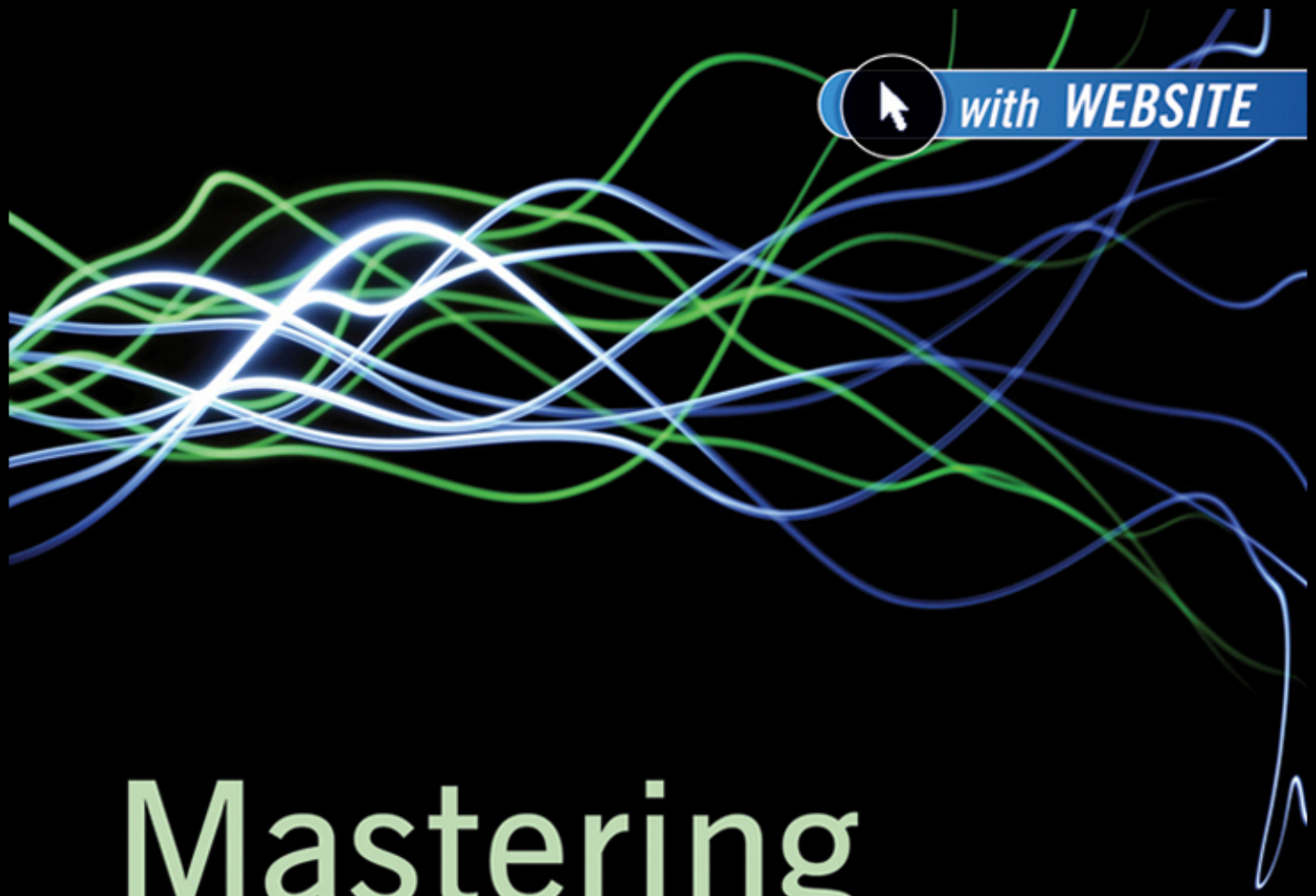


Mastering IDEAScript

THE DEFINITIVE GUIDE

John Paul **Mueller**

Mastering IDEAScript

Mastering IDEAScript

The Definitive Guide

JOHN PAUL MUELLER

John Wiley & Sons, Inc.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

For general information on our other products and services or for technical support, please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993, or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books. For more information about Wiley products, visit our Web site at www.wiley.com.

Library of Congress Cataloging-in-Publication Data:

Mueller, John, 1958-
Mastering IDEAScript: the definitive guide/John Paul Mueller.
p. cm.
Includes index.
ISBN 978-1-118-00448-7 (pbk.); 978-1-118-01783-8 (ebk); 978-1-118-01784-5 (ebk); 978-1-118-01785-2 (ebk)
1. Auditing–Data processing. 2. Accounting–Data processing. 3. Scripting languages (Computer science) 4. Data structures (Computer science) I. Title.
HF5667.12.M37 2011
657.0285'53–dc22

2010045246

Printed in the United States of America.
10 9 8 7 6 5 4 3 2 1

Contents

Preface		xi
Acknowledgments		xvii
CHAPTER 1	Introducing IDEAScript	1
	Understanding Automation	1
	Understanding How You Use Macros	3
	Having Things Your Way	6
	Considering Your Skills	7
	Summary	8
CHAPTER 2	Creating Your First IDEAScript Application	9
	Understanding the Macro Types	9
	Opening the Visual Script Editor	11
	Writing a Hello World Application	15
	Building Your Application	19
	Summary	20
CHAPTER 3	Understanding the Basics of the IDEAScript Editor	21
	Working with Windows	21
	Hiding and Viewing Windows	27
	Working with Menus and Toolbars	29
	Summary	37
CHAPTER 4	Designing Structured Applications	39
	Understanding the Parts of an Application	40
	Understanding the Methods Used to Create an Application	41
	Using the Macro Recorder	45
	Working with Subroutines and Functions	52

Making Your Code Easy to Read 56
Adding Your Application to a Toolbar or Menu 57
Summary 60

CHAPTER 5 Working with Data 63

Understanding Variables and Constants 63
Choosing a Data Type 67
Employing Operators 82
Formatting Data 85
Creating Custom Data Types 89
Summary 91

CHAPTER 6 Using Conditional Statements and Loops 93

Making Decisions Using the If . . . Then . . . Else Statement 93
Choosing between Options Using Select Case 95
Performing Tasks a Specific Number of Times with For . . . Next 99
Performing Tasks Using Conditions with Do Loop 102
Adding Error Trapping to Your Application 106
Redirecting Macro Flow Using GoTo 112
Summary 112

CHAPTER 7 Understanding IDEA Databases 115

Considering the Parts of a Database 115
Introducing the IDEA Database System 118
Opening a Database for Use 119
Checking the Database History 120
Obtaining Field Statistics 122
Setting Database Criteria 125
Indexing a Database 125
Sorting a Database 127
Modifying Database Comments 129
Committing the Database 131
Closing a Database 133
Summary 133

CHAPTER 8 Working with Databases 135

Adding One Database to Another Using Append Database 135
Comparing Two Databases Using CompareDB 138

Working with Keys 140
Exporting a Database Using ExportDatabase 145
Working with Fields Using Field 150
Working with Records 152
Working with Tables 157
Summary 164

