBlkReference(oLayout As AcadLayout, _
                                 sBlkName As String, _
                                 vInsPoint As Variant, _
                                 dRotation As Double, _
                                 dScale As Double _
                                 ) As AcadBlockReference
  On Error Resume Next

  ' Insert the block
  Set AddBlkReference = oLayout.Block. _
                           InsertBlock(vInsPoint, _
                                       sBlkName, _
                                       dScale, dScale, dScale, _
                                       dRotation)

  ' If an error is generated, return Nothing
  If Err Then
    Err.Clear

    Set AddBlkReference = Nothing
  End If
End Function
```

3. On the menu bar, click File ➢ Save.

Displaying Model Space Objects with a Viewport

The most common objects placed on a layout after annotation objects are viewports. Viewports are used to display model space objects at a specific scale.

In the next steps, you will create a new procedure named `AddFloatingViewport` that adds a floating viewport to the specified paper space block with a known center, width, and height.

1. In the code editor window, scroll to the bottom and click after the last `End Function` statement. Press Enter twice.
2. Type the following; the comments are here for your information and don't need to be typed:

```
' Add a floating viewport to a layout
Private Function AddFloatingViewport(oPSpace As AcadPaperSpace, _
                                     vCenterPoint As Variant, _
                                     dWidth As Double, _
                                     dHeight As Double _
                                     ) As AcadPViewport
  On Error Resume Next

  ' Add the Viewport
  Set AddFloatingViewport = oPSpace. _
                               AddPViewport(vCenterPoint, _
                                            dWidth, _
                                            dHeight)

  ' If an error is generated, return Nothing
  If Err Then
    Err.Clear
    Set AddFloatingViewport = Nothing
  End If
End Function
```

3. On the menu bar, click File ➢ Save.

Putting It All Together

Now that you have defined functions that create a layout and plot configuration, insert a block, and add a floating viewport, it is time to put them all to work. In addition to using the functions defined in this exercise, you will use the `createlayer` function from the `clsUtilities` class to create a few new layers if they aren't present in the drawing file.

In these steps, you'll import the class module named `clsUtilities.cls` and define a global variable, which will be used to access the procedures defined in the `clsUtilities` class:

1. In the VBA Editor, in the Project Explorer, right-click the `DrawingSetup` project and choose Import File.
2. When the Import File dialog box opens, browse to and select the `clsUtilities.cls` file in the `MyCustomFiles` folder. Click Open.

The clsUtilities.cls file contains the utility procedures that you created as part of the DrawPlate project or downloaded as part of the sample files for this book.

3. In the Project Explorer, double-click the code module named basDrawingSetup.
4. In the text editor area of the basDrawingSetup component, scroll to the top and add the following on a new line:

   ```
   Private myUtilities As New clsUtilities
   ```

   The createlayer function is now available for use in the basDrawingSetup code module.

In the next steps, you will create a new procedure named CheckPlot. This will be the main procedure that the end user executes from the AutoCAD user interface. This new procedure creates a layout and plot configuration named CheckPlot, inserts the title block stored in the drawing file named b-tblk.dwg, creates a new floating viewport, and outputs the layout using the assigned device to a file named checkplot.dwf.

1. In the code editor window, scroll to the bottom and click after the last End Function statement. Press Enter twice.
2. Type the following; the comments are here for your information and don't need to be typed:

   ```
   ' Creates a function that creates a new layout named CheckPlot,
   ' sets the output device for the layout to the DWF ePlot.pc3 file,
   ' inserts a title block for a ANSI B size sheet of paper and
   ' plots the layout.
   Public Sub CheckPlot()
     On Error Resume Next

     ' Check to see if the CheckPlot layout already exists,
     ' and if so set it current
     Dim oLayout As AcadLayout
     Set oLayout = ThisDrawing.Layouts("CheckPlot")

     If Err Then
       Err.Clear

       ' Store and change the default for creating a viewport
       ' when a new layout is created
       Dim bFlag As Boolean
       bFlag = ThisDrawing.Application. _
                 Preferences.Display.LayoutCreateViewport

       ThisDrawing.Application. _
                 Preferences.Display.LayoutCreateViewport = False

       ' Use the AddLayout function to create
   ```

```
' the CheckPlot layout
Set oLayout = AddLayout("CheckPlot")

' Set the new layout current
ThisDrawing.ActiveLayout = oLayout

' Use the AddPlotConfig function to create
' the CheckPlot plot configuration
Dim oPltConfig As AcadPlotConfiguration
Set oPltConfig = AddPlotConfig("CheckPlot", "DWF6 ePlot.pc3", _
                               "ANSI_B_(17.00_x_11.00_Inches)", _
                               False, "acad.ctb", _
                               acLayout, ac0degrees)

' Assign the plot configuration to the layout
oLayout.CopyFrom oPltConfig

' Use the AddBlkReference function to insert
' the title block named b-tblk.dwg onto the layout
Dim sTitleBlkName As String
sTitleBlkName = ThisDrawing.GetVariable("mydocumentsprefix") & _
                  "\MyCustomFiles\b-tblk.dwg"

Dim dInsPt(2) As Double
dInsPt(0) = 0: dInsPt(1) = 0: dInsPt(2) = 0

Dim oBlkRef As AcadBlockReference
Set oBlkRef = AddBlkReference(oLayout, sTitleBlkName, _
                              dInsPt, 0, 1)

' If a block reference was returned, place it on the Tblk layer
If Not oBlkRef Is Nothing Then
  ' Add the layer for the title block
  oBlkRef.Layer = myUtilities.CreateLayer("TBLK", 8).Name
End If

' Add a viewport to the layout
Dim dCPt(2) As Double
dCPt(0) = 6.375: dCPt(1) = 4.875: dCPt(2) = 0

Dim oVport As AcadPViewport
Set oVport = AddFloatingViewport(ThisDrawing.PaperSpace, _
                                 dCPt, 12.55, 9.55)

' If a floating viewport was returned, place it on the Vport layer
If Not oVport Is Nothing Then
  ' Turn the viewport On
```

```
    oVport.Display True

    ' Add the layer for the viewport and set it to not plottable
    Dim oLayer As AcadLayer
    Set oLayer = myUtilities.CreateLayer("Vport", 9)
    oLayer.Plottable = False

    ' Assign the layer for the viewport
    oVport.Layer = oLayer.Name

    ' Set the scale of the viewport to Fit
    oVport.StandardScale = acVpScaleToFit
  Else
    MsgBox "Warning: The viewport couldn't be created."
  End If

  ' Restore viewport creation for new layouts
  ThisDrawing.Application. _
     Preferences.Display.LayoutCreateViewport = bFlag
Else
  ' Set the new layout current
  ThisDrawing.ActiveLayout = oLayout
End If

' Zoom to the extents of the layout
ThisDrawing.Application.ZoomExtents

' Regen the drawing
ThisDrawing.Regen acActiveViewport

' Re-establish the area to plot is the layout
ThisDrawing.ActiveLayout.PlotType = acLayout

' Prompt the user if the check plot should be created now
If MsgBox("Do you want to create the check plot?", _
          vbYesNo) = vbYes Then
  With ThisDrawing.Plot
    ' Assign the CheckPlot layout for plotting
    .SetLayoutsToPlot Array(oLayout)

    ' Define the name of the DXF file to create
    Dim sDWFName As String
    sDWFName = ThisDrawing.GetVariable("mydocumentsprefix") & _
                    "\MyCustomFiles\checkplot.dwf"

    ' Plot the DWF file and display a message if the
    ' plot was unsuccessful
```

```
          If .PlotToFile(sDWFName) = False Then
            MsgBox "The CheckPlot layout couldn't be output." & _
                   vbLf & "Check the device and plot settings."
          End If
        End With
      End If
    End Sub
```

3. On the menu bar, click File ➢ Save.

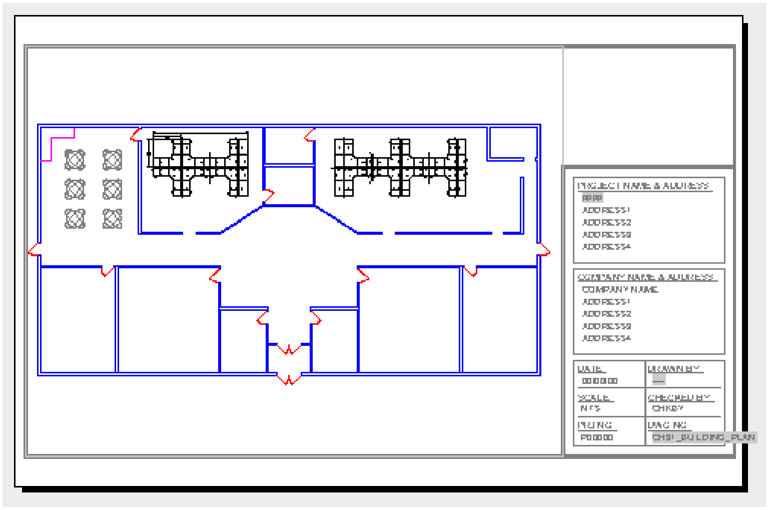
Testing the CheckPlot Procedure

The following steps explain how to test the `CheckPlot` procedure:

1. Switch to the AutoCAD application window.
2. Open `Ch08_Building_Plan.dwg`.
3. At the Command prompt, type **`vbarun`** and press Enter.
4. When the Macros dialog box opens, select the `RoolLabel.dvb!basDrawingsetup.CheckPlot` macro from the list and click Run.

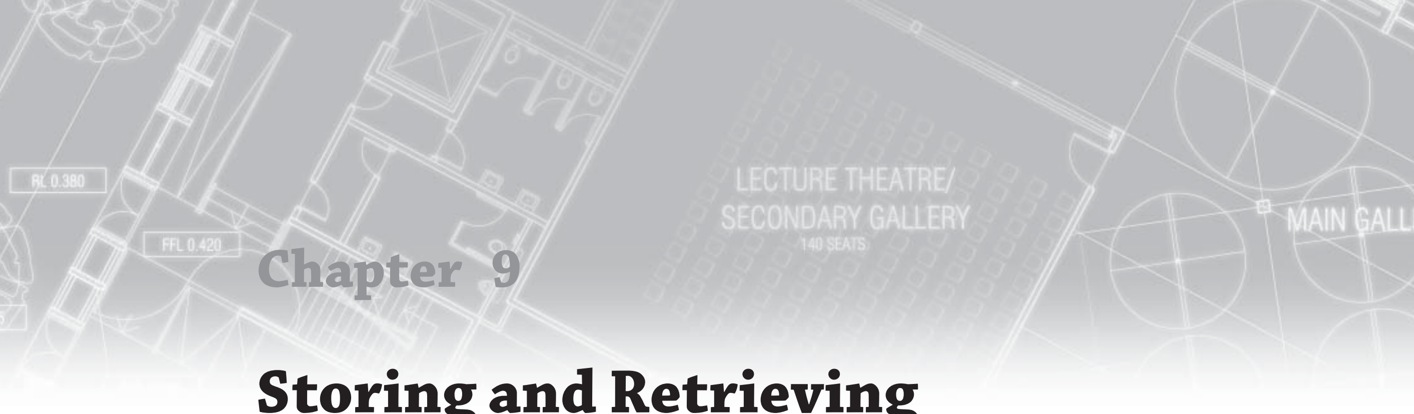
   The new layout named CheckPlot is set as current, as shown in Figure 8.1.

FIGURE 8.1 New layout with a title block



5. When the message box opens, click Yes to create the DWF file in the `MyCustomFiles` folder.

   Open the `checkplot.dwf` file that is generated with the Autodesk Design Review program (`http://usa.autodesk.com/design-review/`) or a similar program.



Chapter 9

Storing and Retrieving Custom Data

There are times when it would be nice to have a custom program store values and then retrieve them at a later time. Although you can use a global variable to temporarily store a value while the custom program remains in memory, global variables do not persist across multiple sessions. Using the AutoCAD® Object library and VBA, you can store values so that they persist between drawing or AutoCAD sessions.

If you want a value to be available when a drawing is open, you can use extended data (Xdata) or a custom dictionary. (I introduced the use of dictionaries in Chapter 8, "Annotating Objects," and how they are used for storing annotation styles such as table and multileader styles.) Xdata can be attached to an object as a way to differentiate one object from another or, in some cases, to affect the way an object might look in the drawing area.

Values can be stored in the Windows Registry and retrieved from any AutoCAD session that your custom program is loaded into. The values stored in the Windows Registry can represent strings, 2D or 3D points, integers, and doubles. As an alternative, the values can be written to a text file and read at a later time. (I discuss how to work with external files in Chapter 12, "Communicating with Other Applications.")

Extending Object Information

Each object in a drawing has a preestablished set of properties that define how that object should appear or behave. For example, these properties are used to define the size of a circle or the location of a line within a drawing. Although you can't use VBA to add a new property to an object, you can append custom information to an object. The custom information that is appended to an object is known as *Xdata*.

Xdata is structured using two arrays. The first array contains the data types for the values to be stored (DXF group codes); the second array contains the values to be stored. The two arrays must contain the same number of elements. As part of the values to be stored, the first value must be an application name to identify the custom program that added the Xdata. After the application name, the array can contain any supported values. Supported values are strings, integers, doubles, and entity names, among others.

The values that make up the Xdata and what they represent is up to you, the creator of the data. Data in the Xdata arrays can be used to identify where an object should be placed or which layer it should be on, to store information about an external database record that is related to an object, or to build relationships between objects in a drawing. The way data is used or enforced is up to you as the programmer.

In addition to Xdata, graphical and nongraphical objects support what are known as *extension dictionaries*. Extension dictionaries are kind of like record tables that can be attached to an object. For example, you could store revision history of a drawing in an extension dictionary that is attached to model space, and then populate the drawing title block with that information. Even AutoCAD uses extension dictionaries to implement features, such as Layer States and Filters, which are attached to the Layer symbol table. I discuss creating custom dictionaries in greater detail in the "Creating and Modifying a Custom Dictionary" section later in this chapter.

Working with Xdata

Attaching Xdata to an object requires you to do some initial planning and perform several steps.

Appending Xdata

The following list outlines the steps that you must perform in order to attach Xdata to an object:

1. Check to see if the object already has Xdata attached and with what application name.

   If Xdata is already attached with the application name you planned to use, skip to the "Replacing Xdata" section.

2. Define and register an application name for your custom program.
3. Define the array that will hold the DXF group codes that will specify the data types for the data values array; the first element in the array should be `1001`, which represents the DXF group code for the application name.
4. Define the array that will hold the data values for the Xdata; the first element in the array should be a string that represents the application name.
5. Get the object to which you wish to append the Xdata.
6. Append the Xdata to the object with the `SetXData` method.

Replacing Xdata

Prior to appending Xdata, you should check to see if the object already has Xdata with your custom program's application name attached to it. If that's the case, you should replace the current Xdata with the new. Follow these steps to modify the Xdata previously attached to an object:

1. Define the values that will make up the Xdata.
2. Define the array that will hold the DXF group codes that will be used to represent the data types of the data values array; the first element in the array should be `1001`, which represents the DXF group code for the application name.
3. Define the array that will hold the data values for the Xdata; the first element in the array should be a string that represents the application name.
4. Get the object for which you wish to replace the Xdata.

5. Use the `GetXData` method to check for the existence of Xdata for the application name.
6. Substitute the current Xdata attached to an object with the new Xdata.
7. Update the object.

Defining and Registering an Application Name

Before you can attach Xdata to an object, you must decide on an application name and then register that name with the current drawing. The application name you choose should be unique to avoid conflicts with other Xdata that could potentially be attached to an object. After you choose an application name, register the name with the `Add` method of the `AcadRegisteredApplications` collection object. The `Add` method accepts a single string argument that is the name of the application you want to register, and it returns the new `AcadRegisteredApplication` object.

The following example demonstrates how to register an application:

```
' Registers the application named MyApp
Dim sAppName as String
sAppName = "MyApp"

Dim oRegApp As AcadRegisteredApplication
Set oRegApp = ThisDrawing.RegisteredApplications.Add(sAppName)
```

Attaching Xdata to an Object

Once you have defined and registered an application name, you can attach Xdata to an object within that drawing. Xdata is made up of two arrays and has a total size limit of 16 KB per object. (See the “Monitoring the Memory Used by Xdata for an Object” sidebar for more information.) The first array defines the data types of the values to be stored using DXF group codes, whereas the second array defines the actual values. The two arrays are used for what is known as a *dotted pair*. A dotted pair in AutoCAD is a relationship of a data type and value that has the format of `(dxftype . value)` to programming languages such as the AutoLISP® and ObjectARX® languages.

The DXF group codes used in the data type array of Xdata must be within the range of 1000 to 1071. Each DXF group code value in that range represents a different type of data, and you can use each DXF group code more than once in the data type array for Xdata. Table 9.1 lists some of the commonly used DXF group codes for Xdata.

TABLE 9.1: Xdata-related DXF group codes

DXF GROUP CODE	DESCRIPTION
1000	String value
1001	Application name

Table 9.1: Xdata-related DXF group codes *(continued)*

DXF group code	Description
`1010`	3D point
`1040`	Real numeric value
`1070`	16-bit (unsigned or signed) integer value
`1071`	32-bit signed integer value

The following arrays define Xdata that contains the application name MyApp, a string value with the text "My custom application," and a double that represents the current date:

```
' Define the data types array for the Xdata
Dim nXdTypes(2) As Integer
nXdTypes(0) = 1001
nXdTypes(1) = 1000
nXdTypes(2) = 1071

' Define the data values array for the Xdata
Dim vXdVals(2) As Variant
vXdVals(0) = "MyApp"
vXdVals(1) = "My custom application"
vXdVals(2) = CLng(ThisDrawing.GetVariable("cdate"))
```

The array that defines the data types of the values in the Xdata must be defined as the integer data type, whereas the data values array for the Xdata should be defined as the variant data type. Once the arrays that will make up the Xdata have been defined, the Xdata can be attached to an object with the `SetXData` method.

The following shows the syntax of the `SetXData` method:

```
object.SetXData dataTypes, dataValues
```

Its arguments are as follows:

`object` The *`object`* argument represents the AutoCAD object that you want to attach Xdata to.

`dataTypes` The *`dataTypes`* argument is an array of integers that represent the types of data values to be stored with the object's Xdata.

`dataValues` The *`dataValues`* argument is an array of variants that represent the data values to be stored with the object's Xdata.

After the Xdata has been attached to an object, you might need to execute the object's `Update` method to refresh the object if the Xdata affects the appearance of the object. I explained how to use the `Update` method in Chapter 5, "Interacting with the User and Controlling the Current View."

This exercise shows how to attach Xdata to a circle:

1. At the AutoCAD Command prompt, type **vbaman** and press Enter.
2. When the VBA Manager opens, click New.
3. Click Visual Basic Editor.
4. In the VBA Editor, in the Project Explorer, double-click the ThisDrawing component.
5. In the code editor window, type the following:

```
Sub AddXDataToCircle()
  ' Registers the application named MyApp
  Dim sAppName As String
  sAppName = "MyApp"

  Dim oRegApp As AcadRegisteredApplication
  Set oRegApp = ThisDrawing.RegisteredApplications.Add(sAppName)

  ' Define the data types array for the Xdata
  Dim nXdTypes(2) As Integer
  nXdTypes(0) = 1001
  nXdTypes(1) = 1000
  nXdTypes(2) = 1071

  ' Define the data values array for the Xdata
  Dim vXdVals(2) As Variant
  vXdVals(0) = "MyApp"
  vXdVals(1) = "My custom application"
  vXdVals(2) = CLng(ThisDrawing.GetVariable("cdate"))

  ' Define center point for the circle
  Dim dCenPt(2) As Double
  dCenPt(0) = 2: dCenPt(1) = 2: dCenPt(2) = 0

  ' Add a circle object to model space
  Dim oCirc As AcadCircle
  Set oCirc = ThisDrawing.ModelSpace.AddCircle(dCenPt, 1)

  ' Assign the Xdata to the circle object
  oCirc.SetXData nXdTypes, vXdVals
End Sub
```

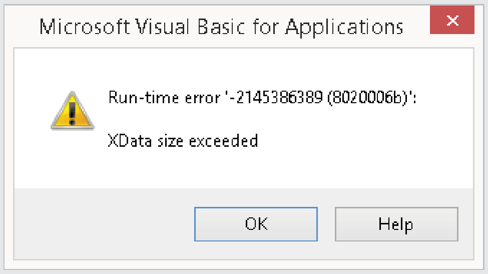
6. Switch to AutoCAD.
7. At the Command prompt, type **vbarun** and press Enter.
8. When the Macros dialog box opens, select the `Global`*`N`*`!ThisDrawing.AddXDataToCircle` macro and click Run.
9. Save the project if you want, but don't close it as you will continue with the project in the next exercise.

A new circle with a center point of 2,2 and radius of 1 is added to model space along with the Xdata attached to it. The circle won't look any different than a circle without the Xdata attached to it because the Xdata doesn't affect the way the AutoCAD program draws the object. However, you can now identify this circle from those that might be created with the `circle` command. For example, you could use Xdata to tag a circle that represents a drill hole in your drawing. By identifying the circle as a drill hole, you make it easier to locate and update the circles that represent drill holes as needed in the drawing.

Monitoring the Memory Used by Xdata for an Object

Each object in a drawing can have a maximum of 16 KB of Xdata attached to it. The 16 KB is the total of all Xdata attached to an object, and not just for one application. If the limit of Xdata is close and you attach additional Xdata that exceeds the limit, the Xdata won't be attached. AutoLISP provides two functions that help to determine the size of the Xdata being attached to an object and the amount of space already being used by the Xdata attached to an object.

The AutoCAD Object library doesn't support any functions that can be used to manage Xdata, but when the limit is exceeded an error is generated. You can use the VBA error-handling features to catch and respond to the error accordingly.


Microsoft Visual Basic for Applications

Run-time error '-2145386389 (8020006b)':

XData size exceeded

Querying and Modifying the Xdata Attached to an Object

Xdata that has been previously attached to an object can be queried and modified by following a process that is similar to the one used to attach Xdata to an object. The `GetXData` method of an object is used to get the Xdata attached to an object for a specific application or all applications. Two arrays are returned by the `GetXData` method. You can use the `IsArray` function to check whether the values returned by the `GetXData` method are empty. If a value of `True` is returned by the `IsArray` function, the object has Xdata attached to it for the specified application name.

The following shows the syntax of the `GetXData` function:

```
object.GetXData appName, dataTypes, dataValues
```

Its arguments are as follows:

`object` The *`object`* argument represents the AutoCAD object that you want to retrieve Xdata from.

`appName` The *`appName`* argument is a string that represents the application name of the Xdata you want to retrieve. Using an empty string returns the Xdata for all applications that have Xdata attached to the object.

dataTypes The *dataTypes* argument must be a variant and is assigned the current types of data that are stored with the object's Xdata. The variant that is returned contains an array of integer values.

dataValues The *dataValues* argument must be a variant and is assigned the current data values that are stored with the object's Xdata.

The following code statements return the Xdata for the application named MyApp if attached to the last object in model space:

```
' Get the last object added to model space
Dim oAcadObj As AcadObject
Set oAcadObj = ThisDrawing.ModelSpace(ThisDrawing.ModelSpace.Count - 1)

' Get the Xdata for the MyApp application name
Dim vXdTypes As Variant, vXdVals As Variant
oAcadObj.GetXData "MyApp", vXdTypes, vXdVals
```

Using an empty string instead of an actual application name returns the Xdata for all applications attached to an object, as shown here:

```
' Get the Xdata for all applications
Dim vXdTypes As Variant, vXdVals As Variant
oAcadObj.GetXData "", vXdTypes, vXdVals
```

This exercise shows how to list the Xdata attached to a dimension with a dimension override:

1. At the AutoCAD Command prompt, type **`dli`** press Enter.
2. At the `Specify first extension line origin or <select object>:` prompt, specify a point in the drawing.
3. At the `Specify second extension line origin:` prompt, specify a second point in the drawing.
4. At the `Specify dimension line location or [Mtext/Text/Angle/Horizontal/Vertical/Rotated]:` prompt, specify a point in the drawing to place the linear dimension.
5. Select the linear dimension that you created, right-click, and then click Properties.
6. In the Properties palette, click the Arrow 1 field under the Lines & Arrows section. Select None from the drop-down list.

   The first arrowhead of the linear dimension is suppressed as a result of a dimension override being created.
7. In the VBA Editor, open the code editor window for the ThisDrawing component of the project you created in the previous exercise. Type the following:

```
Sub RetreiveXDataForLastObject()
  ' Get the last object added to model space
  Dim oAcadObj As AcadObject
  Set oAcadObj = ThisDrawing.ModelSpace(ThisDrawing.ModelSpace.Count - 1)

  ' Get the Xdata attached to the object
```

```
    Dim vXdTypes As Variant, vXdVals As Variant
    oAcadObj.GetXData "", vXdTypes, vXdVals

    ' Check to see whether the value returned is an array
    ' An array means Xdata is present
    If IsArray(vXdTypes) Then
      Dim sMsg As String
      sMsg = "Xdata Values" & vbLf

      Dim nCnt As Integer

      ' Append the values of the Xdata to the sMsg variable
      For nCnt = 0 To UBound(vXdVals)
        sMsg = sMsg & "Value (" & CStr(nCnt) & ") " & vXdVals(nCnt) & vbLf
      Next nCnt

      ' Display the value of the sMsg variable
      MsgBox sMsg
    End If
  End Sub
```

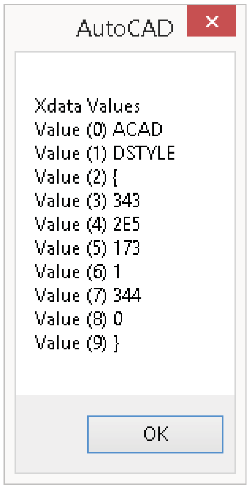
8. Switch to AutoCAD.
9. At the Command prompt, type **`vbarun`** and press Enter.
10. When the Macros dialog box opens, select the `Global`*`N`*`!ThisDrawing.RetreiveXDataForLastObject` macro and click Run.

    Attaching Xdata to the linear dimension is how the AutoCAD program handles dimension overrides for individual dimensions. Figure 9.1 shows what the Xdata attached to the linear dimension looks like as a result of changing the Arrow 1 property in step 6.
11. Save the project if you want, but don't close it, as you will continue with the project in the next exercise.

NOTE I mentioned earlier that Xdata doesn't affect the appearance of an object, and that is still true even when used as we did in the previous exercise. Xdata itself doesn't affect the object, but AutoCAD does look for its own Xdata and uses it to control the way an object might be drawn. If you implement an application with the ObjectARX application programming interface, you could use ObjectARX and Xdata to control how an object is drawn onscreen. You could also control the way an object looks using object overrules with Managed .NET and Xdata. ObjectARX and Managed .NET are the two advanced programming options that Autodesk supports for AutoCAD development. You can learn more about ObjectARX and Managed .NET at `www.objectarx.com`.

As shown in the previous exercise, the `IsArray` function can be used to determine whether Xdata for a specific application is already attached to an object by getting the values returned by the `GetXData` method. If Xdata is already attached to an object for a specific application name, assigning new values with the same application will overwrite the previous Xdata that was attached.

Modifying Xdata that is already attached requires you to get the current Xdata with the `GetXData` method and then re-dimension the array using the `ReDim` and `Preserve` statements. Which approach you use depends on whether you need to replace or modify the existing Xdata.



```
Xdata Values
Value (0) ACAD
Value (1) DSTYLE
Value (2) {
Value (3) 343
Value (4) 2E5
Value (5) 173
Value (6) 1
Value (7) 344
Value (8) 0
Value (9) }
```

FIGURE 9.1 Message box displaying the data values of Xdata that represent a dimension override

This exercise shows how to modify the Xdata of the dimension you created in the previous exercise. You will append values that will assign ACI 40 to the dimension line and ACI 7 to the extension lines overriding the colors assigned to the dimension by its assigned dimension style.

1. In the VBA Editor, open the code editor window for the ThisDrawing component of the project you created earlier in this chapter. Type the following:

```
Sub ReplaceXDataForDimOverride()
  On Error Resume Next

  ' Prompt the user to select an object
  Dim oAcadObj As AcadObject
  ThisDrawing.Utility.GetEntity oAcadObj, Nothing, _
      vbLf & "Select dimension to add overrides: "

  Dim nXdTypesFinal() As Integer
  Dim vXdValsFinal() As Variant

  ' Check to see if an object was selected
  If Not oAcadObj Is Nothing Then
    ' Check to see if the selected object is a dimension
    If TypeOf oAcadObj Is AcadDimension Then

      ' Get the Xdata attached to the object
      Dim vXdTypes As Variant, vXdVals As Variant
```

```
oAcadObj.GetXData "ACAD", vXdTypes, vXdVals

' Check to see whether the value returned is an array
' An array means Xdata is present
If IsArray(vXdTypes) Then
  Dim nCnt As Integer, nNewCnt As Integer
  nCnt = 0: nNewCnt = 0

  ' Append the values of the Xdata to the sMsg variable
  For nCnt = 0 To UBound(vXdVals)

    ' If "{", append the previous value and new values
    If vXdVals(nCnt) = "{" Then
      ' Increase the arrays by 4 additional values to make
      ' room for the new overrides
      ReDim Preserve nXdTypesFinal(nNewCnt + 4)
      ReDim Preserve vXdValsFinal(nNewCnt + 4)

      ' Add the existing Xdata value of "{"
      nXdTypesFinal(nNewCnt) = vXdTypes(nCnt)
      vXdValsFinal(nNewCnt) = vXdVals(nCnt)

      ' Add the data types and values for the new overrides
      ' Dimension line color
      nXdTypesFinal(nNewCnt + 1) = 1070
      vXdValsFinal(nNewCnt + 1) = 176
      nXdTypesFinal(nNewCnt + 2) = 1070
      vXdValsFinal(nNewCnt + 2) = 40

      ' Extension line color
      nXdTypesFinal(nNewCnt + 3) = 1070
      vXdValsFinal(nNewCnt + 3) = 177
      nXdTypesFinal(nNewCnt + 4) = 1070
      vXdValsFinal(nNewCnt + 4) = 200

      ' Increment the array counter by 5 since we added 5 elements
      nNewCnt = nNewCnt + 5
    Else
      ' Not the "{" value, so append the previous value
      ReDim Preserve nXdTypesFinal(nNewCnt)
      ReDim Preserve vXdValsFinal(nNewCnt)

      ' Add the previous values of the Xdata to the new arrays
      nXdTypesFinal(nNewCnt) = vXdTypes(nCnt)
      vXdValsFinal(nNewCnt) = vXdVals(nCnt)
```

```
          ' Increment the array counter by 1
          nNewCnt = nNewCnt + 1
        End If
      Next nCnt
    Else
      ' The following is executed if no Xdata is already applied.
      ' The two arrays define color overrides for the dimension
      ' and extension lines.
      ' Define the data types array for the Xdata
      ReDim nXdTypesFinal(7)
      nXdTypesFinal(0) = 1001: nXdTypesFinal(1) = 1000
      nXdTypesFinal(2) = 1002: nXdTypesFinal(3) = 1070
      nXdTypesFinal(4) = 1070: nXdTypesFinal(5) = 1070
      nXdTypesFinal(6) = 1070: nXdTypesFinal(7) = 1002

      ' Define the data values array for the Xdata
      ReDim vXdValsFinal(7)
      vXdValsFinal(0) = "ACAD": vXdValsFinal(1) = "DSTYLE"
      vXdValsFinal(2) = "{": vXdValsFinal(3) = 176
      vXdValsFinal(4) = 40: vXdValsFinal(5) = 177
      vXdValsFinal(6) = 200: vXdValsFinal(7) = "}"
    End If

    ' Assign the Xdata to the dimension
    oAcadObj.SetXData nXdTypesFinal, vXdValsFinal
    oAcadObj.Update
  End If
 End If
End Sub
```

2. Switch to AutoCAD.
3. At the Command prompt, type **vbarun** and press Enter.
4. When the Macros dialog box opens, select the `Global`*N*`!ThisDrawing.ReplaceXDataForDimOverride` macro and click Run.
5. At the `Select dimension to add overrides:` prompt, select the linear dimension you created in the previous exercise.

   The colors of the dimension and extension lines of the dimension object inherited from the dimension style are now overridden while preserving the first arrow of the dimension being set to None. This is similar to what happens when you select a dimension, right-click, and choose Precision.
6. Save the project if you want, but don't close it as you will continue with the project in the next exercise.

Removing Xdata from an Object

Xdata can be removed from an object when it is no longer needed. You do so by replacing the Xdata attached to an object with a data value array that contains only an application name. When AutoCAD evaluates Xdata with only an application name and no additional data values, it removes the Xdata from the object. Here is an example of Xdata that can be used to remove the Xdata associated with the MyApp application:

```
' Define the data types array for the Xdata
Dim nXdTypes(0) As Integer
nXdTypes(0) = 1001

' Define the data values array for the Xdata
Dim vXdVals(0) As Variant
vXdVals(0) = "MyApp"
```

The following example removes the Xdata list associated with an application named ACAD from a dimension, which removes all overrides assigned to the dimension:

```
Sub RemoveDimOverride()
  On Error Resume Next

  ' Define the data types array for the Xdata
  Dim nXdTypes(0) As Integer
  nXdTypes(0) = 1001

  ' Define the data values array for the Xdata
  Dim vXdVals(0) As Variant
  vXdVals(0) = "Acad"

  Dim oAcadObj As AcadObject
  ThisDrawing.Utility.GetEntity oAcadObj, Nothing, _
      vbLf & "Select dimension to remove overrides: "

  ' Check to see if an object was selected
  If Not oAcadObj Is Nothing Then
    ' Check to see if the selected object is a dimension
    If TypeOf oAcadObj Is AcadDimension Then
      ' Assign the Xdata to the circle object
      oAcadObj.SetXData nXdTypes, vXdVals
    End If
  End If
End Sub
```

Selecting Objects Based on Xdata

You can use the Xdata attached to an object as a way to select or filter out specific objects with the selection-related functions of the `AcadSelectionSet` object. (I explained how to use the optional *`filterType`* and *`filterData`* arguments with the selection-related functions of the

AcadSelectionSet object in Chapter 4, "Creating and Modifying Drawing Objects.") If you want to filter on the Xdata attached to an object, you use the DXF group code 1001 along with the application name from the Xdata.

Here are example code statements that use the SelectOnScreen method to allow the user to select objects in the drawing but keep in the selection set those that have Xdata attached to them with the ACAD application name:

```
Sub SelectObjectsByXdata()
  On Error Resume Next

  ' Define the data types array for the Xdata
  Dim nXdTypes(0) As Integer
  nXdTypes(0) = 1001

  ' Define the data values array for the Xdata
  Dim vXdVals(0) As Variant

  ' Get the selection set named SSAcad
  Dim oSSAcad As AcadSelectionSet
  Set oSSAcad = ThisDrawing.SelectionSets("SSAcad")

  ' If SSMyApp isn't found, add it
  If Err Then
    Err.Clear
    Set oSSAcad = ThisDrawing.SelectionSets.Add("SSAcad")
  Else
    ' Clear the objects in the selection set
    oSSAcad.Clear
  End If

  ' Selects objects containing Xdata
  ' with the application name of ACAD.
  vXdVals(0) = "ACAD"
  oSSAcad.SelectOnScreen nXdTypes, vXdVals

  ' Display the number of objects in the selection set
  MsgBox "Objects that contain Xdata for MyApp are: " & CStr(oSSAcad.Count)

  ' Remove the SSAcad selection set
  oSSAcad.Delete
End Sub
```

Creating and Modifying a Custom Dictionary

Dictionaries are used to store custom information and objects in a drawing and can be thought of as an extension of symbol tables. Dictionaries were introduced with AutoCAD R13 as a way to introduce new symbol-table-like objects without the need to change the drawing file format with each release. Although there is only one type of dictionary in a drawing, dictionaries can be stored in two different ways: per drawing or per object.

The *main dictionary*—also known as the named object dictionary—of a drawing contains nested dictionaries that store multileader and table styles, and even the layouts used to organize and output a drawing. Dictionaries can also be attached to an object, and those are known as *extension dictionaries,* which I explained earlier this chapter.

Custom dictionaries are great for storing custom program settings so that they persist across drawing sessions. You might also use a custom dictionary as a way to store drawing revision history or project information that can be used to track a drawing and populate a title block. In this section, you'll learn how to access, create, query, and modify information stored in a dictionary.

Accessing and Stepping through Dictionaries

The main dictionary of a drawing is accessed using the `Dictionaries` property of the `ThisDrawing` or an `AcadDocument` object. The `Dictionaries` property returns the `AcadDictionaries` collection object, which contains all the dictionaries that aren't attached to an object as an extension dictionary. Dictionaries are similar to working with symbol tables. Once you have the `AcadDictionaries` collection object, use the object's `Item` method or a `For` statement to get an individual dictionary that is represented by an `AcadDictionary` object.

A dictionary can store an object or *extended record*—also known as an Xrecord. An Xrecord is similar to the Xdata that can be attached to an object, which I explain in the "Storing Information in a Custom Dictionary" section later in this chapter. The only difference is that Xrecord data types are in the range of 1–369 instead of more than 1,000 like Xdata. Although VBA can be used to get any dictionary stored in a drawing, not all entries in a dictionary can be accessed with VBA and the AutoCAD Object library. The reason that not all entries in a dictionary are accessible is that some objects aren't part of the AutoCAD Object library.

For example, you can access the dictionaries that store plot and visual styles in a drawing but not the individual entries themselves. The entries of the dictionaries used to store layouts, table styles, and multileader styles are accessible from VBA because the objects in those dictionaries are defined in the AutoCAD Object library.

NOTE If you need to access the entries of other dictionaries, you will need to use the AutoLISP programming language, ObjectARX, or Managed .NET. I discussed how to work with dictionaries using the AutoLISP programming language in Part II, "AutoLISP: Productivity through Programming."

The following example code statements step through and list the names of each table style in the drawing that is stored in the `ACAD_TABLESTYLE` dictionary. The code is followed by sample output.

```
' Lists the table styles in the current drawing
Sub ListTableStyles()
  ' Get the ACAD_TABLESTYLE dictionary
  Dim oDictTblStyle As AcadDictionary
  Set oDictTblStyle = ThisDrawing.Dictionaries("ACAD_TABLESTYLE")

  If Not oDictTblStyle Is Nothing Then
```

```
    Dim sMsg As String
    sMsg = "Table styles in this drawing:" & vbLf

    ' Append the names of each table style to the sMsg variable
    Dim oTblStyleEntry As AcadTableStyle
    For Each oTblStyleEntry In oDictTblStyle
      sMsg = sMsg & oTblStyleEntry.Name & vbLf
    Next oTblStyleEntry

    ' Display the table style names in the Command Line history
    ThisDrawing.Utility.Prompt vbLf & sMsg
  Else
    ThisDrawing.Utility.Prompt vbLf & _
      "Drawing doesn't contain the ACAD_TABLESTYLE dictionary."
  End If
End Sub
```

```
Table styles in this drawing:
BOM - Architectural
BOM - Mechanical
Standard
Title Sheet
```

The existence of an entry in a dictionary can be validated with the `Item` method of the `AcadDictionary` object and an `If` conditional statement or by using the `For` statement to get each entry in the dictionary. If the name of the entry in the dictionary exists, the object is returned; otherwise, an error is generated.

The following shows how to get the Standard table style entry from the `ACAD_TABLESTYLE` dictionary:

```
' Gets the Standard table style in the current drawing
Sub GetStandardTableStyle()
  On Error Resume Next

  ' Get the ACAD_TABLESTYLE dictionary
  Dim oDictTblStyle As AcadDictionary
  Set oDictTblStyle = ThisDrawing.Dictionaries("ACAD_TABLESTYLE")

  If TypeOf oDictTblStyle Is AcadDictionary Then
    ' Get the Standard table style
    Dim oTblStyleEntry As AcadTableStyle
    Set oTblStyleEntry = oDictTblStyle("Standard")

    If Not oTblStyleEntry Is Nothing Then
      MsgBox "Standard table style found."
    End If
  End If
End Sub
```

Creating a Custom Dictionary

As I mentioned earlier, one of the benefits of dictionaries is that you can store custom information or settings related to the programs you create in a drawing. Before a custom dictionary can be used and entries added to it, it must first be created. The `Add` method of the `AcadDictionaries` collection object is used to create a new named object dictionary. When you create a dictionary with the `Add` method, you must pass the method a string that represents the name of the dictionary you wish to create. The `Add` method of the `AcadDictionaries` collection object returns an `AcadDictionary` object.

Here's an example that creates a dictionary named `MY_CUSTOM_DICTIONARY` and adds it to the named object dictionary:

```
' Creates a custom dictionary named MY_CUSTOM_DICTIONARY
Dim oDict As AcadDictionary
Set oDict = ThisDrawing.Dictionaries.Add("MY_CUSTOM_DICTIONARY")
```

In addition to adding a dictionary to the named object dictionary, you can create an extension dictionary on any object that is based on `AcadObject`, which includes most nongraphical and graphical objects in an AutoCAD drawing. Since an `AcadDictionary` object is based on an `AcadObject`, it can also have an extension dictionary, which can make for some interesting and complex data models.

An extension dictionary is similar to the named object dictionary of a drawing, and it can hold nested dictionaries of extended records. If you want to create an extension dictionary, you must first get the extension dictionary of an object with the `GetExtensionDictionary` method. This method returns an `AcadDictionary` object. The `HasExtensionDictionary` property of an object can be used to check whether an object has an extension dictionary attached to it.

These example code statements check whether an extension dictionary exists on the last object in model space:

```
Sub AddExtensionDictionary()
  On Error Resume Next

  ' Get the last object added to model space
  Dim oAcadObj As AcadObject
  Set oAcadObj = ThisDrawing.ModelSpace(ThisDrawing.ModelSpace.Count - 1)

  If Err.Number = 0 Then
    Dim oExDict As AcadDictionary

    ' Check whether an extension dictionary already exists
    If oAcadObj.HasExtensionDictionary Then
      Set oExDict = oAcadObj.GetExtensionDictionary

      MsgBox "Extension dictionary attached." & vbLf & _
             "Number of entries in the extension dictionary: " & _
             oExDict.Count
    Else
```

```
      MsgBox "No extension dictionary attached."