CHAPTER 9 Considering the CaseWare IDEA Object Model 165

Considering the IDEA Object Model 165
Working with the Task Object Model 170
Summary 193

CHAPTER 10 Performing Mathematical Tasks 195

Performing Basic Math 195
Using Advanced Math 198
Employing Analysis 200
Summary 205

CHAPTER 11 Interacting with Arrays 207

Understanding How Arrays Work 207
Creating and Using Arrays 208
Copying Data between Arrays 212
Summary 213

CHAPTER 12 Creating Interactive Dialog Boxes 215

Creating Great Dialog Boxes 216
Using the Basic Controls 218
Obtaining the Visual Appearance You Want 232
Interacting with Dialog Boxes Using Code 235
Adding Pictures to Your Dialog Boxes 244
Summary 247

CHAPTER 13 Locating Information in Databases 249

Performing Searches Efficiently 249
Using the Built-in Search Features 251
Creating a Custom Search 268
Summary 269

CHAPTER 14 Importing and Exporting Data 271

Considering the Import and Export Features 271
Performing Data Extractions 283
Managing PDF Data 290
Managing Text Data 294
Managing Excel Data 306
Managing Access Data 310
Summary 315

CHAPTER 15 Working with Files 317

Considering the File Format 317
Using the File IO Features 318
Using External Variables 346
Summary 348

CHAPTER 16 Working with Other Applications 349

Considering IDEAScript and Visual Basic for Applications (VBA) Differences 349
Understanding the Word and Excel Object Models 350
Running Word from IDEA 351
Running IDEA from Excel 355
Summary 359

CHAPTER 17 Performing Data Analysis Tasks 361

Performing Stratification 361
Performing Summarization 367
Creating a Pivot Table 371
Employing Random Record Sampling Using RandomSample 374
Performing Gap Detection 376
Checking Distribution Using SystematicSample 378
Merging Databases 380
Summary 389

CHAPTER 18 Working with Charts and Graphs 391

Choosing the Correct Chart or Graph 391
Creating a Basic Graph 393

Defining Analytical Charts 399
Summary 405

CHAPTER 19 Defining Reports 407

Defining a Report 407
Outputting Data in PDF Format 412
Outputting Data in Word Format 414
Summary 418

CHAPTER 20 Considering Database Security 419

Considering Programmatic Data Security 419
Choosing the Correct Data Type 421
Validating Data 421
Protecting Dialog Boxes 427
Summary 428

CHAPTER 21 Debugging Your Application 429

Understanding the Kinds of Application Errors 429
Running and Stopping the Application 433
Using Breakpoints 434
Stepping through the Application 435
Using the Watch Window 438
Relying on Message Boxes 444
Summary 445

CHAPTER 22 Performing Project Management Tasks 447

Creating a Plan for Your Application 448
Keeping Track of Application Files 449
Working within a Group 450
Documenting Your Application 451
Summary 453

CHAPTER 23 Converting Visual Script to IDEAScript 455

Considering the Benefits of Using IDEAScript 455
Performing the Conversion 456
Making Changes and Saving the Result 458
Summary 461

About the Author 463

About the Website 465

Index 467

Preface

Most people want to perform tasks faster because, let's face it, time is precious and you'd much rather spend your time doing something other than sitting at your desk waiting for the computer to complete a task. That's where IDEAScript comes into play. By using IDEAScript, you can automate tasks. You can tell the computer to accomplish tasks and let you know when the work is finished. *Mastering IDEAScript: The Definitive Guide* is your window to IDEAScript. It helps you understand how IDEAScript works and how you can use it to do amazing things with IDEA, all without having to sit at your desk to observe the computer doing it. If that sounds interesting, read on!

About This Book

Mastering IDEAScript: The Definitive Guide is designed to help the complete novice develop the skills required to write simple applications using IDEAScript. The overall goal of this book is to make it possible for you to automate all of those tasks that you used to perform manually. Of course, you still have to start the task and interpret the results—even the best automation can't do that for you.

Automation is one of those terms that's used a lot, but is never quite explained by anyone. For the purposes of this book, automation means that you'll be able to write an application that follows a procedure you create—the same procedure you use every day to perform tasks. It's just that simple. Instead of you spending time filling in forms and answering questions, you give all the required information to the computer as part of your application and let it do the work. The result is that you become more productive and spend less time sitting in that chair bored stiff.

The techniques in this book go further, though. Let's say that you have an assistant and want the assistant to help with some of the work. You can write an application that asks the assistant very simple questions and then automates the rest of the task for the assistant so you don't have to help as much. The result is that your assistant also becomes more capable and efficient. By using forms and other techniques described in this book, you make it possible for less-skilled helpers to perform a task using the same approach you do, making the result look the same as if you had done it.

IDEAScript provides a wealth of capabilities and *Mastering IDEAScript: The Definitive Guide* tells you all about them. For example, you might want to remotely control an external application without having to work too hard to do it. Using IDEAScript, you

create the code required to perform the manipulation one time, and then let the computer perform the task for you from then on.

Finally, this book tells you about a few unique tasks you can perform using IDEAScript. Do you have an external file that doesn't quite want to import using the normal techniques provided by IDEA? Well, you can define an application using IDEAScript that makes it possible to import just about anything into IDEA. This book tells you how to accomplish this kind of task.

How This Book Is Organized

This book discusses IDEAScript starting from simple topics and moving on toward more complex topics. The initial topics also focus on tasks that you perform more often. As you progress through the book, you start to discover tasks that are less used, but extremely useful in many situations. In fact, this book may present you with some new ways of accomplishing tasks that you hadn't considered when using the GUI. Here's a list of the chapters in this book:

- **Chapter 1: Introducing IDEAScript:** This chapter helps you understand what automation can help you do. It presents you with the concepts behind using macros and helps you set reasonable goals for working with IDEAScript.
- **Chapter 2: Creating Your First IDEAScript Application:** This chapter begins by helping you understand the two kinds of macros you can create with IDEA: IDEAScript and Visual Script. It then presents you with the editors used to create each macro type. This is also the first chapter where you write an application—something very basic that you can use as a starting point for other applications in the book.
- **Chapter 3: Understanding the Basics of the IDEAScript Editor:** This chapter provides you with the details of using the IDEAScript Editor.
- **Chapter 4: Designing Structured Applications:** This chapter helps you understand the basic parts of an application. You discover how to write code quickly and efficiently by copying it from the help file or from IDEA's history. In addition, you learn how to use the Macro Recorder to create complete applications without writing any code at all. Finally, this chapter shows you how to add your application to an IDEA toolbar or menu so that you can access it quickly.
- **Chapter 5: Working with Data:** This chapter helps you understand what variables and constants are (essentially they're a sort of storage container) and how to use them within your application. This chapter introduces data types, a method of categorizing data stored in variables and constants. Finally, you discover some basic techniques for working with variables and constants.
- **Chapter 6: Using Conditional Statements and Loops:** This chapter begins building on the basic structures you learned about in Chapter 4. In this case, you learn how to perform tasks conditionally using any criteria you want and how to perform tasks multiple times (either a specific number of times or until the application meets certain specifications). This chapter also helps you understand what to do about errors that occur in your application.