      ' If the extension dictionary doesn't exist, it is added
      Set oExDict = oAcadObj.GetExtensionDictionary
    End If
  End If
End Sub
```

If the example code is executed, a message box is displayed, indicating whether an extension dictionary exists for the last object in model space. If the extension dictionary exists, the number of entries in the extension dictionary is returned; otherwise, the extension dictionary is added by the `GetExtensionDictionary` method. Once the extension dictionary is attached to the object, you can then add an Xrecord or nested dictionary to the object's extension dictionary. You'll learn how to add information to a custom dictionary in the next section.

Storing Information in a Custom Dictionary

After a custom dictionary has been created, you add entries to that custom dictionary using the `AddObject` or `AddXrecord` method of the `AcadDictionary` object. The `AddObject` method allows you to add an object based on the `AcDbDictionaryRecord` class that is part of the ObjectARX and Managed .NET APIs, which Autodesk supports for AutoCAD. The `AcDbDictionaryRecord` class or `AcadDictionaryRecord` object isn't available from the AutoCAD Object library. In Chapter 6, I explained how to use the `AddObject` method to create a new table style.

When storing information in a dictionary, use the `AddXrecord` method to add a new Xrecord to the dictionary. The `AddXrecord` method accepts a string that represents the name of the entry to add and returns an `AcadXrecord` object. The name of the entry must be unique to the dictionary.

The following code statements add an Xrecord with the name XR1 to the dictionary named `MY_CUSTOM_DICTIONARY`. The data assigned to the Xrecord contains a string (DXF group code 1), a coordinate value (DXF group code 10), and an integer (DXF group code 71).

```
; Add the Xrecord to the dictionary
Dim oXRec As AcadXRecord
Set oXRec = oDict.AddXRecord("XR1")

' Define the data types array for the Xrecord
Dim nXdTypes(2) As Integer
nXdTypes(0) = 1: nXdTypes(1) = 10: nXdTypes(2) = 71

' Define a point list
Dim dPT(2) As Double
dPT(0) = 5: dPT(1) = 5: dPT(2) = 0

' Define the data values array for the Xrecord
Dim vXdVals(2) As Variant
```

```
vXdVals(0) = "Custom string"
vXdVals(1) = dPT
vXdVals(2) = 11

' Add the arrays to the Xrecord
oXRec.SetXRecordData nXdTypes, vXdVals
```

If you need to make a change to the data contained in an Xrecord, you use the `GetXRecordData` method of the `AcadXrecord` object to get the data type and value arrays of the data stored in the Xrecord. Once you have the two arrays, you can modify their values by using the same steps you used to modify Xdata in the "Querying and Modifying the Xdata Attached to an Object" section earlier in this chapter.

Managing Custom Dictionaries and Entries

After a dictionary or Xrecord has been created and attached, you can change its name, remove it, or replace it. You can freely rename and remove the dictionaries you create; those created by AutoCAD can also be renamed and removed. I recommend being cautious about renaming or removing dictionaries created by other features in the AutoCAD program because doing so could cause problems. Not all dictionaries and entries of a dictionary can be removed; if an entry is referenced by another object, it can't be removed.

The name of an `AcadDictionary` object can be changed using its `Name` property, and the name of an entry in a dictionary can be changed using the `Rename` method of the `AcadDictionary` object. The `Rename` method expects two strings: the current name and the new name. The following shows the syntax of the `Rename` method:

```
object.Rename oldName, newName
```

An `AcadDictionary` object can be removed using its `Delete` method, and the `Remove` method of the `AcadDictionary` object can be used to remove an entry from a dictionary. The `Remove` method expects a string that represents the name of the entry you want to remove from the dictionary. An `AcadObject` object is returned by the `Remove` method that contains the object or Xrecord that is being removed. The following shows the syntax of the `Remove` method:

```
retVal = object.Remove(entryName)
```

Here are examples that rename and remove a custom dictionary:

```
' Renames MY_CUSTOM_DICTIONARY to MY_DICTIONARY
Dim oDict As AcadDictionary
Set oDict = ThisDrawing.Dictionaries("MY_CUSTOM_DICTIONARY")

oDict.Name = "MY_DICTIONARY"

' Removes MY_DICTIONARY
Dim oDict As AcadDictionary
Set oDict = ThisDrawing.Dictionaries("MY_DICTIONARY")

oDict.Delete
```

Here are examples that rename and remove a dictionary entry:

```
' Gets the dictionary MY_CUSTOM_DICTIONARY
Dim oDict As AcadDictionary
Set oDict = ThisDrawing.Dictionaries("MY_CUSTOM_DICTIONARY")

' Renames the entry XR1 to XR_1
oDict.Rename "XR1", "XR_1"

' Removes the entry XR_1
oDict.Remove "XR_1"
```

If you are storing objects and not Xrecords in a dictionary, you can use the `Replace` method of the `AcadDictionary` object to replace an object in an entry. The `Replace` method expects a string that represents the name of the entry you want to replace in the dictionary, and it also expects the new object that should replace the existing object. The following shows the syntax of the `Replace` method:

```
object.Replace entryName, newObject
```

Storing Information in the Windows Registry

The AutoCAD program stores information and setting values using many different methods. Some are proprietary; others are industry standard. Most setting values are stored as part of the drawing using system variables, Xdata, or custom dictionaries. Those settings that aren't stored with the drawing are, for the most part, stored with the AutoCAD user profile. The AutoCAD user profile is maintained in the Windows Registry.

You learned how to work with system variables in Chapter 3, "Interacting with the Application and Documents Objects." I covered Xdata and custom dictionaries earlier in this chapter.

Creating and Querying Keys and Values

You can create and query values in the Windows Registry. The values that you can access in the Windows Registry aren't just related to AutoCAD but are those managed by Windows and other installed applications. The Windows Registry is organized into three main areas (known as *hive keys* but most commonly just *keys*). These keys are as follows:

`HKEY_CLASSES_ROOT` The `HKEY_CLASSES_ROOT` key contains settings related to file extensions and ActiveX libraries that are registered with the local machine. The settings are available to any user logged on to the machine and require elevated or administrative rights to change.

`HKEY_LOCAL_MACHINE` The `HKEY_LOCAL_MACHINE` key contains settings related to the software or hardware configuration of the local machine. The settings are available to any user logged on to the machine and require elevated or administrative rights to change.

`HKEY_CURRENT_USER` The `HKEY_CURRENT_USER` key contains settings related to software and hardware that don't impact the installation of software or the hardware configuration of the local machine. Typically, the settings in this key are driven by the choices made while using a software program. These settings are available only to the user who is currently logged into Windows.

You might occasionally query values in the `HKEY_CLASSES_ROOT` and `HKEY_LOCAL_MACHINE` keys, but the programs you create should treat the values under these keys as read-only. The values in these keys are typically set by an application installer. The `HKEY_CURRENT_USER` key is where you should store any values you want to access between AutoCAD sessions. The values of the `HKEY_CURRENT_USER` key can be queried and added as needed by your programs.

There are three approaches to accessing values in the Windows Registry. The simplest is to use the `SaveSetting` and `GetSetting` functions that are part of the VBA programming language. These methods access values under the following location in the Windows Registry:

```
HKEY_CURRENT_USER\Software\VB and VBA Program Settings
```

The `SaveSetting` function allows you to save a string to the Windows Registry under a user-specified application, section, and key name. Once a value has been stored, you use the `GetSetting` function, which expects the application, section, and key name of the value you want to query.

The following shows the syntax of the `SaveSetting` and `GetSetting` functions:

```
SaveSetting appName, section, key, value
retVal = GetSetting(appName, section, key [, defaultValue])
```

The arguments are as follows:

appName The *appName* argument is a string that specifies the subkey under the VB and VBA Program Settings in the Windows Registry that you want to access.

section The *section* argument is a string that specifies the key under the key represented by the *appName* argument.

key The *key* argument is a string that specifies the key under the key represented by the *section* argument.

value The *value* argument is the string value that you want to store under the key specified. Use an empty string ("") to access the value of the key named `(Default)`.

defaultValue The *defaultValue* argument is an optional string value that should be returned if the key specified doesn't exist.

Here are some examples of writing and reading values to and from the Windows Registry:

```
' Creates a new key with the value of 5.5 under the application
' named CompanyABC123, in a section named HexBolt, and a key named Width
SaveSetting "CompanyABC123", "HexBolt", "Width", "5.5"

' Gets the value of the key CompanyABC123\HexBolt\Width
' If the key doesn't exist, a default value of 5.0 is returned
Dim sWidth As String
sWidth = GetSetting("CompanyABC123", "HexBolt", "Width", "5.0")
```

The GetSetting function requires you to know the name of the value you want to read, but there are times when you might want to read all values under a key. You can use the GetAllSettings function to get the names of all the values under a key. The GetAllSettings function returns a two-dimensional array that contains the key names and their values.

Here is the syntax of the GetAllSettings function:

```
retVal = GetAllSettings(appName, section)
```

The *appName* and section arguments are the same as previously described for the SaveSetting and GetSetting functions.

The following code statements list the keys and their values under HKEY_CURRENT_USER\Software\VB and VBA Program Settings \CompanyABC123\HexBolt:

```
Dim vKeys As Variant, nCnt As Integer

' Query the settings under CompanyABC123\HexBolt
vKeys = GetAllSettings("CompanyABC123", "HexBolt")

' Step through the two-dimensional array
For nCnt = LBound(vKeys, 1) To UBound(vKeys, 1)
  MsgBox "Key: " & CStr(vKeys(nCnt, 0)) & vbLf & _
         "Value: " & CStr(vKeys(nCnt, 1))
Next nCnt
```

If you need to query or create values in other areas of the Windows Registry, you can use the Windows Script Host Object Model, which is an external programming library that can be referenced into your VBA project. The WshShell object contained in the library has the functions RegRead, RegWrite, and RegDelete. In addition to the Windows Script Host Object Model programming library, the Win32 API contains a range of Windows Registry functions that can be used to create, read, and delete keys in any area. You can learn more about the Win32 API functions that are related to the Windows Registry at http://support.microsoft.com/kb/145679. I explain how to work with additional programming libraries and the Win32 API in Chapter 12.

TIP You can access the settings stored in the Windows Registry for the AutoCAD programs installed on your workstation by reading the keys under HKEY_CURRENT_USER\Software\Autodesk\AutoCAD.

Editing and Removing Keys and Values

You can update the data of a value under a key or remove a key or value that is no longer needed. You update a value by using the SaveSetting function, whereas you use the DeleteSetting function to remove a key or value.

Here is the syntax of the DeleteSetting function:

```
DeleteSetting appName [, section] [, key]
```

The *appName*, *section*, and *key* arguments are the same as previously described for the SaveSetting and GetSetting functions. The *section* and *key* arguments are optional for the DeleteSetting function.

The following code statement deletes the sections and keys under HKEY_CURRENT_USER\Software\VB and VBA Program Settings \CompanyABC123:

```
' Removes the settings under the key CompanyABC123
DeleteSetting "CompanyABC123"

' Removes the Width value from under key CompanyABC123\HexBolt
DeleteSetting "CompanyABC123", "HexBolt", "Width"
```

Exercise: Storing Custom Values for the Room Labels Program

In this section, you will modify the VBA project named RoomLabel that was introduced in Chapter 7, "Working with Blocks and External References." The RoomLabel project creates and inserts a room label block into a drawing. The modifications that you will make to the project will allow you to identify the room label block in the drawing and to store values in the Windows Registry and in a custom dictionary.

When the room label block is inserted, Xdata is attached to the block reference and allows you to use it as a way to locate the room label blocks in the drawing. The program lets you choose a starting (or next) number and a prefix. These values are stored as part of the drawing, allowing the program to continue where it last left off, and they can be stored in the Windows Registry as the default values to use when the program is executed for the first time in a drawing.

The key concepts covered in this exercise are as follows:

Attaching Xdata to an Object Extended data (or Xdata) can be used to store custom information with a graphical or nongraphical object. Once attached, the information can be used to filter out objects with specific Xdata values and even manage objects differently through custom programs.

Setting and Querying Information in the Windows Registry The Windows Registry allows you to store values so they can be persisted between AutoCAD sessions and accessed no matter which drawing is current.

Creating and Storing Information in a Custom Dictionary Values assigned to variables in a drawing are temporary, but you can use custom dictionaries to persist values across drawing sessions. The values stored in a drawing can then be recovered by your programs after the drawing is closed and reopened, similar to how system variables work.

NOTE The steps in this exercise depend on the completion of the steps in the "Exercise: Creating and Querying Blocks" section of Chapter 7. If you didn't complete these exercises, do so now or start with the ch09_roomlabel.dvb and ch09_building_plan.dwg sample files available for download from www.sybex.com/go/autocadcustomization. These sample files should be placed in the MyCustomFiles folder within the Documents (or My Documents) folder, or in the location you are using to store the custom program files. Once the files are stored on your system, remove ch09_ from the name of the DVB file.

Attaching Xdata to the Room Label Block after Insertion

Chapter 7 was the last chapter in which any changes were made to the RoomLabel project. At that time, you implemented functionality that created and inserted the room label block, and even set the label value for the attribute in the block. Here you'll modify the RoomLabel_InsertBlkRef procedure so that it attaches some Xdata to the block reference that is inserted into the drawing. The Xdata will help you identify the room label blocks inserted with the RoomLabel project.

The following steps show how to modify the RoomLabel_InsertBlkRef procedure:

1. Load the RoomLabel.dvb file into the AutoCAD drawing environment and display the VBA Editor.
2. In the VBA Editor, in the Project Explorer, double-click the basRoomLabel component.
3. In the code editor window, scroll to the code statement that starts with Private Sub RoomLabel_InsertBlkRef.
4. Type the code shown in bold; the comments are here for your information and don't need to be typed:

```
  ' Changes the attribute value of the "ROOM#"
  ChangeAttValue oBlkRef, vInsPt, "ROOM#", sLabelValue

  ' Create and attach Xdata to assist in selecting Room Labels
  ' Define the data types array for the Xdata
  Dim nXdTypes(1) As Integer
  nXdTypes(0) = 1001: nXdTypes(1) = 1000

  ' Define the data values array for the Xdata
  Dim vXdVals(1) As Variant
  vXdVals(0) = "ACP_RoomLabel": vXdVals(1) = "Room label block"

  ' Attach the Xdata to the block reference
  oBlkRef.SetXData nXdTypes, vXdVals
End Sub
```

5. Click File ➤ Save.

Revising the Main *RoomLabel* Procedure to Use the Windows Registry

The changes you make to the RoomLabel procedure determine which values are used when the procedure is initialized the first time it is executed in a drawing. Previously, the default values were defined in the procedure, but with the changes they can be stored in the Windows Registry.

Follow these steps to update the Global declaration in the basRoomLabel component:

1. In the VBA Editor, in the Project Explorer, double-click the basRoomLabel component.
2. In the code editor window, scroll to the top of the code editor window.
3. Remove the code shown in bold:

```
Private myUtilities As New clsUtilities

' Constant for the removal of the "Command: " prompt msg
Const removeCmdPrompt As String = vbBack & vbBack & vbBack & _
                                  vbBack & vbBack & vbBack & _
                                  vbBack & vbBack & vbBack & vbLf

Private g_nLastNumber As Integer
Private g_sLastPrefix As String
```

4. Click File ➢ Save.

The following steps explain how to get the last number and prefix from the Windows Registry:

1. In the code editor window, scroll to the code statement that starts with `Public Sub RoomLabel()`.
2. In the procedure, locate and select the code statements shown in bold:

```
On Error Resume Next

' Set the default values
Dim nLastNumber As Integer, sLastPrefix As String
If g_nLastNumber <> 0 Then
  nLastNumber = g_nLastNumber
  sLastPrefix = g_sLastPrefix
Else
  nLastNumber = 101
  sLastPrefix = "L"
End If

' Display current values
ThisDrawing.Utility.Prompt removeCmdPrompt & _
                           "Prefix: " & sLastPrefix & _
                           vbTab & "Number: " & CStr(nLastNumber)
```

3. Type the bold code that follows; the comments are here for your information and don't need to be typed:

```
Dim nLastNumber As Integer, sLastPrefix As String
' Check to see if the defaults have been previously
' stored in the Windows Registry
nLastNumber = CInt(GetSetting("ACP_Settings", "RoomLabel", _
                              "FirstNumber", "101"))

sLastPrefix = GetSetting("ACP_Settings", "RoomLabel", _
                         "Prefix", "L")
```

4. In the code editor window, still in the `RoomLabel` procedure, scroll down and locate the following code statement:

```
ThisDrawing.Utility.InitializeUserInput 0, "Number Prefix"
```

5. Revise the code statements in bold; you are adding a new option named Save:

```
basePt = Null

' Set up default keywords
ThisDrawing.Utility.InitializeUserInput 0, "Number Prefix Save"

' Prompt for a base point, number, or prefix value
basePt = ThisDrawing.Utility.GetPoint(, _
         removeCmdPrompt & "Specify point for room label (" & _
         sLastPrefix & CStr(nLastNumber) & _
         ") or change [Number/Prefix/Save]: ")

' If an error occurs, the user entered a keyword or pressed Enter
```

6. In the code editor window, still in the `RoomLabel` procedure, scroll down a few lines to the `Select Case sKeyword` code statement.

7. Type the code shown in bold; the comments are here for your information and don't need to be typed:

```
  Case "Prefix"
    sLastPrefix = ThisDrawing.Utility. _
            GetString(False, removeCmdPrompt & _
                      "Enter new room number prefix <" & _
                      sLastPrefix & ">: ")
  Case "Save"
    ThisDrawing.Utility.InitializeUserInput 0, "Yes No"

    Dim sSaveToDefaults As String
    sSaveToDefaults = ThisDrawing.Utility. _
            GetKeyword(removeCmdPrompt & _
                       "Save current number and prefix " & _
                       "as defaults [Yes/No] <Yes>: ")

    If UCase(sSaveToDefaults) = "YES" Or _
      sSaveToDefaults = "" Then
      ' Save the current room number
      SaveSetting "ACP_Settings", "RoomLabel", _
                  "FirstNumber", CStr(nLastNumber)
      ' Save the current prefix
      SaveSetting "ACP_Settings", "RoomLabel", _
                   "Prefix", sLastPrefix
    End If
End Select
```

8. In the code editor window, still in the `RoomLabel` procedure, scroll to the `End Sub` code statement at the end of the `RoomLabel` procedure.

9. Remove the bold text that follows:

```
  Loop Until IsNull(basePt) = True And sKeyword = ""
```

```
    ' Store the latest values in the global variables
    g_nLastNumber = nLastNumber
    g_sLastPrefix = sLastPrefix
End Sub
```

10. Click File ➢ Save.

Testing the Changes to the *RoomLabel* Procedure

The following steps explain how to use the changes made to the `RoomLabel` procedure:

1. Switch to the AutoCAD application window.
2. Open `ch09_Building_Plan.dwg`. Figure 9.2 shows the plan drawing of the office building that is in the drawing.

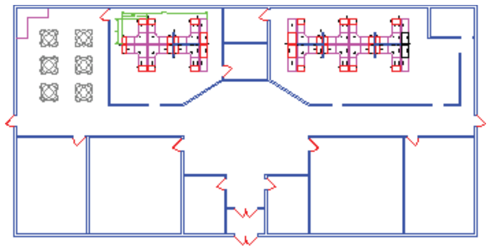


FIGURE 9.2 Plan view of the office building

3. At the Command prompt, type **`vbarun`** and press Enter.
4. When the Macros dialog box opens, select the `RoolLabel.dvb!basRoomLabel.RoomLabel` macro from the list and click Run.
5. At the `Specify point for room label (L101) or change [Number/Prefix/Save]:` prompt, specify a point inside the room in the lower-left corner of the building.

   The room label definition block, `Plan_RoomLabel_Anno` layer, and `My_Custom_Program_Settings` custom dictionary are created the first time the `RoomLabel` procedure is used. The `RoomLabel` block definition should look like Figure 9.3 when inserted into the drawing.

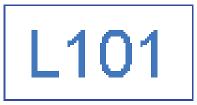


FIGURE 9.3 Inserted `RoomLabel` block

6. At the `Specify point for room label (L101) or change [Number/Prefix/Save]:` prompt, type **`n`** and press Enter.
7. At the `Enter new room number <102>:` prompt, type **`105`** and press Enter.
8. At the `Specify point for room label (L105) or change [Number/Prefix/Save]:` prompt, type **`p`** and press Enter.

9. At the `Enter new room number prefix <L>:` prompt, type **`R`** and press Enter.
10. At the `Specify point for room label (R105) or change [Number/Prefix/Save]:` prompt, specify a point in the large open area in the middle of the building.

    The new room label is marked as R105.

These steps show how to save a new default prefix and starting number:

1. At the `Specify point for room label (R105) or change [Number/Prefix/Save]:` prompt, type **`n`** and press Enter.
2. At the `Enter new room number <105>:` prompt, type **`101`** and press Enter.
3. At the `Specify point for room label (R101) or change [Number/Prefix/Save]:` prompt, type **`p`** and press Enter.
4. At the `Enter new room number prefix <P>:` prompt, type **`F`** and press Enter.
5. At the `Specify point for room label (F101) or change [Number/Prefix/Save]:` prompt, type **`s`** and press Enter.
6. At the `Save current number and prefix as defaults [Yes/No] <Yes>:` prompt, press Enter.
7. Press Enter again to exit the `RoomLabel` procedure.
8. Create a new drawing and execute the `RoomLabel` procedure.

    The starting value is F101. You can change the prefix and number without affecting the default values used each time the program is started.

9. Execute the `RoomLabel` procedure again.

    You should notice that the program starts numbering once again with the default values stored in the Windows Registry. This is as expected since you removed the use of the global variables to hold the last number and prefix. You will address this problem in the next section by writing the last number and prefix to a custom dictionary to persist values in the drawing.

10. Discard the changes to `ch09_Building_Plan.dwg` and the new drawing file.

Persisting Values for the Room Label Procedure with a Custom Dictionary

Instead of using global variables that are lost after a drawing is closed, sometimes it is beneficial to persist values in a drawing for use when the program is executed again. A custom dictionary will be used to persist the last number and prefix used between drawing sessions.

The following steps explain how to add support for storing values in a custom dictionary:

1. Switch to the VBA Editor.
2. In the VBA Editor, in the Project Explorer double-click the `basRoomLabel` component.

3. In the code editor window, scroll up to the code statement that starts with `Public Sub RoomLabel()`.
4. In the procedure, type the bold text that follows; the comments are here for your information and don't need to be typed:

```
nLastNumber = CInt(GetSetting("ACP_Settings", "RoomLabel", _
                             "FirstNumber", "101"))

sLastPrefix = GetSetting("ACP_Settings", "RoomLabel", _
                         "Prefix", "L")

' Gets the custom dictionary "My_Custom_Program_Settings" if it exists
Dim oDict As AcadDictionary
Set oDict = ThisDrawing.Dictionaries("My_Custom_Program_Settings")

Dim oXrecRL As AcadXRecord
Dim nXdType(1) As Integer, vXdValues(1) As Variant

' If the dictionary exists, get the previous values
If Not oDict Is Nothing Then
  Set oXrecRL = oDict("RoomLabel")

  If Not oXrecRL Is Nothing Then
    Dim vXdType As Variant, vXdValue As Variant
    oXrecRL.GetXRecordData vXdType, vXdValue

    Dim nCnt As Integer
    For nCnt = 0 To UBound(vXdType)
      Select Case vXdType(nCnt)
        Case 1
          sLastPrefix = vXdValue(nCnt)
        Case 71
          nLastNumber = vXdValue(nCnt)
      End Select
    Next
  End If
Else
  ' Create the dictionary
  Set oDict = ThisDrawing.Dictionaries.Add("My_Custom_Program_Settings")

  ' Add the default record
  Set oXrecRL = oDict.AddXRecord("RoomLabel")

  nXdType(0) = 1:  vXdValues(0) = sLastPrefix
  nXdType(1) = 71: vXdValues(1) = nLastNumber

  oXrecRL.SetXRecordData nXdType, vXdValues
End If
```

```
Err.Clear

' Display current values
ThisDrawing.Utility.Prompt removeCmdPrompt & _
                           "Prefix: " & sLastPrefix & _
                           vbTab & "Number: " & CStr(nLastNumber)
```

5. In the code editor window, still in the `RoomLabel` procedure, scroll to the `End Sub` code statement at the end of the `RoomLabel` procedure.
6. Type the bold text that follows; the comments are here for your information and don't need to be typed:

```
    End If

    ' If a base point was specified, then insert a block reference
    If IsNull(basePt) = False Then
      RoomLabel_InsertBlkRef basePt, sLastPrefix & CStr(nLastNumber)

      ' Increment number by 1
      nLastNumber = nLastNumber + 1

      ' Update the Xrecord
      nXdType(0) = 1:  vXdValues(0) = sLastPrefix
      nXdType(1) = 71: vXdValues(1) = nLastNumber

      oXrecRL.SetXRecordData nXdType, vXdValues
    End If
  Loop Until IsNull(basePt) = True And sKeyword = ""
End Sub
```

7. Click File ➤ Save.

Retesting the *RoomLabel* Procedure

Follow these steps to test the changes made to the `RoomLabel` procedure:

1. Switch to the AutoCAD application window.
2. Open `ch09_Building_Plan.dwg`.
3. At the Command prompt, type **`vbarun`** and press Enter.
4. When the Macros dialog box opens, select the `RoolLabel.dvb!basRoomLabel.RoomLabel` macro from the list and click Run.
5. At the `Specify point for room label (F101) or change [Number/Prefix/Save]:` prompt, specify a point inside the room in the lower-left corner of the building.
6. Place two other room label blocks.
7. Save the drawing with the name **`RoomLabel Test - VBA.dwg`**, and then close the file.

8. Reopen the `RoomLabel Test - VBA.dwg` file.
9. Execute the `RoomLabel` procedure and press F2. Notice the current values being used are `104` for the number and a prefix of `F`, which were the current values before closing the drawing.
10. Add additional room labels. Keep the drawing file open when done.

Selecting Room Label Blocks

As I mentioned earlier, Xdata can be used to select the room label blocks placed with the `RoomLabel` procedure. Here, you'll create a new procedure named `SelectRoomLabels`, which creates a selection set with only the selected room label blocks. The room label blocks can then be selected using the Previous selection option at any `Select objects:` prompt.

The following steps show how to add the `SelectRoomLabels` procedure:

1. Switch to the VBA Editor.
2. In the VBA Editor, in the Project Explorer double-click the `basRoomLabel` component.
3. In the code editor window, scroll to the end of the code editor window and click after the last code statement. Press Enter twice.
4. Type the following text; the comments are here for your information and don't need to be typed:

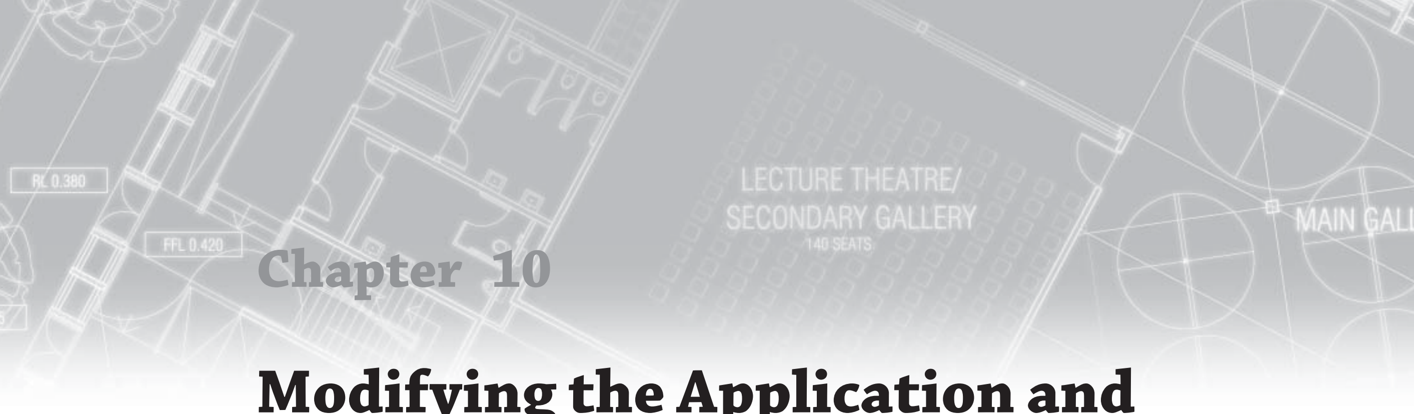
```
Sub SelectRoomLabels()
  On Error Resume Next

  ' Get the select set named SSRoomLabel if it exists
  Dim oSSet As AcadSelectionSet
  Set oSSet = ThisDrawing.SelectionSets("SSRoomLabel")

  If Err Then
    Set oSSet = ThisDrawing.SelectionSets.Add("SSRoomLabel")
  End If

  ' Define the data types array for the Xdata
  Dim nXdTypes(0) As Integer
  nXdTypes(0) = 1001

  ' Define the data values array for the Xdata
  Dim vXdVals(0) As Variant
  vXdVals(0) = "ACP_RoomLabel"

  ThisDrawing.Utility.Prompt _
      removeCmdPrompt & _
      "Select objects to filter on room labels: "

  ' Prompt the user to select objects to filter
  oSSet.SelectOnScreen nXdTypes, vXdVals
  ThisDrawing.Utility.Prompt _
      removeCmdPrompt & _
      "Use the Previous selection method to select room labels." & _
      vbLf

  ' Remove the selection set
  oSSet.Delete
End Sub
```

5. Click File ➤ Save.

The following steps explain how to test the `SelectRoomLabels` procedure:

1. Switch to the AutoCAD application window.
2. If you closed the `RoomLabel Test - VBA.dwg` file from the previous section, reopen it now.
3. At the Command prompt, type **`vbarun`** and press Enter.
4. When the Macros dialog box opens, select the `RoolLabel.dvb!basRoomLabel .SelectRoomLabels` macro from the list and click Run.
5. At the `Select objects:` prompt, type **`all`** and press Enter twice.
6. At the Command prompt, type **`erase`** and press Enter.
7. At the `Select objects:` prompt, type **`p`** and press Enter twice.

   All of the room label blocks have been removed.
8. At the Command prompt, type **`u`** and press Enter.
9. Save and close the drawing file.



Chapter 10

Modifying the Application and Working with Events

The ability to automate the creation and modification of objects in a drawing can be a huge productivity boost to any organization. As a programmer, you should always try to seamlessly integrate your custom programs into existing workflows and make it feel as if they were native to the AutoCAD® application.

You can implement a custom user-interface element to make it easier to start a macro or frequently used AutoCAD command. The user interface elements you implement can start macros from different VBA projects and even custom commands defined in AutoLISP® (LSP) or ObjectARX® (ARX) files that are loaded into the AutoCAD drawing environment. A VBA project can load other custom programs it requires into the AutoCAD environment.

In addition to implementing custom user interface elements, you can use events to help enforce your organization's CAD standards when AutoCAD and third-party commands are used. Events are specially named procedures that can be used to monitor changes to the AutoCAD application, an open drawing, or a specific graphical or nongraphical object in a drawing.

Manipulating the AutoCAD User Interface

The `AcadApplication` and `AcadDocument` objects can be used to manipulate the AutoCAD user interface. The `AcadMenuGroups` collection object returned by the `MenuGroups` property of the `AcadApplication` object allows you to access the customization groups (also known as menu groups) of all loaded CUIx files. A CUIx file is stored externally from the AutoCAD program and contains the definitions of various user interface element types that make up many of the tools displayed in the AutoCAD application window. I explain how to work with the `AcadMenuGroups` collection and `AcadMenuGroup` objects in the next section.

Pull-down menus on the menu bar, toolbars, and tabs on the ribbon are a few of the user interface elements that are stored in CUIx files. Use the CUI Editor (accessed using the `cui` command) to create new CUIx files and modify the user interface elements stored in an existing CUIx file. As an alternative, pull-down menus and toolbars can be customized using the AutoCAD Object library.

Customizing Older Versions of AutoCAD

If you are using AutoCAD 2006 through 2009, customization files had the file extension of `.cui` and not `.cuix`. Prior to AutoCAD 2006, customization files were known as menu files and had the file extension of `.mns`. MNS files were customized using the Customize dialog box (accessed using the `customize` command). You can convert CUI and MNS files to the CUIx file format in the latest release of AutoCAD by using the `cuiimport` command. For more information on working with older customization and menu files, see the `cuiimport` command in the AutoCAD Help system.

Some user interfaces can't be customized, but their display can be toggled using system variables. You can set or get the values system variables using the `SetVariable` and `GetVariable` methods of the `AcadDocument` object. I discuss how to toggle the display of some user interface elements that can be affected by system variables in the "Controlling the Display of Other User Interface Elements" section later in this chapter.

Managing Menu Groups and Loading Customization Files

In recent AutoCAD releases, each CUIx file contains a special name known as the *customization group name*. (In AutoCAD 2005 and earlier, it is the *menu group name*.) The customization group name must be unique for each CUIx file that is loaded into the AutoCAD drawing environment; if the name is already used by a loaded CUIx file, the AutoCAD program won't allow the new CUIx file to be loaded. A loaded CUIx file is represented by an `AcadMenuGroup` object in the `AcadMenuGroups` collection object. You can get the `AcadMenuGroups` collection object of the AutoCAD application by using the `MenuGroups` property of the `AcadApplication` object.

As with other collection objects, you can use the `Count` property to get the number of objects in the collection and the `Item` method to retrieve a specific object. You can use a `For` statement to step through a collection one object at a time if you don't want to retrieve a specific object with the `Item` method.

A customization group (CUIx file) can be loaded either as a base menu group (`acBaseMenuGroup`) or as a partial menu group (`acPartialMenuGroup`). A base menu group forces all other CUIx files to be unloaded before the CUIx file is loaded. A partial menu group is loaded in addition to any CUIx files that are already loaded. You can use the example that follows to see what's currently loaded in your AutoCAD session and help you determine how you wish to load a customization group.

The following example displays a message box for each `AcadMenuGroup` object in the `AcadMenuGroups` collection object. The message box displays the customization group name, the full path to the CUIx file, and how the CUIx file is loaded (base or partial).

```
Sub InfoMenuGroups()
  Dim oMnuGrp As AcadMenuGroup

  For Each oMnuGrp In ThisDrawing.Application.MenuGroups
    With oMnuGrp
      MsgBox "MenuGroup Info: " & vbLf & _
             "Name = " & .Name & vbLf & _
```

```
                "FileName = " & .MenuFileName & vbLf & _
                "Type = " & Switch(.Type = acBaseMenuGroup, "Base", _
                                   .Type = acPartialMenuGroup, "Partial")
      End With
    Next oMnuGrp
  End Sub
```

To load a CUIx file into the AutoCAD drawing environment, use the Load method of the AcadMenuGroups collection object. A customization group, or more specifically a CUIx file, is unloaded using the Unload method for an AcadMenuGroup object.

The following shows the syntax of the Load method:

```
object.Load cuixFileName [, type]
```

Its arguments are as follows:

object The *object* argument represents the variable that contains a reference to the AcadMenuGroups collection object.

cuixFileName The *cuixFileName* argument is a string that specifies the full path to the CUIx file to load.

type The *type* argument is an optional integer that specifies how the CUIx should be loaded. A value of 0 indicates the CUIx file should be loaded as a base customization file, which forces all other CUIx files to be unloaded before the specified CUIx file is loaded. A value of 1 specifies that the CUIx file should be loaded as an additional partial menu. You can also use the constant values acBaseMenuGroup and acPartialMenuGroup instead of 0 and 1 (an approach I recommend).

The following statements load a CUIx file named acp.cuix and unload the customization group named ACP:

```
' Loads the acp.cuix file as a partial file
ThisDrawing.Application.MenuGroups.Load "c:\acp.cuix", acPartialMenuGroup

' Unloads the menu group named ACP
ThisDrawing.Application.MenuGroups("ACP").Unload
```

The properties of the AcadMenuGroup object can also give you information about pull-down menus and toolbars. Look at the Menus and Toolbars properties. I discuss how to access the pull-down menus and toolbars included in a loaded CUIx file in the next section.

Working with the Pull-Down Menus and Toolbars

In recent releases, the ribbon has been the primary focus for accessing tools from the out-of-the-box AutoCAD user interface, but pull-down menus and toolbars still play an important role in custom tool access. The pull-down menus on the menu bar and toolbars displayed in the AutoCAD application window can be customized. You can control the display of pull-down menus and toolbars and modify the items on a pull-down menu or toolbar to align with your customization needs. I explain how to work with pull-down menus and toolbars in the following sections.

NOTE Changes made to a pull-down menu or toolbar can be saved to a CUIx file with the `Save` and `SaveAs` methods of the `AcadMenuGroup` object. The `Save` method saves changes back to the loaded CUIx file and expects a file type; specify a value of `0` for a compiled menu and `1` for a menu source file type. As an alternative, you can use the constant values `acMenuFileCompiled` and `acMenuFileSource` instead of `0` and `1` (and I recommend that you do). The `SaveAs` method saves changes to a specified CUIx file; you must specify a file type just as you do with the `Save` method. For more information on the `Save` and `SaveAs` methods, see the AutoCAD ActiveX Help system.

Customizing Pull-Down Menus and the Menu Bar

The menu bar is an area along the top of most Windows-based applications, and it's used to access a set of pull-down menus. A pull-down menu is displayed by clicking its caption on the menu bar. Each pull-down menu contains a set of items that are typically grouped by function. For example, the Draw pull-down menu contains items used to start a command that creates a new graphical object, as opposed to the Modify pull-down menu, which contains items related to changing an existing drawing object.

NOTE In recent AutoCAD releases, the menu bar is hidden in favor of the ribbon, but you can display it by using the `menubar` system variable. Set the `menubar` system variable to `1` to display the menu bar or `0` to hide it.

Figure 10.1 shows a pull-down menu expanded on the AutoCAD menu bar and how the objects in the AutoCAD Object library are visually related.

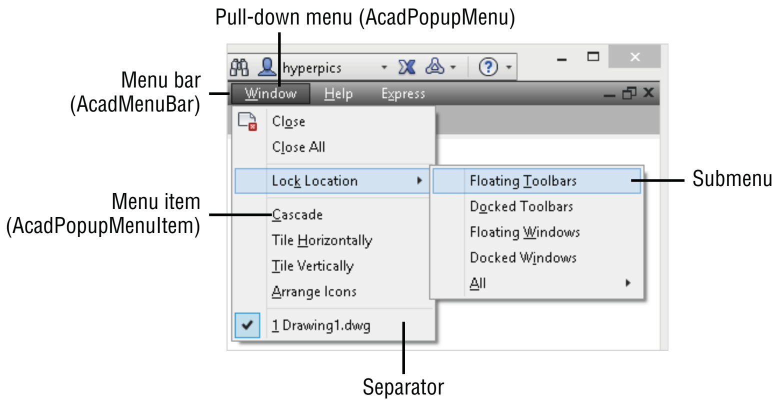


FIGURE 10.1 Visual reference of the objects that make up a pull-down menu

The pull-down menus that are displayed on the AutoCAD menu bar can come from any one of the loaded CUIx files. You access the pull-down menus of a loaded CUIx file using the `AcadMenuGroups` collection object returned by the `MenuGroups` property of the `AcadApplication`, which I discussed earlier, in the section "Managing Menu Groups and Loading Customization Files."

The `Menus` property of an `AcadMenuGroup` object returns an `AcadPopupMenus` collection object that represents the pull-down menus in the associated CUIx file. Use the `Item` method and a `For` statement to get an `AcadPopupMenu` collection object from an `AcadPopupMenus` collection object. You can add a new pull-down menu to an `AcadPopupMenus` collection object by using the `Add` method, which expects a string that represents the name of the new pull-down menu.

The following example code statements display a message box with a listing of the names for each pull-down menu in the `acad.cuix` file:

```
Sub ListAcadMenus()
  Dim sMsg As String
  sMsg = "List of pull-down menus in acad.cuix: "

  Dim oMenuGrp As AcadMenuGroup
  Set oMenuGrp = ThisDrawing.Application.MenuGroups("ACAD")

  Dim oPopMenu As AcadPopupMenu
  For Each oPopMenu In oMenuGrp.Menus
    If oPopMenu.ShortcutMenu = False Then
      sMsg = sMsg & vbLf & "  " & oPopMenu.NameNoMnemonic
    End If
  Next oPopMenu

  MsgBox sMsg
End Sub
```

Table 10.1 lists the properties that can be used to learn more about an `AcadPopupMenu` collection object.

TABLE 10.1: Properties that describe an `AcadPopupMenu` collection object

PROPERTY	**DESCRIPTION**
`Name`	Specifies the pull-down menu name with optional mnemonic characters. The mnemonic characters are used to access the pull-down menu from the keyboard and are displayed when the user holds the Alt key. Figure 10.1 has the mnemonic characters displayed for the pull-down menus and menu items.
`NameOnMnemonic`	Returns the menu name without the mnemonic characters.
`OnMenuBar`	Returns a Boolean value indicating whether the menu is displayed on the menu bar.
`ShortcutMenu`	Returns a Boolean value indicating that the menu is designated as a context menu displayed in the drawing area and not on the menu bar.
`TagString`	Returns the tags assigned to the pull-down menu. Tags are used to uniquely identify an item in a CUIx file.

The menu items on a pull-down menu can be organized into groups using separators and submenus. A submenu is an item that contains additional items; think along the lines of a folder inside of a folder. Menu items are represented by the `AcadPopupMenuItem` object. You can add new menu items to a pull-down menu by using the `AddMenuItem`, `AddSeparator`, and `AddSubMenu` methods of the `AcadPopupMenu` collection object. Existing menu items on a pull-down menu can be accessed by using the `Item` method and a `For` statement with an

AcadPopupMenu collection object. You can remove a menu item or submenu from on a pull-down menu by using the Delete method.

When you use the AddMenuItem, AddSeparator, and AddSubMenu methods, you must use an index value to specify where in the pull-down menu the item should appear. An index of 0 is used to specify the topmost item. In addition to the index, you must use a string to specify the menu item label when using the AddMenuItem and AddSubMenu methods. A third value is required when using the AddMenuItem method: you must specify the macro that should be executed when the menu item is clicked. The AddMenuItem and AddSeparator methods return an AcadPopupMenuItem object, and the AddSubMenu method returns an AcadPopupMenu collection object.

The following code statements create a pull-down menu named ACP—short for AutoCAD Customization Platform—and add a few menu items to it. The ACP pull-down menu is added to the ACAD menu group—which represents the acad.cuix file—but not saved to the CUIx file. Closing the AutoCAD application will result in the removal of the ACP menu.