- **Chapter 7: Understanding IDEA Databases:** This chapter introduces databases from an IDEAScript perspective. You discover how databases are put together and learn how to open databases for processing. After the introductory material, this chapter provides you with the information you need to perform basic tasks, such as indexing, sorting, and closing your database.
- **Chapter 8: Working with Databases:** This chapter takes the next step after basic database management. You discover how to perform some intermediate level tasks, such as comparing databases and exporting them. Finally, this chapter shows you how to work with fields, records, and tables in a database.
- **Chapter 9: Considering the CaseWare IDEA Object Model:** This chapter introduces you to the concept of objects, which are a representation of something in your application. Think of objects as they appear in the real world and you have the basic idea. Once you have the basics of objects down, this chapter introduces you to a number of IDEAScript objects, especially those used to perform tasks.
- **Chapter 10: Performing Mathematical Tasks:** This chapter helps you understand how to perform math tasks using IDEAScript. You'll also see details on using a couple of the analysis-related tasks provided by IDEA.
- **Chapter 11: Interacting with Arrays:** This chapter demonstrates how to use arrays, which provide a method of storing like or associated values together for easy access.
- **Chapter 12: Creating Interactive Dialog Boxes:** This chapter presents techniques for working with complex dialog boxes. IDEAScript lets you create dialog boxes of any complexity so that you can ask the user questions, have the user fill out forms, or interact with the user in other ways. As part of this chapter, you learn how to use the various graphical elements used to create dialog boxes.
- **Chapter 13: Locating Information in Databases:** This chapter helps you learn how to find information within databases. These search techniques can make it quite easy to find any data within the database, even if you aren't quite sure what you're trying to find. IDEA makes it quite easy to perform complex searches without a lot of work on your part—IDEAScript makes things even easier by automating some tasks associated with common searches.
- **Chapter 14: Importing and Exporting Data:** This chapter shows you how you can obtain data from other applications and send IDEA data to other applications. The ability to import and export data is essential in today's world of connected computers. You want to have a number of solutions available to make data accessible for further analysis or for sharing with other people.
- **Chapter 15: Working with Files:** This chapter describes how to work with external files of all types. You can use external files to store other information or import data from external files that IDEA might not support directly. In addition, you can use external files to hold configuration information or even use them to log application errors. In short, external files are really important when creating moderately complex IDEAScript applications.
- **Chapter 16: Working with Other Applications:** This chapter describes how to make other applications work with IDEA in a number of ways. In this chapter, you discover specifically how to use IDEA with both Microsoft Word and Microsoft Excel, but the techniques shown will work with other applications, too. Once you

complete this chapter, you can create multi-application scenarios and automate data manipulation tasks even further.

- **Chapter 17: Performing Data Analysis Tasks:** This chapter discusses advanced database manipulation tasks that you may not use very often, but will find essential to accomplish certain goals. The examples in this chapter discuss advanced database manipulation tasks such as working with pivot tables and finding gaps within data sequences.
- **Chapter 18: Working with Charts and Graphs:** This chapter describes how to create graphical presentations of data within a database. Graphics are exciting and usually the best way to present complex information. In addition, by using charts and graphs, you can help others see patterns in data and present a specific message with your data that isn't possible using other means.
- **Chapter 19: Defining Reports:** This chapter describes how to create nicely formatted reports using database data. In many cases, the way you present data will affect how the viewer receives it, so this chapter is essential if you want to create output with a certain level of fit and finish.
- **Chapter 20: Considering Database Security:** This chapter discusses the difficult topic of security in a very simple manner. When you finish this chapter, you'll be able to create a relatively secure application without a lot of work. Because security is such an important topic today, you'll want to read this chapter if you plan to share your application with anyone else or if you routinely work on sensitive data.
- **Chapter 21: Debugging Your Application:** This chapter presents methods for finding errors in your application and fixing them. Most applications have errors at some point in their lifetime. Fixing these errors makes the application more reliable, easier to use, more efficient, and definitely a pleasure to work with as well.
- **Chapter 22: Performing Project Management Tasks:** This chapter helps you understand how to work with your application in a larger company environment. In many cases, you'll find that your application becomes popular after you show it to other people. In other cases, you might find yourself working with other people to create a database management solution in the form of an application. It's important to know how to work with IDEAScript in a group environment.
- **Chapter 23: Converting Visual Script to IDEAScript:** This chapter demonstrates how to convert your Visual Script macro into an IDEAScript macro. Visual Script can be a little limiting because you don't have good control over every feature the application can do. Using IDEAScript is more flexible and makes it possible to perform advanced database management.

What You Need to Use This Book

This book doesn't assume that you have any knowledge about programming or have any programming skills. In fact, it assumes that you don't have either programming knowledge or skills. However, you do need to know the basics of working with Windows and you must at least be familiar with using IDEA. This book doesn't tell you how to

perform tasks such as working with a mouse and it assumes that you know how to use IDEA to perform at least simple tasks such as extracting a database.

You won't need any special equipment or software to use this book. All you need is your copy of IDEA. If you want to perform some advanced tasks, such as working with other applications, you do need the other applications. For the purposes of this book, if you have a copy of Microsoft Excel and Microsoft Word handy, you'll be able to work with the examples in Chapter 16.

Conventions Used in This Book

This book doesn't use many conventions. It emphasizes simple text for most purposes. You'll see special terms in *italics* on first use, followed by a definition of that term. Website URLs, all code, file/folder names, and file/folder locations appear in `Courier New` font type. In procedural text, **Bold** is used for user interface elements and text you need to enter. This book also provides three special kinds of text as follows:

- **Notes** are additional information that doesn't fit within the flow of text. You might find the information useful because it augments the information found in the paragraphs. In some cases, notes provide sources of additional information or help you understand a concept more clearly. In almost every case, you can skip a note without losing any essential information, but whenever possible, stop to read the notes to get the special information they contain.
- **Tips** provide you with insights on how to do something more efficiently, faster, or with less work. Often, tips provide best practices for working with IDEAScript. Even though tips aren't essential reading, you'll want to read tips whenever possible to get the most out of this book and IDEAScript.
- **Warnings** tell you about things you should avoid doing. In many cases, warnings tell you about situations that will cause data loss or at least cause your application to crash. You should always pay special attention to warnings. In fact, you should note warnings that especially affect you and go back to them later when writing your application. Never skip warnings.

In addition to these three special kinds of text, this book does use a special format for code. You'll see both code snippets (short sections of incomplete code) and code listings (longer sections of complete examples) in the following format:

```
' This is a comment.
MsgBox "This is code"
```

Acknowledgments

Thanks to my wife, Rebecca, for working with me to get this book completed. I really don't know what I would have done without her help in researching and compiling some of the information that appears in this book. She also did a fine job of proofreading most of my rough draft.

Matt Wagner, my agent, deserves credit for helping me get the contract in the first place and taking care of all the details that most authors don't consider. I always appreciate his assistance. It's good to know that someone wants to help.

Finally, I would like to thank Andrew Coles, Vanessa Muckleston, Andy Sloman, Christine Dahlgren, and the rest of the editorial and production staff for their assistance in bringing this book to print. It's always nice to work with such a great group of professionals.

CHAPTER 1

Introducing IDEAScript

You've just completed the same analysis for the fiftieth time and wonder if there isn't an easier way to get the job done. Yes, using IDEA is fast and easy, but there must be a way to make things faster still. Of course, you can try to find some way to improve your own efficiency or try to perform the analysis a few less times, but there are limitations to that approach and they often require additional work on your part. Why not have someone else, or more importantly, something else, do the work for you? That's what scripting is all about. This book tells you everything you need to know in order to make the computer work for you, rather than you work for it. In this chapter, you discover just how much benefit you can obtain by spending a few hours learning to tell the computer what to do. As you go through this chapter, you learn how to:

- Understand how automation can make working with IDEA easier.
- Consider the ways in which you can use automation.
- Decide which forms of automation to pursue first.
- Determine how your skills can help you use automation best.

Understanding Automation

The computer community will use all kinds of technical terms you don't understand to describe scripting. In fact, the word scripting itself sounds foreign and technical. What this book really describes is automation, and you use automation every day. When you go to the gas station and fill your car with gas, you're using automation. After all, you don't have to pump the gas from the storage tank yourself—you let the gas pump do the work. When you go to the store, the cashier uses a cash register and scanner to total the amount of money you owe for food—no one uses pen and paper any longer. The cashier is employing yet another kind of automation. You get home and click a button—the garage door opens. The garage door is yet more automation. In fact, it won't take long for you to find automation everywhere in your life. Why not automate IDEA as well?

All forms of automation rely on some kind of control. When you pump gas, you press buttons or tell the gas pump to begin pumping in some other way. Controls inside the gas pump automatically stop the flow of gas. At the store, the act of clicking a few keys on the

cash register and scanning the items provides control over the adding process. The garage door opens when you click a button on its control. Likewise, *scripting* is a form of control over IDEA that you exercise using special words and phrases. As you can see, scripting isn't anything new—you've already been exercising control over things all your life.

Scripting is a little more complex than pumping gas, scanning groceries, or opening a garage door, but it also does a lot more for you. The complexity comes in the form of a procedure you must write. Of course, you've already been doing that task for a long time too. Any time you have someone stay at your house to water the plants or ask a coworker to perform a task, you write a procedure for them—you tell them what you want them to do and when to do it. Automating tasks in IDEA is no different. You use control words to write a procedure that IDEA performs for you. This procedure is called a *macro* and you use macros to tell IDEA how to automate tasks for you. The following sections describe the benefits of automating tasks in IDEA in more detail.

How Does Automation Benefit You?

The main reason you're reading this book is to gain a new skill that benefits you in some way. After all, why bother to learn something that isn't going to help you in some way? The following list outlines the benefits you should consider as you read this book:

- You can perform work faster.
- The results you obtain will contain fewer errors.
- Any analysis is performed more consistently.
- Your work becomes more interesting because you can focus on unique tasks.
- You don't have to remember how to perform complex procedures because the procedure is contained in the macro.
- It's easy to justify actions you take based on the consistency of your macros.

Tip

There are many ways in which learning to script will benefit you that this book can't cover. For example, if you know how to script and none of the other people in your organization do, you'll likely find that your job security is greater and you'll receive promotions more often. Many people are afraid of scripting, but you're brave enough to give it a try. You'll find that scripting is actually quite easy and straightforward as the book progresses.

How Does Automation Benefit Others?

Believe it or not, your new skill will also benefit others. When you know how to create macros, you become an important asset to others who don't know how to perform this task or simply want to benefit from what you've learned. The following list outlines the benefits others will receive from your macros:

- People can use your macros to obtain the same benefits you obtain.
- Your organization can perform analysis in a consistent fashion, making the analysis easier for everyone to understand.
- The reports and other output you generate will make it easier to see trends.
- It's possible to create a *workflow* (a standard method of performing a task) for the entire organization.
- A single employee absence won't mean that work stops.

Best Practices for Using Automation

Given the benefits of automation, you may be tempted to use automation all the time. However, automation isn't always the answer; you must use some discretion in employing automation. For example, you wouldn't want a completely automated plane—having a pilot is important for safety reasons. The following list provides best practices you should follow when considering automation:

- Always choose tasks that you'll repeat. The more often you need to repeat a task, the better a candidate it is for scripting.
- Always choose well-defined tasks. In order to write a procedure, you must understand the task completely.
- Always plan your macros carefully and completely so that the procedure works as you expect it should. The planning process begins when you separate tasks that will automate well from those that won't.
- Whenever possible, create macros that everyone in your organization can use, rather than focus on macros for personal needs. When everyone benefits, the time you use to write the macro is paid off faster.
- Whenever possible, write down the procedure you use and then test the procedure carefully. This act is no different from any other automation you use. For example, you'd expect that a cashier would receive training that relies on written and tested procedures.
- Avoid writing macros that are too complicated for your current skill level. Discover scripting a step at a time. As this chapter progresses, you'll learn tricks you can use to avoid getting in over your head.
- Never assume that the macro you write for your machine will work on another machine until you test it on that machine. Just as a procedure for one cash register may not work on another, you can't assume your macro will work on every machine. As the book progresses, you'll discover methods for testing your macros to ensure they work as anticipated.

Understanding How You Use Macros

Macros, the written procedures used for scripting, can perform all kinds of tasks. In fact, the number of tasks you can perform is literally limited only by your imagination. Some

people have written games and done all kinds of other interesting things with macros. Of course, most people use macros for more practical purposes. The following sections describe some of the common tasks you can perform with macros.

Interacting with Databases

The task you perform most often in IDEA is interacting with a database of some sort. From your perspective, you're performing a data analysis. However, from the perspective of the IDEA application, you're manipulating data found in databases. Whatever the perspective, being able to get the data you need is important, and it's often repetitious. Macros that help you get the information you need are probably the singular most important kind of macro that you can write.

Starting with Chapter 7, you begin working with databases and discover that no matter the source, databases often have similar needs and requirements when writing a macro. Chapter 8 shows you how to interact with databases, while Chapter 9 describes the model IDEAScript uses to work with databases. When working with IDEAScript, you can access two kinds of databases:

- Internal IDEA databases
- External databases such as SQL Server

This book helps you work with both kinds of databases. The external database information starts in Chapter 14 and you see some advanced techniques in Chapter 17. In fact, you'll find that you can access data in all its forms, even an Excel document (see Chapter 16) or a text file (see Chapter 15) on your hard drive. Procedures that might seem complex when you perform them by hand suddenly become easy and fast when you use a macro to perform them.

Customizing the IDEA Interface

You can attach (bind) your macros to the IDEA interface. By adding buttons that access your macros, you can customize the IDEA interface to meet your specific needs. Your macros, in essence, become part of the IDEA application and make using IDEA easier. Chapter 4 tells you how to add your macros to the IDEA application interface.