```
Sub AddACPMenu()
  On Error Resume Next

  Dim oMenuGrp As AcadMenuGroup
  Set oMenuGrp = ThisDrawing.Application.MenuGroups("ACAD")

  Dim oPopMenu As AcadPopupMenu
  Set oPopMenu = oMenuGrp.Menus("ACP")

  If Err Then
    Err.Clear

    Set oPopMenu = oMenuGrp.Menus.Add("ACP")

    oPopMenu.AddMenuItem 0, "Draw Plate", _
                     Chr(3) & Chr(3) & _
                     "(vl-vbaload (findfile ""drawplate.dvb""))" & _
                     "(vl-vbarun " & _
                     """DrawPlate.dvb!basDrawPlate.CLI_DrawPlate"") "
    oPopMenu.AddSeparator 1

    Dim oPopSubMenu As AcadPopupMenu
    Set oPopSubMenu = oPopMenu.AddSubMenu(2, "Additional Tools")
    oPopSubMenu.AddMenuItem 0, "First Program", _
                        Chr(3) & Chr(3) & _
                        "(vl-vbaload (findfile ""firstproject.dvb""))" & _
                        "(vl-vbarun " & _
                        """firstproject.dvb!ThisDrawing.FirstMacro"") "
    oPopSubMenu.AddMenuItem 1, "BOM", _
                        Chr(3) & Chr(3) & _
                        "(vl-vbaload (findfile ""furntools.dvb""))" & _
                        "(vl-vbarun ""FurnTools.dvb!basFurnTools.FurnBOM"") "
  End If
End Sub
```

Figure 10.2 shows what the ACP pull-down menu would look like if you added it to the menu bar.

Figure 10.2
ACP pull-down menu

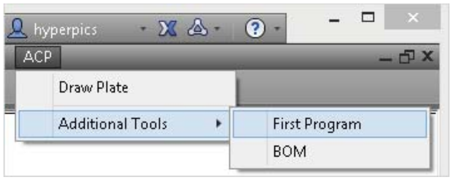


Table 10.2 lists the properties that can be used to change or learn more about an `AcadPopupMenuItem` object.

Table 10.2: Properties that describe an `AcadPopupMenuItem` object

Property	Description
`Caption`	Returns a menu item's text as it appears on a pull-down menu.
`Check`	Specifies a Boolean value indicating whether the menu item is checked. When the item is selected, a check mark is displayed to the left of its label. This is typically used to indicate a setting value, such as whether the UCS icon is displayed or the mode is enabled.
`Enable`	Specifies a Boolean value indicating whether the menu item is enabled. When disabled, the menu item can't be clicked.
`EndSubMenuLevel`	Specifies the nesting level in which the menu item appears on a submenu; the value is an integer.
`HelpString`	Specifies the help string to be displayed in the status bar when the cursor is over the pull-down menu item.
`Index`	Returns the index of the menu item (its location on the pull-down menu or submenu).
`Label`	Specifies the complete label for the menu item. This includes the text that is displayed in the user interface, mnemonic characters, and the DIESEL (Direct Interpretively Evaluated String Expression Language) macro that can be used to control the behavior of the menu item. DIESEL can be used to check or disable the menu item.
`Macro`	Specifies the macro that should be executed when the menu item is clicked. Use `Chr(3)` to represent the pressing of the Esc key to cancel the current command. Autodesk recommends that you use at least two instances of `Chr(3)` in a macro.
`SubMenu`	Returns a Boolean value indicating whether the menu item is a submenu.
`TagString`	Returns the tags assigned to the menu item. Tags are used to uniquely identify an item in a CUIx file.
`Type`	Returns an integer based on the menu item's type: 0 (or `acMenuItem`) for a menu item, 1 (or `acMenuSeparator`) for a separator bar, or 2 (or `acMenuSubMenu`) for a submenu.

A pull-down menu from an AcadPopupMenus collection object can be added to or removed from the menu bar. The leftmost place on the AutoCAD menu bar is specified by passing the location argument 0. Table 10.3 lists the methods that can be used to manage pull-down menus on the menu bar.

Table 10.3: Methods used to manage pull-down menus on the menu bar

Method	Description
InsertInMenuBar InsertMenuInMenuBar	Inserts an AcadPopupMenu collection object onto the AutoCAD menu bar. The InsertInMenuBar method accepts a single argument that is an integer specifying the pull-down menu's location on the menu bar. The InsertMenuInMenuBar method accepts a string that represents the name of the pull-down menu from the AcadPopupMenus collection object to insert onto the menu bar and an integer that specifies the pull-down menu's location on the menu bar.
RemoveFromMenuBar RemoveMenuFromMenuBar	Removes an AcadPopupMenu collection object from the AutoCAD menu bar. The RemoveFromMenuBar method doesn't require any argument values. The RemoveMenuFromMenuBar method accepts an integer that specifies the location of the pull-down menu to remove from the menu bar.

The MenuBar property of the AcadApplication object returns an AcadMenuBar collection object that contains the pull-down menus displayed on the menu bar. You can step through the collection object to see which pull-down menus are on the menu bar before adding or removing a pull-down menu.

The following example code statements check to see whether the pull-down menu with the name ACP is on the menu bar. If the pull-down menu isn't on the menu bar, the ACP pull-down menu is inserted onto the menu bar from the ACAD customization group.

```
Sub InsertACPMenu()

  ' Get the menu bar from the application
  Dim oMenubar As AcadMenuBar
  Set oMenubar = ThisDrawing.Application.MenuBar

  ' Set the default test condition to False
  Dim bMenuFound As Boolean
  bMenuFound = False

  ' Step through the pull-down menus on the menubar for ACP
  Dim oPopMenu As AcadPopupMenu
  For Each oPopMenu In ThisDrawing.Application.MenuBar
    If UCase(oPopMenu.NameNoMnemonic) = "ACP" Then
      ' Exit if the ACP menu is already on the menu bar
      bMenuFound = True
      Exit For
    End If
  Next oPopMenu

  ' If not found on the menu bar, insert ACP
  If bMenuFound = False Then
    Dim oMenuGrp As AcadMenuGroup
    Set oMenuGrp = ThisDrawing.Application.MenuGroups("ACAD")

    On Error Resume Next

    ' Insert the ACP menu
    oMenuGrp.Menus("ACP").InsertInMenuBar oMenubar.Count
  End If
End Sub
```

TIP Since AutoCAD 2006, workspaces have been used to control the display of pull-down menus on the menu bar. However, using a combination of CUIx files and the AutoCAD Object library, you can ensure a pull-down menu is available from the menu bar no matter which workspace is current.

Customizing Toolbars

Toolbars were among the first visual user interfaces that most Windows-based applications implemented as an alternative to pull-down menus. In recent AutoCAD releases, the ribbon has replaced much of the functionality that is part of a toolbar. However, it is beneficial to use both the ribbon and toolbars at the same time. For example, using the Layers toolbar, you can switch layers or see which layer is current without needing to switch to the Home tab on the ribbon. Less switching of interface elements means you can spend more time on design-related tasks.

A toolbar can be docked along one of the edges between the application and drawing windows, or in a floating state. Since toolbars can take up a fair amount of space onscreen, the number of tools that they provide access to is typically a small subset of those found on a pull-down menu. Like a pull-down menu, all the tools on a toolbar typically perform related tasks.

TIP By default, toolbars are hidden in recent AutoCAD releases. You can display a toolbar by using the Toolbars submenu on the Tools pull-down menu of the AutoCAD menu bar or the `toolbar` command. If the AutoCAD menu bar is hidden, set the `menubar` system variable to 1.

Figure 10.3 shows the Modify toolbar with the Array flyout expanded and shows how the objects in the AutoCAD Object library are visually related.

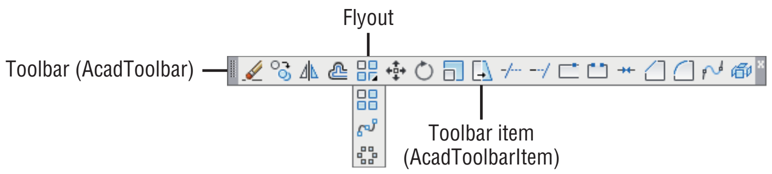


FIGURE 10.3 Visual reference of the objects that make up a toolbar

The toolbars that are displayed in the AutoCAD user interface can come from any one of the loaded CUIx files. You access the toolbars of a loaded CUIx file using the `AcadMenuGroups` collection object returned by the MenuGroups property of the AcadApplication, which I discussed earlier, in the section "Managing Menu Groups and Loading Customization Files."

The Toolbars property of an AcadMenuGroup object returns an AcadToolbars collection object that represents the toolbars in the associated CUIx file. You use the Item method and a For statement to get an AcadToolbar collection object from an AcadToolbars collection object. A new toolbar can be added to an AcadToolbars collection object by using the Add method; a string that represents the name of the toolbar is expected.

The following example code statements display a message box with a listing of the names for each toolbar in the acad.cuix file:

```
Sub ListAcadToolbars()
  Dim sMsg As String
  sMsg = "List of toolbars in acad.cuix: "

  Dim oMenuGrp As AcadMenuGroup
  Set oMenuGrp = ThisDrawing.Application.MenuGroups("ACAD")

  Dim cnt As Integer, nPrevNameChars As Integer
  cnt = 0: nPrevNameChars = 0

  Dim oTbar As AcadToolbar
  For Each oTbar In oMenuGrp.Toolbars

    ' Display the toolbar names in two columns
    If InStr(1, CStr(cnt / 2), ".") = 0 Then
      sMsg = sMsg & vbLf & "  " & oTbar.Name
    Else
      sMsg = sMsg & vbTab

      ' If the previous toolbar name was greater than or
      ' equal to 9 characters add a second tab
      If nPrevNameChars <= 9 Then
        sMsg = sMsg & vbTab
      End If

      sMsg = sMsg & "  " & oTbar.Name
    End If

    ' Get the number of characters in the toolbar name
    nPrevNameChars = Len(oTbar.Name)
    cnt = cnt + 1
  Next oTbar

  MsgBox sMsg
End Sub
```

Table 10.4 lists the properties that can be used to learn more about an AcadToolbar collection object. The toolbar must be visible before you can call many of its properties.

Table 10.4: Properties that describe an `AcadToolbar` collection object

Property	Description
`DockStatus`	Returns an integer that indicates where the toolbar is docked on the application window: `0` (or `acToolbarDockTop`) top `1` (or `acToolbarDockBottom`) bottom `2` (or `acToolbarDockLeft`) left `3` (or `acToolbarDockRight`) right `4` (or `acToolbarFloating`) floating
`FloatingRows`	Specifies the number of rows that the toolbar should conform to when floating.
`Height`	Returns the height of the toolbar in pixels when docked or floating.
`HelpString`	Specifies the help string to be displayed in the status bar when the cursor is over the button item on the toolbar.
`LargeButtons`	Returns a Boolean value that indicates whether the toolbar is shown using large or small button images. `True` is returned when large button images are being used.
`Left`	Specifies the left edge of the toolbar in pixels. The value is calculated from the left edge of the screen; the leftmost position is `0`.
`Name`	Specifies the toolbar's name.
`TagString`	Returns the tags assigned to the toolbar. Tags are used to uniquely identify an item in a CUIx file.
`Top`	Specifies the top edge of the toolbar in pixels. The value is calculated from the top edge of the screen; the topmost position is `0`.
`Visible`	Specifies whether the toolbar is visible onscreen. `True` indicates the toolbar is visible.
`Width`	Returns the width of the toolbar in pixels when docked or floating.

The button items on a toolbar can be organized into groups using separators and flyouts. A flyout is kind of like a submenu on a pull-down menu, but a flyout is a nested toolbar that is referenced by another toolbar and accessed from a button item. When the flyout is clicked, the most recent button on the flyout is used, but if the mouse cursor is over the button and the mouse button is held, the other button items of the nested toolbar can be selected.

Button items are represented by the `AcadToolbarItem` object. You can add new button items to a toolbar by using the `AddSeparator` and `AddToolbarButton` methods of the `AcadToolbar` collection object. Existing button items on a toolbar can be accessed by using the `Item` method and a `For` statement with an `AcadToolbar` collection object. You can remove a button item or flyout from a toolbar by using the `Delete` method.

When you use the AddSeparator and AddToolbarButton methods, you must specify an index location that specifies where the new item should appear on the toolbar. Index 0 is the leftmost item. In addition to an index, the AddToolbarButton method requires you to specify the following to add a button:

- Name: string value
- HelpString: string value
- Macro: string value
- Optionally, if the button should be a flyout: Boolean value

The AddSeparator and AddToolbarButton methods return an AcadToolbarItem object. When creating a flyout with the AddToolbarButton method, the *Macro* argument, although ignored, must have a value other than "", and True must be specified for the optional argument. After the flyout button is created, the AttachToolbarToFlyout method must be called on the AcadToolbarItem object returned by the AddToolbarButton method to attach a toolbar to the flyout button. The AttachToolbarToFlyout method expects the name of the customization group that the toolbar is part of and the toolbar name as assigned to its Name property.

A button item isn't very helpful without an image. You assign images to a button item by using the SetBitmaps method of an AcadToolbarItem object. If the image files are stored in the AutoCAD support file search paths, only the filenames of the small and large images need to be specified. If they are stored elsewhere, you must specify the full path to the images. You can use the GetBitmaps method on an existing button item to get the names of the small and large images used by a button item.

TIP Controls such as the Layer drop-down list or the Quick Find Text text box can't be added to a toolbar using the AutoCAD Object library. These controls must be added to a toolbar in a CUIx file with the CUI Editor (accessed by calling the cui command). The CUIx file can then be loaded and the toolbar displayed using a VBA program.

The following code creates a new toolbar named ACP—short for AutoCAD Customization Platform—and adds a few button items to the new toolbar and a CAD Standards toolbar as a flyout button. The ACP toolbar is added to the ACAD customization group—which represents the acad.cuix file—but not saved to the CUIx file.

```
Sub AddACPToolbar()
  On Error Resume Next

  Dim oMenuGrp As AcadMenuGroup
  Set oMenuGrp = ThisDrawing.Application.MenuGroups("ACAD")

  Dim oTbar As AcadToolbar
  Set oTbar = oMenuGrp.Toolbars("ACP")

  If Err Then
    Err.Clear
```

```
        Set oTbar = oMenuGrp.Toolbars.Add("ACP")

        Dim oTbarItem As AcadToolbarItem
        Set oTbarItem = oTbar. _
          AddToolbarButton(0, "Draw Plate", _
                           "Draws a plate with 4 bolt holes", _
                           Chr(3) & Chr(3) & _
                           "(vl-vbaload (findfile ""drawplate.dvb""))" & _
                           "(vl-vbarun " & _
                           """DrawPlate.dvb!basDrawPlate.CLI_DrawPlate"") ")
        oTbarItem.SetBitmaps "drawplate_16.bmp", "drawplate_32.bmp"

        Set oTbarItem = oTbar. _
          AddToolbarButton(1, "BOM", _
                           "Creates a BOM for the furniture blocks", _
                           Chr(3) & Chr(3) & _
                           "(vl-vbaload (findfile ""furntools.dvb""))" & _
                           "(vl-vbarun ""FurnTools.dvb!basFurnTools.FurnBOM"") ")
        oTbarItem.SetBitmaps "bom_16.bmp", "bom_32.bmp"

        oTbarItem.AddSeparator 2

        Set oTbarItem = oTbar. _
          AddToolbarButton(3, "CAD Standards", _
                           "CAD Standards toolbar", _
                           "Flyout", _
                           True)
        oTbarItem.AttachToolbarToFlyout "ACAD", "CAD Standards"
      End If
    End Sub
```

Figure 10.4 shows what the new ACP toolbar would look like if created using the example code statements. The images shown for the first two button items are part of this chapter's sample files.

Figure 10.4
New ACP toolbar

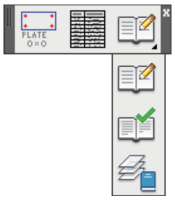


Table 10.5 lists the properties that can be used to change or learn more about an `AcadToolbarItem` object.

Table 10.5: Properties that describe an `AcadToolbarItem` object

Property	**Description**
`CommandDisplayName`	Specifies the text that mentions which commands are being used by the macro.
`Flyout`	Returns a Boolean value that indicates whether the button item is a flyout. `True` indicates the button is a flyout.
`HelpString`	Specifies the help string to be displayed in the status bar when the cursor is over the button item on the toolbar.
`Index`	Returns the index of the button item (its location on the toolbar).
`Macro`	Specifies the macro that should be executed when the button item is clicked. Use `Chr(3)` to represent pressing the Esc key to cancel the current command. Autodesk recommends the use of at least two instances of `Chr(3)` in a macro.
`Name`	Specifies the name for the button item.
`TagString`	Returns the tags assigned to the button item. Tags are used to uniquely identify an item in a CUIx file.
`Type`	Returns an integer based on the button item type: 0 (or `acToolbarButton`) button item 1 (or `acToolbarSeparator`) separator bar 2 (or `acToolbarControl`) control 3 (or `acToolbarFlyout`) flyout

A toolbar from an `AcadToolbars` collection object can be displayed and then docked or set to floating in the AutoCAD application window. A toolbar can be docked using the `Dock` method or set to floating with the `Float` method of the `AcadToolbar` collection object. The `Dock` method expects a single argument value of an integer between `0` and `3`—the same values as the `DockStatus` property mentioned in Table 10.4. The `Float` method expects three integer values that represent the top and left edges of the toolbar and how many rows the toolbar should be displayed with.

Why Won't My Toolbars Stay Put?

The order in which toolbars are docked isn't very straightforward, and the AutoCAD Object library is somewhat limited in this area. If you want to control the order in which toolbars are displayed with the AutoCAD Object library, you must undock all the toolbars from an edge of the application window and then re-dock them in a right-to-left or bottom-to-top order. There is no equivalent to the `AcadMenuBar` collection object to determine which toolbars are visible, so you must step through the `AcadMenuGroups` collection object returned by the `MenuGroups` property of the `AcadApplication` object. Then use the `AcadMenuGroup` object's `Toolbars` property and step through each toolbar and see which toolbars are visible and the edge they are displayed along. If you need absolute control over the placement of toolbars, consider using a CUIx file to define a workspace and set the workspace as current.

The following example code statements hide all toolbars and then redisplay three toolbars. The toolbars displayed are the ACP toolbar that could be created with the previous example, and then the standard AutoCAD Layers and Draw toolbars. The ACP and Layers toolbars will be docked below the ribbon, and the Draw toolbar will be floating near the center of the AutoCAD application window.

```
Sub DisplayToolbars()
  On Error Resume Next

  Dim oMenuGrp As AcadMenuGroup
  Dim oTbar As AcadToolbar

  ' Hide all toolbars
  For Each oMenuGrp In ThisDrawing.Application.MenuGroups
    For Each oTbar In oMenuGrp.Toolbars
      oTbar.Visible = False
    Next oTbar
  Next oMenuGrp

  ' Display the ACP, Layers, and Draw toolbars in the ACAD menugroup
  Set oMenuGrp = ThisDrawing.Application.MenuGroups("ACAD")

  ' Display the ACP toolbar, if found
  Set oTbar = oMenuGrp.Toolbars("ACP")
  oTbar.Visible = True
  oTbar.Dock acToolbarDockTop

  ' Display the Layers toolbar
  Set oTbar = oMenuGrp.Toolbars("Layers")
  oTbar.Visible = True
  oTbar.Dock acToolbarDockTop

  ' Display the Draw toolbar near the center of the
  ' AutoCAD application window
  Set oTbar = oMenuGrp.Toolbars("Draw")
  oTbar.Visible = True
  oTbar.Float (ThisDrawing.Application.Height / 2) + _
              ThisDrawing.Application.WindowTop, _
              (ThisDrawing.Application.Width / 4) + _
              ThisDrawing.Application.WindowLeft, _
              1
End Sub
```

TIP Starting with AutoCAD 2006, workspaces are used to control the display of toolbars in the AutoCAD user interface. However, using a combination of CUIx files and the AutoCAD Object library, you can ensure a toolbar is displayed no matter which workspace is current.

Controlling the Display of Other User-Interface Elements

Not all user interface elements of the AutoCAD application can be customized using the AutoCAD Object library. However, you can affect the display of some user interface elements by using the properties of the `AcadPreferencesDisplay` object, system variables, or commands. I mentioned the `AcadPreferencesDisplay` object in Chapter 3, "Interacting with the Application and Documents Objects." The following explains how you can control the display of some additional elements in the AutoCAD user interface:

Menu Bar You can control the display of the menu bar with the `menubar` system variable.

Layout Tabs The display of the layout tabs along the bottom of the drawing window can be toggled by setting the `DisplayLayoutTabs` property of the `AcadPreferencesDisplay` object to `True` or `False`. The following hides the layout tabs:

```
ThisDrawing.Application.Preferences.Display. _
    DisplayLayoutTabs = False
```

Scroll Bars The display of the scroll bars in the drawing window can be toggled by setting the `DisplayScrollBars` property of the `AcadPreferencesDisplay` object to `True` or `False`. The following hides the scroll bars:

```
ThisDrawing.Application.Preferences.Display. _
    DisplayScrollBars = False
```

Status Bars You can control the display of the drawing and application window status bars with the `statusbar` system variable.

TIP Workspaces stored in a CUIx file control the display of many user interface elements in the AutoCAD application window. You can set a workspace as current that is defined in a loaded CUIx file by using the `wscurrent` system variable.

Using External Custom Programs

VBA projects can use macros defined in other VBA projects and third-party commands as long as they are loaded into the AutoCAD drawing environment. Macros and commands can also be executed from user interface elements, such as pull-down menus, toolbars, and the ribbon, as you saw in the "Manipulating the AutoCAD User Interface" section. You shouldn't rely on a custom program file being loaded when you need it—you should load the custom program file before you try to call the macro or command.

NOTE If a custom program file is already loaded, loading a custom program file again typically doesn't affect the AutoCAD drawing environment or the current drawing. However, you will want to test what happens when reloading a custom program in your AutoCAD drawing environment, because some programs might execute code statements when a program is being loaded. The code statements that are executed could affect the objects and settings in the current drawing.

The following outlines how you can work with a custom program from a VBA project:

VBA Project The `LoadDVB` method of the `AcadApplication` object allows you to specify the full path of a DVB file you wish to load. Once loaded, the `RunMacro` method can be used to execute a macro in a DVB file from another VBA project. You must use the format *filename.dvb![projectname.]modulename.macro* to specify the macro to execute. For example, to execute the macro CLI_DrawPlate in the code module basDrawPlate of the DrawPlate.dvb, you would use DrawPlate.dvb!basDrawPlate.CLI_DrawPlate.

The UnloadDVB method lets you unload a DVB file when it is no longer needed. Be sure to specify the full path to the DVB file you wish to unload. If you are looking for information on getting your VBA program files loaded into the AutoCAD drawing environment, see Chapter 13, "Handling Errors and Deploying VBA Projects."

ObjectARX (ARX) File The LoadARX method of the AcadApplication object allows you to specify the full path of the ARX file to load. Once the file is loaded, use the SendCommand or PostCommand method of the AcadDocument object to execute one of the defined commands. You can unload an ARX file when it is no longer needed with the UnloadARX method; you must specify the full path to the ARX file you want to unload. You can get an array of all loaded ARX files with the ListARX method and determine whether the ARX file you need is already loaded.

Managed .NET DLL (AutoCAD 2005 and Later) Use the netload command with the SendCommand or PostCommand method of the AcadDocument object to load a Managed .NET DLL. The following shows an example of how to load the file named layerutils.dll:

```
ThisDrawing.SetVariable "filedia", 0
ThisDrawing.SendCommand "netload layerutils.dll" & vbCr
ThisDrawing.SetVariable "filedia", 1
```

AutoLISP (LSP/VLX/FAS) File Use the AutoLISP load function with the SendCommand or PostCommand method of the AcadDocument object to load an AutoLISP file. The following shows an example of how to load the file named layerutils.lsp:

```
ThisDrawing.SendCommand "(load ""layerutils.lsp"")" & vbCr
```

JavaScript (JS) File (AutoCAD 2014 and Later) Use the webload command with the SendCommand or PostCommand method of the AcadDocument object to load a JS file. The following shows an example of how to load the file named layerutils.js:

```
ThisDrawing.SendCommand "webload layerutils.js" & vbCr
```

Working with Events

There are two types of programming paradigms: *linear* and *event-driven*. In linear programming, code statements are executed in a specific and known order, typically a first-to-last approach. In event-driven programming, events are triggered by the actions of the user or messages from an application. Most modern applications that get input from the user using a dialog box or controls of some sort rely on event-driven programming. From a programming perspective, events are specially named procedures that are triggered under specific conditions. I discuss dialog boxes—or UserForms as they are known in VBA—and controls in Chapter 11, "Creating and Displaying User Forms."

Many of the objects in the AutoCAD Object library support event-driven programming. The AutoCAD application, document (or drawing), and graphical and nongraphical objects all support different types of events. Events can be used to monitor changes to the application and drawing windows, or even to enforce CAD standards for the objects in a drawing. For example, you can watch for the start of a dimension-related command or the `hatch` command and set a specific layer as current before the command accepts input from the user.

By default, the events of the current drawing are accessible from the `ThisDrawing` object in an AutoCAD VBA project. To use the events of other objects, such as the `AcadApplication` or a graphical object, you must declare a variable of the object type with the `WithEvents` keyword. The variable must be declared at the global level of `ThisDrawing` or within class and `UserForm` modules so that it persists beyond the current procedure.

The following are example code statements that declare an `AcadApplication` and `AcadBlock` object with events:

```
Public WithEvents oAcadApp As AcadApplication

Private WithEvents oTitleBlk As AcadBlock
```

Once you declare a variable of an object type with the `WithEvents` keyword, you must assign the variable an object by using the `Set` keyword. The assignment of an object to the variable is typically done when the VBA project is loaded using the `AcadStartup` procedure or when a procedure is executed. AutoCAD looks for and automatically loads a VBA project named `acad.dvb` and executes the procedure `AcadStartup` after the VBA project is loaded. In Chapter 13, I explain techniques that can be used to automatically load a VBA project file when the AutoCAD program starts up.

The following is an example of an `AcadStartup` procedure that assigns the `AcadApplication` object of the current drawing to the *`oAcadApp`* variable:

```
Public Sub AcadStartup()
  Set oAcadApp = ThisDrawing.Application
End Sub
```

Even though the procedure is named `AcadStartup`, it isn't executed automatically unless it is included in a DVB file named `acad.dvb`. Once you have declared a variable using the `WithEvents` keyword and assigned an object to the variable, you can then define a procedure that uses an exposed event of the object in your program.

The following steps explain how to add a procedure for an object event:

1. In the Project Explorer, double-click the `ThisDrawing` component.

   The code editor window opens. You will be working with the Object and Procedure drop-down lists (see Figure 10.5).

2. In the code editor window, click the Object drop-down list. Choose an object that you want to interact with when an event occurs.

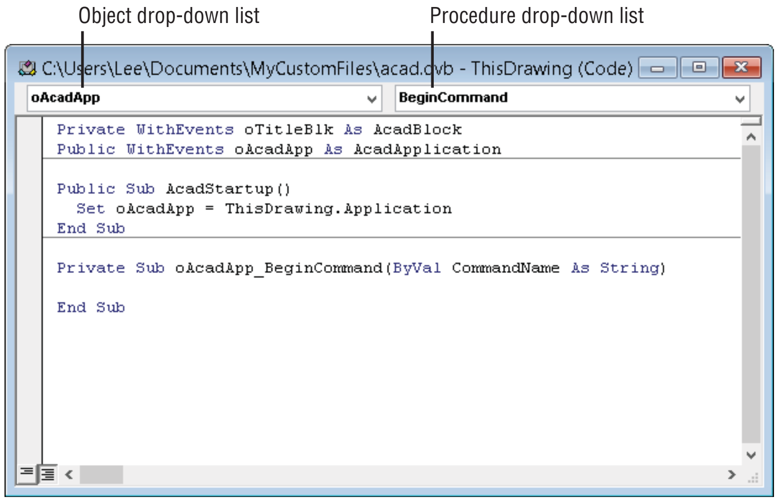
   After you make a selection in the Object drop-down list, the event you select in the Procedure drop-down list will be added to the code editor window. You can remove the event if it isn't the one you want.

3. Click the Procedure drop-down list and choose the event you want to use in your program.

A private procedure is added to the code editor window with the appropriate name and arguments. The following shows what the BeginCommand event looks like for a variable named *g_oAcadApp* that is of the AcadApplication object type:

```
Private Sub g_oAcadApp_BeginCommand(ByVal CommandName As String)

End Sub
```

FIGURE 10.5
Selecting an object and event to add


Object drop-down list
Procedure drop-down list
C:\Users\Lee\Documents\MyCustomFiles\acad.dvb - ThisDrawing (Code)
oAcadApp
BeginCommand

```
Private WithEvents oTitleBlk As AcadBlock
Public WithEvents oAcadApp As AcadApplication

Public Sub AcadStartup()
  Set oAcadApp = ThisDrawing.Application
End Sub

Private Sub oAcadApp_BeginCommand(ByVal CommandName As String)

End Sub
```

After the procedure is added to your project, you then add the code statements that should be executed when the proper conditions are met in the AutoCAD drawing environment for the event to be triggered.

NOTE Don't use the SendCommand or PostCommand method with an event-triggered procedure. The SendCommand and PostCommand methods are delayed and executed only after the AutoCAD program becomes idle, and the AutoCAD program typically doesn't enter an idle state until all the procedures triggered by events have been executed.

Table 10.6 lists some of the most commonly used events that the AcadApplication and AcadDocument objects support. For a full list of events, view the object's class in the Object Browser of the VBA Editor or the AutoCAD ActiveX Help system.

TABLE 10.6: Common events for the AcadApplication and AcadDocument objects

Events	Supported objects	Description
Activate Deactivate	AcadDocument	Occurs when a drawing window receives (Activate) or loses (Deactivate) focus as a result of switching drawing windows.
AppActivate AppDeactivate	AcadApplication	Occurs when the AutoCAD application window receives (AppActivate) or loses (AppDeactivate) focus as a result of switching applications.

Table 10.6: Common events for the `AcadApplication` and `AcadDocument` objects *(continued)*

Events	Supported objects	Description
`BeginClose`	`AcadDocument`	Occurs immediately after a request to close a drawing is made.
`BeginCommand` `EndCommand`	`AcadApplication`, `AcadDocument`	Occurs when a command begins or ends. Useful in determining which command a user is using.
`BeginDocClose`	`AcadDocument`	Occurs before a drawing is completely closed. Useful if you don't want to allow the drawing to be closed.
`BeginLisp` `EndLisp`	`AcadApplication`, `AcadDocument`	Occurs when the evaluation of an AutoLISP program or statement begins or ends. Useful in determining which AutoLISP programs a user is using.
`BeginOpen` `EndOpen`	`AcadApplication`	Occurs before or after a drawing file is opened.
`BeginPlot` `EndPlot`	`AcadApplication`, `AcadDocument`	Occurs before or after a layout is plotted.
`BeginQuit`	`AcadApplication`	Occurs before the application window is closed.
`BeginSave` `EndSave`	`AcadApplication`, `AcadDocument`	Occurs before or after a drawing file is saved.
`LayoutSwitched`	`AcadDocument`	Occurs when focus is switched from one layout to another.
`NewDrawing`	`AcadApplication`	Occurs when a new drawing is being created.
`ObjectAdded` `ObjectErased` `ObjectModified`	`AcadDocument`	Occurs when an object is added to, erased from, or modified in a drawing.
`SysVarChanged`	`AcadApplication`	Occurs when a change to a system variable is being made. Not all system variables trigger this event. In the system variables database I maintain on my website, `www.hyperpics.com/system_variables`, I indicate whether or not a system variable triggers this event.

In addition to the events listed in Table 10.6, the `AcadObject` object supports an event named `Modified`. The `AcadObject` object is the base class used to implement graphical and nongraphical objects, such as `AcadLine`, `AcadCircle`, `AcadLayers`, and `AcadLayer`. You can use the

Modified event to monitor changes to a specific object. However, instead of declaring a variable with events for a single object, it is often more efficient to use the ObjectModified event of the AcadDocument object.

Listing 10.1 shows an example program that logs the commands and first expressions of an AutoLISP program that are used after the BeginLog procedure is executed. Logging is disabled when the EndLog procedure is executed or when AutoCAD is closed. Using these two procedures, you can track the use of the custom programs and figure out which commands your users frequently use. I discuss how to write to a text file in Chapter 12, "Communicating with Other Applications."

LISTING 10.1: Custom command logging functionality using events

```
Private Sub AcadDocument_BeginCommand(ByVal CommandName As String)
  LogActivity CommandName
End Sub

Private Sub AcadDocument_BeginLisp(ByVal FirstLine As String)
  LogActivity FirstLine
End Sub

Private Sub LogActivity(sActivity As String)
  On Error Resume Next

  ' Create a new text file named Data.txt
  Dim ofsObj As FileSystemObject
  Set ofsObj = CreateObject("Scripting.FileSystemObject")
  sLogName = ThisDrawing.GetVariable("MyDocumentsPrefix") & _
             "\cmdtracker.log"

  ' Open or create the log file
  If ofsObj.FileExists(sLogName) Then
    Set oTxtStreamData = ofsObj.OpenTextFile(sLogName, ForAppending)
  Else
    Set oTxtStreamData = ofsObj.CreateTextFile(sLogName, False)
  End If

  ' Write text to the log file
  oTxtStreamData.WriteLine sActivity
  oTxtStreamData.Close

  Set ofsObj = Nothing
End Sub

Public Sub DisplayLog()
  ' Open the log in NotePad
  Dim oShell As Object
```

```
    Set oShell = CreateObject("WScript.Shell")
    oShell.Run "Notepad.exe " & _
               ThisDrawing.GetVariable("MyDocumentsPrefix") & _
               "\cmdtracker.log"
  End Sub
```

The code in Listing 10.1 is in the sample file `ch10_CodeListings.dvb` that is available from this book's website. When loaded, the code writes to a text file named `cmdtracker.log` in your `My Documents` (or `Documents`) folder. The following shows an example of the output containing commands and a single AutoLISP expression:

```
CIRCLE
(alert "Hello ACP!")
ERASE
LINE
ZOOM
ZOOM
```

Exercise: Extending the User Interface and Using Events

In this section, you will create a new VBA project that loads a customization file named `acp.cuix` into the AutoCAD drawing environment and uses events to help enforce some basic layer standards. The `acp.cuix` file contains a custom ribbon tab, pull-down-menu, and toolbar, all named ACP. Using the `AcadMenuGroups` collection object, you will load the CUIx file and then use properties of the customization group to display the pull-down menu and toolbar in the AutoCAD user interface.

The ribbon tabs in one or more loaded CUIx files can be merged. For example, you can merge a custom ribbon tab named ACP with the standard Home tab. Merging ribbon tabs can make it easier to control where or when a custom ribbon tab is displayed on the ribbon. The process of merging two or more ribbon tabs requires you to assign the same alias value to the ribbon tabs you want to merge. To merge a custom ribbon tab with the Home tab, you would assign the custom ribbon tab the alias `ID_TabHome`.

The events that you define in this exercise will allow you to monitor the opening of a drawing file and instances of several different commands. The `BeginCommand` event allows you to perform tasks before a command is started, whereas the `EndCommand` event allows you to perform tasks after a command has been completed. The `BeginCommand` and `EndCommand` events will be used to watch for the use of hatch- and dimension-related commands; to check the current layer, when necessary; to set a specific layer as current before the command is started; and, when necessary, to restore the previous layer after the command has been completed.

The key concepts I cover in this exercise are as follows:

Loading a CUIx File CUIx files can be used to add new and arrange user interface elements in the AutoCAD drawing environment. Using the AutoCAD Object library, you can load a CUIx file and even control the display of some of the user interface elements defined in the CUIx file.

Implementing Application and Document Events Events are used to define how the user can interact with the controls on the `UserForm` and the code statements that the `UserForm` should execute when loading or unloading.

NOTE The steps in this exercise depend on the Chapter 10 sample files (`ch10_acp.cuix` and `ch10_hexbolt.dvb`) available for download from `www.sybex.com/go/autocadcustomization`. If you completed all the exercises presented earlier in this book, you do not need to extract `ch10_drawplate.dvb` and `ch10_furntools.dvb` from the sample files archive. Place the sample files from the archive in the `MyCustomFiles` folder under the `Documents` (or `My Documents`) folder, or in the location you are using to store your custom program files. Once the sample files are extracted on your system, remove the characters `ch10_` from the filenames.

Loading the *acp.cuix* File

Here you load a customization (CUIx) file that contains a ribbon tab, pull-down menu, and toolbar with a few of the tools you have created in various exercises throughout this book.

The following steps explain how to create a procedure that loads a CUIx file, and then controls the display of the pull-down menu and toolbar named ACP:

1. Create a new VBA project with the name `Environment`; make sure to also change the default project name of `ACADProject` to `Environment` in the VBA Editor.
2. In the VBA Editor, in the Project Explorer, double-click the `ThisDrawing` component.
3. In the code editor window, type the following code. Throughout this exercise, I've included the comments for your information, and you don't have to type them.