Performing Calculations

Analysis normally includes performing comparisons and employing equations to calculate specific values. Of course, you want to be sure you perform the right comparisons and obtain the correct calculations. Fortunately, your computer is far faster and significantly more accurate in both comparisons and calculations, so this is one area that you really should let the computer take care of for you. Chapter 6 tells you how to perform comparisons, while Chapter 10 addresses math requirements.

Designing New Application Features

Binding macros to the IDEA interface isn't the only kind of customization you can do. In addition, you can create your own interface elements as dialog boxes. You can use dialog boxes for two purposes:

- To output information to the user.
- To obtain input from the user.

You can do everything from telling the user of your macro the status of a calculation to asking the user for the name of the database they want to use for analysis. Chapter 12 shows you how to create interactive dialog boxes that let you do amazing things.

Importing and Exporting Data

Most businesses need to exchange data in some form or another. Depending on the kind of analysis you perform, you might have to obtain data from many different businesses. Unfortunately, businesses use different applications and different methodologies to store information. Trying to remember all of the methods used to access this information can prove daunting. Fortunately, macros make fast work of importing and exporting data as needed. Chapter 14 describes the resources that IDEAScript provides for importing and exporting data.

Data comes in many forms. Normally, you'll work with databases, but that isn't always the case. Besides the database chapters in this book, you can discover how to interact directly with files in Chapter 15 and Excel in Chapter 16.

Controlling Other Applications

One of the most useful ways to use macros is to control other applications. It's inconvenient and time wasting to have to interact with more than one application at a time. If you can perform at least part of that work by using a macro, you save time and can maintain a focus on IDEA.

Nothing limits the kinds of applications you can control. If you want to start a copy of Excel and use it to create a chart or graph, you can do so with the information found in Chapter 18. Chapter 19 shows how to control other applications, such as Word, to create reports.

Warning

Never execute macros unless you know what task the macro performs and that the macro is safe to use. Executing a macro that you don't know about can cause damage to your data or produce unreliable results. In addition, macros can cause significant problems on your machine, such as installing a virus. The macros that are completely safe are the ones you understand and obtain from a reliable source.

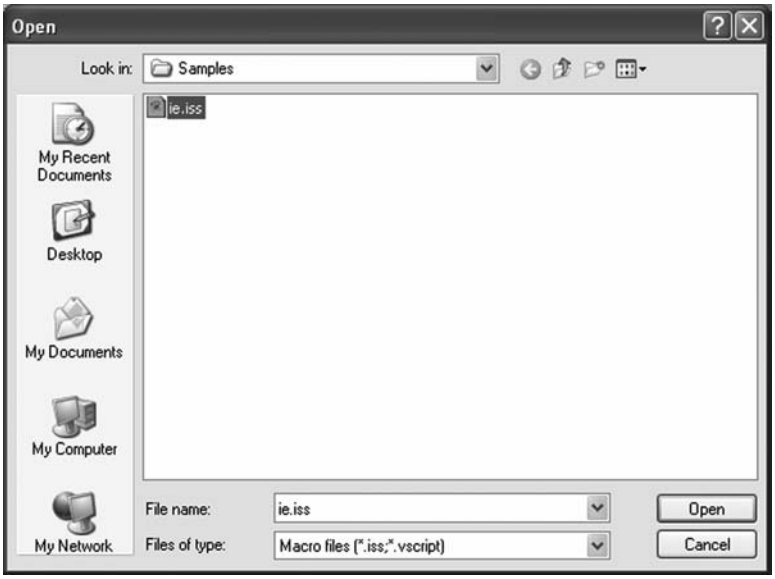


FIGURE 1.1 The Open Dialog Box Lets You Choose a Macro File on Your Hard Drive

You're probably thinking that controlling another application sounds too hard, but it really isn't. The first exercise in this book is to execute a macro that starts a copy of Internet Explorer. This macro actually comes with your copy of IDEA. Just follow these steps.

1. Open IDEA and select **Tools** > **Macros** > **Open**. This command displays the **Open** dialog box shown in Figure 1.1. IDEA automatically selects the `Samples` folder for you, which contains a macro named `ie.iss`. If you don't see the `Samples` folder, you can find it by locating `\My Documents\IDEA\Samples` in the **Look in** field.
2. Select `ie.iss` and click **Open**. IDEA opens the **IDEAScript Editor** shown in Figure 1.2. Don't worry about the editor for now; Chapter 2 tells you how to work with it. The text you see in the right **Editor** pane is a macro and we're going to execute it.
3. Click **Run Script**. The **Run Script** button is the blue right-pointing arrow on the toolbar. You can also press **F5**. IDEA executes the macro and opens a copy of Internet Explorer for you. Congratulations! You just executed your first macro.
4. Close Internet Explorer. Select **File** > **Exit** in the **IDEAScript Editor**.

Having Things Your Way

For many people, the idea of scripting can become overwhelming. At first, you can't quite accept that you can actually write macros, but then, once you get used to the idea, all kinds of macro ideas start coming to mind. It's nice to have things your way. Once

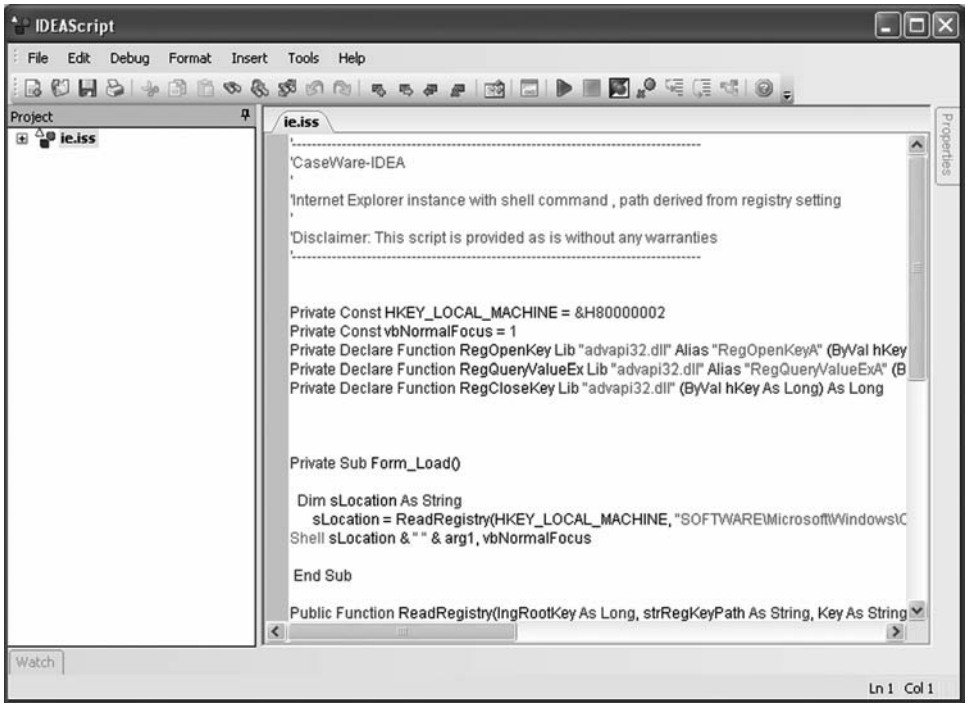


FIGURE 1.2 The IDEAScript Editor Lets You Write Macros That You Can Save to Disk

you discover the full capability of scripting, you'll find that you can do a host of things that you didn't think were possible in the past.

Of course, that list of ideas can become a burden, too. Make sure you write your ideas down because it's all too easy to forget something you want to do. Even if you don't know how to write the macro today, write it down. As you work with IDEAScript, your skills will improve and today's impossible task will become quite possible tomorrow. Make sure you prioritize your list. Use these criteria for prioritization:

- **Skill Level:** Your skill level determines the macros you can create today.
- **Existing Knowledge:** Macros that play to knowledge you already possess, say a math macro if you're already a math expert, should be a priority.
- **Pressing Need:** Personal or organizational needs can act as a great motivator to finish the macro. Some people try for a short time and then give up—finishing the macro is the only way to build the knowledge you need.
- **Interest:** Some projects are definitely more interesting than others. It's more likely that you'll finish a macro that interests you, so be sure to tackle these macros first.

Considering Your Skills

Even though macro writing is like many other things you've already done and is basically writing a procedure for IDEA to follow, it's still a skill. As you write more macros, you'll

learn more about the IDEAScript language and be able to create procedures that are more complex than those you create at first. The best way to learn scripting is to start slowly and discover new commands one at a time until you become proficient.

Some of the skills you already possess will help as you discover IDEAScript. For example, if you already have solid math skills, you'll find that writing macros that perform math tasks is significantly easier. You may even want to focus on math-related macros when you first begin scripting. Some people already know quite a bit about databases, so working on database-related macros is easier. Don't force yourself to start out with something too difficult—ease into scripting.

You should create some goals for yourself based on the scripting needs you discover as you work with IDEA. Put these goals on your To Do list—the same as you would anything else you want to learn. When you have a little additional time or you're waiting for another task to complete, take some time to learn a new IDEAScript command and then begin employing it in your macros. It won't be long before you'll be writing complex macros without any trouble at all.

No one's asking you to memorize anything. The purpose of this book is to act as your memory. As you work through the book, you'll discover that IDEA provides a number of other useful aids to make writing macros easier. Writing macros should be something you do to improve your work experience, not ruin your mood.

Summary

This chapter has started you on the road to a new kind of experience—scripting. The most important idea to take from this chapter is that anyone can write a macro as long as they fully understand the task at hand. While not every task is suitable for automation, many tasks are and you should make full use of this capability in IDEA to reduce your workload. The macros you create help both you and everyone else in your organization, so writing good macros is essential.

One of the most important aspects of using automation is to employ it correctly. Of course, only you can decide when automation applies. Before you go to the next chapter, consider a few places in which automation will help you and your organization. Using the information in this chapter, write down the pros and cons of using automation for the tasks you define. Present your list and reasoning to other people and see if they agree that automation is the right choice for the tasks you list. This exercise will save you considerable time trying to automate tasks that you really shouldn't automate.

Now that you have a list of tasks you want to automate, Chapter 2 takes the next step and begins to show how to create macros. Of course, your first macros will be very simple. You want to make scripting fun and easy to perform, so these initial steps are important. The macro in Chapter 2 is functional and you can even show it off to your friends. However, you'll create significantly more interesting macros as the book progresses.

CHAPTER 2

Creating Your First IDEAScript Application

An IDEAScript *application* is a group of one or more macros that perform a specific task, such as a particular kind of analysis on a database. The application can contain dialog boxes and other user interface features that make it possible to interact with the user and create a more flexible result. You can even compile your IDEAScript application to execute it at the command line without ever opening IDEA.

Many people experience a mental block when they attempt to perform some tasks. For example, most people have a hard time starting to write something when faced with a blank page. Writing an application can be the same way. Many people look at the blank editor screen and simply don't know what to do next. Just as book authors use a number of tricks to avoid the blank page, you can use some tricks to avoid the blank editor screen. This chapter helps you get started by showing you how to create simple applications. Often, you can start with a simple application and keep adding to it until the resulting application does everything you need.

Before you can create a macro, however, you need to know how to perform tasks such as opening the editors and interacting with them in a meaningful way. In fact, you will learn that IDEA actually provides two different editors and you need to decide which editor to use for your application. IDEA also lets you create an *executable program*—one that you can start from within Windows without starting IDEA. Executables are really interesting because they let you start macros without starting the editor every time.

Understanding the Macro Types

A *macro* is always a method of creating a procedure using special control words. You execute the macro by telling IDEA to perform the procedure. However, IDEA provides two different methods for creating a macro. The first, IDEAScript, is more powerful and flexible because it lets you work with the control words directly. The second, Visual Script, is easier because you tell IDEA what you want to do using a graphical interface

and then IDEA creates the control words for you. The following sections describe each macro creation method in more detail.

Considering IDEAScript

IDEAScript provides you with the ultimate in flexibility when creating macros for IDEA. When working with IDEAScript, you can control the entire IDEA object model, work with dialog boxes, interact with the operating system, and even control other applications. However, with this kind of power comes a certain level of complexity. Don't let the complexity overwhelm you—take things slowly and you'll find that it's quite manageable. After all, everyone who's ever written a macro has started out knowing nothing about performing this task. Even so, it will require a little time to learn how to use IDEAScript.

You select IDEAScript when you want the best control over your application and need to perform significant tasks. IDEAScript is the right tool for anyone who has time to learn the special control words used to communicate with IDEA. Anyone reading this book falls into that category. You've already shown your interest in having the additional power over IDEA by reading this book.

Considering Visual Script

Visual Script provides a graphical method of creating a macro. You essentially describe what you want to see and IDEA provides it for you. Visual Script is very good for creating a macro for repeating tasks. For example, you can use it to create a macro that imports a file, runs some tasks to analyze data, and then provides results where the auditor takes over.

Using Visual Script is incredibly easy, but it also limits what you can do in a significant way. For example, you can't create a message box describing the result of executing a task directly. You save Visual Script macros as `.vscript` files.

One of the more important Visual Script limitations is that you can't run a Visual Script macro on IDEA Server. IDEA Server allows the users to connect to a server and run tasks on a server farm. Consequently, Visual Script isn't a very good solution for larger enterprises.

It's possible to use Visual Script as a method for starting an IDEAScript application if you want to completely avoid the blank page issue. All you do is create the basics of your application using the Visual Script Editor and then convert the Visual Script macro to an IDEAScript macro. Chapter 23 explains how to convert Visual Script macros to IDEAScript macros.

You aren't completely without resources when working with Visual Script. For example, you can use macros (either IDEAScript or Visual Script) that you or someone else creates using IDEAScript. Consequently, it's possible to use Visual Script as a means of gluing modules together—to hide complexity from view. With the right coding technique, you can obtain the simplicity of Visual Script matched with the flexibility of IDEAScript.

Note

You can't use macros that require any sort of input information other than the information that's available from a database or other external source. When you insert a macro in Visual Script, the macro is executed in its entirety. Consequently, any macros you use within Visual Script must be self-contained modules (black boxes) that require nothing on the part of the individual using it.