```
Public Sub LoadACPMenu()
  On Error Resume Next

' Load the acp.cuix file
  ' Get the MenuGroups object of the application
  Dim oMenuGrps As AcadMenuGroups
  Set oMenuGrps = ThisDrawing.Application.MenuGroups

  ' Get the ACP menugroup if it is loaded
  Dim oMenuGrp As AcadMenuGroup
  Set oMenuGrp = oMenuGrps("ACP")

  ' If an error is returned, the ACP menugroup doesn't exist
  If Err Then
    Err.Clear
    oMenuGrps.Load ThisDrawing.GetVariable("MyDocumentsPrefix") & _
                "\MyCustomFiles\acp.cuix", acPartialMenuGroup
  End If

' Display the ACP toolbar
  Dim oTbar As AcadToolbar
  Set oTbar = oMenuGrp.Toolbars("ACP")

  ' If an error is returned, the ACP toolbar wasn't found
  If Not Err Then
    If oTbar.Visible = False Then
      oTbar.Visible = True
      oTbar.Dock acToolbarDockTop
    End If
  End If

' Add the ACP menu to the menubar
  ' Display the menubar
  ThisDrawing.SetVariable "menubar", 1

  ' Get the MenuBar object of the application
  Dim oMenuBar As AcadMenuBar
  Set oMenuBar = ThisDrawing.Application.MenuBar

  Dim oPopMenu As AcadPopupMenu
  Set oPopMenu = oMenuBar.Item("ACP")

  ' If an error is returned, the ACP menu isn't on the menubar
  If Err Then
    Err.Clear

    ' Add the ACP menu to the far right on the menubar
    oMenuGrp.Menus.InsertMenuInMenuBar "ACP", oMenuBar.Count
  End If
End Sub
```

4. Click File ➢ Save.

Specifying the Location of DVB Files

The macros that are defined in the `acp.cuix` file expect that the AutoCAD program can locate the DVB files in its support file search paths. Chapter 13 explains more about setting up the AutoCAD support file search paths and the use of trusted paths in recent releases.

The following steps explain how to add the `MyCustomFiles` folder under the `Documents` (or `My Documents`) folder, or in the location you are using to store your custom program files to the AutoCAD support file search paths:

1. Click the Application menu button ➢ Options (or at the Command prompt, type **`options`** and press Enter).
2. When the Options dialog box opens, click the Files tab.
3. Select the Support File Search Path node. Click Add and then click Browse.
4. In the Browse For Folder dialog box, browse to the `MyCustomFiles` folder that you created for this book in the `Documents` (or `My Documents`) folder, or browse to the folder that contains your DVB files.
5. Select the folder that contains your DVB files and click OK.
6. Click OK to save the changes to the Options dialog box.

Adding the Document Events

Document events allow you to monitor changes that occur in the drawing window and the objects and other elements in the associated drawing file. Using the `BeginCommand` and `EndCommand` events, you can ensure that specific settings are in place—either before or after a command has been executed. In this exercise, you will be using these events to make sure a specific layer is set as current to ensure that dimensions are placed on the `Dim` layer and that hatch and gradient fills are placed on the `Hatch` layer. Using the same approach, you should warn users when they are about to draw on layer 0, which isn't ideal unless they are creating blocks.

The following steps explain how to add the `BeginCommand` and `EndCommand` events:

1. In the code editor window, scroll to the top and type the text shown in bold:

```
Private g_sPrevLyr As String

Public Sub LoadACPMenu()
```

2. In the code editor window, click the Object drop-down list and choose AcadDocument. The Object drop-down list is in the upper-left corner of the code editor window.
3. Click the Procedure drop-down list and choose BeginCommand.
4. If a procedure other than `AcadDocument_BeginCommand` was added before step 2, remove that procedure.
5. Type the code in bold that follows:

```
Private Sub AcadDocument_BeginCommand(ByVal CommandName As String)
  On Error Resume Next

  ' Store the current layer
  g_sPrevLyr = ThisDrawing.ActiveLayer.Name

  ' Create and switch layers based on the command used
  Select Case CommandName
    Case "HATCH", "BHATCH", "GRADIENT"
      CreateSetLayer "Hatch", acRed
    Case "DIMLINEAR", "DIMDIAMETER", "DIMROTATED"
      CreateSetLayer "Dim", acGreen
  End Select
End Sub
```

6. Add the `EndCommand` event and type the code in bold that follows:

```
Private Sub AcadDocument_EndCommand(ByVal CommandName As String)
  On Error Resume Next

  ' Restore the previous layer if they are different
  If g_sPrevLyr <> "" Then
    ThisDrawing.ActiveLayer = ThisDrawing.Layers(g_sPrevLyr)
  End If
End Sub
```

7. Click File ➢ Save.

The BeginCommand event uses a procedure named CreateSetLayer to create a particular layer or to set that particular layer as current. The following steps add the CreateSetLayer procedure:

1. In the code editor window, click after the End Sub statement of the EndCommand event procedure and press Enter twice.
2. Type the following code:

```
Private Sub CreateSetLayer(sName As String, nColor As ACAD_COLOR)
  On Error Resume Next

  ' Get the layer if it exists
  Dim oLyr As AcadLayer
  Set oLyr = ThisDrawing.Layers(sName)

  ' If an error is returned, the layer doesn't exist
  If Err Then
    Err.Clear

    ' Create the layer and assign it a color
    Set oLyr = ThisDrawing.Layers.Add(sName)
    oLyr.color = nColor
  End If

  ' Set the layer current
  ThisDrawing.ActiveLayer = oLyr
End Sub
```

3. Click File ➢ Save.

Implementing an Application Event

Application events allow you to monitor changes that are made to the application window, as well as some of the events that are available as document events. Using the EndOpen event, you can make sure specific settings and even applications are available before the user begins working in the drawing. In this exercise, you will be using the EndOpen event to make sure the drawing opens in model space by setting the system variable tilemode to 0 and then adjusting the view of the drawing to show the extents of the objects in model space. EndOpen can also be useful if you are trying to create a batch operation and want to know when it is safe to begin making changes to the drawing.

To define a variable of the AcadApplication object type with events, in the code editor window, scroll to the top and add the following code:

```
Private WithEvents g_oAcadApp As AcadApplication
```

Now that you have defined a variable with events in the global scope of the `ThisDrawing` component, you can access the variable from the Object drop-down list and add an available event:

1. In the code editor window, click the Object drop-down list and choose g_oAcad. The Object drop-down list is in the upper-left corner of the code editor window.
2. Click the Procedure drop-down list and choose EndOpen.
3. If a procedure other than `g_oAcad_EndOpen` was added after step 1 as a result of choosing g_oAcad from the Object drop-down list, remove that procedure. Choosing a different object adds a default procedure, if not already defined, which might not be the procedure you want to work with.
4. Type the code in bold that follows:

```
Private Sub g_oAcadApp_EndOpen(ByVal FileName As String)
  On Error Resume Next

  ' Set the ModelSpace tab current
  ThisDrawing.SetVariable "tilemode", 1

  ' Zoom to the extents of the drawing
  ThisDrawing.Application.ZoomExtents

  ' Zoom out a bit
  ThisDrawing.Application.ZoomScaled 0.8, acZoomScaledRelative
End Sub
```

5. Click File ➤ Save.

Even though you added the `EndOpen` event, it won't be triggered until a reference to the AutoCAD application has been assigned to the `g_oAcad` variable. In the next section, you will add an `AcadStartup` procedure, which assigns the AutoCAD application object to the `g_oAcad` variable and executes the `LoadACPMenu` procedure.

Defining the *AcadStartup* Procedure

The `AcadStartup` procedure in this VBA project assigns the `AcadApplication` object of the current drawing to the global variable *`g_oAcad`* that was defined at the top of the code window of the `ThisDrawing` component. In addition to assigning the `AcadApplication` object to the *`g_oAcad`* variable, the `AcadStartup` procedure executes the `LoadACPMenu` procedure, which loads the `acp.cuix` file and displays the custom user interface elements in the AutoCAD drawing environment.

The following steps explain how to add the `AcadStartup` procedure:

1. In the code editor window, scroll to the bottom and click after the last code statement. Then press Enter twice.

2. Type the following code:

```
Public Sub AcadStartup()
  ' Assign the current Application object to the g_oAcad variable
  Set g_oAcadApp = ThisDrawing.Application

  ' Load the ACP menu
  LoadACPMenu

  ' Execute the loading of the ACP menu a second time
  ' The ACP menu isn't always added the first time
  LoadACPMenu
End Sub
```

3. Click File ➢ Save.

Testing the *AcadStartup* Procedure

Follow these steps to test the `AcadStartup` procedure in the `Environment.dvb` file:

1. Switch to the AutoCAD application and use the `vbarun` command to execute the `environment.dvb!ThisDrawing.AcadStartup` macro.

   The `acp.cuix` file is loaded and the custom user interface elements in the file are displayed, as you can see in Figure 10.6. The ACP menu is displayed on the menu bar, the ACP toolbar is docked below the ribbon, and the ACP ribbon tab is merged with the Home tab.

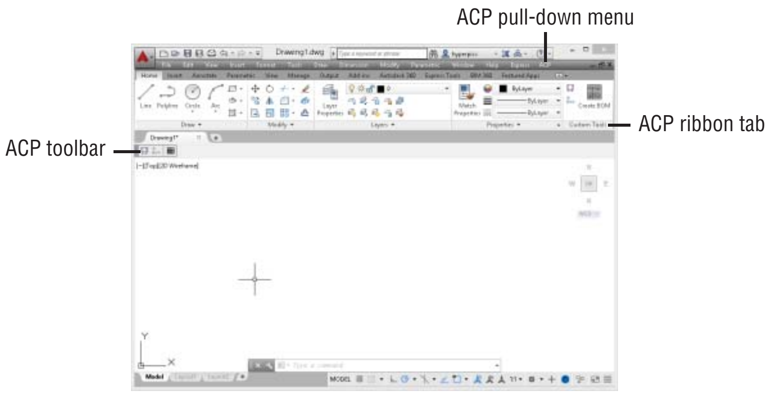


FIGURE 10.6 Custom user interface elements added by loading `acp.cuix`

2. Click one of the custom tools on the ribbon, pull-down menu, or toolbar. If the File Loading - Security Concern message is displayed (AutoCAD 2013 or later), click Load.

   The DVB file in the macro is loaded and then a specific macro is executed. If the message "Macro not found" is displayed in the Command History window, make sure you added the correct folder to the AutoCAD support file search paths and renamed the DVB files as needed.

3. Press Esc to cancel the macro.

Testing the Application and Document Events

To test the application and document events that are part of the `Environment.dvb` file, follow these steps:

1. Create a new drawing based on the `acad.dwt` drawing template file.
2. At the Command prompt, type **`layer`** and press Enter.

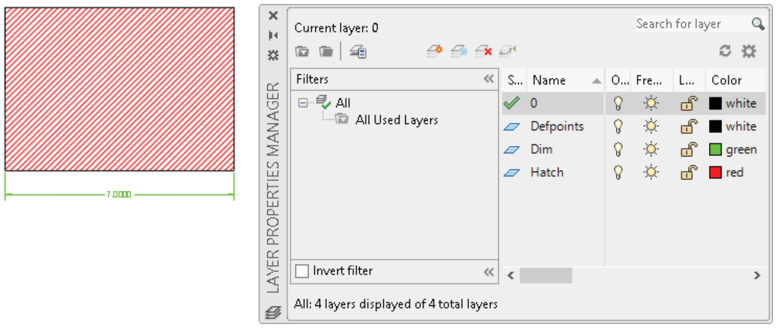
   Notice there is only one layer and it is named `0`.
3. At the Command prompt, type **`rectang`** and press Enter. Draw a rectangle.

   The new rectangular object is drawn on layer 0.
4. At the Command prompt, type **`hatch`** and press Enter.
5. Follow the prompts of the `hatch` command and specify a point inside the rectangle to apply a hatch fill.

   The hatch object is added to the `Hatch` layer that was created and set as current when the `hatch` command was started. The previous layer is restored after the `hatch` command ends.
6. At the Command prompt, type **`dimlinear`** and press Enter.
7. Follow the prompts of the `dimlinear` command, and specify the lower corners of the rectangle to create the linear dimension.

   The dimension object is added to the `Dim` layer that was created and set as current when the `dimlinear` command was started. The previous layer is restored after the `dimlinear` command ends.
8. Click the Layout1 tab.
9. Save the drawing with the name **`ch10_exercise`** to the `MyCustomFiles` folder, or in the location you are using to store the exercise files from this book.
10. Close the drawing.
11. Reopen the `ch10_exercise.dwg` file.

    Notice the drawing opens in model space and the extents of the objects in the drawing are displayed. Figure 10.7 shows the results of the document and applications.



S...	Name	O...	Fre...	L...	Color
✓	0				■ white
	Defpoints				■ white
	Dim				🟩 green
	Hatch				🟥 red

All: 4 layers displayed of 4 total layers

FIGURE 10.7 Layers created as a result of the `BeginCommand` event and the use of the `hatch` and `dimlinear` commands

Chapter 11

Creating and Displaying User Forms

Input from end users is either the key to a flexible and efficient program or its Achilles' heel. It all depends on how you gather and use that input. Up to this point, the input that you have been getting from the user has been requested at the AutoCAD® Command prompt. There is nothing bad about getting input only from the Command prompt, but it can be a limiting approach.

VBA programs support the ability to implement dialog boxes by adding a `UserForm` object to a project. Standard interactive controls that you are already familiar with from other Windows-based programs can be added to a user form to get input from the user. User forms allow a user to see values that might normally be hidden behind a set of prompts and provide input for only those options they are interested in changing. A user form can also be used to stitch multiple procedures together into a single, easy-to-use interface.

Adding and Designing a User Form

Many Windows-based programs use dialog boxes to get nonsequential input from the user and to provide feedback. A dialog box in a VBA project is known as a *UserForm object*. A user form, or dialog box, uses objects known as *controls*. A control can be of various types and sizes, and it usually accepts input from the mouse and/or keyboard that is attached to a workstation. In more recent years, input can come in the form of touch as well. Touch input is interpreted in a manner similar to mouse input. As a user clicks or types in a control, procedures known as *events* are executed. Events allow your program time to validate and manipulate the values provided through the control.

Adding a User Form to a VBA Project

With a VBA project loaded in the VBA Editor, a `UserForm` object can be added to the project. The default `UserForm` contains only a Close button in the upper-right corner, as shown in Figure 11.1. You can add a new `UserForm` object to a VBA project using one of the following methods:

- On the menu bar, click Insert ➢ UserForm.
- In the Project Explorer, right-click over the project and choose Insert ➢ UserForm.

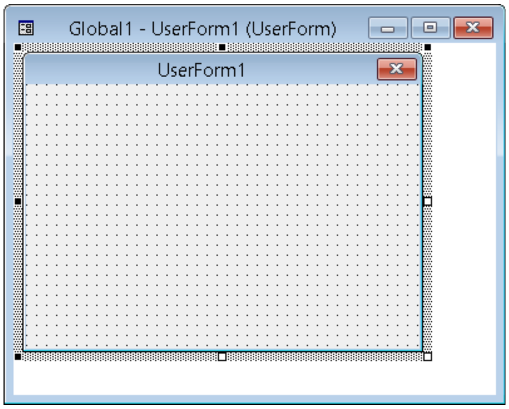


Figure 11.1
Default `UserForm` displayed in the UserForm editor window

When a new `UserForm` object is added to a VBA project, it is displayed in the UserForm editor window. You can perform the following tasks with the UserForm editor window:

- Add controls from the Toolbox window; see the "Placing a Control on a User Form" section later in this chapter for more information.
- Reposition, resize, group, and align controls.
- Use the Properties window to change the appearance of the user form or controls; see the "Changing the Appearance of a User Form or Control" section later in this chapter.
- Define the behavior of the user form as it is being loaded or when the user interacts with controls; see the "Defining the Behavior of a User Form or Control" section later in this chapter.

As I explained with naming variables in Chapter 2, "Understanding Visual Basic for Applications." Hungarian notation should be used to help identify a variable's data type. Hungarian notation is also typically used with `UserForm` objects and controls. The standard Hungarian notation used for a `UserForm` object name is `frm`.

TIP If you have a `UserForm` in another project that you want to reuse, export the `UserForm` to a form (FRM) file and then import it into your project. Right-click over a `UserForm` in the Project Explorer and choose Export File to export the component. To import a previously exported component, right-click over a project and choose Import File.

Considering the Design of a User Form

A user form often provides your users with their first impression of your program. Users typically don't see the code that is running behind the scenes where all the real magic happens. As in real life, first impressions can be hard to shake. The user forms you create for your programs should have a familiar feel, as if the user has been using them forever.

When creating a user form, consider the following basic guidelines:

- Controls with the most importance should be placed in the upper-left corner, whereas the least frequently used should be located in the lower area of the user form.
- The flow in a user form should be top-down and left-to-right.
- Controls should be aligned along the top edge when placed horizontally or along their left edge when placed vertically.
- Controls of the same type should be of a similar size.
- Organize and group related options together.
- Don't crowd the controls on a user form—be sure to put some space between the controls. Be aware that too much space can make a user form feel empty.
- Keep text labels and messages short and meaningful.
- Buttons used to accept or cancel the changes made should be placed horizontally along the lower right or vertically along the right edge of the user form.
- The button used to accept changes should be to the left of or above the button used to cancel the changes made.

You should also consider the following as you design a user form:

Will the user form be used to get input or provide feedback? User forms used to get input are displayed temporarily and then dismissed, whereas those used to provide feedback remain onscreen until they are dismissed. A good comparison might be dialog boxes versus palettes in the AutoCAD program.

Will the text on a user form need to be available in more than one language? Localizing text on a user form affects how controls are laid out and their size. German text strings on average are longer than most other languages, whereas text strings in languages such as Hebrew and Korean can be taller. As you design your dialog boxes, consider the impact that other languages might have on the width or height of the controls on the user form.

What should the look of a user form and its controls be? You can make your user form and its controls as vibrant as the latest summer clothing line or use a fancy font, but that doesn't mean you should. If you look at the dialog boxes in the applications you use every day, colors are commonly limited to identifying a tool through the use of an image or to communicate information about a problem. Fonts that are chosen are easy to read. The default color choices in most dialog boxes are friendly to those who are color-blind. Although only a small percentage of the population is color-blind, it is a factor that should be considered.

TIP For ideas on how to design your user forms, take a look at the dialog boxes you use every day to see how they are laid out and how they present information.

The guidelines and recommendations I mentioned are basic and the main ones I apply when creating a user form. You might also want to check with your organization to see if it has specific guidelines you should follow. Microsoft offers design guidelines and recommendations for Windows developers to help create similar and familiar dialog boxes and interfaces. I recommend you take a look at the guidelines Microsoft publishes, but remember that these are guidelines and not the be-all, end-all.

You can read more about Microsoft's recommendations for the Windows user experience with the following resources:

- Windows User Experience Interaction Guidelines (`www.microsoft.com/en-us/download/details.aspx?id=2695`)
- Designing a User Interface (`http://msdn.microsoft.com/en-us/library/windows/desktop/ff728820(v=vs.85).aspx`)
- Common UI Controls and Text Guidelines (`http://msdn.microsoft.com/en-us/library/windows/desktop/bb246433(v=vs.85).aspx`)

Placing and Arranging Controls on a User Form

Maybe you've never thought of yourself as the next Leonardo da Vinci, painting the next great work of art, but a well-designed user form can seem like a work of art. Okay, maybe not so much, but a new user form is similar to a blank canvas. You will select colors and fonts, and place and lay out controls, with precision and care.

You select controls from the Toolbox window and place them on the user form. You can then modify the position and size of your controls using grip editing. Grip editing in the UserForm editor window is similar to grip editing in the AutoCAD drawing window.

In addition to changing a control's position and size after it has been placed on a user form, you can change the control's properties using the Properties window or the control's interactive behavior. I explain how to change the appearance and define a control's interactive behavior in the "Changing the Appearance of a User Form or Control" and "Defining the Behavior of a User Form or Control" sections later in this chapter.

Placing a Control on a User Form

The Toolbox window, shown in Figure 11.2, is used to add controls to a user form. When you're editing a `UserForm` object in the UserForm editor window, the Toolbox window should be displayed. If the window isn't displayed, choose Toolbox on the Standard toolbar or from the View menu on the VBA Editor menu bar.

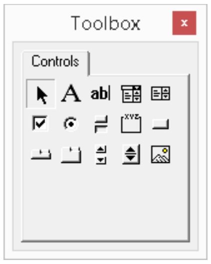


FIGURE 11.2
Controls that can be added to a user form are displayed in the Toolbox window.

From the Toolbox window, you can add a control to a `UserForm` object using any of the following methods:

- Click the icon that represents the control you want to add. Move your cursor to the UserForm editor window, and then click and drag to create the new control. This method allows you to specify both the location and the size of the control.
- Click the icon that represents the control you want to add. Move your cursor over the UserForm editor window and click. This method allows you to specify the upper-left corner of the new control; its size is set to a default value.
- Click and drag the icon that represents the control you want to add to the `UserForm`. This method allows you to specify the middle of the new control; its size is set to a default value.

By default, controls are placed on the `UserForm` using a grid system. The spacing of the grid is set to 6 points in the horizontal and vertical directions. The grid starts in the upper-left corner of the `UserForm` with a value of 0,0. The X value increases as you move to the right, and the Y value increases when moving down. You can toggle between showing and hiding the grid, specify the grid spacing, and toggle snap to grid on and off from the General tab of the Options dialog box in the VBA Editor. To display the Options dialog box, choose Tools ➢ Options on the VBA Editor's menu bar.

You can fine-tune the placement of a control with the control's `Left` and `Top` properties in the Properties window. You can also adjust the height and width of the control using the control's `Height` and `Width` properties. I explain how to change the properties of a control in the "Changing the Appearance of a User Form or Control" section.

Deciding Which Control to Use

The type of control you use depends on the information needed from the user. If you need the user to choose between a value of on or off, it wouldn't be productive to have the user type On or Off as it is more work and increases the potential for errors. You should become familiar with the 14 common controls that are available on the Toolbox window and the type of user interaction they support. Figure 11.3 shows the use of several of the common controls in two user forms.

The following describes the icons on the Toolbox window and the controls they represent:

Select Object The Select Object icon in the upper-left corner of the Toolbox window isn't used to place a control on the user form; it enables Object Selection mode. Click the icon again to exit Control Creation mode.

Label A label is used to display descriptive text and messages. Use the control's `Caption` property to change or get the current text. When naming a label, use the Hungarian notation prefix of `lbl`, such as `lblPlateWidth` or `lblPlateHeight`.

TextBox A text box allows the user to enter a text string. Use the control's `Value` property to change or get its current text. When naming a text box, use the Hungarian notation prefix of `txt`, such as `txtPlateWidth` or `txtPlateHeight`.

ComboBox A combo box (or drop-down list) allows the user to enter a text string or choose a predefined value from a list. Use the control's `Value` property to change or get its current value. You use the `AddItem` method of the control to add items to the drop-down list. When naming a combo box, use the Hungarian notation prefix of `cbo` or `cmb`, such as `cmbSectionViews.`

ListBox A list box allows the user to choose a predefined value from a list. Use the control's `Value` property to change or get its current value. You use the `AddItem` method of the control to add items to the list. When naming a list box, use the Hungarian notation prefix of `lst`, such as `lstBoltSizes`.

CheckBox A check box allows the user to indicate a value of on/off or true/false. This control is often used when the user can make multiple choices, such as wanting to use the Midpoint and Endpoint object snap modes. Use the control's `Value` property to change or get its current value. When naming a check box, use the Hungarian notation prefix of chk, such as `chkHiddenLines` or `chkAddLabel`.

OptionButton An option button (or radio button) allows the user to indicate a value of on/off or true/false. This control is often used when the user can choose only one out of multiple choices, such as using a straight line or spline segment for a leader line. Use the control's `Value` property to change or get its current value. When naming an option button, use the Hungarian notation prefix of `opt` or `rad`, such as `optSideView` or `radTopView`.

ToggleButton A toggle button allows the user to indicate a value of on/off or true/false. This control is similar to a CheckBox control, but it typically shows an image reflecting the current state of the control. Use the control's `Value` property to change or get its current value, or use the `Picture` property to display an image instead of text. When naming a toggle button, use the Hungarian notation prefix of `tgl`, such as `tglAddLabel`.

CommandButton A command button allows the user to start an action. This control is commonly used to accept or cancel the changes made to a user form or to display a nested user form. Use the control's `Caption` property to change or get its current display text, or use the `Picture` property to display an image instead of text. When naming a command button, use the Hungarian notation prefix of `cmd`, such as `cmdOK` or `cmdCancel`.

ScrollBar and SpinButton A scroll bar and spin button allows the user to specify a value within a range of two numeric values. Use the control's `Value` property to change or get its current value. When naming a scroll bar or spin button control, use the Hungarian notation prefix of `sb` or `spb`, respectively, such as `sbLength` or `spbHeight`.

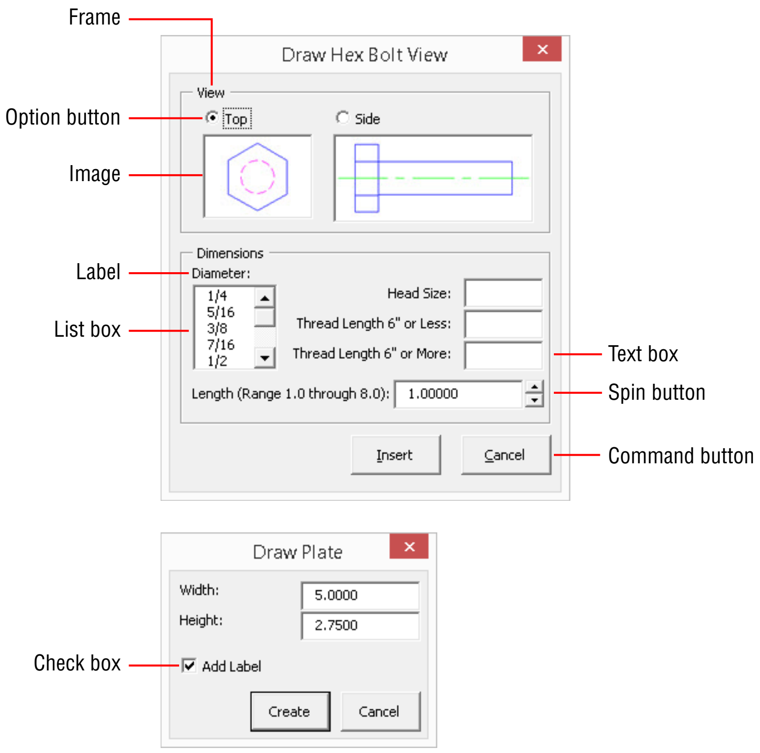
Image An image allows the user to start an action or get visual feedback about a value they might have chosen. Use the control's `Picture` property to specify the image to be displayed. When naming an image, use the Hungarian notation prefix of `img`, such as `imgTopView` or `imgSideView`.

NOTE Hungarian notation is used as a way to help identify the data type of a variable or the type an object represents. Its use is optional but highly recommended.

There are times when you might need to use a specialized control for the type of input or feedback you want to provide. The AutoCAD program installs two additional controls that you can use in a user form:

AutoCAD Control (`AcCtrl`) Allows you to embed an instance of the AutoCAD application in a user form. With this control, you can open a drawing and even use the control's `PostCommand` method to send command macros to the drawing to automate tasks.

AutoCAD Focus Control (`AcFocusCtrl`) I explain this control and its purpose in the "Keeping the Focus on Your User Form" sidebar later in this chapter.



Draw Hex Bolt View

View

(● Top) ○ Side

Dimensions

Diameter:

1/4
5/16
3/8
7/16
1/2

Head Size:

Thread Length 6" or Less:

Thread Length 6" or More:

Length (Range 1.0 through 8.0): 1.00000

Draw Plate

Width: 5.0000

Height: 2.7500

☑ Add Label

Figure 11.3 Common controls employed in user forms

Autodesk and third-party developers offer additional controls that can be used to display the thumbnail of a drawing or slide file, controls that mimic the standard AutoCAD color and linetype drop-down lists, data grids, and much more. To access the additional controls that Autodesk offers, you must be a registered Autodesk Developer Network (ADN) partner.

Autodesk Developer Network Partners

If you plan to make a career out of developing custom applications for others to use with AutoCAD, an ADN membership is optional but recommended. The membership grants you access to most of Autodesk's product offers for a flat annual fee and provides direct access to a professional support team that helps developers when they are stuck.

You can become an ADN partner by going to `www.autodesk.com/adn` or, if you are an Autodesk User Group International (AUGI) member, go to `https://www.augi.com/adn-membership-offer`. As of this writing, AUGI members can upgrade to the Professional level and get a complimentary ADN membership, which is a great deal for $100.

You can locate additional controls by searching the Internet on the keywords "free activex controls" or "purchase activex controls" along with VBA or VB6. Here are a few sites where you can get ActiveX controls that you can place on a user form:

`http://download.cnet.com`

`www.componentsource.com`

You can add third-party controls that you've installed and registered on your workstation to the Toolbox window by doing the following:

1. In the Toolbox window, right-click and choose Additional Controls.
2. When the Additional Controls dialog box opens, check the control to display on the Toolbox window and click OK.

NOTE Be careful when using an uncommon control, because it might not be available on other workstations or it may be available only on the 32- or 64-bit release of Windows.

Grouping Related Controls

Controls on a user form can be grouped in two different ways: for editing in the UserForm editor window or visually for user interaction. When controls are grouped in the UserForm editor window with the Group option, it doesn't affect how a user interacts with the controls when a user form is displayed in the AutoCAD drawing environment, but it does make editing and repositioning controls easier. To group controls in the UserForm editor window, hold down the Ctrl key and select the controls you want to group. Then right-click and choose Group.

After controls are grouped, clicking a control in the group selects the group. Once a group is selected, you can drag an individual control or the group's boundary to reposition all the controls in the group. With the group selected, you also can edit the common properties of all the controls in the group from the Properties window. I explain how to edit the properties of a control in the "Changing the Appearance of a User Form or Control" section later in this chapter. If you want to edit a single control in a group, select the group and then select the individual control you wish to edit. If a grouping of controls is no longer needed, you can ungroup the controls by selecting the group, right-clicking, and then choosing Ungroup from the context menu.

Grouping controls visually in the UserForm editor window can be achieved using the following controls from the Toolbox window:

Frame A frame graphically groups related controls and is a container object. You add a frame on a user form and then add the controls to be grouped over the frame. You can add an existing control to a frame by dragging it from the user form onto the frame and dropping it. As an alternative, you can cut a control from the user form and paste it to the frame. If you wish to cut and paste a control, select the frame before trying to paste it. Controls placed in the frame are moved or hidden when it is repositioned or its visibility changes. Use the control's `Caption` property to change or get its current display text. When naming a frame, use the Hungarian notation prefix of `fra` or `fam`, `frmViewStyle`, or `famBoltDimensions`, for example.

NOTE A container object, such as a `UserForm` or frame control, is used to hold and organize controls without the need for additional code statements. Controls placed on a UserForm or frame are displayed automatically when the `UserForm` or frame is shown or visible.

Tab Strip A tab strip control graphically groups related controls with the use of tabs. Unlike the frame control, a tab strip isn't a container object; this makes additional work for you. To control the display of controls with a tab strip, you use the tab strip's `Click` event to know when a tab is being switched and then use code statements to change the `Visible` property of the controls that should be hidden to `False` and those that should be displayed to `True`. Use the control's `SelectedItem` property to get information about the current page. When naming a tab strip, use the Hungarian notation prefix of `tb` or `tab`, such as `tbDrawHexBolt` or `tabDrawPlate`.

MultiPage A MultiPage control graphically groups related controls on different pages (or tabs). The MultiPage control is a container object like the frame control. You set one of the control's pages current, and then add the controls to the page that should be visible when that page is current. You add or manage pages on the control by right-clicking one of the pages and choosing the desired option. An existing control can be added to a page by dragging and dropping it from the user form onto the page, or by cutting and pasting the control to the page. If you cut a control from the user form, select the page before trying to paste it. Controls placed on the page are moved when it is repositioned or hidden when the page isn't current. Use the control's `SelectedItem` property to get information about the current page. When naming a MultiPage control, use the Hungarian notation prefix of `mp`, such as `mpDrawHexBolt`.

NOTE I recommend using the MultiPage control if you want to control the visibility of controls with tabs. The MultiPage control is a container object that doesn't require you to provide any additional code to determine which controls should be visible when a specific tab is current. The control requires less coding but offers fewer options to define how it should appear. The tab strip supports horizontal tabs, vertical tabs, and tabs displayed as buttons, whereas the MultiPage control only supports horizontal tabs along the top of the control.

Managing Controls on a User Form

Once a control has been placed on a user form or in a container control, such as a frame or MultiPage control, you can interactively manipulate, duplicate, remove, and change the display order of a control. The following explains how:

Moving a Control You can move a control by selecting and dragging it on the user form. As the control is dragged, it snaps to the grid based on the current spacing values. If you need to move a control off the grid, disable grid snapping or use the Properties window (which I explain in the "Changing the Appearance of a User Form or Control" section later in this chapter). You can disable grid snapping in the VBA Editor's Option dialog box (on the menu bar, choose Tools ➢ Options and click the General tab) and clear the Align Controls To Grid check box.

NOTE You can arrange command buttons along the bottom or right edge of a user form by selecting the command buttons you wish to arrange and choosing an option from the Arrange Buttons submenu on the Format menu bar.

Duplicating a Control An existing control and its properties can be duplicated to create a new control. To create a copy, right-click the control and choose Copy; then right-click and choose Paste. The name of the control is changed to its default value, so be sure to give the new control a meaningful name. The procedures (events) that define the user interaction behavior for a control aren't duplicated. I discuss how to define the behavior of a control in the "Defining the Behavior of a User Form or Control" section later in this chapter.

Removing a Control You can remove a control from a user form by selecting the control and pressing the Delete key or by right-clicking over the control and choosing Delete. You aren't prompted to confirm the removal of the control, so be careful about removing a control. Sometimes it is best to move a control off to the side and set its `Visible` property to `False`—just in case you need the control later. By setting the `Visible` property to `False`, you ensure that the control isn't accessible to the user of the user form when shown. If you determine later that the control is no longer needed, delete it.

Aligning a Control Although you can use the grid to align controls on a user form, it doesn't always produce the look and feel you want when it comes to controls of different types and sizes. You can align one control to the edge of another control. Hold the Ctrl key while selecting controls; the last control selected is designated as the anchor control (its grips are white filled instead of black). All of the selected controls will be aligned with the anchor control. Right-click one of the selected controls, choose Align, and then choose one of the alignment options. You can also align controls to the closest grid point while the Align Controls To Grid option is disabled by selecting one or more controls, right-clicking, and then choosing Align ➢ To Grid. The alignment tools can also be found on the Format menu and UserForm toolbar. If you want to center controls on the user form, select the controls you want to center, choose Format ➢ Center In Form, and then choose one of the suboptions.

Resizing a Control The size of a control can be adjusted by selecting the control and then using the grips that are displayed along the control's boundary. When the Align Controls To Grid option is enabled, as you drag a grip it snaps to the nearest grid point. Disable Align Controls To Grid or use the Properties window to adjust the size of a control when you don't want it to land on one of the grid points. If you want multiple controls to have the same height, width, or both, select the controls you want to make the same size. The last control selected defines the height and width that will be applied to all selected controls. Then right-click one of the selected controls, choose Make Same Size, and then choose one of the available options. The resizing tools can also be found on the Format menu and UserForm toolbar.

Spacing Controls Equally The spacing between two or more controls can be evenly distributed, increased, decreased, or removed altogether. Select two or more controls, choose Format ➢ Horizontal Spacing or Vertical Spacing, and then choose one of the available suboptions from the menu bar. The distance used to equally space the controls is based on the two outermost selected controls in the horizontal or vertical directions.

Controlling the Display Order of a Control The display order isn't something that needs to be specified too often, but you can adjust the order in which controls are displayed. Sometimes, you want to ensure that a control is displayed in front of another control. For example, you might want a text box to be displayed in front of a tab strip. You can adjust the display order of a control by right-clicking over the control and choosing Bring Forward or Send Backward. The display order of a control can also be changed using the `ZOrder` method of the control while the VBA project is being executed. The display order tools can also be found on the Format menu and UserForm toolbar.

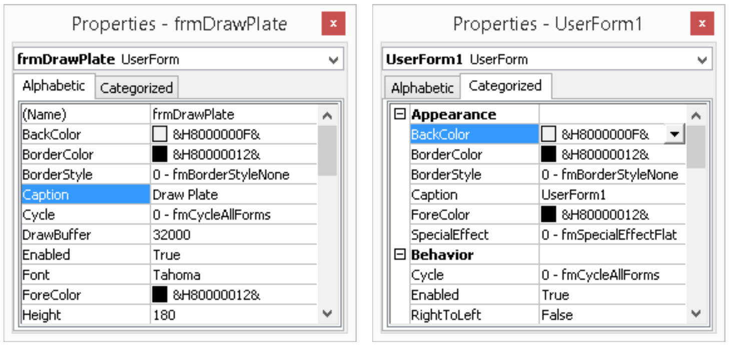
NOTE When one or more controls is selected, dragging one of the controls repositions all the selected controls. If you drag the grip of a selected control, all controls are resized accordingly.

Changing the Appearance of a User Form or Control

You can change the appearance of a user form or control in *design time* or *runtime*. Design-time is the time you spend developing an application before executing the procedures you have written. All of the objects you create or modify in an AutoCAD drawing are done during runtime; runtime is the time that occurs while a program is executing (or running).

During design-time, you add a `UserForm`, and then place and size controls on the `UserForm`. Like manipulating graphical and nongraphical objects in the AutoCAD drawing environment, the appearance of a `UserForm` or control can also be changed during runtime using code statements.

When a `UserForm` or control is selected in the UserForm editor window, its properties are displayed in the Properties window (see Figure 11.4). Display the Properties window by clicking View ➢ Properties Window, by choosing Properties Window on the Standard toolbar, or by pressing F4. To change a property of a `UserForm` or control, display the Properties window, select a property, and then change the property's value. Most of the properties displayed in the Properties window can also be changed at runtime.



Properties - frmDrawPlate

frmDrawPlate UserForm

Alphabetic | Categorized

(Name)	frmDrawPlate
BackColor	□ &H8000000F&
BorderColor	■ &H80000012&
BorderStyle	0 - fmBorderStyleNone
Caption	Draw Plate
Cycle	0 - fmCycleAllForms
DrawBuffer	32000
Enabled	True
Font	Tahoma
ForeColor	■ &H80000012&
Height	180

Properties - UserForm1

UserForm1 UserForm

Alphabetic | Categorized

⊟ **Appearance**	
BackColor	□ &H8000000F&
BorderColor	■ &H80000012&
BorderStyle	0 - fmBorderStyleNone
Caption	UserForm1
ForeColor	■ &H80000012&
SpecialEffect	0 - fmSpecialEffectFlat
⊟ **Behavior**	
Cycle	0 - fmCycleAllForms
Enabled	True
RightToLeft	False

FIGURE 11.4 View and change the property values in the Properties window.

TIP You can click the drop-down list at the top of the Properties window to choose which control to work with on the active `UserForm`. Click the Alphabetic and Categorized tabs below the drop-down list to specify whether the properties are displayed by name or in groups of similar purpose.

Figure 11.4 shows the Properties window with the properties of a `UserForm` named `frmDrawPlate`. Each `UserForm` has a property named `Caption` that controls the text displayed in its title bar. The text can be changed at design-time using the `Caption` property in the Properties window or at runtime using the `Caption` property, as shown in the following statement:

```
frmDrawPlate.Caption = "Draw Plate"
```

As you can see in Figure 11.4, there is a large number of properties that you can change to affect the appearance of a `UserForm` or control. In addition to properties that affect the appearance of a `UserForm` or control, there are properties that affect the behavior of a control during runtime. Table 11.1 lists some of the properties that affect the appearance or behavior of a `UserForm` or control.

Table 11.1: Common `UserForm` or control properties

Property	Description
`Cancel`	Determines which command button is used to discard changes; `CommandButton` set to `True` is executed when the user presses Esc.
`Default`	Determines which command button is used to accept changes; `CommandButton` set to `True` is executed when no other command button has focus and the user presses Enter.
`Enabled`	Determines whether a control can receive focus; `True` indicates the user can interact with the control.
`Font`	Specifies the font, font style, and size of the text displayed for a control.
`GroupName`	Specifies the name of a group. It is used to create a mutually exclusive group for CheckBox and OptionButton controls without using a Frame control.
`Height`	Specifies the height of a control or `UserForm`.
`Left`	Specifies the coordinate value for the leftmost edge of a control. The greater the value, the farther to the right on the `UserForm` the control is placed. A value of `0` specifies the control is positioned adjacent to the left edge of the `UserForm`.
`ListStyle`	Specifies the list style for a ComboBox or ListBox control.
`Locked`	Determines whether the user can change the value of a control; `True` indicates the value can't be changed.
`Style`	Specifies whether the user can enter information or only choose a listed value from a ComboBox.
`TabStop`	Determines whether the user can navigate to a control by pressing the Tab key; `True` indicates the control can be navigated to with the Tab key. Use the `TabIndex` property to set the tab order.

Property	Description
`Tag`	A property that can be used to store a custom or secondary value.
`Top`	Specifies the top edge of the control. The greater the value, the farther down on the `UserForm` the control is placed. A value of `0` specifies the control is positioned adjacent to the top edge of the `UserForm`.
`Visible`	Determines whether the control is visible at runtime; `True` indicates the control is visible.
`Width`	Specifies the width of a control or `UserForm`.

TIP User forms and the various control types have many properties in common; there are also many unique properties. Select a property in the Properties window and press `F1` to access help related to that property. This can be a great way to learn about properties.

Defining the Behavior of a User Form or Control

You might have noticed that some properties affect the behavior of a control. Properties alone don't define every behavior of a control. Consider what happens when the user enters text in a text box, clicks a command button or check box, or even chooses an option from a list. When a user interacts with a control, VBA looks for and executes specially named procedures known as *events*. I discussed how an event could be created to monitor changes to the application, drawing, or an object in a drawing in Chapter 10, "Modifying the Application and Working with Events."

`Click` is a commonly used event and is typically associated with a command button. VBA will execute the control's `Click` event if one has been defined and the user clicks the button. The same is true for other types of controls. The `KeyPress` event of a text box control can be used to determine which key the user pressed, and the `Change` event is used to notify you when the user makes a selection change in a list box.

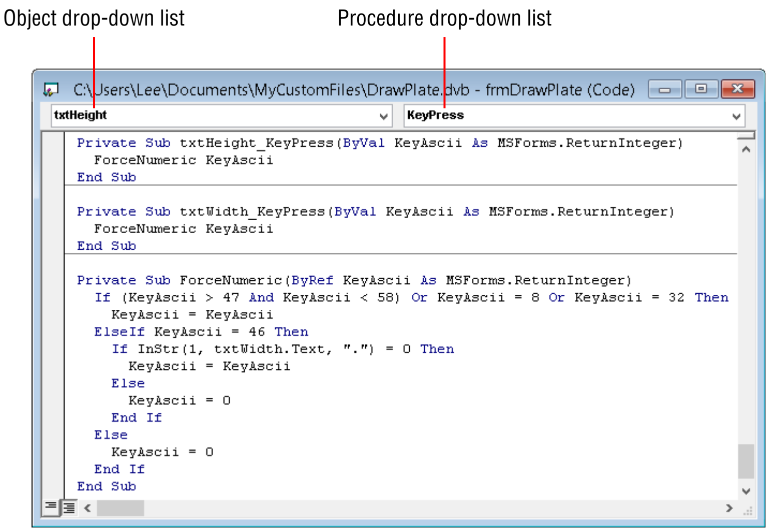
In addition to using events to get information about a control while the user is interacting with it, events are used to let you know when a `UserForm` component is being loaded or unloaded. The `Initialize` event is executed when a `UserForm` is being loaded the first time during a session, and the `QueryClose` and `Terminate` events are executed when a `UserForm` is being closed or unloaded from memory.

The following steps explain how to add an event to a `UserForm` or control:

1. In the Project Explorer, right-click over a `UserForm` component and choose View Code to display the `UserForm` in the UserForm editor window.

   The code editor window opens and looks just as it always has, but you will need to work with the Object and Procedure drop-down lists now (see Figure 11.5).

FIGURE 11.5
Selecting an object and event to add


Object drop-down list
Procedure drop-down list
C:\Users\Lee\Documents\MyCustomFiles\DrawPlate.dvb - frmDrawPlate (Code)
txtHeight
KeyPress

```
Private Sub txtHeight_KeyPress(ByVal KeyAscii As MSForms.ReturnInteger)
  ForceNumeric KeyAscii
End Sub

Private Sub txtWidth_KeyPress(ByVal KeyAscii As MSForms.ReturnInteger)
  ForceNumeric KeyAscii
End Sub