Visual Script may not be the option of choice for you, but it could be an option for less skilled workers. You can create IDEAScript modules that others put together as needed to perform tasks in your organization. Consequently, you shouldn't view Visual Script from the limited perspective of a personal tool.

Opening the Visual Script Editor

Before you can create a Visual Script macro, you must open the associated editor. Start IDEA and then select **Tools** > **Macros** > **New** > **Visual Script**. You see the **Visual Script Editor** shown in Figure 2.1.

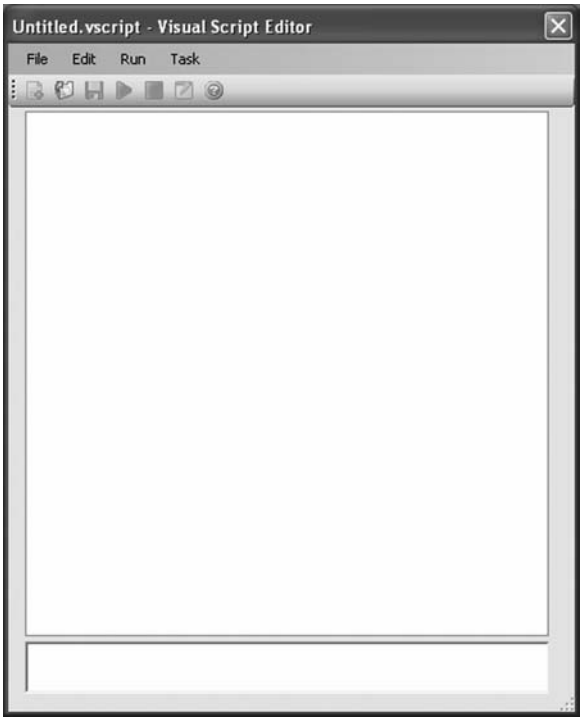


FIGURE 2.1 The Visual Script Editor Lets You Create Visual Script Macros

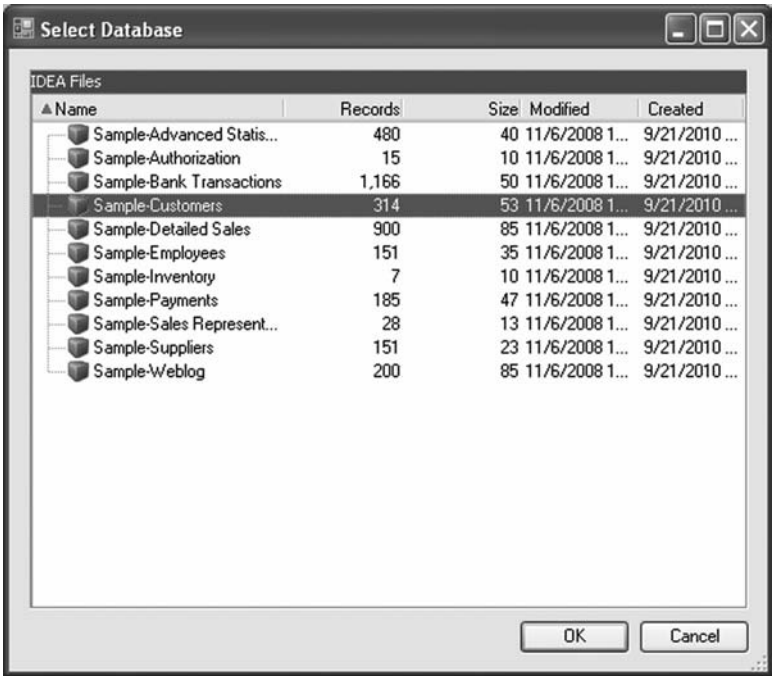


FIGURE 2.2 Choose a Database You Want to Use

Unfortunately, the display looks a bit blank, but that's easily fixed. To begin a new macro, simply open a database or select a macro you want to execute. For the purpose of this example, let's assume you want to find all of the customers in the `Sample-Customers` database that have a credit limit equal to or greater than $10,000.00. The following steps show you how easy it is to create the macro.

1. Select **Task** > **Insert** > **File** > **Open Database**. You'll see the **Select Database** dialog box shown in Figure 2.2. This dialog box shows all of the databases that you can access, which are the sample databases when you first install IDEA (found in your current working folder).
2. Select the database you want to use (`Sample-Customers` in this case) and click **OK**. IDEA closes the dialog box and adds an entry for the database to the **Visual Script Editor**. Now that you have a database open, you can do something with it. For the purposes of the example, we'll create an indexed extraction.
3. Select **Task** > **Insert** > **Data** > **Extractions** > **Indexed Extraction**. You'll see the **Indexed Extraction** dialog box shown in Figure 2.3 (which has already been filled out in the screenshot).
4. In the **Field** box, select **CREDIT_LIM**. In the **Value is** fields, select >= and then type **10000**. In the **File name** field, type **10K Customers**. Each of these actions defines part of the task you want performed on the `Sample-Customers` database. In this

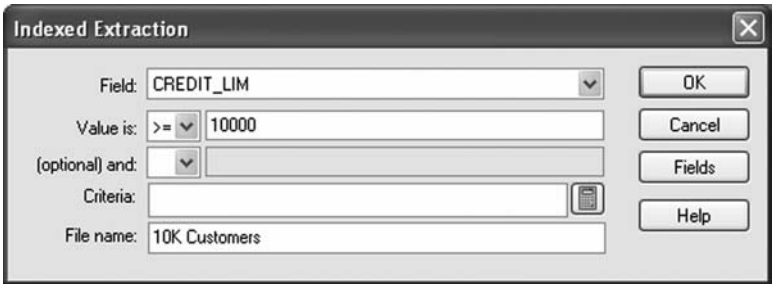


FIGURE 2.3 Tell IDEA What You Want to Do with the Database

case, you're telling IDEA to extract all of the records in the `Sample-Customers` database that have a value equal to or greater than 10000 in the CREDIT_LIM (credit limit) field and place them in the `10K Customers` database.

5. Click **OK**. IDEA will display a dialog box asking whether you want to perform the task now. Click **No**. We'll run the task later. Now, at this point, you'll probably want to close the `Sample-Customers` database because you don't need it anymore.
6. Select **Task** > **Insert** > **File** > **Close Database**. Again, IDEA will ask whether you want to perform the task now. Click **No**. At this point, you should have a procedure completed like the one shown in Figure 2.4. That's all a macro is really—a procedure that you define for IDEA to perform.

Note

The reason the task message appears is that you may need to program a task based on the output of a previous task. This feature allows you to selectively run only a single task at a time—something that you can't do using IDEAScript.

Let's try executing the macro. Select **Run** > **Run** and you'll see the `Sample-Customers` database open, IDEA will create the required extraction, and then you'll see the `Sample-Customers` database close. At the bottom of the **Visual Script Editor** window, you'll see "The Visual Script has completed without any errors."