Private Sub ForceNumeric(ByRef KeyAscii As MSForms.ReturnInteger)
  If (KeyAscii > 47 And KeyAscii < 58) Or KeyAscii = 8 Or KeyAscii = 32 Then
    KeyAscii = KeyAscii
  ElseIf KeyAscii = 46 Then
    If InStr(1, txtWidth.Text, ".") = 0 Then
      KeyAscii = KeyAscii
    Else
      KeyAscii = 0
    End If
  Else
    KeyAscii = 0
  End If
End Sub
```

2. In the code editor window, click the Object drop-down list box. Choose UserForm to add an event for the `UserForm` or one of the controls on the `UserForm` to add an event for the selected control.

   When you make a selection in the Object drop-down list, the event selected in the Procedure drop-down list is added to the code editor window. Remove the event if it isn't the one you want.

3. Click the Procedure drop-down list and choose the event you want to use in your program.

   A private procedure is added to the code editor window with the appropriate name and arguments. The following shows what the `KeyPress` event looks like for a text box control named `txtHeight`:

```
Private Sub txtHeight_KeyPress(ByVal KeyAscii As MSForms.ReturnInteger)

End Sub
```

TIP You can double-click any `UserForm` or control (except a text box) from the UserForm editor window to add a `Click` event for that object. For text boxes, a double-click adds a `Change` event.

Table 11.2 lists some of the most commonly used events for `UserForms` and controls.

Table 11.2: Common `UserForm` or control events

Event	Description
`Change`	Executed when a change to a control's `Value` property occurs. You can use this event to validate the current value of a control and change the value as needed.
`Click`	Executed when the user clicks on the user form or a control or selects a value from a list. This event is often used to perform tasks related to accepting or discarding the values in a user form, such as an OK or Cancel button.
`DblClick`	Executed when the user double-clicks on the user form or a control. This event is often used to implement secondary click event for a control that already has a `Click` event. The speed with which the double-click must occur is based on the input settings for the operating system.
`Enter`	Executed when a control receives focus from another control. You can use this event to inform the user of the type of input expected before the control receives focus.
`Exit`	Executed when a control loses focus to another control. You can use this event to perform final validation of a control's value.
`Initialize`	Executed when a `UserForm` is loaded with the `Load` statement or displayed using the `Show` method. You can use this event to initialize the values of the controls on the `UserForm`.
`KeyPress`	Executed when the user provides input using a physical or onscreen keyboard. You can use this event to restrict the characters that the user can provide as input. For example, you can restrict a value to numeric characters only.
`QueryClose`	Executed when a request for the `UserForm` to be unloaded is made with the `Unload` statement or the Close button is clicked. You can use this event to veto and not allow the `UserForm` to be unloaded. The event isn't triggered when a `UserForm` is hidden.
`Terminate`	This is the last event executed before a `UserForm` is unloaded from memory. You can use this event to do any final cleanup of variables and store values to the Windows Registry so they can be restored the next time the `UserForm` is displayed. The event isn't triggered when a `UserForm` is hidden.

NOTE For more information on the events mentioned in Table 11.2 and other supported events, see the Microsoft Visual Basic Help available from the Help menu on the VBA Editor's menu bar.

Displaying and Loading a User Form

Once you've designed your user form, you need to get it in front of the users. Before you display a user form, you must decide if it should be displayed in a *modal* or *modeless* state. The modal state forces the user to interact only with your user form while it is displayed; no other tasks can be performed in the AutoCAD drawing environment while the user form is displayed. Dialog boxes such as the Insert (`insert` command) and Options (`options` command) in the AutoCAD program are examples of modal dialog boxes—you must click OK or Cancel to get back to the drawing environment.

The modeless state allows the user to interact with your user form and the AutoCAD drawing environment without first closing the user form. There are a number of examples of this behavior in AutoCAD. The ribbon, toolbars, Properties palette, and Tool Palettes window are all examples of modeless user interfaces. Use the modeless state if your user form provides real-time feedback (similar to the Properties palette) or, like the ribbon or a toolbar, is designed to allow the user to start a tool. User forms that are displayed in the modeless state typically don't have a traditional Accept or Cancel button.

Showing and Hiding a User Form

A `UserForm` object is displayed onscreen with the `Show` method. Once the form is displayed, the user can interact with its controls. The `Hide` method is used to hide the user form but will keep it loaded in memory to preserve the values a user might have entered for the next time the user form is displayed. The `Show` method accepts an optional integer value, which is used to indicate whether the user form should be displayed in the modal or modeless state; modal is the default display state. A value of `1` indicates the user form should be displayed in the modal state; a value of `0` specifies a modeless state. As an alternative, you can use the constant values `vbModal` and `vbModeless` in place of the integer values.

NOTE The `Initialize` event of a `UserForm` is executed the first time a form is displayed in the current AutoCAD session. This procedure allows you to set up the default values for the controls on a `UserForm` before it is displayed. I discussed how to use events to define how a user can interact with a user form and its controls in the "Defining the Behavior of a User Form or Control" section.

The `Hide` method doesn't accept any values. When the `Hide` method is executed, the `UserForm` remains loaded in memory but is no longer displayed. It can be redisplayed using the `Show` method. It is common practice to hide a `UserForm` when the user might need to select objects or a point in the drawing area, and then redisplay it after the user has finished interacting with the drawing area. The current `UserForm` can be referenced by using the object name `Me`. `Me` is a self-reference, and it is commonly used when a control needs to reference the `UserForm` where it is located.

Here are examples of displaying and hiding a `UserForm`:

```
' Displays a UserForm named frmDrawPlate
frmDrawPlate.Show vbModal

' Hides the UserForm in which a control is placed
```

```
Me.Hide

' Redisplays a UserForm which was hidden by a control
Me.Show
```

Keeping the Focus on Your User Form

When the AutoCAD program is in the foreground, the frontmost application, it wants to keep all attention on itself. The palettes in the AutoCAD environment fight for attention when the cursor passes over them. The same thing happens when you display a user form in the modeless state; the AutoCAD program tries to grab attention away from your user form.

You tell AutoCAD that your modeless user form should be allowed to have focus while the user interacts with it by adding an AutoCAD Focus Control (`AcFocusCtrl`) control to the user form. The control isn't visible to the user when the user form is displayed, so its placement on the user form doesn't matter. I explained how to add controls to a user form in the "Placing a Control on a User Form" section earlier in this chapter.

To add the `AcFocusCtrl` control to the Toolbox window, follow these steps:

1. On the Toolbox window, right-click and choose Additional Controls.
2. When the Additional Controls dialog box opens, check AcFocusCtrl and click OK.

Loading and Unloading a User Form

A `UserForm` can be loaded into and unloaded from memory. The `Show` method, mentioned in the previous section, loads and immediately displays a `UserForm`. There are times when you might only want to load a `UserForm` into memory and manipulate its controls without displaying it immediately.

For example, if you have a program that uses one or more nested `UserForm objects`, similar to the Options dialog box, you can preload the nested `UserForm` objects into memory so they are all initialized and ready to go when needed. The loading of nested `UserForm` objects is typically handled in the `Initialize` event of the main `UserForm` in your program. The `Load` statement is used to load a `UserForm` into memory.

Once a `UserForm` is no longer needed, you can remove it from memory to free up system resources using the `Unload` statement. When a `UserForm` is hidden with the `Hide` method, it remains in memory and all of the control values are preserved until the project is unloaded from memory. (Projects are unloaded from memory as a result of unloading the VBA project or closing the AutoCAD program.) If you want to preserve control values between AutoCAD sessions, write the current values of the controls on the `UserForm` to the Windows Registry. Then, restore the values from the Windows Registry as part of the `Initialize` event. I discussed how to store custom values in the Windows Registry in Chapter 9, "Storing and Retrieving Custom Data."

The `Load` and `Unload` statements require the name of an object for their single argument value. The name must be of a `UserForm` in the current VBA project.

Here are examples of loading and unloading a `UserForm`:

```
' Loads the frmMySettings UserForm into memory
Load frmMySettings

' Unloads the frmMySettings UserForm from memory
Unload frmMySettings
```

Exercise: Implementing a User Form for the *DrawPlate* Project

In this section, you will add a user form to the `DrawPlate` project that was originally introduced in Chapter 4, "Creating and Modifying Drawing Objects." The dialog box replaces the width and height prompts of the `CLI_DrawPlate` procedure and adds an option to control the creation of the label. The key concepts I cover in this exercise are:

Creating a User Form and Adding Controls A user form and the controls placed on it are used to get input from or provide feedback to a user.

Implementing Events for a User Form and Controls Events are used to define how the user can interact with the controls on the user form and the code statements that the user form should execute when loading or unloading.

Displaying a User Form Once a user form has been created, it can be displayed in the AutoCAD drawing environment. The choices a user makes can then be used to control the behavior and output of a custom program.

NOTE The steps in this exercise depend on the completion of the steps in the "Exercise: Adding Annotation to a Drawing" section of Chapter 6, "Annotating Objects." If you didn't complete the steps, do so now or start with the `ch11_drawplate.dvb` sample file available for download from `www.sybex.com/go/autocadcustomization`. Place this sample file in the `MyCustomFiles` folder under the `Documents` (or `My Documents`) folder, or in the location you are using to store your custom program files. Once the sample file is stored on your system, remove the characters `ch11_` from the filename.

Adding the User Form

Chapter 6 was the last chapter in which any changes were made to the `DrawPlate` project. At that time, you implemented functionality that added a label to the plate that is drawn. Here you add a user form to get the width and height values to draw a plate.

The following steps explain how to add the user form:

1. Load the `DrawPlate.dvb` file into the AutoCAD drawing environment and display the VBA Editor.
2. In the VBA Editor, in the Project Explorer right-click the `DrawPlate` project and choose Insert ➢ UserForm from the context menu.
3. In the Properties window, click the `(Name)` field and type **`frmDrawPlate`**.

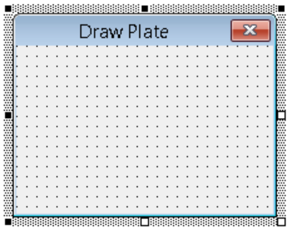
If the Properties window isn't displayed, click View ➢ Properties Window.

4. Change the following `UserForm` properties to the indicated values:

    - `Caption` = **`DrawPlate`**
    - `Height` = **`122`**
    - `Width` = **`158`**

    Figure 11.6 shows what the `UserForm` should look like after you have updated its properties.

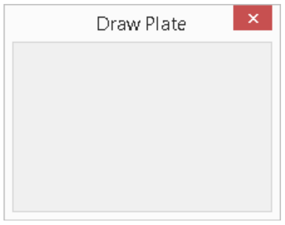
FIGURE 11.6 New `UserForm` in the editor window



5. Click File ➢ Save.
6. With the UserForm editor window active, click Run ➢ Run Sub/UserForm.

    Figure 11.7 shows what the user form looks like when executing.

FIGURE 11.7 New user form running in the AutoCAD drawing environment



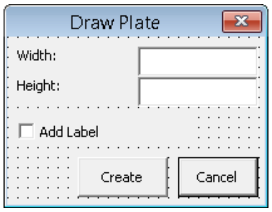
7. Click the Close button in the upper-right corner of the user form.

Adding Controls to the User Form

Controls are used to get input from a user. The type of controls you use depends on the type of input needed from the user. The Draw Plate user form will include two labels, two text boxes, a check box, and two command buttons. The labels are used to indicate the values that are expected for the text boxes. The two text boxes are used to get the width and height values for the plate, whereas the check box will be used to indicate whether a label should be placed in the drawing when the plate is drawn. The two command buttons will be used to draw the plate or exit the dialog box.

Figure 11.8 shows what the finalized user form will look like when completed.

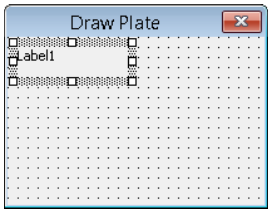
FIGURE 11.8
Completed Draw Plate user form



The following steps explain how to add two labels to the user form:

1. In the Project Explorer, double-click the `frmDrawPlate` component.
2. In the Toolbox window, click the Label icon.
3. In the UserForm editor window, click and drag to create the label shown in Figure 11.9.

FIGURE 11.9
The label control added to the user form



4. In the Toolbox window, click and drag the Label icon, and release the mouse button over the UserForm editor window when the outline of the control appears below the first label control placed.
5. Select the control labeled `Label1`.
6. In the Properties window, change the following properties of the `Label1` control to the indicated values:

`(Name) = `**`lblWidth`**	`Left = `**`6`**
`Caption = `**`Width:`**	`Top = `**`6`**
`Height = `**`18`**	`Width = `**`72`**

7. Select the `Label2` control and change its properties to the following:

`(Name) = `**`lblHeight`**

`Caption = `**`Height:`**

`Top = `**`24`**

8. Select the second label, and then press and hold the Ctrl key. Select the first label control you placed.

   The first label control should have white-filled grips.
9. Right-click over the selected controls and choose Align ➢ Lefts.

10. Right-click over the selected controls and choose Make Same Size ➢ Both.

The following steps explain how to add two text boxes to the user form:

1. In the Toolbox window, use the TextBox icon and place two text boxes. Place a text box to the right of each label.
2. Select the first text box placed, the one to the right of the label with the caption `Width:`.
3. In the Properties window, change the following properties of the `TextBox1` control to the indicated value:

`(Name)` = **`txtWidth`**	`Left` = **`78`**
`Height` = **`18`**	`Width` = **`72`**

4. In the Properties window, change the following property of the `TextBox2` control to the indicated value:

`(Name)` = **`txtHeight`**

5. Select the second text box, and then press and hold the Ctrl key. Select the first text box control you placed.
6. Right-click over the selected controls and choose Align ➢ Lefts.
7. Right-click over the selected controls and choose Make Same Size ➢ Both.
8. Select the first text box control you placed, and then press and hold the Ctrl key. Select the label with the caption `Width:`.
9. Right-click over the selected controls and choose Align ➢ Tops.
10. Align the tops of the second text box and label.

The following steps explain how to add a check box to the user form:

1. In the Toolbox window, click the CheckBox icon and place a check box below the second label.
2. In the Properties window, change the following properties of the CheckBox1 control to the indicated values:

`(Name)` = **`chkAddLabel`**	`Left` = **`6`**
`Caption` = **`Add Label`**	`Top` = **`48`**
`Height` = **`18`**	`Width` = **`108`**

The following steps explain how to add two command boxes to the user form:

1. In the Toolbox window, use the CommandButton icon and place two command boxes along the bottom of the form below the check box.
2. In the Properties window, change the following properties of the `CommandButton1` control to the indicated values:

`(Name) =` **`cmdCreate`**	`Left =` **`42`**
`Caption =` **`Create`**	`Top =` **`72`**
`Default =` **`True`**	`Width =` **`54`**
`Height =` **`24`**	

3. Change the following properties of the `CommandButton2` control to the indicated values:

`(Name) =` **`cmdCancel`**	`Cancel =` **`True`**
`Caption =` **`Cancel`**	`Left =` **`102`**

4. Select the second command button, and then press and hold the Ctrl key. Select the first command button you placed.
5. Right-click over the selected controls and choose Align ➢ Tops.
6. Right-click over the selected controls and choose Make Same Size ➢ Both.
7. Click File ➢ Save.

Displaying a User Form

The `Show` method of a `UserForm` is used to display it in the AutoCAD drawing environment.

The following steps explain how to create a procedure that displays the user form:

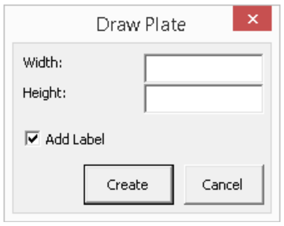
1. In the Project Explorer, double-click the `basDrawPlate` component.
2. In the code editor window, scroll to the end of the code editor window.
3. Click after the last code statement and press Enter twice. Type the following:

```
Public Sub DrawPlate()
  frmDrawPlate.Show
End Sub
```

4. Click File ➢ Save.
5. Switch to the AutoCAD application window.
6. At the Command prompt, type **`vbarun`** and press Enter.
7. When the Macros dialog box opens, select the `DrawPlate.dvb!basDrawPlate.DrawPlate` macro from the list and click Run.

   The Draw Plate user form is displayed in the AutoCAD drawing environment, as shown in Figure 11.10.

FIGURE 11.10 Completed Draw Plate user form in the AutoCAD drawing environment



8. Interact with the controls on the dialog box. Type **`acb123`** in the text boxes and click the command buttons.

   Notice you can enter text in the text boxes and check the check box. The command buttons don't do anything at the moment, and the text boxes accept any text characters entered with the keyboard.

9. Click the Close button in the upper-right corner of the user form.

Implementing Events for a User Form and Controls

Events are used to control what happens when a user clicks a button, types text in a text box, or even when a `UserForm` is loaded during an AutoCAD session. You will define the `Initialize` event for the `UserForm` to assign default values to the text boxes. In addition to setting up the default values for the text boxes, you will define a custom procedure that restricts the user to entering numeric values only into the text values. The custom procedure will be used with the `KeyPress` event of the text boxes.

The final events you will set up are related to the `Click` event of the two command buttons. When the Create button is clicked, it will use the values in the user form and prompt the user for the first corner of the plate. The Cancel button dismisses or hides the dialog box without drawing the plate.

The following steps explain how to set up the global variables and constants that will be used by the procedures of the Draw Plate user form:

1. In the Project Explorer, right-click the `frmDrawPlate` component and choose View Code.
2. In the code editor window, type the following:

```
Private myUtilities As New clsUtilities

Private g_drawplate_width As Double
Private g_drawplate_height As Double
Private g_drawplate_label As Boolean

' Constants for PI and removal of the "Command: " prompt msg
Const PI As Double = 3.14159265358979
Const removeCmdPrompt As String = vbBack & vbBack & vbBack & _
                                  vbBack & vbBack & vbBack & _
                                  vbBack & vbBack & vbBack & vbLf
```

3. Click File ➢ Save.

The following steps add the `Initialize` event for the `UserForm` and assign the default values to the controls:

1. In the code editor window, click the Object drop-down list and choose UserForm. The Object drop-down list is in the upper-left corner of the code editor window.
2. Click the Procedure drop-down list and choose Initialize.
3. If a procedure other than `UserForm_Initialize` was added before step 2, remove the procedure.
4. Between the `Private Sub UserForm_Initialize()` and `End Sub` code statements, type the following:

```
Private Sub UserForm_Initialize()
  ' Define the width and height for the plate, and enable label placement
  If g_drawplate_width = 0 Then g_drawplate_width = 5#
  If g_drawplate_height = 0 Then g_drawplate_height = 2.75
  If g_drawplate_label = 0 Then g_drawplate_label = True

  Me.txtWidth.Text = Format(g_drawplate_width, "0.0000")
  Me.txtHeight.Text = Format(g_drawplate_height, "0.0000")
  Me.chkAddLabel.Value = g_drawplate_label
End Sub
```

5. Click File ➢ Save.

The following steps define a custom procedure named `ForceNumeric`, which restricts input to numeric values only. The procedure is then assigned to the `KeyPress` event for the `txtWidth` and `txtHeight` controls.

1. In the code editor window, scroll to the end of the code editor window.
2. Click after the last code statement and press Enter twice. Type the following:

```
Private Sub ForceNumeric(ByRef KeyAscii As MSForms.ReturnInteger)
  If (KeyAscii > 47 And KeyAscii < 58) Or KeyAscii = 8 Or KeyAscii = 32 Then
    KeyAscii = KeyAscii
  ElseIf KeyAscii = 46 Then
    If InStr(1, txtWidth.Text, ".") = 0 Then
      KeyAscii = KeyAscii
    Else
      KeyAscii = 0
    End If
  Else
    KeyAscii = 0
  End If
End Sub
```

The procedure is passed the ASCII value of the key that is pressed, and if it isn't a number between 0 and 9, a period, a backspace (8), or a carriage return (32), the character returned is a `Null` value.

3. Add the `KeyPress` event for the `txtWidth` control and type the text in bold to modify the procedure:

```
Private Sub txtWidth_KeyPress(ByVal KeyAscii As MSForms.ReturnInteger)
  ForceNumeric KeyAscii
End Sub
```

4. Repeat step 3 for the `txtHeight` control.
5. Click File ➢ Save.

The following steps define the `Click` event for the Cancel button:

1. In the Project Explorer, right-click the `frmDrawPlate` component and choose View Object.
2. In the UserForm editor window, double-click the command button labeled Cancel.

3. The code editor, window is displayed and the `Click` event for the `cmdCancel` control is added. Type the text in bold to complete the event:

```
Private Sub cmdCancel_Click()
  Me.Hide
End Sub
```

4. Click File ➢ Save.

The following steps define the `Click` event for the Create button, which is a variant of the `CLI_DrawPlate` function.

1. Add the `Click` event to the `cmdCreate` control. Between the `Private Sub cmdCreate_Click` and `End Sub` code statements, type the following:

```
Private Sub cmdCreate_Click()
  Dim oLyr As AcadLayer

  ' Hide the dialog so you can interact with the drawing area
  Me.Hide

  On Error Resume Next

  Dim sysvarNames As Variant, sysvarVals As Variant
  sysvarNames = Array("nomutt", "clayer", "textstyle")

  ' Store the current value of system variables to be restored later
  sysvarVals = myUtilities.GetSysvars(sysvarNames)

  ' Set the current value of system variables
  myUtilities.SetSysvars sysvarNames, Array(0, "0", "STANDARD")

  ' Get recently used values from the global variables
  Dim width As Double, height As Double
  width = Me.txtWidth.Text
  height = Me.txtHeight.Text

  ' Prompt for a base point
  Dim basePt As Variant
  basePt = Null
  basePt = ThisDrawing.Utility.GetPoint(, _
           removeCmdPrompt & "Specify base point for plate: ")

  ' If a base point was specified, then draw the plate
  If IsNull(basePt) = False Then
    ' Create the layer named Plate or set it current
    Set oLyr = myUtilities.CreateLayer("Plate", acBlue)
    ThisDrawing.ActiveLayer = oLyr

    ' Create the array that will hold the point list
    ' used to draw the outline of the plate
```

```
Dim dPtList(7) As Double
dPtList(0) = basePt(0): dPtList(1) = basePt(1)
dPtList(2) = basePt(0) + width: dPtList(3) = basePt(1)
dPtList(4) = basePt(0) + width: dPtList(5) = basePt(1) + height
dPtList(6) = basePt(0): dPtList(7) = basePt(1) + height

' Draw the rectangle
myUtilities.CreateRectangle dPtList

' Create the layer named Holes or set it current
Set oLyr = myUtilities.CreateLayer("Holes", acRed)
ThisDrawing.ActiveLayer = oLyr

Dim cenPt1 As Variant, cenPt2 As Variant
Dim cenPt3 As Variant, cenPt4 As Variant
Dim dAng As Double, dDist As Double

' Calculate the placement of the circle in the lower-left corner.
' Calculate a new point at 45 degrees and distance of 0.7071 from
' the base point of the rectangle.
cenPt1 = ThisDrawing.Utility.PolarPoint(basePt, PI / 4, 0.7071)
myUtilities.CreateCircle cenPt1, 0.1875

' Calculate the distance between the first and second corners of the
' rectangle.
dDist = myUtilities.Calc2DDistance(dPtList(0), dPtList(1), _
                                   dPtList(2), dPtList(3))

' Calculate and place the circle in the lower-right
' corner of the rectangle.
dAng = myUtilities.Atn2(dPtList(2) - dPtList(0), _
                        dPtList(3) - dPtList(1))
cenPt2 = ThisDrawing.Utility.PolarPoint(cenPt1, dAng, dDist - 1)
myUtilities.CreateCircle cenPt2, 0.1875

' Calculate the distance between the second and third corners of the
' rectangle.
dDist = myUtilities.Calc2DDistance(dPtList(2), dPtList(3), _
                                   dPtList(4), dPtList(5))

' Calculate and place the circle in the upper-right
' corner of the rectangle.
dAng = myUtilities.Atn2(dPtList(4) - dPtList(2), _
                        dPtList(5) - dPtList(3))
cenPt3 = ThisDrawing.Utility.PolarPoint(cenPt2, dAng, dDist - 1)
myUtilities.CreateCircle cenPt3, 0.1875
```

```
  ' Calculate and place the circle in the upper-left
  ' corner of the rectangle.
  dAng = myUtilities.Atn2(dPtList(6) - dPtList(0), _
                          dPtList(7) - dPtList(1))
  cenPt4 = ThisDrawing.Utility.PolarPoint(cenPt1, dAng, dDist - 1)

  Dim oEnt As AcadEntity
  Set oEnt = myUtilities.CreateCircle(cenPt4, 0.1875)

  ' Force an update to the last object to display it when
  ' the dialog reappears.
  oEnt.Update

  If Me.chkAddLabel.Value = True Then
    ' Get the insertion point for the text label
    Dim insPt As Variant
    insPt = Null
    insPt = ThisDrawing.Utility.GetPoint(, _
            removeCmdPrompt & "Specify label insertion point " & _
                              "<or press Enter to cancel placement>: ")

    ' If a point was specified, place the label
    If IsNull(insPt) = False Then
      ' Define the label to add
      Dim sTextVal As String
      sTextVal = "Plate Size: " & _
                 Format(ThisDrawing.Utility. _
                   RealToString(width, acDecimal, 4), "0.0###") & _
                 "x" & _
                 Format(ThisDrawing.Utility. _
                   RealToString(height, acDecimal, 4), "0.0###")

      ' Create label
      Set oLyr = myUtilities.CreateLayer("Label", acWhite)
      ThisDrawing.ActiveLayer = oLyr

      Dim oMtext As AcadMText
      Set oMtext = myUtilities.CreateText(insPt, _
                          acAttachmentPointMiddleCenter, _
                          0.5, 0#, sTextVal)

      ' Use update to force the display of the label
      ' as it is the last object drawn before the form
      ' is redisplayed.
      oMtext.Update
    End If
  End If
End If
```

```
    ' Restore the saved system variable values
    myUtilities.SetSysvars sysvarNames, sysvarVals

    ' Save previous values to global variables
    g_drawplate_width = width
    Me.txtWidth.Text = Format(g_drawplate_width, "0.0000")
    g_drawplate_height = height
    Me.txtHeight.Text = Format(g_drawplate_height, "0.0000")
    g_drawplate_label = Me.chkAddLabel.Value

    ' Show the dialog box once done
    Me.show
  End Sub
```

2. Click File ➢ Save.

Testing the User Form and Controls

The following steps explain how to test the user form and the `DrawPlate` procedure in the `DrawPlate.dvb` file:

1. Switch to the AutoCAD application and use the `vbarun` command to execute the `DrawPlate.dvb!basDrawPlate.DrawPlate` macro.

   The Draw Plate user form is displayed.

2. In the Draw Plate user form, in the Width text box, clear the current value and type **`abc`**.

   Notice the text box ignores the characters `abc` as they are being typed.

3. In the Width text box, clear the current value and type **`4`**.
4. In the Height text box, clear the current value and type **`4`**.
5. Clear the Add Label check box.
6. Click Create.
7. At the `Specify base point for the plate:` prompt, pick a point in the drawing area to draw the plate and holes based on the width and height values specified.

   AutoCAD draws the completed plate without the label, as expected.

8. Run the macro again.
9. In the Draw Plate user form, select the Add Label check box.
10. Click Create and specify the insertion point for the plate and label.

AutoCAD draws the completed plate with a label this time. Figure 11.11 shows the results of the plates drawn with and without the label.

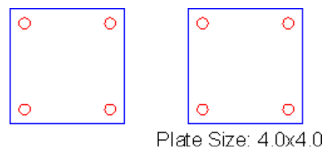


Figure 11.11
Completed plates

Chapter 12

Communicating with Other Applications

Everything up until this point has been focused on learning VBA, automating tasks in the AutoCAD® drawing environment, and manipulating the AutoCAD program itself. The VBA programming language also supports features that can be used to get information from outside of the AutoCAD program.

Using VBA, you can read and write text files that are stored on disc and leverage other libraries registered on your workstation with the ActiveX technology. Microsoft Windows comes preinstalled with a number of libraries that can be used to parse the information stored in an XML file or manipulate the files and directories on the discs that are accessible from your workstation. If you have Microsoft Office installed, you can also access Microsoft Word, Excel, and Access to read and write information to DOC, DOCX, XLS, XLSX, ACCDB, or MDB files.

Referencing a Programming Library

When a new VBA project is created, the Microsoft VBA and AutoCAD Object libraries are referenced by default. You can reference other libraries that are installed and registered on your workstation using the References dialog box. Here are examples of other programming libraries:

AutoCAD/ObjectDBX™ Common Type Library (`axdb<version>enu.tlb`**)** This library allows you to access the objects of a drawing without loading the drawing into the AutoCAD drawing environment first.

AcSmComponents 1.0 Type Library (`acsmcomponents<version>.tlb`**)** Using this library, you can automate tasks related to the Sheet Set Manager in the AutoCAD drawing environment.

Microsoft Excel Object Library (`excel.exe`**)** If you need to access the Excel application, use this library.

Microsoft Word Object Library (`msword<version>.olb`**)** Using the Microsoft Word Object Library, you can access the Word application.

Real World Scenario

Extending Nondrafting Workflows with VBA

Your boss has just come from the latest conference. He's excited about agile systems and how implementing Agile processes can make projects go smoother. After sitting in a few meetings, you realize how much extra work this could be in the short term, but you can see how it will help deliver more projects on time in the long term. So, you decide to participate in the pilot project using Agile processes.

One of the new processes that drafters will be responsible for is entering project team queries into an Excel spreadsheet. The spreadsheet will be used by the team to address issues during the daily meeting, report project status in Microsoft Project at each handoff point in a drawing, and notify the team of queries and handoffs by email. Using VBA, you help facilitate the information exchange. Your custom programs allow drafters to export status updates, handoffs, and queries to Excel and Project, and the interface in AutoCAD allows the drafters to respond to a query, send a request for more information, and update a project's status from within any drawing for the project.

The following explains how to add a reference to a third-party library in a VBA project:

1. In the VBA Editor, from the Project Explorer select a loaded project to set it as current.
2. On the menu bar, click Tools ➢ References.
3. When the References dialog box opens, scroll to the library you want to reference.
4. Click the check box next to the programming library to reference.

   If the programming library you want to load is not referenced, click Browse and select the library to load. Click Open.
5. Click OK.

Creating and Getting an Instance of an Object

Most of the objects that you have learned to work with were created using an object method defined in the AutoCAD Object library or using the `New` keyword. I explained how to use the `New` keyword to create a new instance of an object in the "Working with Objects" section of Chapter 2, "Understanding Visual Basic for Applications."

When using a library registered on your workstation, you can let VBA know that you want to use the library by referencing it first or simply creating an instance of an object that can be instantiated. Referencing a library as part of your VBA project is known as *early binding*. Early binding allows you to browse the objects and members of a library using the Object Browser in the VBA Editor as well as the IntelliSense (type-ahead) feature of the code editor windows. I mentioned how to reference a library in the "Referencing a Programming Library" section earlier.

The alternative to early binding is known as *late binding*. Late binding is when you use a programming library without first adding a reference to the library in your project. Early binding is the more popular approach when working with a programming library, but it does have a limitation.

Early binding forces your program to use a specific release of a programming library, whereas late binding allows you to work with any version of a programming library registered on your workstation. An example of when you might want to use late binding instead of early binding is when you want to create a program that can target and take advantage of the features in different releases of Word.

NOTE Late binding is more flexible when deploying an application to workstations that could have different versions of a programming library than you are using. However, early binding does make development and debugging easier because you can take advantage of IntelliSense for the library in the VBA Editor.

Creating a New Instance of an Object

The `New` keyword can only be used when you use early binding. To create a new instance of an object with early or later binding, you can use the `CreateObject` function. The `CreateObject` function expects a class ID for the object you want to create. Class ID values are defined in the Windows Registry, but the vendor of the library should have these documented as part of the library's documentation.

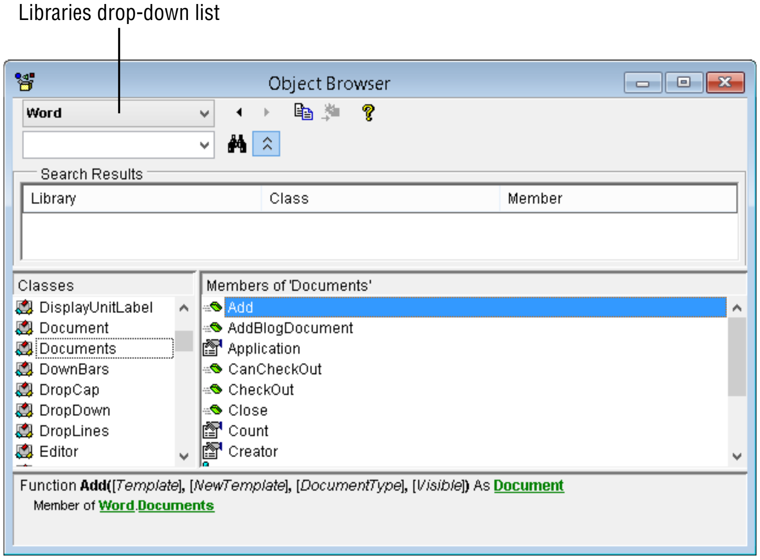
The following shows the syntax of the `CreateObject` function:

```
retObj = CreateObject(classID [, servername])
```

Its arguments are as follows:

`retObj` The *`retObj`* argument represents the object that is returned.

`classID` The *`classID`* argument is a string that represents the object to be created. The string is in the form of *`appname.objecttype`*[*`.version`*]. When the library has already been referenced in a project, the value of *`appname`* must match the name of the library you are calling exactly as it appears in the Libraries drop-down list in the Object Browser of the VBA Editor (see Figure 12.1). The value of *`objecttype`* specifies the type of object to be created, whereas *`version`* is an optional version number that could include a major and/or minor version number. Not all object types support a version number.



Function **Add(**[*Template*], [*NewTemplate*], [*DocumentType*], [*Visible*]**)** As **Document**
Member of **Word.Documents**

FIGURE 12.1 Check the Libraries drop-down list in the Object Browser and use the `appname` listed there.

An example of a major version number might be 19, and an example of a major and minor number might be 19.1. The `AcadApplication` object in the AutoCAD Object library supports both major and minor versions and is based on the release of AutoCAD. Table 12.1 lists some common class IDs for the `Application` object in the AutoCAD Object library.

Table 12.1: Common class IDs for the AutoCAD Object library `Application` object

Class ID	Specifies
`AutoCAD.Application`	Any version of AutoCAD
`AutoCAD.Application.19`	AutoCAD 2013 release
`AutoCAD.Application.19.1`	AutoCAD 2014 release
`AutoCAD.Application.20`	AutoCAD 2015 release

Be sure to refer to each library's documentation for object versioning information.

`servername` The *`servername`* argument is an optional string that represents the name of the server on the network where the object should be created. If no value or a value of `""` is provided, the object is created locally in memory on your workstation.

If the object specified by the *`classID`* argument can't be created, an error is generated. You will need to handle those errors. I show an example of how to handle the errors generated by the `CreateObject` function later in this chapter, and you can learn more about error handling in Chapter 13, "Handling Errors and Deploying VBA Projects."

Two objects must be created using the `CreateObject` function before they can be used: the File System object (`FileSystemObject`) from the Scripting Object library and the `Application` object in the Word Object library. The `FileSystemObject` object provides access to the files and folders on a local or network drive and allows you to automate the file management tasks related to your projects. For example, you could use the `Application` object of the Word Object library to generate sections of a bid specification or a cost estimation document from the information in a drawing. I discuss more about the File System and Word application objects in the "Working with Microsoft Windows" and "Working with Microsoft Office Applications" sections later in this chapter.

Here are examples of creating new instances of a filesystem or the Word application object:

```
' Create a new instance of the FileSystemObject without
' referencing the Microsoft Scripting Runtime (scrrun.dll)
Dim ofsObj as Object
Set ofsObj = CreateObject("Scripting.FileSystemObject")
```

```
' Create a new instance of the Word application without
' referencing the Microsoft Word 15.0 Object Library (msword15.olb)
Dim oWordApp as Object
Set oWordApp = CreateObject("Word.Application")
```

If you have both Word 2003 and Word 2013 installed on the same machine (a fair number of tech writers and editors have those two versions available for compatibility with client files or because they have an extensive macro set that was developed with 2003 and didn't translate smoothly to 2010 and later versions), you can specify which version of the application to work with by adding the version number to the class ID. Here are examples of creating a new instance of an `Application` object from the Word 2003 Object library and Word 2013 Object library with the `CreateObject` function:

```
' Create a new instance of the Word 2003 application
Dim oWordApp2003 as Object
Set oWordApp2003 = CreateObject("Word.Application.11")

' Create a new instance of the Word 2013 application
Dim oWordApp2013 as Object
Set oWordApp2013 = CreateObject("Word.Application.15")
```

When you create a new object, whether with the `New` keyword or the `CreateObject` function, you should consider whether the object should be removed from memory when you are done with it or left resident. Remove the object from memory if you don't plan to use it again or want the user to interact with the object; such is the case if you create an instance of the Word application. Some objects support a method such as `Close` or `Quit` that removes the object from memory, whereas some require you to set the object to `Nothing`.

For example, the following example creates a new instance of the File System object and gets the filesystem type of the C: drive.

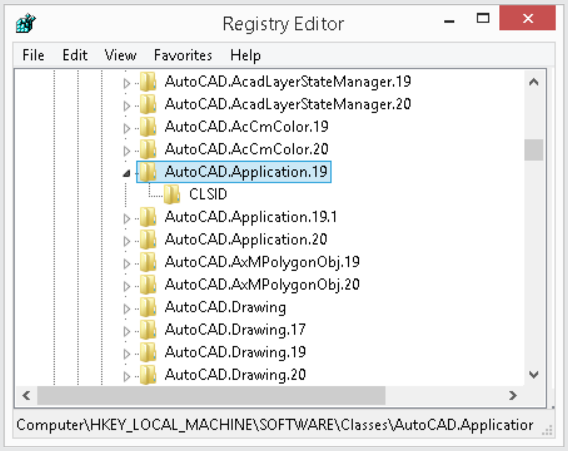
```
Dim ofsObj As Object
Set ofsObj = CreateObject("Scripting.FileSystemObject")

' Display the file system type of the C: drive
Dim oDrv As Object
Set oDrv = ofsObj.Drives("C")
MsgBox oDrv.FileSystem

' Release the object
Set ofsObj = Nothing
```

I discuss more about the File System object in the "Accessing the Filesystem" section later in this chapter.

Obtaining a List of the Class IDs Registered on Your Workstation

The `HKEY_LOCAL_MACHINE\SOFTWARE\Classes` key in the Windows Registry contains a list of all class IDs and their versions that can be created with the `CreateObject` function. You can display the Windows Registry by doing the following:

1. Right-click the Windows Start button on Windows XP or Windows 7, or right-click in the lower-left corner of the screen on Windows 8.
2. Click Run.
3. When the Run dialog box opens, type **`regedit`** and click OK to open the Registry Editor.



AutoCAD.AcadLayerStateManager.19
AutoCAD.AcadLayerStateManager.20
AutoCAD.AcCmColor.19
AutoCAD.AcCmColor.20
AutoCAD.Application.19
CLSID
AutoCAD.Application.19.1
AutoCAD.Application.20
AutoCAD.AxMPolygonObj.19
AutoCAD.AxMPolygonObj.20
AutoCAD.Drawing
AutoCAD.Drawing.17
AutoCAD.Drawing.19
AutoCAD.Drawing.20

Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Classes\AutoCAD.Applicatior

Getting an In-Memory Instance of an Object

In some cases, you don't need to create a new instance of an object but can work with an instance of an object that is already running in memory. The `GetObject` function returns an instance of an object that is running in memory or can create an object that is representative of a file. For example, you can get an instance of the Word `Application` object running in memory or create a Word `Document` object based on a DOC or DOCX file that is stored on disc.

The following shows the syntax of the `GetObject` function:

```
retObj = GetObject([pathName] [, classID])
```

Its arguments are as follows:

`retObj` The *`retObj`* argument represents the object that is returned.

`pathName` The *`pathName`* argument is an optional string that represents the filename containing the type of object to create and return. The filename could be an executable or a file created by an application, but the filename must correspond to a valid class ID in a registered programming library.

classID The *classID* argument is an optional string that represents the type of object in memory to be returned. For information on the structure of the class ID value, review the syntax of the *CreateObject* function in the "Creating a New Instance of an Object" section earlier in this chapter.

Here are two examples of the GetObject function:

```
' Gets an instance of the Word application running in memory
Dim oWordApp as Word.Application
Set oWordApp = GetObject("Word.Application")

' Creates an instance of a Document object of the
' Word Object library based on a file
Dim oWordApp as Word.Document
Set oWordApp = GetObject("c:\Users\Lee\Documents\MyCustomFiles\ch12_circles.doc")
```

NOTE Be cautious of setting the object returned by GetObject to Nothing; doing so could cause potential problems for the application that originally created the instance of the object. In most cases, unless the object is created as a result of specifying a file, I wouldn't set the object to Nothing.

Listing 12.1 shows a custom function named GetCreateObject that tries to get an instance of an object in memory before attempting to create a new instance of the object. This function can be helpful when creating a new instance of the AutoCAD or Word application. The code in Listing 12.1 can be found in the ch12_code_listings.dvb file that is available for download from www.sybex.com/go/autocadcustomization.

LISTING 12.1: Getting and creating an instance of an object based on a class ID

```
' Try and get an instance of an object before
' creating a new instance based on a class ID
Function GetCreateObject(classID As String) As Object
  On Error Resume Next

  ' Try and get an instance of the object
  Set GetCreateObject = GetObject(, classID)

  If Err Then
    Err.Clear

    On Error GoTo ErrHandler

    ' Create a new instance of the object
    Set GetCreateObject = CreateObject(classID)
  End If

  Exit Function
' If an error is generated when creating the
' new instance of the object, raise an error
ErrHandler:
  Err.Raise Err.Number, Err.Source, Err.Description, _
            Err.HelpFile, Err.HelpContext
End Function
```

Here is an example of how the `GetCreateObject` function can be used to get or create an instance of the Word application:

```
Sub WordAppInstance()
  On Error GoTo ObjectNotCreated

  Dim wordApp As Object
  Set wordApp = GetCreateObject("Word.Application")

  ' Make the application visible
  wordApp.Visible = True

  ' Create a new document
  wordApp.Documents.Add

  Exit Sub

ObjectNotCreated:
    MsgBox "MS Word couldn't be started." & vbLf & _
           "Verify MA Word is installed or the install isn't corrupt."
End Sub
```

If Microsoft Word isn't installed on the workstation, a message box with the text `MS Word couldn't be started.` is displayed when the previous example is executed.

Accessing a Drawing File from outside of AutoCAD

There are times when you want to get information from a drawing file that isn't opened in the current instance of AutoCAD. Although you could use the `Open` method of the `AcadDocument` or `AcadDocuments` objects, it takes time to open the drawing because you have to wait for the AutoCAD program to create a new document window and regenerate the drawing as it is being opened in the AutoCAD user interface.

Using the AutoCAD/ObjectDBX Common Type Library that comes with the AutoCAD program, you can access and modify graphical and nongraphical objects stored in a DWG file. Although there are limitations—such as not being able to allow the user to select objects to work with or use AutoCAD commands—it is much faster if you need to work with several to several hundreds of drawings quickly.

To use the AutoCAD/ObjectDBX Common Type Library in a VBA project, follow these steps:

1. Create a new or open an existing VBA project.
2. In the VBA Editor, click Tools ➢ References.

3. When the References dialog box opens, scroll to and check AutoCAD/ObjectDBX Common *<version>* Type Library.

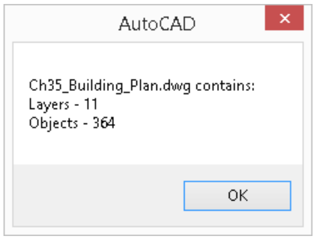
   You typically want to always work with the latest version on your workstation, but if you need to work with a specific version make sure you choose that one. Version 18 represents AutoCAD 2010 through 2012, 19 represents AutoCAD 2013 and 2014, and 20 represents AutoCAD 2015. If you have more than one instance of the library registered, be sure to reference the one in *<drive>*:`\Program Files\Common Files\Autodesk Shared`.

4. Click OK.

The following example shows how to open a DWG file in memory and list the number of layers and objects in model space that are present in the drawing. Figure 12.2 shows the resulting message box when the example code is executed.

```
Sub ReadDrawingEx()
  Dim sFlrPath As String, sDWGName As String
  sFlrPath = ThisDrawing.GetVariable("MyDocumentsPrefix") & _
             "\MyCustomFiles\"
  sDWGName = "Ch12_Building_Plan.dwg"

  Dim oDWGFile As New AxDbDocument
  oDWGFile.Open sFlrPath & sDWGName

  MsgBox sDWGName & " contains:" & vbLf & _
         "Layers - " & CStr(oDWGFile.Layers.Count) & vbLf & _
         "Objects - " & CStr(oDWGFile.ModelSpace.Count)

  ' Close the drawing file
  Set oDWGFile = Nothing
End Sub
```

FIGURE 12.2 Information from an externally opened drawing file



```
Ch35_Building_Plan.dwg contains:
Layers - 11
Objects - 364
```

If you make changes to the DWG file, be sure to save the changes with the `SaveAs` method. Because there is no `Close` method, you simply set the variable that contains the `Document` object that represents the DWG file to `Nothing`, as shown in the previous example.

Working with Microsoft Windows

Microsoft provides a number of programming libraries that allow you to access and use some of the features defined in the Windows operating system with your own programs.

These programming libraries can save you time and help to implement user experiences in your custom programs that are found in many other Windows-based programs. Although it isn't possible to introduce all of the Windows programming libraries that are available in this book, I will show you examples from a few of my favorite libraries.

Here are some Windows programming libraries that I suggest you take a look at:

- Microsoft Scripting Runtime
- Windows Script Host Object Model
- Microsoft Shell Controls and Automation
- Windows 32-bit API

For additional information on these libraries, use your favorite Internet search engine and do a search of the library names with VB6 as one of the keywords.

Accessing the Filesystem

The Microsoft Scripting Runtime library is your gateway to the Windows filesystem. The library allows you to access information about the files, folders, and drives on your workstation. To use the Microsoft Scripting Runtime library, reference the library as part of your VBA project or use late binding with the class ID `Scripting.FileSystemObject`.

You can use the Microsoft Scripting Runtime library to perform the following tasks:

- List the names and types of drives attached to a workstation
- Create and manage the folders on a drive
- Check to see if a file exists
- Get information about a file or folder, including the extension of a file, parent folder of a folder, or name of a special folder

NOTE You can also use the Microsoft Scripting Runtime library to create, read from, and write to a text file. I discuss accessing the content of a text file in the "Reading and Writing Text Files" section later in this chapter.

Here is how to create an instance of a File System object (`FileSystemObject`):

```
Dim ofsObj As New FileSystemObject
```

or

```
Dim ofsObj As Object
Set ofsObj = CreateObject("Scripting.FileSystemObject")
```

NOTE Don't forget to reference the Microsoft Scripting Runtime library in your VBA project before using the `New` keyword to create an instance of the File System object.

Getting the Drives Attached to a Workstation

When you first begin to work with the filesystem, it is a common tendency to want to work with a file. After all, a file is typically what you are creating with AutoCAD or any other application. Files are the lowest item in the filesystem hierarchy; drives are the very top.

The `FileSystemObject` object allows you to access all the drives attached to a workstation using the `Drives` collection object. Information about a drive can be obtained using the properties and methods of a `Drive` object.

The following example steps through the `Drives` collection object and displays a message box containing a drive's letter designation and the filesystem used to format it:

```
Sub ListDrives()
  Dim ofsObj As Object
  Set ofsObj = CreateObject("Scripting.FileSystemObject")

  ' Display the drive letter and file system
  Dim oDrv As Object
  For Each oDrv In ofsObj.Drives
    ' Check to see if the drive is ready
    If oDrv.IsReady Then
      MsgBox "Letter: " & oDrv.DriveLetter & vbLf & _
             "File System: " & oDrv.FileSystem
    Else
      MsgBox "Letter: " & oDrv.DriveLetter & vbLf & _
             "File System: *Drive not ready*"
    End If
  Next oDrv

  ' Release the object
  Set ofsObj = Nothing
End Sub
```

Table 12.2 lists the methods of the `FileSystemObject` object that can be helpful when working with the drives attached to a workstation.

Table 12.2: Drive-related methods of the `FileSystemObject` object

Method	Description
`DriveExists`	Returns `True` if the drive specified exists
`GetDrive`	Returns a `Drive` object based on a drive letter or UNC path
`GetDriveName`	Returns the drive name in a file path

Working with Folders and Special Folders

Folders are used to organize the files on a drive. You can use folders to organize your custom programs or all the drawing files related to a client project. The `Folder` object is used to get information about a folder on a drive, whereas the `Folders` collection object is used to access all the subfolders contained in a folder or drive. The `FileSystemObject` object can also be used to get the folders Microsoft has designated as special folders. There are three special folders: `Windows`, `System`, and `Temp`. Special folders are obtained using the `GetSpecialFolder` function and an integer value of `0` to `2`. Pass the `GetSpecialFolder` function a value of `0` to get the `Windows` folder. Use a value of `1` to get the `System` folder or `2` to get the `Temp` folder.

The following example lists the subfolders in the AutoCAD installation folder and the location of the `Windows` folder:

```
Sub ListFolders()
  Dim ofsObj As New FileSystemObject

  ' Get the AutoCAD install folder
  Dim oAcadFldr As Folder
  Set oAcadFldr = ofsObj.GetFolder(ofsObj. _
                  GetFile(ThisDrawing.Application.FullName).ParentFolder.Path)

  ' Get the subfolders of the AutoCAD install folder
  Dim oFldr As Folder
  Dim sFldrs As String

  For Each oFldr In oAcadFldr.SubFolders
    sFldrs = sFldrs & vbLf & "  " & oFldr.Name
  Next oFldr

  ' Output the names of the AutoCAD install subfolders
  ThisDrawing.Utility.Prompt vbLf & oAcadFldr & sFldrs

  ' Get the Windows folder
  Dim oWinFldr As Folder
  Set oWinFldr = ofsObj.GetSpecialFolder(0)

  ' Get information about the Windows folder
  ThisDrawing.Utility.Prompt vbLf & "Windows install folder: " & _
                             vbLf & "  " & oWinFldr.Path

  ' Release the object
  Set ofsObj = Nothing
End Sub
```

The following shows an example of the output from the previous example:

```
C:\Program Files\Autodesk\AutoCAD 2015
  AcWebBrowser
  AdExchange
```

```
    AdlmRes
    CER
    Content Explorer
    Drv
  Windows install folder:
    C:\Windows
```

Table 12.3 lists some of the other methods of the `FileSystemObject` object that can be helpful when working with the folders on a drive.

Table 12.3: Folder-related methods of the `FileSystemObject` object

Method	**Description**
`CopyFolder`	Copies the folder and its files from a source to a destination location
`CreateFolder`	Creates a new folder and returns a `Folder` object
`DeleteFolder`	Removes a folder
`FolderExists`	Returns `True` if the folder specified exists
`GetAbsolutePathName`	Returns a string that represents the absolute path of a file
`GetBaseName`	Returns a string that represents the base path of a file
`GetParentFolderName`	Returns a string that represents the parent path of a file
`MoveFolder`	Moves a folder and its files from a source to a destination location

Getting Information about a File

Files are the lowest item in the file hierarchy, but they are also the most important because they hold the information created by an application or the Windows operating system. The `File` object is used to get information about a file stored in a folder, whereas the `Files` collection object is used to access all the files in a folder.

The following example lists the files in the `Fonts` folder in the AutoCAD installation folder:

```
Sub ListFiles()
  Dim ofsObj As Object
  Set ofsObj = CreateObject("Scripting.FileSystemObject")

  Dim sAcadFontsFldr As String
  sAcadFontsFldr = ofsObj.GetParentFolderName( _
                        ThisDrawing.Application.FullName)

  ' Get the AutoCAD Fonts folder
  Dim oAcadFontsFldr As Object
```

```
   Set oAcadFontsFldr = ofsObj.GetFolder(sAcadFontsFldr & "\Fonts")

   ' Get the Files of the Fonts folder
   Dim oFile As Object
   Dim sFiles As String

   For Each oFile In oAcadFontsFldr.Files
     sFiles = sFiles & vbLf & "  " & oFile.Name
   Next oFile

   ' Output the names of the files
   ThisDrawing.Utility.Prompt vbLf & oAcadFontsFldr.Path & sFiles

   ' Release the object
   Set ofsObj = Nothing
 End Sub
```

The following shows an example of the output from the previous example:

```
C:\Program Files\Autodesk\AutoCAD 2015\Fonts
  @extfont2.shx
  AcadEref.shx
  aehalf.shx
  AMDTSymbols.shx
  amgdt.shx
  amgdtans.shx
  bigfont.shx
```

Table 12.4 lists some of the other methods of the `FileSystemObject` object that can be helpful when working with the files on a drive or in a folder.

Table 12.4: File-related methods of the `FileSystemObject` object

Method	Description
`CopyFile`	Copies a file from a source to a destination location
`DeleteFile`	Removes a file
`FileExists`	Returns `True` if the file specified exists
`GetExtension`	Returns a string that represents the extension of the file based on the specified path
`GetFile`	Returns the `File` object based on the specified path
`GetFileName`	Returns a string that represents the name of the file based on the specified path
`GetFileVersion`	Returns a string that represents the version of the file based on the specified path
`MoveFile`	Moves a file from a source to a destination location

Manipulating the Windows Shell

The Windows Script Host Object Model library can be helpful in manipulating some of the features found in the Windows shell. To use this library, reference it as part of your VBA project or use a late bind to the class ID `WScript.Shell`.

You can use the Windows Script Host Object Model library to perform the following tasks:

- Create a desktop shortcut
- Get and set environment variables

Here is how to create an instance of the Windows Scripting Shell object:

```
Dim ofsObj As New WshShell
```

or

```
Dim ofsObj As Object
Set ofsObj = CreateObject("WScript.Shell")
```

NOTE Don't forget to reference the Windows Script Host Object Model library in your VBA project before using the `New` keyword to create an instance of the Windows Scripting Shell object.

The following shows an example of creating a desktop shortcut:

```
Sub CreateDesktopShortcut()
  Dim oWshObj As New WshShell

  ' Get the Desktop and Documents folder locations
  Dim sDskFldr As String, sDocsFldr As String
  sDskFldr = oWshObj.SpecialFolders("Desktop")
  sDocsFldr = oWshObj.SpecialFolders("MyDocuments")

  ' Create the shortcut file
  Dim oShrtObj As WshShortcut
  Set oShrtObj = oWshObj.CreateShortcut(sDskFldr & "\My AutoCAD.lnk")

  ' Set the target and properties of the shortcut
  oShrtObj.TargetPath = ThisDrawing.Application.FullName
  oShrtObj.Arguments = "/w ""3D Modeling"""
  oShrtObj.Description = "Custom AutoCAD Desktop Shortcut"
  oShrtObj.WindowStyle = WshNormalFocus
  oShrtObj.Hotkey = "Ctrl+Alt+A"
  oShrtObj.WorkingDirectory = sDocsFldr
  oShrtObj.IconLocation = ThisDrawing.Application.FullName & ",0"
  ' Save the shortcut
  oShrtObj.Save

  ' Release the object
  Set oShrtObj = Nothing
End Sub
```

The following custom procedures demonstrate how to get and set the values of environment variables in the Windows operating system:

```
' Shows how to use expanding environment strings
' Usage: ExpEnvStr "%TEMP%\\MYDATA"
' Results of sample: "C:\\DOCUME~1\\Lee\\LOCALS~1\\Temp\\MYDATA"
Public Function ExpEnvStr(strToExpand As String) As String
  Dim oWshObj As New WshShell

  ' Expand the string and any variables in the string
  ExpEnvStr = oWshObj.ExpandEnvironmentStrings(strToExpand)

  ' Release the object
  Set oWshObj = Nothing
End Function

' Retrieve the value of the environment variable
' Usage: GetEnvStr "SYSTEM", "PROCESSOR_ARCHITECTURE"
' Results of sample: "AMD64"
' Alt Usage: GetEnvStr "SYSTEM", "USERID"
' Results of sample: "L123"
Public Function GetEnvStr(VarType As String, VarName As String) As String
  Dim oWshObj As New WshShell

  ' Get a reference to the Environment
  Dim envVars As WshEnvironment
  Set envVars = oWshObj.Environment(VarType)

  ' Get the value of the variable
  GetEnvStr = envVars(VarName)

  ' Release the object
  Set oWshObj = Nothing
End Function

' Set the value to an environment variable
' Usage: SetEnvStr "SYSTEM", "USERID", "L123"
```

```
Public Function SetEnvStr(VarType As String, VarName As String, _
                          VarValue As String) As String
  Dim oWshObj As New WshShell

  ' Get a reference to the Environment
  Dim envVars As WshEnvironment
  Set envVars = oWshObj.Environment(VarType)

  ' Set the variable to the provided value
  oWshObj.Environment(VarType) = VarValue

  ' Release the object
  Set oWshObj = Nothing
End Function
```

Using the Win32 API

Buried deep in the Windows operating system lies a powerful programming library known as the Win32 API (or the Windows API in recent years). This programming library was introduced with Windows 95 but was still present in the latest release of Windows (Windows 8.1) available when this book was written. Although the Win32 API was originally introduced with Windows 95 (the first 32-bit release of Windows), the Win32 library contains functions that were introduced with Windows 3.1 (a 16-bit release) and later 64-bit releases of Windows.

Much of the information around using the Win32 API has gone dormant over the years since the introduction of VB.NET and its rebranding as the Windows API, but there are resources on the Internet that you will find useful. You can use the following resources to learn how to implement the Win32 API in your VBA programs:

Using the Win32 API This tutorial (available at `www.vb6.us/tutorials/using-win32-api`) provides an overview for using the Win32 API.

Visual Basic Win32 API Declarations This download (available from `www.microsoft.com/en-gb/download/details.aspx?id=12427`) installs a TXT file that contains the declarations of the functions and data types in the Win32 API.

If you prefer a book to electronic references, I suggest tracking down a copy of the *Visual Basic Programmer's Guide to the Win32 API* written by Dan Appleman (Sams, 1999). Other Win32 books also were written in the late 1990s, so you should be able to find something.

Although the Win32 API can take a while to learn and understand, it does offer many great functions that can be used to implement familiar interfaces and access features in the depths of the Windows operating system from your VBA programs. You will use the `GetOpenFileName` and `GetSaveFileName` functions from the Win32 API later in the "Exercise: Reading and Writing Data" section to prompt the user to select a file or specify a filename using a dialog box. The `Ch26_ExSamples.dvb` sample file that comes with this book also shows an example of the `GetSaveFileName` function, which allows the user to specify a location and filename in which to save a file using a standard file navigation dialog box.

Reading and Writing Text Files

VBA supports the ability to read and write ASCII or Unicode text files. You can read a text file and use the contents of the file to create general notes and disclaimers for known building conditions, or even populate project information in a title block. In addition to reading the contents of a file, you can write data to a text file, which is useful for exporting a bill of materials (BOM) containing the quantity and parts in a drawing or listing the properties of nongraphical objects or system variables to help identify CAD standard violations in a drawing.

Text files can be used to define a number of file types, such as CSV, text (TXT), HTM/HTML, or even XML. The File System (`FileSystemObject`) object, which I introduced in the "Accessing the Filesystem" section earlier in this chapter, can also be used to read and write a text file.

NOTE You can use libraries registered on your workstation and ActiveX to parse the contents of an XML file or access files that can be opened from an application in the Microsoft Office suite. I explain how to parse XML files in the "Parsing Content in an XML File" section later in this chapter. How to work with applications in the Microsoft Office suite will be discussed in the "Working with Microsoft Office Applications" section, also later in this chapter.

Opening and Creating a File

Content can be read or written to an existing text file stored on a local or network drive, or a text file can be created to store new content. The `OpenTextFile` function of the `FileSystemObject` object is used to open an existing file, whereas the `CreateTextFile` function can be used to create a new file. Whether you use the `OpenTextFile` or `CreateTextFile` function, both functions return a `TextStream` object. The `TextStream` object is then used to read and write the contents of a text file in memory.

The following shows the syntax of the `OpenTextFile` function:

```
retObj = OpenTextFile(filename [, mode] [, create] [, format])
```

Its arguments are as follows:

`retObj` The *`retObj`* argument represents the `TextStream` object that is returned.

`filename` The *`filename`* argument is a string that represents the file you want to open or create when the *`create`* argument is set to `True` and the file wasn't found.

`mode` The *`mode`* argument is an optional integer value that represents how the file should be opened. By default, the file is open for read only. Table 12.5 provides a basic description of the supported integer values and the corresponding constants that can be used in their place.

TABLE 12.5: File modes available for the `OpenTextFile` statement

MODE	DESCRIPTION
1 or `ForReading`	Content of the file can only be read.
2 or `ForWriting`	Content of the file can be read or written.
8 or `ForAppending`	New content added to the file is appended to the end of the file. Content cannot be read.

create The *create* argument is an optional Boolean that determines whether the file specified by the *filename* argument should be created if it wasn't found. A value of True creates the file if it wasn't found.

format The *format* argument is an optional integer value that determines the format of the file: ASCII or Unicode. By default, the file is opened as an ASCII file. Table 12.6 provides a basic description of the supported integer values and the corresponding constants that can be used in their place.

TABLE 12.6: File formats available for the OpenTextFile statement

MODE	DESCRIPTION
-2 or TriStateUseDefault	File format is set to the system default; ASCII or Unicode.
-1 or TriStateTrue	File format is indicated as Unicode.
0 or TriStateFalse	File format is indicated as ASCII.

NOTE As an alternative to the OpenTextFile function, you can use the OpenAsTextStream function of the File object in the Microsoft Scripting Runtime library to open a text file and return a TextStream object. See the section "Getting Information about a File" earlier in this chapter to learn how to work with a File object.

The following shows the syntax of the CreateTextFile function:

```
retObj = CreateTextFile(filename [, overwrite] [, unicode])
```

Its arguments are as follows:

retObj The *retObj* argument represents the TextStream object that is returned.

filename The *filename* argument is a string that specifies the name of the file you want to create.

overwrite The *overwrite* argument is an optional Boolean that determines whether the file specified by the *filename* argument should be overwritten if it already exists. A value of True results in the existing file being overwritten by the newly created file.

unicode The *unicode* argument is an optional Boolean that determines whether the file specified by the *filename* argument should be created with the ASCII or Unicode format. A value of True results in the file being created with the Unicode format.

Here are a few examples of opening and creating a text file:

```
' Create an instance of the File System object
Dim ofsObj As New FileSystemObject

' Open the text file Data.txt for read
Dim oTxtStreamData As TextStream
Set oTxtStreamData = ofsObj.OpenTextFile("c:\Dataset\Data.txt")
```

```
' Create the text file BOM.txt, and overwrite if found
Dim oTxtStreamBOM As TextStream
Set oTxtStreamBOM = ofsObj.CreateTextFile("c:\Dataset\BOM.txt", True)
```

NOTE Trying to open a file that is read-only or that is stored in a read-only location with the write or append access mode results in a permissions error. Make sure to add proper error handling to check to see if the file is read-only. You can also use the `Attributes` property of a `File` object that is returned using the `GetFile` method of the `FileSystemObject` object to determine whether a file is read-only.

As the *`filename`* argument can specify any text file on a local or network drive, the name of the file and path you choose can affect the sustainability of your custom program. When you specify the *`filename`* argument for the `open` function, consider the following:

Static Filenames When you need to read the contents from a file, using a static filename might be ideal, but static filenames don't work well when you want to write data to a file. When creating a file, allow the user to specify a filename either using the AutoCAD Command prompt or a file-navigation dialog box, or as part of a user form.

Hard-Coded Paths I recommend against placing specific file paths in a custom program. Rather than hard-coding (typing the actual path to a particular file) a path or drive as part of a filename, use paths stored in system or environment variables related to the operating system or returned by the `File` object. For example, you can get the paths to `My Documents` (or `Documents`) or the temporary files folder with the AutoCAD system variables `mydocumentsprefix` and `tempprefix`.

If you just want to create a temporary file, you can use the `GetTempName` function of the File System object to generate a unique filename. Then, use the `CreateTextFile` function to create the `TextStream` object for that file. If you want to keep the temporary file, you can use the `MoveFile` function of the `File` object to keep the file and give it a more meaningful name.

Reading Content from a File

Once a `TextStream` object has been obtained, you can use its various read methods to step through the content. You can choose to read a specific number of characters at a time, read one line, or read all content into a string. The `Read` function allows you to specify a number of characters to read from the text stream into a string. Each successive call to the function gets the next characters in the text stream.

Reading a specific number of characters at a time until you reach the end of the text stream can be helpful in some situations, such as when you are reading a space-delimited file, but in most cases you want to read an entire line in the text stream. A line is defined as a text string that ends with a new linefeed character, which has an ASCII code value of `10`. Use the `ReadLine` function to read a line of text from a text stream. Similar to the `Read` function, each successive call to the `ReadLine` function gets the next line in the text stream.

When using the `Read` or `ReadLine` function, an `Input Past End of File` error will be generated when there are no additional characters or lines to be read from the text stream. You should check the `AtEndOfStream` property of the `TextStream` object to see if the end of the file has been reached before you continue to read the content of the text stream. If you want to read all the content from a text stream, use the ReadAll function to get a string containing all the content.

NOTE Be careful with mixing the use of the Read, ReadLine, and ReadAll functions when reading the content from the same text stream. Each time one of the read functions is called, the file pointer is moved forward a specific number of characters or a line. The file pointer specifies where in the text stream the next read function begins.

The Read function expects a single integer value that represents the number of characters to be read from the text stream, whereas the ReadLine and ReadAll functions don't accept any values. The Read, ReadLine, and ReadAll functions all return a string value.

Here are examples of reading content from a text stream with the Read, ReadLine, and ReadAll functions:

```
' Create an instance of the File System object
Dim ofsObj As New FileSystemObject

' Open the text file Data.txt for reading
Dim oTxtStreamData As TextStream
Set oTxtStreamData = ofsObj.OpenTextFile("c:\Dataset\Data.txt")

' Read the first 10 chracters of the content
ThisDrawing.Utility.Prompt vbLf & oTxtStreamData.Read(10) & vbLf

' Read the next line or remainder of the current line
ThisDrawing.Utility.Prompt vbLf & oTxtStreamData.ReadLine & vbLf

' Read the rest of the file
ThisDrawing.Utility.Prompt vbLf & oTxtStreamData.ReadAll & vbLf
```

TIP If you know there is some content in the text stream that you want to skip over, you can use the Skip and SkipLine functions. The skip functions are used to advance the file pointer a specific number of characters or to the next line in the text stream. The next read function called starts at the new location of the file pointer.

As you read content from the text stream, you can get your current location using the Columns or Line property. The Columns property lets you know how many characters from the left you have read in the current line, and the Line property lets you know which line you are on in the file. You can use the AtEndOfStream property to see if you have reached the end of the text stream, and when reading characters with the Read function, you can use the AtEndOfLine property to see if you have reached the end of the current line when reading one character at a time.

Writing Content to a File

Writing data to a text stream is similar to reading data from a text stream. You can write a string with or without using the ASCII code value of 10 (the linefeed character). The linefeed character is used to indicate the end of a line in the text stream. A line can be created with content or blank lines can be written.

The `Write` and `WriteLine` functions are used to write a string to a text stream. The difference between the two functions is that the `WriteLine` function adds the linefeed character to the end of the string and forces a new line in the text stream. The `WriteBlankLines` function is used (just as its name indicates) to write blank lines to a text stream. Both the `Write` and `WriteLine` functions expect a string value that represents the content that should be written to the file, whereas the `WriteBlankLines` function expects a single integer value that represents the number of blank lines that should be written.

Here are examples of writing content to a text stream with the `Write`, `WriteLine`, and `WriteBlankLine` functions:

```
' Create an instance of the File System object
Dim ofsObj As New FileSystemObject

' Create a new text file named Data.txt
Dim oTxtStreamData As TextStream
Set oTxtStreamData = ofsObj.CreateTextFile("c:\Dataset\Data.txt")

' Write a content to the file without adding the new linefeed character
oTxtStreamData.Write "BLOCK" & vbTab & "TAG" & vbTab

' Append to the current line and add the new linefeed character
oTxtStreamData.WriteLine "PART" & vbTab & "DESCRIPTION"

' Write a blank line
oTxtStreamData.WriteBlankLine 1
```

Closing a File

Each text stream that represents an opened or new text file created with the `OpenTextFile` or `CreateTextFile` function must be closed using the `Close` function. Closing the text stream saves the changes to the file and removes the text stream from memory to free up system resources. Text streams that aren't closed might remain open in memory, and that memory is not available to other applications until AutoCAD is closed or the VBA project is unloaded.

Typically, when the procedure ends and the `Close` function hasn't been called, the text stream is closed automatically, but I wouldn't rely on this approach. It is always good practice to close the text stream when it is no longer needed and not to rely on the system. The `Close` function doesn't accept any values.

Here is an example of the `Close` function:

```
' Close the text stream
oTxtStreamData.Close
```

Parsing Content in an XML File

XML files were once used primarily for working with data on the Internet, but today they are used by many applications and are a way to transfer information between different applications.

Although text files can be nice for generating basic reports that can be printed or for storing content, they aren't really designed as a data repository or for working with large amounts of data. XML files are readable both by humans (when not compiled) and by applications without specialized software, but unlike a text file, they can help to enforce a data structure. You can create an XML file using the functions I mentioned in the "Reading and Writing Text Files" section earlier, but reading an XML file can be simplified using the Microsoft XML library.

The following is an example of an XML file that contains the contents of three general notes that could be found in a drawing or on a title sheet of a drawing set:

```
<?xml version="1.0"?>
<catalog>
   <note id="n001">
      <updated_date>2014-09-01</updated_date>
      <name>ADA Turn Radius</name>
      <description>ADA REQUIRES A MINIMUM TURN RADIUS OF 60" (1525MM) FOR
WHEELCHAIRS.</description>
   </note>
   <note id="n002">
      <updated_date>2014-05-14</updated_date>
      <name>Dimension Reference</name>
      <description>ALL DIMENSIONS INDICATED ARE FOR REFERENCE AND
COORDINATION PURPOSES ONLY.</description>
   </note>
   <note id="n003">
      <updated_date>2014-04-14</updated_date>
      <name>Electrical Contractor</name>
      <description>ALL ELECTRICAL WORK SHALL BE COMPLETED WITH NEW
MATERIALS AND CONDUCTED BY THE ELECTRICAL CONTRACTOR UNLESS
NOTED.</description>
   </note>
</catalog>
```

NOTE Spaces in an attribute value are interpreted as literal spaces in XML. In the previous example, the `description` attributes aren't indented for this reason. If I had indented the values of the `description` attributes, the spaces or tab characters would become part of the value when the XML file is parsed.

A note is represented by the `Note` element and is a child of the root element `Catalog`. Each `Note` element has an attribute named `id`, which is used to uniquely identify the note in the XML file and three children nodes that describe the `Note` element. The three children nodes are named `updated_date`, `name`, and `description`.

The following shows an example VBA procedure that reads each `Note` element in an XML file, and then outputs the values of the attribute and children nodes of the `Note` element to the AutoCAD Command prompt:

```
Sub ReadXML()
  ' Specify the XML file to open
  Dim sXMLFile As String
  sXMLFile = ThisDrawing.GetVariable("MyDocumentsPrefix") & _
```

```
                "\MyCustomFiles\ch12_notes.xml"

  ' Open the XML file
  Dim oXMLDoc As New MSXML2.DOMDocument
  oXMLDoc.async = False
  oXMLDoc.validateOnParse = False
  oXMLDoc.Load sXMLFile

  ' Get the root node of the XML file
  ' In the ch12_notes.xml file, the root node is Catalog
  Dim oNodeCatalog As MSXML2.IXMLDOMNode
  Set oNodeCatalog = oXMLDoc.documentElement

  Dim oNote As MSXML2.IXMLDOMNode
  Dim oNoteChild As MSXML2.IXMLDOMNode

  ' Get the nodes under the catalog node
  ' In the ch12_notes.xml file, the children nodes are Note
  For Each oNote In oNodeCatalog.ChildNodes

    ' Get and output the first attribute of the Note
    ' In the ch12_notes.xml file, the first attribute is ID
    ThisDrawing.Utility.Prompt vbLf & "ID: " & _
                           oNote.Attributes(0).Text & vbLf


    ' Get and output the children nodes of the Note node
    ' In the ch12_notes.xml file, the children nodes are updated_date,
    ' name, and description.
    For Each oNoteChild In oNote.ChildNodes
      ThisDrawing.Utility.Prompt vbLf & "  " & UCase(oNoteChild.BaseName) & _
                             ": " & oNoteChild.Text & vbLf
    Next oNoteChild
  Next oNote

  ThisDrawing.Utility.Prompt vbLf

  ' Release the object
  Set oXMLDoc = Nothing
End Sub
```

The following shows what the output looks like in the AutoCAD Command Line history after executing the previous example:

```
ID: n001
  UPDATED_DATE: 2014-09-01
```

```
    NAME: ADA Turn Radius
    DESCRIPTION: ADA REQUIRES A MINIMUM TURN RADIUS OF 60" (1525MM)
                 FOR WHEELCHAIRS.
  ID: n002
    UPDATED_DATE: 2014-05-14
    NAME: Dimension Reference
    DESCRIPTION: ALL DIMENSIONS INDICATED ARE FOR REFERENCE AND COORDINATION
                 PURPOSES ONLY.
  ID: n003
    UPDATED_DATE: 2014-04-14
    NAME: Electrical Contractor
    DESCRIPTION: ALL ELECTRICAL WORK SHALL BE COMPLETED WITH NEW MATERIALS AND
                 CONDUCTED BY THE ELECTRICAL CONTRACTOR UNLESS NOTED.
```

Before using the previous example, be sure to reference the Microsoft XML Library. There might be several versions of the Microsoft XML Library registered on your workstation; reference the latest version on your workstation. If you reference the Microsoft XML 6.0v library, you will need to change the code statement `Dim oXMLDoc As New MSXML2.DOMDocument` to **`Dim oXMLDoc As New MSXML2.DOMDocument60`**.

For more information on parsing an XML file and using the Microsoft XML library, I recommend starting with the "A Beginner's Guide to the XML DOM" topic on the Microsoft Developer Network site (`http://msdn.microsoft.com/en-us/library/aa468547.aspx`). You can also use your Internet browser to locate additional resources on working with XML files, or buy a book from your favorite retailer.

Working with Microsoft Office Applications

The Microsoft Office suite installs a number of programming libraries that allow you to manipulate the contents in the files that the applications of the suite can create and access the application's settings. For example, you can create an instance of the Microsoft Word application, and then create or open a DOC or DOCX file. Once a document has been created or opened, you can step through and manipulate the content of the document or print the document to an available system printer.

The following libraries allow you to create an instance of an application and work with the files that can be created or modified using the Microsoft Office suite:

- Microsoft Word *<version>*.0 Object Library (`msword.olb`)
- Microsoft Excel *<version>*.0 Object Library (`excel.exe`)
- Microsoft Outlook *<version>*.0 Object Library (`msoutl.olb`)
- Microsoft PowerPoint *<version>*.0 Object Library (`msppt.olb`)
- Microsoft Access *<version>*.0 Object Library (`msacc.olb`)
- Microsoft Publisher *<version>*.0 Object Library (`mspub.tlb`)

Referencing the Correct Version of the Microsoft Office Library

The text `<version>` in the previous list represents the version of Microsoft Office installed on your workstation. For Microsoft Office 2013, `<version>` would be a value of 15. Here is a listing of the version numbers from recent releases of the Microsoft Office suite:

- Microsoft Office 2013 - 15
- Microsoft Office 2010 - 14
- Microsoft Office 2007 - 12
- Microsoft Office 2003 - 11
- Microsoft Office 2000 - 10

If you are not sure which version of Microsoft Office suite might be installed on a workstation, you can use late binding, as I explained earlier in this chapter, to create an instance of one of the applications in the Microsoft Office suite. Using early binding does make development and debugging easier, though, but you can switch to late binding later, after you have debugged your program and are ready to deploy it.

The following shows how you can create an instance of the Microsoft Word application and create a new blank document using late binding:

```
Sub CreateWordApp ()
  On Error Resume Next

  Dim oWordApp As Object
  Set oWordApp = CreateObject("Word.Application")

  ' Create a new drawing
  oWordApp.Documents.Add

  ' Make the application visible
  oWordApp.Visible = True
End Sub
```

You can use the following code to create a reference to the Microsoft Excel object using late binding:

```
Dim oExcelApp As Object
Set oExcelApp = CreateObject("Excel.Application")
```

The Microsoft Access Object library can be used to manipulate information in a database file, but you can also access the tables and queries of an Access database file without having Access installed. You can use the following programming libraries when you want to work with an Access database file without having Access installed:

- Microsoft ActiveX Data Objects 2.8 Library (`msado28.tlb`)
- Microsoft DAO 3.6 Object Library (`dao360.dll`)

The object libraries for each of the applications in the Microsoft Office suite are very extensive, and it would take an entire book to do them justice. If you want to learn more about creating VBA programs that interact with the applications in the Microsoft Office suite, I recommend checking out *Mastering VBA for Microsoft Office 2013,* by Richard Mansfield (John Wiley & Sons, 2013). You can also use your favorite Internet browser and search engine to access resources online for learning to use VBA with the applications in the Microsoft Office suite.

I have created several examples for this book that demonstrate how you can connect information from an AutoCAD drawing to Microsoft Word and Excel. You can find these custom procedures in the `ch12_mswin_office.dvb` file that can be downloaded from `www.sybex.com/go/autocadcustomization`.

The DVB file contains the following custom procedures:

`createmsworddoc` The `createmsworddoc` procedure creates a new Word document and saves it with the name `ch12_apc_word_sample.doc` to the `MyCustomFiles` folder. The new Word document file is populated with information about some of the nongraphical objects in the current drawing.

`printmsworddoc` The `printmsworddoc` procedure opens the `ch12_apc_word_sample.doc` file that was created with the `createmsworddoc` procedure and placed in the `MyCustomFiles` folder. The Word document file is then printed using the default system printer.

`extractattributestoexcel` The `extractattributestoexcel` function creates a new spreadsheet file named `ch12_attributes.xls` in the `MyCustomFiles` folder. The handle, tag, and text string for each attribute in the block references of the current drawing are extracted to columns and rows in the spreadsheet. Open the `ch12_building_plan.dwg` file in AutoCAD before executing the function.

`updateattributesfromexcel` The `updateattributesfromexcel` function reads the information from the spreadsheet file named `ch12_attributes.xls` in the `MyCustomFiles` folder. The extracted handle in the spreadsheet is used to get the attribute reference and then update the tag and text string value that are present in the spreadsheet. Since handles are unique by drawing, you must open the original drawing that the attributes were extracted from. Make changes to the third column in the spreadsheet file, such as C2436 to CC2436, before opening the `ch12_building_plan.dwg` file in AutoCAD before executing the function.

Along with the custom procedures that demonstrate how to work with Microsoft Word and Excel files, there are a few functions that demonstrate how to connect to an Access database (MDB) file using Database Access Object (DAO) and ActiveX Data Object (ADO). The database library you use depends on which release of Windows you are using or the Microsoft Office version installed. You can find these custom procedures in the `ch12_mswin_office.dvb` file that can be downloaded from `www.sybex.com/go/autocadcustomization`.

The DVB file contains the following custom functions:

`accessdatabasedao` The `accessdatabasedao` procedure makes a connection to the Access database `ch12_employees.mdb` located in the `MyCustomFiles` folder. Once a connection to the database is made, the records in the Employees table are read and modified. Use this function when working with Access 2007 and earlier.

`accessdatabaseado` The `accessdatabaseado` function makes a connection to the Access database `ch12_employees.mdb` located in the `MyCustomFiles` folder. Once a connection to the database is made, the records in the Employees table are read and modified. Use this function when working with Access 2007 and later.

Exercise: Reading and Writing Data

In this section, you will create a new VBA project and modify the `FurnTools` project to introduce several new procedures that read data from and write data to text files. The first main procedure reads information from a text file and uses that information to add new layers to a drawing.

The second main procedure is an extension of the BOM program in the FurnTools project that you created in Chapter 7. Instead of adding a table grid to a drawing, this new procedure exports the BOM content to a comma-delimited (CSV) file that can be imported into a database or spreadsheet program.

The key concepts I cover in this exercise are as follows:

Referencing a Programming Library Programming libraries allow you to access additional features and utilities that are not part of the core VBA programming language.

Locating and Prompting for External Files Files that a custom program might rely on can be located in the AutoCAD support file search paths before they are used, or the user can be prompted for a filename and location.

Opening, Reading, and Writing Data in External Files Data files can be opened before the data in the file can be read or data can be written to. Once file access is no longer needed, the file should be closed.

NOTE The steps in this exercise depend on the completion of the steps in the "Exercise: Creating and Querying Blocks" section of Chapter 7. If you didn't complete the steps, do so now or start with the ch12_furntools.dvb sample file available for download from www.sybex.com/go/autocadcustomization. This sample file should be placed in the MyCustomFiles folder under the Documents (or My Documents) folder, or the location you are using to store the DVB files. After the files are saved to the location you are using to store DVB files, remove ch12_ from the filename. You will also need the files ch12_building_plan.dwg, ch12_layers.dat, ch12_clsDialogs.cls, and ch12_clsUtilities.cls for this exercise.

Creating Layers Based on Data Stored in a Text File

Often you start a drawing from a drawing template that contains a default set of layers, but any layers that are not used can accidentally be removed with the purge or -purge command. To restore the missing layers, you could create a drawing that contains your default layers and insert it into your drawing. As an alternative on Windows, you could restore the layers using the Content Explorer™ palette or the DesignCenter ™ palette. An additional approach to restoring layers (or other named standards) is through the use of external data files and VBA.

The ch12_layers.dat file (part of the sample files supplied with this book) contains information that can be used to create layers in a drawing. The DAT file is tab-delimited and contains three pieces of information about each layer—layer name, color, and linetype:

```
; AutoCAD Customization Platform
; Layer data file used to setup layers
Plan_Cabinets       6     Continuous
Plan_Dimensions     3     Continuous
```

You will use the createlayer function defined in the ch12_clsUtilities.cls file (exported as part of the exercise in Chapter 4, "Creating and Modifying Drawing Objects") to create the new layers. In addition to using the ch12_clsUtilities.cls file, you will use a function defined in the ch12_clsDialogs.cls file to let the user select a file on their local or network drive. The functions in the ch12_clsDialogs.cls file use the Win32 API.

In these steps, you'll create a new VBA project named `LayerTools` with a custom procedure named `LoadLayers` that will read and use the data stored in the file named `layers.dat` to create new layers in a drawing:

1. Create a new VBA project with the name `LayerTools`. Make sure to also change the default project name (`ACADProject`) to `LayerTools` in the VBA Editor.
2. In the VBA Editor, in the Project Explorer, right-click the new project and choose Import File.
3. When the Import File dialog box opens, browse to and select the `ch12_clsUtilities.cls` file in the `MyCustomFiles` folder. Click Open.

   The `ch12_clsUtilities.cls` file contains the utility procedures that you created as part of the `DrawPlate` and the `FurnTools` projects.
4. Import the `ch12_clsDialogs.cls` file into the new project from the `MyCustomFiles` folder.
5. In the Project Explorer, right-click the new project and choose Insert ➤ Module. Change the name of the new module to **`basLayerTools`**.
6. In the text editor area of the `basLayerTools` component, type the following; the comments are here for your information and don't need to be typed:

```
Private myUtilities As New clsUtilities
Private myDialogs As New clsDialogs

' Creates layers based on the values in the ch12_layers.dat file.
Sub LoadLayers()

  ' Select the layer data file, if not found
  ' in the AutoCAD support file search paths
  Dim sLayerDataFile As String
  sLayerDataFile = myUtilities.FindFile("ch12_layers.dat")

  ' If the file wasn't found then prompt the user
  If sLayerDataFile = "" Then
    ' Check to see if a previous file name is in the Windows Registry
    Dim sLastLoc As String
    sLastLoc = GetSetting("Sybex", "ACP", "LastLayerDataFile")

    ' Make sure the value in the Windows Registry is valid
    If sLastLoc <> "" Then
       sLastLoc = myUtilities.FindFile(sLastLoc)
    End If

    ' If the file is not valid, prompt for the file
    If sLastLoc = "" Then
```

```
        sLayerDataFile = myDialogs.SelectOpenFile( _
              "Select Layer Data File", "", "ch12_layers.dat", _
              "Data File (*.dat)" & Chr(0) & "*.dat")
      Else
        sLayerDataFile = sLastLoc
      End If

      ' Store the last location to the Windows Registry
      If sLayerDataFile <> "" Then
        SaveSetting "Sybex", "ACP", "LastLayerDataFile", sLayerDataFile
      End If
    End If

    ' Check to see if the user selected a file
    If sLayerDataFile <> "" Then

      On Error Resume Next

      ' Create a new instance of the File System object
      Dim ofsObj As New FileSystemObject

      ' Check to see if the value passed was a file or not
      Dim oFile As File
      Set oFile = ofsObj.GetFile(sLayerDataFile)

      Dim oTextStream As TextStream
      Set oTextStream = oFile.OpenAsTextStream(ForReading)

      ' Skip the first two lines in the text stream as they are comments
      oTextStream.SkipLine
      oTextStream.SkipLine

      ' Read the text from the stream
      Dim vLineData As Variant
      While Not oTextStream.AtEndOfStream
        ' Split the line into elements based on tab characters
        vLineData = Split(oTextStream.ReadLine, vbTab)

        ' Create the new layer
        Dim oLayer As AcadLayer
        Set oLayer = myUtilities.CreateLayer(CStr(vLineData(0)), _
                                    CInt(vLineData(1)))
        ' Assign the linetype to the layer
        oLayer.Linetype = vLineData(2)
      Wend
    End If
  End Sub
```

7. Click File ➢ Save.

The procedure can't be executed yet, because you need to define a new utility procedure named FindFile in the imported clsUtilities component. You will do so in the next section.

Searching for a File in the AutoCAD Support Paths

The files that a custom program relies on should be found within the AutoCAD support search file paths or in a location that the custom program can always find, such as the ProgramData folder. The AutoCAD Object library doesn't have a native function that can be used to check to see if a file is found in the AutoCAD support file search paths, but you can use the SupportPaths property of the AcadPreferencesFiles object to get a list of the support paths and then use the FileExists function of the File System object to check for the existence of the file.

In these steps, you'll add the FindFile function to the imported version of the ch12_clsUtilities.cls file:

1. On the menu bar, click Tools ➢ References.
2. When the References dialog box opens, scroll to the Microsoft Scripting Runtime library and click the check box next to it. The library should now be checked.
3. Click OK.
4. In the Project Explorer, double-click the clsUtilities component.
5. In the text editor area of the clsUtilities component, scroll to the bottom of the last procedure and press Enter a few times. Then, type the following. (The comments are here for your information and don't need to be typed.)

```
' Returns a string containing the full path to the file if it is found
' in the AutoCAD support file search path.
' Function expects a string representing the name of
' the file you want to find.
Function FindFile(sFileName As String) As String
  On Error Resume Next

  ' Create a new instance of the File System object
  Dim ofsObj As FileSystemObject
  Set ofsObj = CreateObject("Scripting.FileSystemObject")

  ' Check to see if the value passed was a file or not
  Dim oFile As File
  Set oFile = ofsObj.GetFile(sFileName)

  If Err Then
    Err.Clear

    Dim sSupportPaths As String, sPath As Variant
```

```
      ' Get the Support File paths
      sSupportPaths = ThisDrawing.Application.Preferences.Files.SupportPath

      ' Split the support paths delimited by a semicolon
      For Each sPath In Split(sSupportPaths, ";")

        ' Check to see if the file exists in the path
        If ofsObj.FileExists(CStr(sPath) & "\" & sFileName) Then

          ' Return the full path to the file
          FindFile = CStr(sPath) & "\" & sFileName

          ' Exit the For statement
          Exit Function
        End If
      Next
    Else
      ' Return the file name as it is a full path
      FindFile = sFileName

      Exit Function
    End If

    FindFile = ""
  End Function
```

6. Click File ➤ Save.
7. In the Project Explorer, right-click the `clsUtilities` component and choose Export File.
8. When the Export File dialog box opens, browse to the `MyCustomFiles` folder and click Save. The name in the File Name text box should be `clsUtilities.cls`; if it isn't, enter **`clsUtilities`** and then click Save. If you have a previous version of this file, be sure you want to overwrite that file.

   Now when you import the `clsUtilities.cls` file into a future project, the `FindFile` function will be available in that project. If you need this function in an existing project, you will need to remove the existing `clsUtilities` component and import this file.

Adding Layers to a Drawing with the *LoadLayers* Procedure

The `LayerTools.dvb` file now contains the `LoadLayers` procedure, which uses the `createlayer` function defined in the `clsUtilities` component and the `SelectOpenFile` function defined in the `clsDialogs` component.

NOTE The following steps require a data file named `ch12_layers.dat`. If you didn't download the sample files previously, download them now from `www.sybex.com/go/autocadcustomization`. Place these sample files in the `MyCustomFiles` folder under the `Documents` (or `My Documents`) folder.

The following steps explain how to use the `LoadLayers` procedure in the `LayerTools.dvb` file:

1. Create a new drawing.
2. At the Command prompt, type **`vbarun`** and press Enter.
3. When the Macros dialog box opens, select the `LayerTools.dvb!basLayerTools.LoadLayers` macro from the list and click Run.
4. If the Select Layer Data File dialog box opens, browse to and select the `ch12_layers.dat` file, which you should have copied to the `MyCustomFiles` folder under the `Documents` (or `My Documents`) folder. Click Open.

   The Select Layer Data File dialog box is only displayed if the VBA program couldn't locate the `ch12_layers.dat` file in the AutoCAD support search file paths.
5. On the ribbon, click Home tab ➢ Layers panel ➢ Layer Properties.
6. Open the `ch12_layers.dat` file in Notepad.
7. Click at the end of the last line; the line starts with `Plan_Walls`.
8. In the text editor area, type the following. (Press the Tab key rather than typing the text <tab>.)

   ```
   Title_Block<tab>7<tab>Continuous
   ```
9. Save the changes to the `ch12_layers.dat` file.
10. In AutoCAD, execute the `LoadLayers` macro again with the `vbarun` command; notice that the layer `Title_Block` is now added to the drawing.

Writing Bill of Materials to an External File

In Chapter 7 you created a VBA project that allowed you to extract the attributes of a block and then quantify the results before creating the BOM in the drawing. Here you will create a procedure named `FurnBOMExport` that allows you to export the BOM data generated with the `ExtAttsFurnBOM` procedure output to a comma-delimited text (CSV) file instead of adding it to the drawing as a table grid as you did with the `FurnBOM` procedure. You could then use the CSV file and import the BOM into a costing or ordering system.

Using these steps, you will create the custom procedure named `FurnBOMExport` in the `FurnTools.dvb` file, which you created in Chapter 7.

1. Load the `FurnTools.dvb` file into the AutoCAD drawing environment and display the VBA Editor.
2. In the VBA Editor, in the Project Explorer, right-click the `FurnTools` project and choose Import File.
3. When the Import File dialog box opens, browse to the `MyCustomFiles` folder and select the `ch12_clsDialogs.cls` file (or the `clsDialogs.cls` file you exported in the previous section). Click Open.

4. On the menu bar, click Tools ➢ References.
5. When the References dialog box opens, scroll to the Microsoft Scripting Runtime library in the list and click the check box next to it. The library should now be checked.
6. Click OK.
7. In the Project Explorer, double-click the `basFurnTools` component to edit the code in the code editor window.
8. In the code editor window, scroll to the top of the window and after the code statement `Private myUtilities As New clsUtilities` press Enter.
9. Type the following:

```
Private myDialogs As New clsDialogs
```

10. Scroll to the bottom of the code editor window, click after the End statement of the last procedure, and press Enter twice.
11. In the code editor window, type the following:

```
' Exports the extracted attribute information to an external data file
Sub FurnBOMExport()
  On Error Resume Next

  ' Get the blocks to extract
  Dim oSSFurn As AcadSelectionSet
  Set oSSFurn = ThisDrawing.SelectionSets.Add("SSFurn")

  ' If an error is generated, selection set already exists
  If Err Then
    Err.Clear

    Set oSSFurn = ThisDrawing.SelectionSets("SSFurn")
  End If

  ' Define the selection set filter to select only blocks
  Dim nDXFCodes(0) As Integer, nValue(0) As Variant
  nDXFCodes(0) = 0
  nValue(0) = "INSERT"

  Dim vDXFCodes As Variant, vValues As Variant
  vDXFCodes = nDXFCodes
  vValues = nValue

  ' Allow the user to select objects in the drawing
  oSSFurn.SelectOnScreen vDXFCodes, vValues

  ' Proceed if oSSFurn is greater than 0
  If oSSFurn.Count > 0 Then
    Dim sBOMDataFile As String
    sBOMDataFile = myDialogs.SelectSaveFile("Create CSV File", "", "", _
                        "Comma-delimited File (*.csv)" & Chr(0) & "*.csv")

    ' Check to see if the user selected a file
    If sBOMDataFile <> "" Then
      ' Extract and quantify the parts in the drawing
      Dim vAttList As Variant
      vAttList = ExtAttsFurnBOM(oSSFurn)

      On Error Resume Next

      ' Create a new instance of the File System object
      Dim ofsObj As New FileSystemObject

      ' Check for a file extension, if not present append one
      If ofsObj.GetExtensionName(sBOMDataFile) = "" Then
        sBOMDataFile = sBOMDataFile & ".csv"
      End If

      ' Create a new text file based on the selected file
      Dim oTextStream As TextStream
      Set oTextStream = ofsObj.CreateTextFile(sBOMDataFile)

      ' Write the header line to the file
      oTextStream.WriteLine "QTY,LABELS,PARTS"

      ' Step through the list
      Dim vItem As Variant
      For Each vItem In vAttList
        vItem = Split(vItem, vbTab)
        oTextStream.WriteLine CStr(vItem(0)) & "," & _
                              CStr(vItem(1)) & "," & _
                              CStr(vItem(2))
      Next vItem

      ' Close the file
      oTextStream.Close
    End If

    ' Remove the selection set
    oSSFurn.Delete
  End If
End Sub
```

12. Click File ➢ Save.

Using the *FurnBOMExport* Procedure

The following steps explain how to use the `FurnBOMExport` procedure that is defined in the `FurnTools.dvb` file. Before starting the steps, download the sample `ch12_building_plan.dwg` from `www.sybex.com/go/autocadcustomization`. Place the sample file in the `MyCustomFiles` folder under the `Documents` (or `My Documents`) folder.

1. Open `ch12_building_plan.dwg`.
2. At the Command prompt, type **`vbarun`** and press Enter.
3. When the Macros dialog box opens, select the `FurnTools.dvb!basFurnTools .FurnBOMExport` macro from the list and click Run.
4. At the `Select objects:` prompt, select the furniture blocks in the plan and press Enter.
5. When the Create CSV File dialog box opens, browse to the `MyCustomFiles` folder or the folder in which you want to create the CSV file.
6. In the File Name text box, type **`furnbom`** and click Save.
7. Open Windows Explorer or File Explorer, and browse to the location of the `furnbom.csv` file.
8. Open the file in Notepad or even an application like Microsoft Excel.

   Figure 12.3 shows the results of opening the `furnbom.csv` file in Excel.

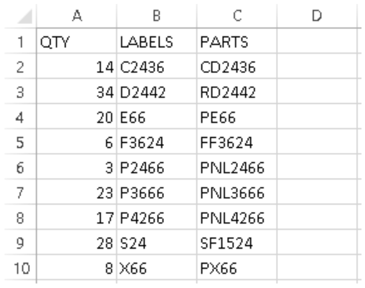
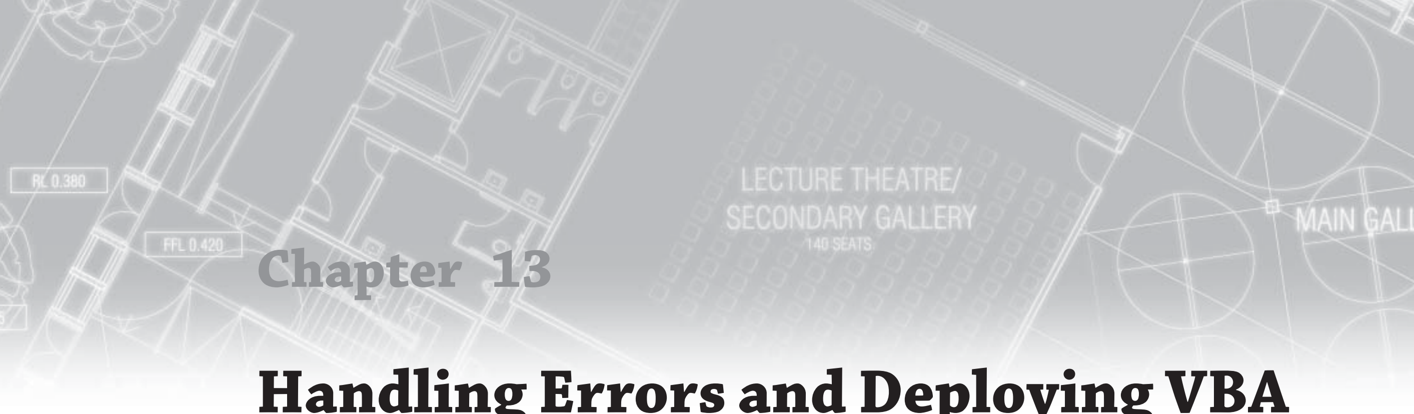


FIGURE 12.3 BOM content in Excel

	A	B	C	D
1	QTY	LABELS	PARTS	
2	14	C2436	CD2436	
3	34	D2442	RD2442	
4	20	E66	PE66	
5	6	F3624	FF3624	
6	3	P2466	PNL2466	
7	23	P3666	PNL3666	
8	17	P4266	PNL4266	
9	28	S24	SF1524	
10	8	X66	PX66	



Chapter 13

Handling Errors and Deploying VBA Projects

What separates a good programmer from a great programmer is often the ability to implement error handling that catches an error and exits the program cleanly, thus avoiding system crashes and unwanted changes to a drawing.

The ability to predict where something might go wrong in your program can help you locate potential problems—errors or bugs, as programmers commonly refer to them. If you hang around any programmers, you might have heard the term *debugging*; it is the industry-standard term used for the process of locating and resolving problems in a program. Conditional statements can be used to identify and work around potential problems by validating values and data types used in a program.

Once you have tested a program for potential problems and handled the errors generated, you are ready to deploy the program for use.

Catching and Identifying Errors

The VBA programming language supports two statements that are designed to assist in handling errors. The `On Error` and `Resume` statements allow you to execute your code and specify the code statements that should be executed if an error occurs. Along with these two statements, the `Err` object can be used to get information about the error that was generated. You can use this information for evaluation and error handling or, when necessary, to pass an error forward from a custom function for the calling program to evaluate and handle.

For example, you might have a procedure that works with a text file and accepts a string that contains the file it should work with. If the procedure is passed a string but it doesn't represent a proper filename, your procedure should handle the error but also raise the error so that the calling procedure can use the error handling of the VBA programming language to continue.

Recovering and Altering Execution after an Error

There is nothing more frustrating to end users than a program that misbehaves or terminates without warning and provides them with no information about what went wrong. No program is ever perfect, but you should make every attempt to ensure the users of your custom program the best possible experience by adding proper error handling.

The `On Error` statement is what you will be using to catch and handle any errors that occur during the execution of a custom procedure. There are two variants of the `On Error` statement:

- `On Error Resume Next` instructs VBA to ignore any error it encounters and continue execution with the next code statement.
- `On Error GoTo <Line Number or Label>` instructs VBA to move execution to a specific label or line number in the current procedure when an error occurs.

The `On Error Resume Next` statement is the most frequently used variant of the `On Error` statement, as you typically want to execute the next code statement in a procedure to try to recover from the error.

The following is an example of a procedure that tries to get a layer named `ACP-Doors`:

```
Private Function GetLayerACP_Doors() As AcadLayer
  Set GetLayer = ThisDrawing.Layers("ACP-Doors")
End Function
```

In this procedure, if the layer doesn't exist, the function suddenly terminates and VBA displays an error message. This isn't ideal because the VBA program doesn't perform the expected task and the default error message displayed isn't helpful (see Figure 13.1).

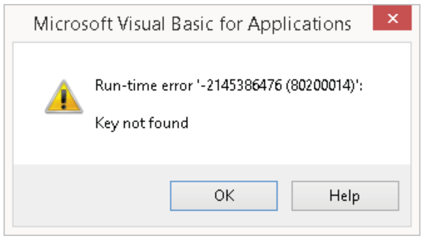


FIGURE 13.1 The layer wasn't found in the drawing.

```
Microsoft Visual Basic for Applications

Run-time error '-2145386476 (80200014)':

Key not found
```

If the `On Error Resume Next` statement is added to a procedure and inserted before a code statement that could generate an error, no error message is displayed. The procedure that calls the `GetLayerACP_Doors` procedure would need to check the value that is returned for a valid `AcadLayer` object or a value of `Nothing`. Here is what the function would look like with an `On Error Resume Next` statement added:

```
Private Function GetLayerACP_Doors() As AcadLayer
  On Error Resume Next
  Set GetLayer = ThisDrawing.Layers("ACP-Doors")
End Function
```

The `On Error GoTo` *`<Line Number or Label>`* statement can be helpful when you don't want execution to continue to the next code statement in a procedure after an error occurs. The statement `On Error GoTo` *`<Line Number>`* is used to move execution to a code statement within the procedure that starts with a specified line number. VBA does not automatically assign line numbers, and not all code statements need to have a line number. To use this statement, you must manually enter the line number in front of each code statement that should have a line number. The lines don't need to be numbered sequentially, but the numbers specified must be greater than 0; a line number of 0 indicates that error handling should be disabled in the procedure.

The following shows an example of the `On Error GoTo <Line Number>` statement. An error is generated in the procedure when the `Add` method of the `AcadLayers` collection object is executed as a result of the < and > characters, which are not valid, in the layer name. When the error occurs, execution is moved to the code statement with line number 9 to its left.

```
Public Sub CreateLayer()
1  On Error GoTo 9

   Dim sName As String
3  sName = "<Bad Name>"

5  ThisDrawing.Layers.Add sName

   MsgBox "Layer " & sName & " was added."

   Exit Sub

9  Err.Clear
   ThisDrawing.Utility.Prompt _
       vbLf + "Error: Layer couldn't be created."

   GoTo 1
End Sub
```

NOTE In the previous example, the `GoTo` statement is used without `On Error` to move execution back to the code statement numbered 1 in the procedure. The `GoTo` statement can also be used to move execution to a label in a procedure. I discuss how to use labels next.

As an alternative to using line numbers to specify a location within a procedure, you can use labels that have more meaningful names than 1, 5, 9, and so on. A label starts on a new line in a procedure and ends with a colon. For example, you could create labels with the names `LayerNotFound`, `BadName`, and `ErrHandler`. `LayerNotFound` might contain code statements that should be executed if a layer wasn't found, `BadName` might contain code statements to handle an invalid name passed to the `Add` method of the `AcadLayers` collection, and `ErrHandler` might be a generic label to handle all other errors that are generated. Although multiple labels can be used in a procedure, only one can be used with the `On Error GoTo <Label>` statement in a procedure.

The following shows an example of the `On Error GoTo <Label>` statement. In this case, an error could be generated by the code statement `Set oLayer = ThisDrawing.Layers(sName)` as a result of the layer `ACP-Door` not being found in the drawing. When the error occurs, execution is moved to the label `LayerNotFound` in the procedure where the layer is added to the drawing. The newly added layer is assigned to the `oLayer` variable. Once the layer is added, execution is returned to the `oLayer.color = acBlue` statement using the `Resume Next` statement.

```
Public Sub GetLayer()
  On Error Resume Next

  Dim sName As String
  sName = "ACP-Door"
  On Error GoTo LayerNotFound

  Dim oLayer As AcadLayer
  Set oLayer = ThisDrawing.Layers(sName)
  oLayer.color = acBlue

  Exit Sub

LayerNotFound:
  Set oLayer = ThisDrawing.Layers.Add(sName)

  Resume Next
End Sub
```

NOTE In the previous example, the `Resume Next` statement is used without `On Error` to move execution to the code statement immediately after the code statement that originally generated the error. `Resume` can also be used to move execution back to the code statement that originally caused the error. If you want execution to resume at a specific code statement, you can use the statement `Resume` *`<Line Number or Label>`*.

Getting Information About the Recent Error

The `Err` object is part of the VBA programming language, and it holds information about the most recent error that was generated during the execution of a procedure. If you want to learn more about the `Err` object, you can look up `ErrObject` (not `Err`) in the VBA Help system or the Object Browser of the VBA Editor. You can use a combination of the `On Error` and `If` statements to determine whether an error occurred. The value of the `Err` object's `Number` property is 0 by default, and is changed to a nonzero number when an error occurs.

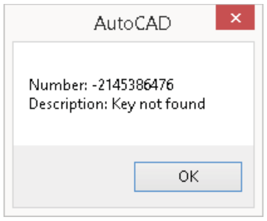
The value of the `Number` property isn't always very helpful or decipherable by us humans. For example, the value of 5 could mean "Invalid procedure call or argument" for one software vendor's object library but have a different meaning for another library from a different vendor. You will want to contact the software vendor or use your favorite Internet search engine to see if you can obtain a listing of error values and their meaning. For humans, the `Err` object also has a `Description` property. The `Description` property of the `Err` object provides a basic explanation of the error that occurred, but even this can be a bit cryptic if you don't understand the terminology used.

The following example first tries to get the layer 10101 in the `AcadLayers` collection object of the current drawing. If the layer exists, no error is generated and nothing happens. If the layer doesn't exist, an error is returned and the code statements in the `If` statement are executed.

```
Private Sub CreateLayer10101()
  On Error Resume Next

  Dim obj As AcadLayer
  Set obj = ThisDrawing.Layers("10101")

  If Err.Number <> 0 Then
```

```
        MsgBox "Number: " & CStr(Err.Number) & vbLf & _
               "Description: " & Err.Description

        ThisDrawing.Layers.Add "10101"
      End If
    End Sub
```

The first time the procedure is executed, an error occurs and a message, shown in Figure 13.2, is displayed, indicating that the key (the layer in this case) wasn't found. As part of the `If` statement, the layer is added to the drawing, and executing the procedure a second time results in no error or message being displayed because the layer already exists.



Number: -2145386476
Description: Key not found

FIGURE 13.2 Custom message containing information about a recent error

Table 13.1 lists the other properties of the `Err` object that can be used to get information about the most recent error.

TABLE 13.1: `Err` object–related properties

PROPERTY	**DESCRIPTION**
`HelpContext`	Specifies a long value that represents the context ID of a help topic in the help file specified by the `HelpFile` property related to the error.
`HelpFile`	Specifies a string value that represents the help file in which information can be found about the error.
`LastDLLError`	Returns a long value that contains an error code if the error was generated at the operating system level. This property is read-only.
`Source`	Specifies a string value that represents the application or object library in which the error occurred.

The `Err` object supports two methods: `Clear` and `Raise`. The `Clear` method allows you to reset the `Err` object to its default state so that you can continue execution of your procedure and handle additional errors. Though not used as frequently, the `Raise` method can be used to generate an error from a custom procedure within a program. The error that is generated can then be caught with the `On Error` statement by the calling procedure.

The following example shows two custom procedures, a subroutine and a function, that are used to create a new layer. The `On Error` and `Err` objects are used to handle the errors that might occur.

```
Public Sub AddLayer()
  On Error Resume Next

  ' Call the CreateLayer function with a bad layer name
  CreateLayer "<BadName>", acBlue

  ' If an error occurs in the CreateLayer function,
  ' display a message
  If Err.Number <> 0 Then
    MsgBox "Number: " & CStr(Err.Number) & vbLf & _
           "Description: " & Err.Description
  End If
End Sub

' Creates a new layer and returns the AcadLayer object
Private Function CreateLayer(sName As String, _
                             nClr As ACAD_COLOR) As AcadLayer

  On Error Resume Next

  ' Try to get the layer first and return it if it exists
  Set CreateLayer = ThisDrawing.Layers(sName)

  ' If layer doesn't exist create it
  If Err.Number <> 0 Then
    Err.Clear

    On Error GoTo ErrHandler
    Set CreateLayer = ThisDrawing.Layers.Add(sName)
    CreateLayer.color = nClr
  End If

  ' Exit the function if it gets this far
  Exit Function

' If an error occurs when the layer is created, raise an error
ErrHandler:
  Err.Raise Err.Number, Err.Source, Err.Description, _
            Err.HelpFile, Err.HelpContext
End Function
```

In the previous example, the `AddLayer` procedure passes the `CreateLayer` procedure a name and color. The layer name that is passed is invalid and causes an error to occur in the `Add` method of the `AcadLayers` collection object. The `On Error Resume Next` statements are used to keep execution going, whereas the `On Error GoTo` *`<Label>`* statement allows execution to be moved to the general error handler. The general error handler in the `CreateLayer` procedure uses the `Raise` method of the `Err` object to pass an error up to the `AddLayer` procedure so that it can handle what should be done next.

TIP Instead of checking for a value that isn't equal to 0 in the previous examples (`If Err.Number <> 0 Then`), you could simply check to see if the `Err` object has a value using a statement such as `If Err Then`.

Debugging a VBA Project

Debugging is a process that steps through a program and inspects either each code statement—one at a time—or an entire procedure and looks for problems in the code. Maybe your procedure expects a string and instead it is passed a numeric value that generates an error; debugging can be helpful in figuring out just where the problematic value is coming from in your program.

The tools that you can use to debug your code range from simply displaying a message to employing the more integrated solutions found in the VBA Editor. Displaying messages at the Command prompt or in a message box can be a low-tech solution and allow nondevelopers to provide you with troubleshooting information as they use your custom programs. The debugging tools that the VBA Editor offers are more efficient at debugging problems when compared to putting code statements in your program to display messages to the user.

Debugging Through Messages

One of the simplest forms of debugging a program is to display messages at the AutoCAD® Command prompt or in a message box during execution. These messages are displayed periodically as your program executes to let you know which code statements are about to be executed next. You can think of this form of debugging much like the children's game "Red Light, Green Light." Every so often you use a unique message in your program so you have an understanding of progress during the execution of the program. In the game "Red Light, Green Light," you randomly shout out "Red Light" or "Green Light" to ensure people are paying attention and to keep the game moving.

To debug through messages, you place a messaging function every 5 to 10 statements in a custom procedure; place the debugging messages too frequently (or infrequently), and they become less useful. The `Prompt` method of the `AcadUtility` object and the VBA `MsgBox` functions are the most commonly used techniques for displaying a message; I tend to lean away from the `MsgBox` function as it adds unnecessary extra steps to executing a procedure. Once debugging is completed, you can comment out the messaging code statements so they are not displayed to the end user.

The following is an example of a custom procedure that contains two errors and demonstrates how messaging functions can be used to help identify the bad statements. You will step through this code as part of the exercise later in this chapter under the "Stepping Through the BadCode VBA Project" section.

```
Sub BadCode()
  ' Prompt for string
  Dim sVal As String
  sVal = ThisDrawing.Utility.GetString(True, vbLf & "Enter a string: ")

  ' If str is not empty, continue
  If IsEmpty(sVal) = False Then
    ThisDrawing.Utility.Prompt vbLf & "DEBUG: Inside IF"
```

```
      ' Error 1, - should be &
      ThisDrawing.Utility.Prompt vbLf & "Value entered: " - sVal

      ' Prompt for integer
      Dim nVal As Integer
      nVal = ThisDrawing.Utility.GetInteger(vbLf & "Enter an integer: ")

      ThisDrawing.Utility.Prompt vbLf & "DEBUG: Ready to divide"

      ' Error 2, if the user enters 0, cannot divide by 0
      ThisDrawing.Utility.Prompt vbLf & "Divisor: " & CStr(2 / nVal)
    Else
      ThisDrawing.Utility.Prompt vbLf & "DEBUG: If...Else"
    End If

    ThisDrawing.Utility.Prompt vbLf & "DEBUG: Outside IF"
  End Sub
```

If you execute the previous example, the following prompts are displayed the first time you execute the procedure:

```
Enter a string: Hello World!
DEBUG: Inside IF
```

If you change `"Value entered: " - sVal` to `"Value entered: " & sVal` and execute the procedure again, the following messages are displayed at the AutoCAD Command prompt if 0 is entered when prompted for an integer:

```
DEBUG: Inside IF
Value entered: Hello World!
Enter an integer: 0
DEBUG: Ready to divide
```

If a value other than 0 is entered, the following messages are displayed:

```
DEBUG: Inside IF
Value entered: Hello World!
Enter an integer: 2
DEBUG: Ready to divide
Divisor: 1
DEBUG: Outside IF
```

TIP You can use the messaging approach mentioned in this section and adapt it to create a debug log file that can be used to identify problems with your programs after you deploy them. I discussed how to create and write to an external file in Chapter 12, "Communicating with Other Applications."

Using the VBA Editor Debug Tools

Although the On Error statement and Err object are useful in catching and handling errors during execution, the VBA Editor offers several tools that can be helpful in determining what happened during the execution of a procedure that led to an error. The VBA Editor offers the following features that can be used to debug a program:

- Output values to and execute a code statement using the Immediate window
- Interrupt the execution of a procedure and step through the code statements of a procedure in real time using breakpoints
- Check the value of a variable during the execution of a procedure and get notified of when the value changes by establishing a watch on a variable

Outputting Information to the Immediate Window

The Immediate window of the VBA Editor allows you to view the debug output from a program. You can display the Immediate window by pressing Ctrl+G or by clicking Immediate Window in the View menu or Debug toolbar. Although the Immediate window is used primarily to output debug information, it can also be used to execute a single code statement. To execute a code statement in the Immediate window, type a code statement such as MsgBox ThisDrawing.WindowTitle and press Enter.

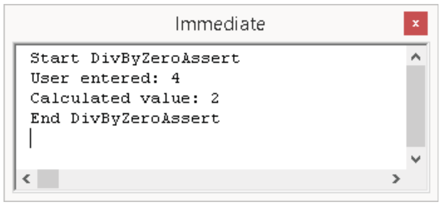
When you use the VBA Debug object, the resulting values and messages are output to the Immediate window, where they aren't visible to the user running the VBA project from the AutoCAD user interface. The Debug object supports two output methods: Print and Assert. The Print method accepts most data types with the exception of an object and displays the value to the Immediate window.

The following shows an example of using the Print method to output values and messages to the Immediate window:

```
Sub DivByZeroDebug()
  Debug.Print "Start DivByZeroDebug"

  Dim nVal As Integer, nDiv As Integer
  nVal = ThisDrawing.Utility.GetInteger(vbLf & "Enter an integer: ")

  Debug.Print "User entered: " & nVal

  If nVal <> 0 Then
    nDiv = nVal / 2
    MsgBox nDiv
    Debug.Print "Calculated value: " & nDiv
  End If

  Debug.Print "End DivByZeroDebug"
End Sub
```

Figure 13.3 shows the results of the `Debug.Print` statements in the Immediate window before the error occurred. Using this technique, you can then deduce that the error occurred after the last message output in the Immediate window and before the next `Debug.Print` statement.

Figure 13.3
Outputting debug information to the Immediate window



```
Start DivByZeroAssert
User entered: 4
Calculated value: 2
End DivByZeroAssert
```

Asserting Your Position

An *assert* in programming is a controlled condition that should always evaluate to `True`, and when it is `False` the program goes into debug mode. In VBA programming, an assert causes the execution of a program to be interrupted and focus to be brought to the position of the code statement containing the assert in the code editor window, where you can evaluate what might have gone wrong. You add an assert to your program using the `Assert` method of the `Debug` object.

While on the surface the `Assert` method might not seem helpful, it can be a useful debugging tool. Instead of using breakpoints, which I cover in the next section, you can suspend the execution of a program when a condition is `False`, thereby reducing the time it takes to debug a program because you engage in the task of debugging only when a specific condition is met.

The following shows an example of using the `Assert` method:

```
Sub DivByZeroAssert()
  Dim nVal As Integer, nDiv As Integer
  nVal = ThisDrawing.Utility.GetInteger(vbLf & "Enter an integer: ")

  If nVal <> 0 Then
    nDiv = nVal / 2
    MsgBox nDiv
  Else
    Debug.Assert False
  End If
End Sub
```

It is good practice to add an assert when you want to check an assumption at runtime. You may want to use an assert with complex conditionals or to test if an error is generated.

The `Assert` method won't execute if a project is password protected and hasn't been unlocked for editing. For information on password-protecting a project, see the "Protecting a Project" section later in this chapter.

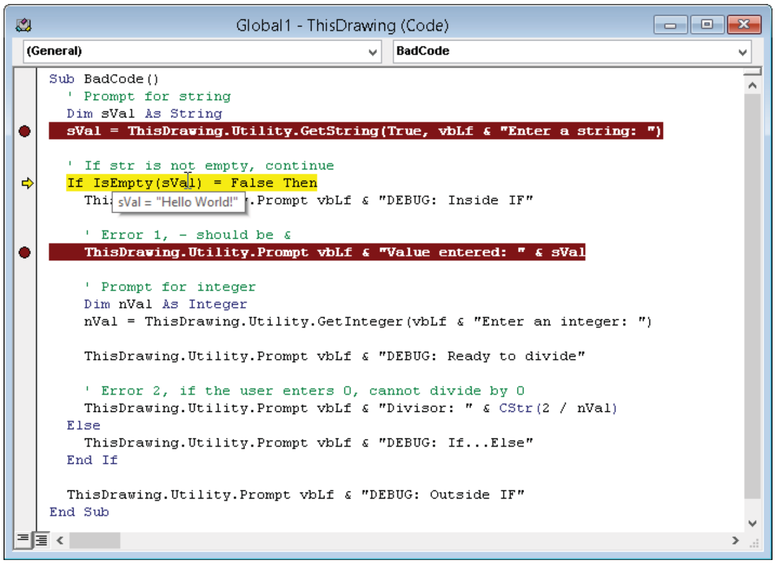
Stepping Through a Procedure

The VBA Editor enables you to step through a program while it is being executed with the use of a feature known as *breakpoints*. Breakpoints allow you to specify a position in a VBA program at which execution should be suspended. While the program is suspended, you can check the current values of a variable and move execution forward one code statement at a time using the code stepping tools, where execution is currently suspended.

While you are in the code editor window, you can set breakpoints quickly by doing any of the following:

- Clicking in the left margin adjacent to a code statement
- Placing the cursor on the desired line and pressing F9
- Right-clicking on a code statement and choosing Toggle ➢ Breakpoint from the context menu

When a breakpoint is placed, a circle is displayed in the left margin and the code statement is highlighted; see Figure 13.4. (By default, the circle and highlight are maroon colored; you can change the color using the Options dialog of the VBA Editor.) Click a breakpoint that is set in the left margin to remove it.



```
Sub BadCode()
  ' Prompt for string
  Dim sVal As String
  sVal = ThisDrawing.Utility.GetString(True, vbLf & "Enter a string: ")

  ' If str is not empty, continue
  If IsEmpty(sVal) = False Then
    Thi sVal = "Hello World!" .Prompt vbLf & "DEBUG: Inside IF"

    ' Error 1, - should be &
    ThisDrawing.Utility.Prompt vbLf & "Value entered: " & sVal

    ' Prompt for integer
    Dim nVal As Integer
    nVal = ThisDrawing.Utility.GetInteger(vbLf & "Enter an integer: ")

    ThisDrawing.Utility.Prompt vbLf & "DEBUG: Ready to divide"

    ' Error 2, if the user enters 0, cannot divide by 0
    ThisDrawing.Utility.Prompt vbLf & "Divisor: " & CStr(2 / nVal)
  Else
    ThisDrawing.Utility.Prompt vbLf & "DEBUG: If...Else"
  End If

  ThisDrawing.Utility.Prompt vbLf & "DEBUG: Outside IF"
End Sub
```

Figure 13.4 Suspend the execution of a program with breakpoints for debugging.

TIP If you wish to remove all breakpoints from a VBA project (not just those in the active code editor window), you can press Ctrl+Shift+F9 or choose Debug ➢ Clear All Breakpoints from the VBA Editor menu bar.

Once one or more breakpoints have been set, you can execute the procedure from the VBA Editor or the AutoCAD user interface with the `vbarun` command. Execution starts and is suspended when the first breakpoint is encountered. The VBA Editor moves to the foreground and the code editor receives focus when execution is suspended. A yellow arrow—known as the *execution point*—and highlighted code indicate the code statement that will be executed next (see Figure 13.4).

TIP You can drag the yellow arrow up or down to control which code statement will be executed next when stepping code statements or resuming normal execution.

While execution is suspended, you can position the cursor over a variable to see its current value in a tooltip (see Figure 13.4). You can also see the current values of all variables in the current procedure using the Locals window or those that are being watched in the Watches window. I discuss the Locals and Watches windows in the next section.

Execution of the procedure can be continued by stepping into, over, or out of a code statement. To step through the code statements of a procedure, you choose one of the following options on the Debug menu or toolbar.

Choose Step Into when you want to step through each code statement in a procedure and continue stepping into other procedures that are called.

Use Step Over when you want to step through each code statement in a procedure but don't want to continue stepping into other procedures that are called.

The Step Out tool resumes normal execution for the code statements in the current procedure. If execution is suspended in a procedure that was called from the original procedure, normal execution is resumed for the called procedure and is suspended when execution returns to the calling procedure.

Normal execution can be restored by choosing Continue on the Run menu or Debug toolbar.

Debugging can be terminated by choosing Reset on the Run menu or Debug toolbar.

Watching Variable Values

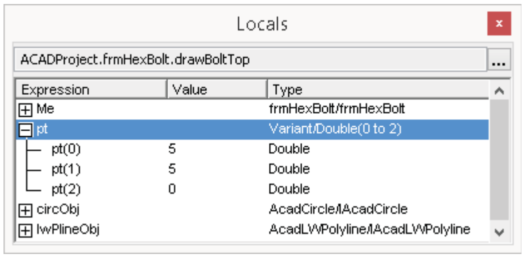
Many people like to watch birds or go whale watching. As a programmer, I have often found watching variable values enjoyable. The VBA Editor allows you to view the current value of one or more variables or see the result of a code statement while a program is executing. It can also let you know when the value of a variable changes or evaluates to `True`.

In the VBA Editor, either the Locals window or the Watches window can be used to view the current value of the variables in a procedure while execution is suspended using a breakpoint or when an assertion occurs:

Locals Window The Locals window, shown in Figure 13.5, allows you to view the value of each local variable in the procedure that currently has execution focus. (The procedure that has the execution focus is the procedure that contains the next code statement to be executed.) You don't need to follow any extra steps to get the variables to display in the window. You can display the Locals window by choosing Locals Window from the View menu or by clicking the Locals Window icon on the Debug toolbar. When the Locals window is displayed, click the ellipsis button in the upper-right corner of the window to open the Call Stack window. The Call Stack window allows you to view the variables dimensioned in either the procedure that has execution focus or the calling procedure that is currently executing.

Watches Window The Watches window, shown in Figure 13.6, allows you to view the value of a specific variable or the return value of a statement in a similar fashion to how the Locals window does. However, the Watches window displays the values only for the variables you are interested in knowing more about. You can also use the Watches window to be notified when the value of a variable changes or whenever the value of a variable or code statement evaluates to `True`. Display the Watches window by choosing Watch Window from the View menu or by clicking the Watches Window icon on the Debug toolbar.

FIGURE 13.5 Viewing local variables with the Locals window

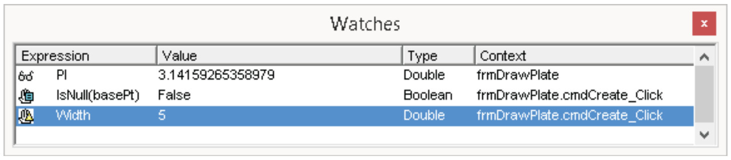


Locals

ACADProject.frmHexBolt.drawBoltTop

Expression	Value	Type
⊞ Me		frmHexBolt/frmHexBolt
⊟ pt		Variant/Double(0 to 2)
pt(0)	5	Double
pt(1)	5	Double
pt(2)	0	Double
⊞ circObj		AcadCircle/IAcadCircle
⊞ lwPlineObj		AcadLWPolyline/IAcadLWPolyline

FIGURE 13.6 Watching variables and statements with the Watches window



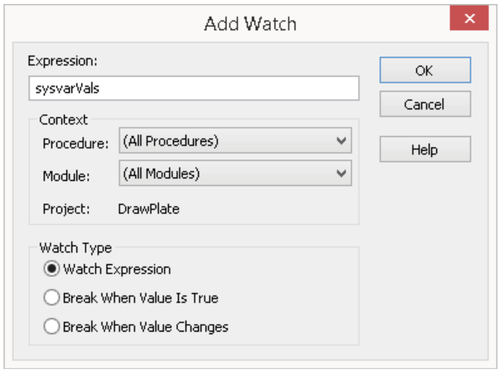
Watches

Expression	Value	Type	Context
PI	3.14159265358979	Double	frmDrawPlate
IsNull(basePt)	False	Boolean	frmDrawPlate.cmdCreate_Click
Width	5	Double	frmDrawPlate.cmdCreate_Click

To add a variable or statement to the Watches window, use the following steps:

1. In the code editor window, select a variable or code statement to add to the Watches window.
2. Right-click the highlighted text and click Add Watch.
3. When the Add Watch dialog box (see Figure 13.7) opens, verify that the variable or statement is displayed in the Expression text box. If not, close the Add Watch dialog box and try again. As an alternative, you can simply type into the Expression text box the code statement or variable name you want to watch.

FIGURE 13.7 Adding a watch with the Add Watch dialog box



Add Watch

Expression: sysvarVals

Context
Procedure: (All Procedures)
Module: (All Modules)
Project: DrawPlate

Watch Type
[✓ Watch Expression]
Break When Value Is True
Break When Value Changes

4. Optionally, you can change the context in which the value of the variable or code statement should be evaluated. The default context is based on where you highlighted the text in your project. You can set the procedure or module settings to (All Procedures) or (All Modules) to display the current value of a variable or code statement no matter what procedure is currently being executed.
5. Optionally, you can change the watch type. The default is Watch Expression, which displays the current value of the variable or returns the value of the code statement in the specified context. Choose Break When Value Is True or Break When Value Changes if you want to debug your program when the value either is `True` or changes while executing the program.
6. Click OK.

   You can modify a watch by selecting it in the Watches window and right-clicking. When the context menu opens, choose Edit Watch (to make changes to the watch entry) or Delete Watch (to remove the watch).

NOTE Watches are not saved with the VBA project, and they are not maintained between AutoCAD sessions. Unloading a VBA project or closing AutoCAD removes any watches that were added.

Deploying a VBA Project

After you have spent countless hours, days, or even weeks writing a program, handling errors, and debugging a VBA program, it all comes down to deploying the program for others to use. When you are ready to deploy a VBA project, you should consider the following:

- How will the VBA project be loaded into the AutoCAD drawing environment?
- How will the user start a VBA macro?
- Should a user be able to make changes to the code and components of a VBA project?

Loading a VBA Project

VBA programs are stored in DVB project files that must be loaded into AutoCAD before they can be used. A number of methods can be used to load a DVB file. These methods fall into one of two categories: manual or automatic. Most DVB files are loaded using one of the manual techniques.

Manually Loading a VBA Project File

AutoCAD is a graphics- and resource-intensive application, and it loads components into memory only as each is needed. DVB files are typically rather small in size, but loading a large number (or even several that include complex user forms) into the AutoCAD drawing environment can impact performance. For this reason, you should load a DVB file only as it is needed and then unload the file once it is no longer needed. I don't suggest loading a DVB file, executing a macro, and then unloading the DVB file immediately because that can affect the user's experience with your custom programs and even with AutoCAD. All DVB files are unloaded from AutoCAD when the current session is terminated, but you can use the `vbaunload` command to unload a specific VBA project while AutoCAD is still running.

Use the following techniques to manually load a DVB file into AutoCAD:

Open VBA Project Dialog Box (`vbaload` Command) The Open VBA Project dialog box allows you to browse to where your DVB files are stored and select which file to load. After selecting a DVB file, click Open to load the file into memory. I discussed how to use this command in Chapter 1, "Understanding the AutoCAD VBA Environment."

Load/Unload Applications Dialog Box (`appload` Command) The Load/Unload Applications dialog box allows you to browse to where your DVB files are stored and select which files to load. After selecting a DVB file, click Load to load the file into memory. I explain how to load a DVB file with the Load/Unload Applications dialog box in the "Using the Load/Unload Applications Dialog Box to Load a DVB File" section later in this chapter.

Drag and Drop DVB and other types of custom program files can be dragged and dropped onto an open drawing window in the AutoCAD drawing environment. When you drop a DVB file onto an open drawing window, AutoCAD prompts you to load the file and/or to enable the macros contained in the VBA project file.

AutoLISP® `vl-vbaload` Function The AutoLISP function `vl-vbaload` allows you to load a DVB file from a script file, from a command macro defined in a CUI/CUIx file, at the AutoCAD Command prompt, or even from a LSP file. When you use the `vl-vbaload` function, it searches the paths that are listed under the Support File Search Path node in the Options dialog box. You should avoid using absolute file paths with the `vl-vbaload` function; if your drive mappings or folder structure changes, the DVB file will fail to load.

The following is an example of loading a DVB file named `drawplate.dvb` with the `vl-vbaload` function:

```
(vl-vbaload "drawplate.dvb")
```

Automatically Loading a VBA Project File

Manually loading DVB files doesn't always create the best user experience. Keep in mind, though, that you don't want all your DVB files to be loaded at startup because it takes away some of the computing resources from the operating system and the AutoCAD program.

You will recall that in Chapter 1, I introduced the following techniques for automatically loading a DVB file into the AutoCAD drawing environment:

Startup Suite (`appload` Command) The Startup Suite is part of the Load/Unload Applications dialog box (`appload` command). When a DVB file is added to the Startup Suite, the file is loaded when the first drawing of a session is opened. Removing a file from the Startup Suite causes the file not to be loaded in any future AutoCAD sessions. If you want to use the Startup Suite to load DVB files, you must add the files to the Startup Suite on each workstation and AutoCAD user profile. I discuss how to add DVB files to the Startup Suite in the "Using the Load/Unload Applications Dialog Box to Load a DVB File" section later in this chapter.

Using an `acad.dvb` File Each time you start AutoCAD, it looks for a file named `acad.dvb` and loads it automatically if found in the AutoCAD support file search paths. In addition to loading the file, if the VBA project contains a public procedure of the subroutine type named `AcadStartup`, the macro is executed at startup.

TIP You can use one of the LSP files that is automatically loaded at startup or with each drawing that is created to load one or more DVB files. For a listing of other files that are loaded automatically when the AutoCAD program is started, see Chapter 20, "Authoring, Managing, and Loading AutoLISP Programs," in *AutoCAD Platform Customization: User Interface, AutoLISP, VBA, and Beyond*, or check the AutoCAD Help system.

Plug-in Bundles Plug-in bundles allow you to load DVB and other custom program files in AutoCAD 2013 or later, and they are supported on both Windows and Mac OS. A plug-in bundle is a folder structure with a special name and metadata file that describes the files contained in the bundle. I discuss plug-in bundles in the "Loading a Project with a Plug-in Bundle" section later in this chapter.

Using the Load/Unload Applications Dialog Box to Load a DVB File

The Load/Unload Applications dialog box (which you open with the `appload` command) is the easiest way to load a DVB file into the AutoCAD drawing environment. Some of the other methods for loading a DVB file provide better integration into an end user's workflow, but they require you to define where the DVB files are located. I describe how to set up and identify the folders where the AutoCAD program should look for custom files in the "Specifying the Location of and Trusting a Project" section later in this chapter.

The following steps provide an overview of how to load a DVB file with the Load/Unload Applications dialog box.

1. On the ribbon, click Manage tab ➢ Customization panel ➢ Load Application (or at the Command prompt, type **`appload`** and press Enter).
2. When the Load/Unload Applications dialog box opens, browse to and select a DVB file. Click Load.

TIP If the Add To History check box is selected when you click Load, AutoCAD adds the selected file to a list box on the History tab. Click the History tab and then select the file you want to load. Then click Load to load the file.

3. If the File Loading - Security Concern message box is displayed, click Load. You'll learn which paths contain custom files that should be trusted in the "Specifying the Location of and Trusting a Project" section and the sidebar "Restricting Custom Applications" later in this chapter.
4. Click Close to return to the drawing area.

You can use the following steps to add a DVB file to the Startup Suite:

1. On the ribbon, click the Manage tab ➢ Customization panel ➢ Load Application (or at the Command prompt, type **`appload`** and press Enter).
2. When the Load/Unload Applications dialog box opens, click Contents in the Startup Suite section.
3. When the Startup Suite dialog box opens, click Add.

4. In the Add File To Startup Suite dialog box, browse to and select a DVB file. Click Open.
5. In the Startup Suite dialog box, click Close.
6. In the Load/Unload Applications dialog box, click Close.

Loading a Project with a Plug-in Bundle

A plug-in bundle, as I previously mentioned, is one of the methods that can be used to deploy your DVB files. Fundamentally, a bundle is simply a folder structure with its topmost folder having `.bundle` appended to its name and a manifest file with the filename `PackageContents.xml` located in the topmost folder.

You can use Windows Explorer or File Explorer to define and name the folder structure of a bundle. You can create the `PackageContents.xml` file with a plain ASCII text editor such as Notepad. You will also need a bit of assistance from AutoLISP to load a DVB file into the AutoCAD drawing environment with the bundle.

The following is a sample `PackageContents.xml` file that defines the contents of a bundle named `DrawPlate_VBA.bundle` that contains three files: a help file named `DrawPlate_VBA.htm`, a LSP file named `DrawPlateLoader.lsp`, and the VBA project file named `DrawPlate.dvb`:

```
<?xml version="1.0" encoding="utf-8"?>
<ApplicationPackage
    SchemaVersion="1.0"
    AppVersion="1.0"
    Name="Plate Generator (VBA)"
    Description="Draws a rectangle plate with four bolt holes."
    Author="HyperPics, LLC"
    ProductCode="{144819FE-3A2B-4D8A-B49C-814D0DBD45B3}"
    HelpFile="./Contents/DrawPlate_VBA.htm"
>

  <CompanyDetails
     Name="HyperPics, LLC"
     Url="http://www.hyperpics.com"
  />

  <RuntimeRequirements
     OS="Win32|Win64"
     SeriesMin="R19.0"
     Platform="AutoCAD*"
     SupportPath="./Contents/"
  />

  <Components Description="Windows OSs">
    <ComponentEntry Description="Loader file"
       AppName="DrawPlateMainLoader"
       Version="1.0"
       ModuleName="./Contents/DrawPlateLoader.lsp">
    </ComponentEntry>
```

```
      <ComponentEntry Description="Main file"
         AppName="DrawPlateMain"
         Version="1.0"
         ModuleName="./Contents/DrawPlate.dvb">
      </ComponentEntry>
   </Components>
</ApplicationPackage>
```

The folder structure of the bundle that the `PackageContents.xml` file refers to looks like this:

```
DrawPlate_VBA.bundle
      PackageContents.xml
   Contents
      DrawPlate.dvb
      DrawPlate_VBA.htm
      DrawPlateLoader.dvb
```

I have provided the `DrawPlate_VBA.bundle` as part of the sample files for this book, but you will also learn how to create the `DrawPlate_VBA.bundle` yourself later in this chapter. To use the bundle with AutoCAD, copy the `DrawPlate_VBA.bundle` folder and all of its contents to one of the following locations so that all users can access the files:

- `ALLUSERSPROFILE%\Application Data\Autodesk\ApplicationPlugIns` (Windows XP)
- `ALLUSERSPROFILE%\Autodesk\ApplicationPlugIns` (Windows 7 or Windows 8)

If you want a bundle to be accessible only by specific users, place the bundle into the following location under each user's profile:

- `APPDATA%\Autodesk\ApplicationPlugIns`

For additional information on the elements used to define a `PackageContents.xml` file, perform a search in the AutoCAD Help system on the keyword "PackageContents.xml."

NOTE The `appautoload` system variable controls when bundles are loaded into AutoCAD. By default, bundles are loaded at startup, when a new drawing is opened, and when a plug-in is added to the `ApplicationPlugins` folder. You can use the `appautoloader` command to list which bundles are loaded or to reload all the bundles that are available to AutoCAD.

Specifying the Location of and Trusting a Project

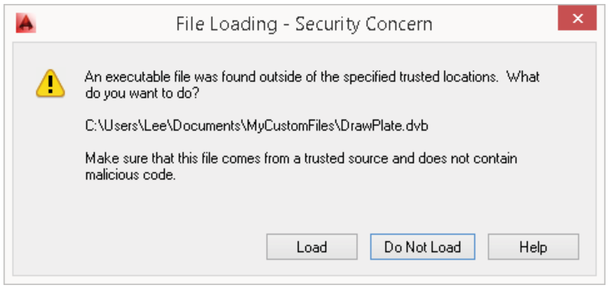
The DVB files that you create or download from the Internet can be placed in any folder on a local or network drive. I recommend placing all your custom files in a single folder on a network drive so they can be accessed by anyone in your company who might need them. Placing the files in a network location makes rolling out changes easier as well. You might consider using the name `DVB Files` or `VBA Project Files` for the folder that contains your DVB files.

I also recommend marking any folder(s) that contain custom files on the network as read-only for everyone except for those designated to make updates to the files. Marking the folders as read-only helps prevent undesired or accidental changes.

Regardless of the folder name you use or where you choose to place your DVB files, you need to let AutoCAD know where these files are located. To do so, add each folder that contains DVB files to the Support File Search Path and the Trusted Locations settings accessible through the Options dialog box.

The support file search paths are used by AutoCAD to locate custom files, such as those that contain block definitions, linetype patterns, AutoLISP programs, and VBA projects. Use the Options dialog box to add the folders that contain DVB files to the support file search paths of AutoCAD.

If you are using AutoCAD 2013 SP1 or later, when you try to load a DVB file AutoCAD checks to see if the DVB file being loaded is from a *trusted location*. A folder that you identify as a trusted location contains DVB files that are safe to be loaded without user interaction. The Trusted Locations setting in the Options dialog box or the `trustedpaths` system variable are used to specify trusted locations. Any DVB file that isn't loaded from a trusted location results in the File Loading - Security Concern message box (see Figure 13.8) being displayed.


File Loading - Security Concern

An executable file was found outside of the specified trusted locations. What do you want to do?

C:\Users\Lee\Documents\MyCustomFiles\DrawPlate.dvb

Make sure that this file comes from a trusted source and does not contain malicious code.

FIGURE 13.8
This security warning informs you that a DVB file is being loaded from an untrusted location.

The File Loading - Security Concern message box indicates why it might not be a good idea to load the file if its origins aren't known. Loading files with an unknown origins could introduce malicious code. The end user then must decide to load (or not load) the file before the AutoCAD program can complete the load. When adding new trusted locations, make sure you limit the number of folders you trust. Further, trusted folders should be marked as read-only to avoid the introduction of unknown DVB files or other custom programs to the folders. For more information on trusted paths, see the `trustedpaths` system variable in the AutoCAD Help system.

NOTE A folder that you identify as a trusted location must also be listed in the Support File Search Paths setting of the Options dialog box.

The following steps explain how to add a folder to the support file search paths and trusted locations used by AutoCAD:

1. Click the Application menu button ➢ Options (or at the Command prompt, type **`options`** and press Enter).
2. When the Options dialog box opens, click the Files tab.

The following steps explain how to add a folder to the AutoCAD support file search paths:

1. Select the Support File Search Path node. Click Add and then click Browse.
2. In the Browse For Folder dialog box, browse to and select the folder that contains your DVB files.
3. Click OK.

The following steps explain how to add a folder to the AutoCAD trusted locations:

1. Select the Trusted Locations node and click Add, and then click Browse.
2. In the Browse For Folder dialog box, browse to and select the folder that contains your DVB files.
3. Click OK.
4. If the selected folder is not read-only, the Trusted File Search Path - Security Concern dialog box will be displayed. Click Continue to add the folder.
5. Click OK to save the changes to the Options dialog box.

If the location of your custom programs changes, you can replace an existing folder in the Options dialog box by expanding the Support File Search Path or Trusted Paths node and selecting the folder you want to replace. After selecting the folder you want to replace, click Browse and then select the new folder.

Restricting Custom Applications

Starting with AutoCAD 2013 SP1, Autodesk introduced some new security measures to help reduce potential threats or viruses that could affect AutoCAD and the drawing files you create. These security measures allow you to do the following:

- Disable the loading of executable code when AutoCAD is started using `/nolisp` (AutoCAD 2013 SP1) or `/safemode` (AutoCAD 2014 and later)
- Automatically load and execute specially named files: `acad.lsp`, `acad.fas`, `acad.vlx`, `acaddoc.lsp`, `acaddoc.fas`, `acaddoc.vlx`, and `acad.dvb`

In AutoCAD 2014 and later, you can use the `secureload` system variable to control whether AutoCAD loads files only from trusted locations or allows you to load custom files from any location. I recommend setting `secureload` to 2 and loading custom files only from a secure and trusted location. However, the default value of 1 for `secureload` is also fine; it displays a message box when AutoCAD tries to load a file from a nontrusted location. Don't set `secureload` to 0, thereby disabling the security feature, because it could result in your system loading a malicious program.

Starting a Macro with AutoLISP or a Command Macro

Executing a VBA macro from the Macros dialog box can be a bit overwhelming to an end user since the dialog box lists all the available macros from each of the VBA projects that are currently loaded into the AutoCAD drawing environment. Most end users are accustomed to starting a command from the user interface or even typing a command at the AutoCAD Command prompt.

A VBA macro can be executed from a command macro in the user interface or at the Command prompt using the AutoLISP `vl-vbarun` function or the `-vbarun` command. Both methods achieve the same result and can be used interchangeably.

The following examples show how to execute the `CLI_DrawPlate` procedure defined in the `basDrawPlate` code module of the `DrawPlate.dvb` file with the `vl-vbarun` function and the `-vbarun` command using the AutoLISP `command` function:

```
; Execute macro with vl-vbarun
(vl-vbarun "DrawPlate.dvb!basDrawPlate.CLI_DrawPlate")

; Execute macro with command function and -vbarun command
(command "._-vbarun" "DrawPlate.dvb!basDrawPlate.CLI_DrawPlate")
```

The following shows how to execute the `CLI_DrawPlate` procedure defined in the `basDrawPlate` code module of the `DrawPlate.dvb` file with the `-vbarun` command at the AutoCAD Command prompt:

```
Command: -VBARUN
Macro name: DrawPlate.dvb!basDrawPlate.CLI_DrawPlate
```

TIP If for some reason a DVB file with the same name is loaded from different locations, you can specify the absolute file path to a DVB file to ensure the correct macro is executed. For example, you could use

```
(vl-vbarun "c:\\users\\lee\\documents\\mycustomfiles\\DrawPlate.dvb!basDrawPlate.
CLI_DrawPlate") instead of (vl-vbarun "DrawPlate.dvb!basDrawPlate.CLI_DrawPlate")
```

Although VBA doesn't allow you to create a custom command that end users can enter at the Command prompt like AutoLISP, ObjectARX®, or Managed .NET does, you can use AutoLISP as a wrapper to execute a VBA procedure.

The following shows how to use AutoLISP to define a custom command named `-drawplate_vba` that an end user could use to execute the `CLI_DrawPlate` macro:

```
(defun c:-drawplate_vba ( )
  (vl-vbarun "DrawPlate.dvb!basDrawPlate.CLI_DrawPlate")
)
```

Grouping Actions into a Single Undo

When a VBA macro is executed, users tend to expect certain things to occur before or after the use of any custom program. Users expect that any changes to system variables will be restored if those variables affect drawings, and they expect that when they type **`u`** and press Enter any changes to the drawing will be rolled back. A single undo record isn't always created when a VBA program is executed, especially when the `SendCommand` or `PostCommand` method of the `AcadDocument` object is used. I discussed these methods in Chapter 3, "Interacting with the Application and Documents Objects."

It is good practice to call the `StartUndoMark` and `EndUndoMark` methods of the `AcadDocument` object when a VBA program makes changes to a drawing. The `StartUndoMark` method should be called before the first change is made, and the `EndUndoMark` method should be called after the last change is made. The methods instruct AutoCAD to group the operations between the two methods into a single undo record, making it easier for the user to roll back any changes made by a VBA program.

Protecting a Project

A lot of time and effort can go into developing a VBA project, and a VBA project may include information about proprietary processes or intellectual property. The VBA Editor offers a way to password-protect the code and components of a VBA project. When a VBA project is password-protected, the VBA project can be loaded and macros can be executed without entering a password. But when anyone wishes to edit the code and components, the password must be entered.

The following steps explain how to password-protect a VBA project:

1. Load a DVB file into the AutoCAD drawing environment.
2. In the VBA Editor, click Tools ➢ <*Project Name*> Properties.
3. When the <*Project Name*> - Project Properties dialog box opens, click the Protection tab.
4. On the Protection tab, check the Lock Project For Viewing check box and enter a password in the Password and Confirm Password text boxes.
5. Click OK.

Exercise: Deploying the DrawPlate VBA Project

In this section, you will continue to work with the DrawPlate project that was introduced in Chapter 4, "Creating and Modifying Drawing Objects." If you completed the exercises, you also worked with the DrawPlate project throughout this book by adding annotations, getting input from the user at the Command prompt, and even implementing a user interface to get input from the user.

The key concepts I cover in this exercise are as follows:

Using Breakpoints and Stepping Through Code Statements Suspending a VBA program during execution can be used to step through the code statements that define a procedure and to view the current values of the variables used by a procedure.

Adding Error-Handling Statements Using `On Error GoTo` statements and labels to implement error handling can help reduce problems that an end user might encounter when using a custom program.

Using Undo Grouping Wrapping methods into an undo grouping allows any changes that are made by a custom program to be rolled back and restores the drawing to the state it was in before it was executed.

Identifying the Locations of Your DVB Files AutoCAD must be able to find your DVB files and needs to know which locations are trusted.

Creating and Deploying Plug-in Bundles Plug-in bundles can make deploying VBA programs easier than having to set up support file search paths and trusted locations on multiple machines, and they allow you to support multiple releases of a program with much greater ease.

NOTE The steps in this exercise depend on the completion of the steps in the "Exercise: Implementing a UserForm for the *DrawPlate* Program" section of Chapter 11, "Creating and Displaying User Forms." If you didn't complete the steps, do so now or start with the ch13_drawplate.dvb sample file available for download from www.sybex.com/go/autocadcustomization. You will also need the ch13_badcode.dvb, ch13_packagecontents.xml, ch13_drawplate_vba.htm, and ch13_drawplateloader.lsp sample files. Place these sample files in the MyCustomFiles folder under the Documents (or My Documents) folder or in the location you are using to store the DVB files. Once you've stored the sample files on your system, remove the characters ch13_ from the name of each file.

Stepping Through the BadCode VBA Project

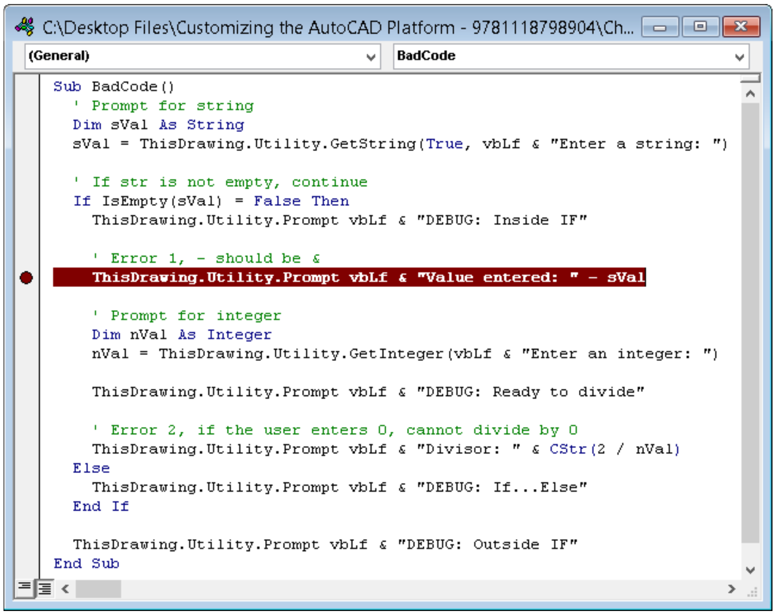
In this exercise, you'll work with the badcode.dvb file that came with this book and was shown in the "Debugging Through Messages" section. Stepping through a program code statement by code statement allows you to identify what is happening in your code, determine whether it is executing as expected or if an error occurs, and see which branches of a program are being followed based on the results of the logical tests. Additionally, you can view the current values of the variables in the program at specific times to ensure they have the correct data before they are passed to a function.

The following steps explain how to set a breakpoint and add watches to the Watches window:

1. Open the VBA Editor and load the badcode.dvb file.
2. In the Project Explorer of the VBA Editor, double-click the basBadCode component.
3. In the code editor window, locate the statement that is below the comment ' Error 1, - should be &. Click in the left margin adjacent to the code statement to set a breakpoint.

   The code editor window should now look like Figure 13.9.

FIGURE 13.9 Breakpoint set in the code editor window

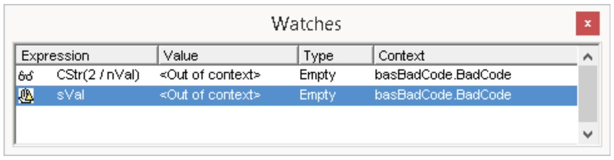

C:\Desktop Files\Customizing the AutoCAD Platform - 9781118798904\Ch...
(General)
BadCode

```
Sub BadCode()
  ' Prompt for string
  Dim sVal As String
  sVal = ThisDrawing.Utility.GetString(True, vbLf & "Enter a string: ")

  ' If str is not empty, continue
  If IsEmpty(sVal) = False Then
    ThisDrawing.Utility.Prompt vbLf & "DEBUG: Inside IF"

    ' Error 1, - should be &
    ThisDrawing.Utility.Prompt vbLf & "Value entered: " - sVal

    ' Prompt for integer
    Dim nVal As Integer
    nVal = ThisDrawing.Utility.GetInteger(vbLf & "Enter an integer: ")

    ThisDrawing.Utility.Prompt vbLf & "DEBUG: Ready to divide"

    ' Error 2, if the user enters 0, cannot divide by 0
    ThisDrawing.Utility.Prompt vbLf & "Divisor: " & CStr(2 / nVal)
  Else
    ThisDrawing.Utility.Prompt vbLf & "DEBUG: If...Else"
  End If

  ThisDrawing.Utility.Prompt vbLf & "DEBUG: Outside IF"
End Sub
```

4. In the code editor window, in the first few code statements of the `BadCode` procedure locate the variable `sVal`.
5. Double-click the `sVal` text to select it and then right-click the selected text. Choose Add Watch from the context menu.
6. When the Watches window opens, in the Watch Type section choose Break When Value Changes and click OK.
7. If the Watches window is in the way, drag it away from the code editor window.
8. In the code editor window, right-click and choose Add Watch.
9. When the Watches window opens, replace the text in the Expression text box with **`CStr(2 / nVal)`** and click OK.

   The Watches window should now look like Figure 13.10.

FIGURE 13.10
Current watches added to the Watches window



Expression	Value	Type	Context
CStr(2 / nVal)	<Out of context>	Empty	basBadCode.BadCode
sVal	<Out of context>	Empty	basBadCode.BadCode

The following steps explain how to step through the code statements of the `BadCode` procedure:

1. Switch to AutoCAD.
2. At the Command prompt, type **`vbarun`** and press Enter.
3. When the Macros dialog box opens, select the `BadCode` procedure from the Macros list. Click Run.
4. At the `Enter a string:` prompt, type **`Hello World!`** and press Enter.

   Execution of the `BadCode` procedure is suspended as a result of the watch setup for the `sVal` variable. The `If IsEmpty(sVal) = False Then` code statement is also highlighted, indicating which code statement will be executed next when execution resumes.
5. Review the current value of the `sVal` variable in the Watches window. The value of the `sVal` variable in the Watches window should now be listed as "Hello World!"
6. In the VBA Editor, click Run ➢ Continue to resume normal execution.

   Execution is suspended again when the breakpoint is reached.
7. In the VBA Editor, click Debug ➢ Step Into to execute the highlighted code statement and move execution to the next code statement.
8. When the Microsoft Visual Basic error message is displayed, click Debug.

The type mismatch error is the result of the text `- sVal` in the code statement.

9. In the highlighted statement, change the text `"Value entered: " - sVal` to **`"Value entered: " & sVal`**.
10. Press F8 to execute the highlighted code statement and move execution to the next code statement.
11. Click Debug ➢ Clear All Breakpoints to remove the breakpoint that was set.
12. Click Run ➢ Continue to resume normal execution.
13. Switch to AutoCAD.
14. At the `Enter an integer:` prompt, type **4** and press Enter.
15. Press F2 to expand the Command Line history.

    The Command Line history shows the following messages:

```
DEBUG: Inside IF
Value entered: Hello World!
Enter an integer: 4
Command:
DEBUG: Ready to divide
Divisor: 0.5
DEBUG: Outside IF
```

Implementing Error Handling for the Utility Procedures

As you make changes to the procedures in the `clsUtilities` class module, notice how easy it can be to implement error handling for your utility functions.

The following steps explain how to update the `CreateLayer` procedure to handle general problems and pass the error to the calling procedure:

1. Load the `drawplate.dvb` file that you last updated in the exercises for Chapter 11, or rename the file `ch13_drawplate.dvb` to **`drawplate.dvb`** and then load the renamed file.
2. In the Project Explorer, double-click the `clsUtilities` component.
3. In the code editor window, scroll to the `CreateLayer` procedure and add the bold text:

```
Public Function CreateLayer(sName As String, _
                            nClr As ACAD_COLOR) As AcadLayer

  On Error Resume Next

  ' Try to get the layer first and return it if it exists
  Set CreateLayer = ThisDrawing.Layers(sName)

  ' If layer doesn't exist create it
  If Err Then
```

```
    Err.Clear

    On Error GoTo ErrHandler
    Set CreateLayer = ThisDrawing.Layers.Add(sName)
    CreateLayer.color = nClr
  End If

  ' Exit the function if it gets this far
  Exit Function

' If an error occurs, raise an error
ErrHandler:
  Err.Raise Err.Number, Err.Source, Err.Description, _
            Err.HelpFile, Err.HelpContext
End Function
```

4. Click File ➢ Save.

The following steps explain how to update the `CreateRectangle`, `CreateText`, and `CreateCircle` procedures to handle general problems and pass the error to the calling procedure:

1. In the code editor window, scroll to the procedures in the code and add the bold text:

```
Public Function CreateRectangle(ptList As Variant) As AcadLWPolyline
  On Error GoTo ErrHandler

  Set CreateRectangle = ThisDrawing.ActiveLayout.Block. _
                        AddLightWeightPolyline(ptList)
  CreateRectangle.Closed = True

  ' Exit the function if it gets this far
  Exit Function

' If an error occurs, raise an error
ErrHandler:
  Err.Raise Err.Number, Err.Source, Err.Description, _
            Err.HelpFile, Err.HelpContext
End Function

Public Function CreateText(insPoint As Variant, _
                           attachmentPt As AcAttachmentPoint, _
                           textHeight As Double, _
                           textRotation As Double, _
                           textString As String) As AcadMText
  On Error GoTo ErrHandler

  Set CreateText = ThisDrawing.ActiveLayout.Block. _
                   AddMText(insPoint, 0, textString)
  ' Sets the text height, attachment point, and rotation of the MText object
  CreateText.height = textHeight
  CreateText.AttachmentPoint = attachmentPt
  CreateText.insertionPoint = insPoint
  CreateText.rotation = textRotation

  ' Exit the function if it gets this far
  Exit Function

' If an error occurs, raise an error
ErrHandler:
  Err.Raise Err.Number, Err.Source, Err.Description, _
            Err.HelpFile, Err.HelpContext
End Function

Public Function CreateCircle(cenPt As Variant, circRadius) As AcadCircle
  On Error GoTo ErrHandler

  Set CreateCircle = ThisDrawing.ActiveLayout.Block. _
                     AddCircle(cenPt, circRadius)

  ' Exit the function if it gets this far
  Exit Function

' If an error occurs, raise an error
ErrHandler:
  Err.Raise Err.Number, Err.Source, Err.Description, _
            Err.HelpFile, Err.HelpContext
End Function
```

2. Click File ➢ Save.

Implementing Error Handling and Undo Grouping for the Main Procedures

The following steps explain how to update the `CLI_DrawPlate` procedure to handle general problems when drawing the objects that form the plate and use undo grouping to make rolling back changes easier:

1. In the Project Explorer, double-click the `basDrawPlate` component.
2. In the code editor window, scroll to the `CLI_DrawPlate` and add the text in bold:

```
Public Sub CLI_DrawPlate()
  Dim oLyr As AcadLayer

  On Error Resume Next
  ' Start an undo mark here
  ThisDrawing.StartUndoMark

  Dim sysvarNames As Variant, sysvarVals As Variant
  sysvarNames = Array("nomutt", "clayer", "textstyle")
```

3. Scroll to the `If IsNull(basePt) = False Then` statement and add the text in bold:

```
' If a base point was specified, then draw the plate
If IsNull(basePt) = False Then
  On Error GoTo ErrHandler

  ' Create the layer named Plate or set it current
  Set oLyr = myUtilities.CreateLayer("Plate", acBlue)
  ThisDrawing.ActiveLayer = oLyr
```

4. Scroll to the `Dim insPt As Variant` statement and add the text in bold:

```
myUtilities.CreateCircle cenPt4, 0.1875

On Error Resume Next

' Get the insertion point for the text label
Dim insPt As Variant
insPt = Null

insPt = ThisDrawing.Utility.GetPoint(, _
        removeCmdPrompt & "Specify label insertion point " & _
                          "<or press Enter to cancel placement>: ")

' If a point was specified, placed the label
If IsNull(insPt) = False Then
  On Error GoTo ErrHandler

  ' Define the label to add
  Dim sTextVal As String
```

5. Scroll to the `Loop Until IsNull(basePt) = True And sKeyword = ""` statement and add the text in bold:

```
      myUtilities.CreateText insPt, acAttachmentPointMiddleCenter, _
                             0.5, 0#, sTextVal
    End If
  End If

  On Error Resume Next
Loop Until IsNull(basePt) = True And sKeyword = ""
```

```
' Restore the saved system variable values
myUtilities.SetSysvars sysvarNames, sysvarVals
```

6. Scroll to the `End Sub` statement and add the text in bold:

```
  ' Save previous values to global variables
  g_drawplate_width = width
  g_drawplate_height = height

  ' End an undo mark here
  ThisDrawing.EndUndoMark

  Exit Sub

ErrHandler:
  ' End an undo mark here
  ThisDrawing.EndUndoMark

  ' Rollback changes
  ThisDrawing.SendCommand "._u "
End Sub
```

7. Click File ➢ Save.

The following steps explain how to update the `cmdCreate_Click` procedure to handle general problems when drawing the objects that form the plate and use undo grouping to make rolling back changes easier:

1. In the Project Explorer, right-click the `frmDrawPlate` component and choose View Code.
2. In the code editor window, scroll to the `cmdCreate_Click procedure`, or select `cmdCreate` from the Object drop-down list and then choose Click from the Procedure drop-down list to the right of the Object drop-down list at the top of the code editor window.
3. Add the text in bold:

```
Private Sub cmdCreate_Click()
  Dim oLyr As AcadLayer

  ' Hide the dialog so you can interact with the drawing area
  Me.Hide

  On Error Resume Next

  ' Start an undo mark here
  ThisDrawing.StartUndoMark

  Dim sysvarNames As Variant, sysvarVals As Variant
  sysvarNames = Array("nomutt", "clayer", "textstyle")
```

4. Scroll to the `If IsNull(basePt) = False Then` statement and add the text in bold:

```
' If a base point was specified, then draw the plate
 If IsNull(basePt) = False Then
  On Error GoTo ErrHandler

  ' Create the layer named Plate or set it current
  Set oLyr = myUtilities.CreateLayer("Plate", acBlue)
  ThisDrawing.ActiveLayer = oLyr
```

5. Scroll to the `If Me.chkAddLabel.Value = True Then` statement and add the text in bold:

```
myUtilities.CreateCircle cenPt4, 0.1875

If Me.chkAddLabel.Value = True Then
  On Error Resume Next

  ' Get the insertion point for the text label
  Dim insPt As Variant
  insPt = Null
  insPt = ThisDrawing.Utility.GetPoint(, _
          removeCmdPrompt & "Specify label insertion point " & _
                           "<or press Enter to cancel placement>: ")

  ' If a point was specified, placed the label
  If IsNull(insPt) = False Then
    On Error GoTo ErrHandler

    ' Define the label to add
    Dim sTextVal As String
```

6. Scroll of the `End Sub` statement and add the text in bold:

```
  ' Save previous values to global variables
  g_drawplate_width = width
  Me.txtWidth.Text = Format(g_drawplate_width, "0.0000")
  g_drawplate_height = height
  Me.txtHeight.Text = Format(g_drawplate_height, "0.0000")
  g_drawplate_label = Me.chkAddLabel.Value
  ' End an undo mark here
  ThisDrawing.EndUndoMark

  ' Show the dialog box once done
  Me.show

  Exit Sub

ErrHandler:
  ' End an undo mark here
  ThisDrawing.EndUndoMark

  ' Rollback changes
  ThisDrawing.SendCommand "._u  "

  ' Show the dialog box once done
  Me.show
End Sub
```

7. Click File ➤ Save.

Configuring the AutoCAD Support and Trusted Paths

If you can't or don't plan to use a bundle to deploy your custom programs, you must let AutoCAD know where your DVB files are stored and whether they can be trusted. Without the trusted file designation, AutoCAD will display the File Loading - Security Concern message box each time a custom program is loaded in AutoCAD 2013 SP1 or later. And consider this: How can AutoCAD run a program it can't find?

The following steps explain how to add the folder named `MyCustomFiles` to the support file search paths and trusted locations used by AutoCAD:

1. Click the Application menu button ➤ Options (or at the Command prompt, type **`options`** and press Enter).
2. When the Options dialog box opens, click the Files tab.
3. Select the Support File Search Path node. Click Add and then click Browse.
4. In the Browse For Folder dialog box, browse to the `MyCustomFiles` folder that you created for this book in the `Documents` (or `My Documents`) folder, or browse to the folder that contains your DVB files.
5. Select the folder that contains your DVB files and click OK.
6. With the new path still highlighted, press F2. Press Ctrl+C, or right-click and choose Copy.
7. Select the Trusted Locations node. Click Add.
8. With focus in the in-place text editor, press Ctrl+V, or right-click and choose Paste. Then press Enter to accept the pasted path.
9. If the Trusted File Search Path - Security Concern message box appears, click Continue.
10. Click OK to save the changes to the Options dialog box.

Creating *DrawPlate_VBA.bundle*

Plug-in bundles are a relatively new concept in AutoCAD, but they make deploying your custom programs much easier. After all, a bundle is simply a folder structure that you can copy between machines no matter which operating system you are using. Bundles are supported in AutoCAD 2013–based products and later.

The following steps explain how to create a bundle named `DrawPlate_VBA.bundle`:

1. Launch Windows Explorer or File Explorer based on your version of the operating system. Right-click the Windows Start button in Windows XP or Windows 7, or right-click in the lower-left corner of the screen on Windows 8. Click Windows Explorer or File Explorer.
2. Browse to the `MyCustomFiles` folder under the `Documents` (or `My Documents`) folder. Right-click in an empty area and choose New ➢ Folder.
3. Type **`DrawPlate_VBA.bundle`** and press Enter.
4. Double-click the `DrawPlate_VBA.bundle` folder.
5. Create a new folder under the `DrawPlate_VBA.bundle` folder and name the new folder **`Contents`**.
6. From the sample files that are available with this book and those that you created, copy the following files into the appropriate folder (see Table 13.2).

Table 13.2: Files for `DrawPlate_VBA.bundle`

Filename	Folder
`packagecontents.xml`	`DrawPlate.bundle`
`drawplateloader.lsp`	`Contents`
`drawplate.dvb`	`Contents`
`drawplate_vba.htm`	`Contents`

The `drawplateloader.lsp` file loads the `drawplate.dvb` file and then defines two custom functions named `-drawplate_vba` and `drawplate_vba`. The `-drawplate_vba` function also supports contextual help; when the function is active, you can press F1 to display the `drawplate_vba.htm` file.

Deploying and Testing *DrawPlate_VBA.bundle*

Plug-in bundles must be placed within a specific folder before they can be used. You learned which folders a bundle can be placed in earlier in the section "Loading a Project with a Plug-in Bundle."

The following steps explain how to deploy a bundle named `DrawPlate_VBA.bundle`:

1. In Windows Explorer or File Explorer, browse to the `DrawPlate_VBA.bundle` folder you created in the previous exercise.
2. Select the `DrawPlate_VBA.bundle` folder and right-click. Choose Copy.
3. In the Location/Address bar of Windows Explorer or File Explorer, type one of the following and press Enter:

- In Windows XP, type **`%ALLUSERSPROFILE%\Application Data\Autodesk\ApplicationPlugIns`**.
- In Windows 7 or Windows 8, type **`%ALLUSERSPROFILE%\Autodesk\ApplicationPlugIns`**.

4. Right-click in the file list and choose Paste.

The following steps explain how to test `DrawPlate.bundle`:

1. In AutoCAD, create a new drawing.
2. At the Command prompt, type **`-drawplate_vba`** and press Enter.

   You should see the familiar `Specify base point for plate or [Width/Height]:` prompt. Before you created the bundle, you had to load the `drawplate.dvb` file and then start the macro with the `vbarun` command to access the functionality. As a reminder, the `-drawplate_vba` function is defined as part of the `drawplateloader.lsp` file that is used to also load the `DrawPlate.dvb` file.

NOTE If the `-drawplate_vba` function isn't available in the drawing, check the current value of the `appautoload` system variable. The `appautoload` system variable controls when a bundle should be loaded. The default value of the `appautoload` system variable is 14, which indicates a bundle should be loaded at startup, when a new drawing is opened, or when a new bundle has been added to one of the plug-in folders.

3. When the `-drawplate_vba` function starts, press Esc to end the function.
4. At the Command prompt, type **`drawplate_vba`** and press Enter.

   You should see the Draw Plate `UserForm` that you defined in Chapter 11, "Creating and Displaying User Forms."
5. Click Cancel to exit the dialog box.

Index

Note to the Reader: Throughout this index **boldfaced** page numbers indicate primary discussions of a topic. *Italicized* page numbers indicate illustrations.