To save the macro so you can view it later, select **File** > **Save**. You'll see the **Save Visual Script As** dialog box. Type a name, such as **10K Customers**, in the **File name** field and click **Save**. Select **File** > **Exit** to close the **Visual Script Editor** window. To see the results of the macro, double-click the `Sample-Customers\10K Customers` database entry in the **File Explorer** window. You'll see the output shown in Figure 2.5.

Of course, the question now is whether the macro actually produced code. If you convert this macro using the process described in Chapter 23, you'll get the macro shown in Listing 2.1.

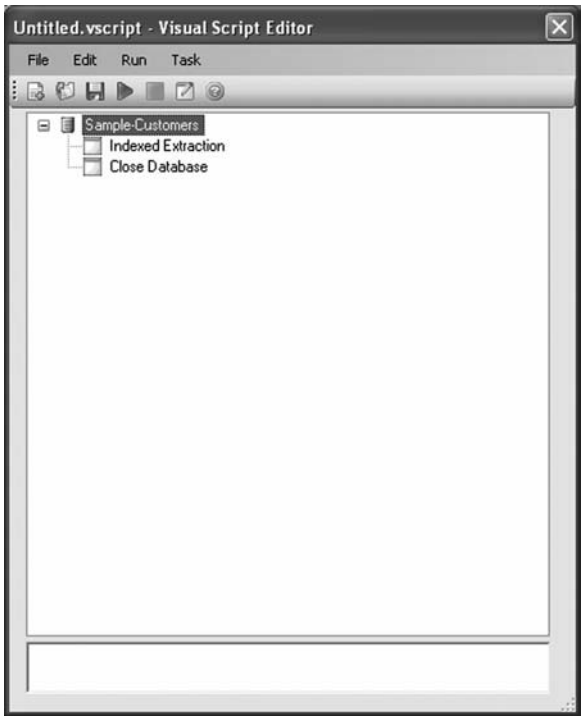


FIGURE 2.4 The Completed Macro Is Simply a Procedure

10K Customers.IMD

	CUSTNO	COMPANY	FIRST_NAME	LAST_NAME	COUNTRY	STATUS	CREDIT_LIM
1	10000	Timekeepers	MARIU	EUGENIA	ARGENTINA	A	10000
2	10201	Sanford Fine Jewels	CHABIRAJI	SAWYER	SOUTH AFRICA	A	10000
3	20035	Krysstal Jewels	OLAV	HARALDSSON	FAROE ISLANDS	A	10000
4	20273	Toon Town Watches	NANCY	SOUFFLOT	NEW ZEALAND	A	10000
5	20849	Exclusive Malaysian Jewels	YASMINE	YUSOFF	MALAYSIA	A	10000
6	21055	Cheap Jewellery	RACHEL	GUSTAV	AUSTRIA	A	10000
7	21089	Fifth Avenue Jewellery	DEVENDRA	PADE	GREENLAND	A	10000
8	21450	Your Jeweller's Jewellery and Gift Shop	SIMONE	MERRIOTT	JAMAICA	A	10000
9	42003	Classy Rings & Things	KIMBERLY	FOLEY	U.S.A.	A	10000
10	50003	Buenos Aires Jewelry	CLEMENTINE	MENDOSA	ARGENTINA	A	10000
11	21254	Finest Quartz Watches	PHILIPP	STAHELIN	NORWAY	A	11000
12	40134	Fine Cut Jewelry	CAROL	DESPATIE	BARBADOS	A	11000
13	10102	Johnson Bancock Fine Collectibles	JENNIFER	DE FREITAS	SOUTH AFRICA	A	12000
14	20462	The Watch Guy	RONALD	PAULSEN	U.S.A.	A	12000
15	20820	Argentinian Estate Jewellery	JULIO	DALIE	ARGENTINA	A	12000
16	20845	Malaysian Fine Jewels	NAMIKO	ABE	MALAYSIA	A	12000
17	20861	Happy Corner Watches	KENNY	LIM	MALAYSIA	I	12000
18	21341	Tang's Jewellers	LAWRENCE	SIA	SINGAPORE	A	12000
19	21466	Copenhagen's	MUHI	GROZDANIC	DENMARK	A	12000
20	61300	Coisas Preciosas	PETER	BORGES	BRAZIL	A	12000
21	92321	Fabuleux	MARTIN	SIMARD	FRANCE	A	12000
22	10203	Ananzi Watches	KATHARINE	BURROWS	SOUTH AFRICA	A	13000

FIGURE 2.5 Seeing the Output Tells You That the Macro Worked

LISTING 2.1 Output of the 10K Customers Macro

```
Sub Main
   Call IndexedExtraction()     'Sample-Customers.imd
End Sub

' Data: Indexed Extraction
Function IndexedExtraction
   Const WI_IE_NUMFLD = 1
   Const WI_IE_CHARFLD = 2
   Const WI_IE_TIMEFLD = 3

   Set db = Client.OpenDatabase("Sample-Customers.imd")
   Set task = db.IndexedExtraction
   task.IncludeAllFields
   task.FieldToUse = "CREDIT_LIM"
   task.FieldValueIs2 WI_IE_GTEQUAL, 10000.00, WI_IE_NUMFLD
   dbName = "10K Customers.imd"
   task.OutputFilename = dbName
   task.PerformTask
   Set task = Nothing
   Set db = Nothing
   Client.OpenDatabase (dbName)
End Function
```

It isn't important that you understand the code shown in Listing 2.1 right now. In fact, after you've discovered more about IDEAScript, you can come back to this listing and read it with ease. What's important is to know that creating a macro using the Visual Script Editor produces real code that you can modify later.

Writing a Hello World Application

You've probably wanted to write your first IDEAScript macro from the beginning of this chapter. Well, the wait is over. To begin this section, select **Tools** > **Macros** > **New** > **IDEAScript**. You'll see the **IDEAScript Editor** shown in Figure 2.6.

Notice that IDEA automatically starts a macro for you so that you don't face a blank page. The keyword *Sub* is short for subroutine. A macro consists of one or more subroutines. You must tell IDEA where the subroutine starts and ends. Placing `Sub` as the first word on a line always starts a subroutine. The *End Sub* keyword combination shows the end of the subroutine. The word *Main* is the name of the subroutine. You must always provide a name for the subroutines you create.

The left side of the display contains the Project window, while the right side contains the Editor window where you type the code for your macro. Now that we're ready for that first macro, the following sections help you create your first macro and show it off to your friends.

FIGURE 2.6 The IDEAScript Editor Is Where You Type the Code for the Macros You Want to Create

Typing in the Editor Window

You type code words into the Editor window just as you would a word processor document, spreadsheet, or other editor. When you create a macro, you place the text you type between the `Sub` and `End Sub` lines in the window. This placement tells IDEA that the words you've typed are part of a procedure.

Fortunately, you don't have to go it alone when you use IDEAScript, because the Editor window comes with several aids. The first is IntelliSense. When you type something, the Editor window helps you out by telling you about the additional information you can provide as a tooltip. Figure 2.7 shows a typical example of IntelliSense information. Don't worry about the other aids for right now, you'll find them discussed in later sections of the book.

For this example, type **MsgBox("Hello World")**. The `MsgBox` *function* (a kind of command) tells IDEA to display a message box that contains the words `Hello World`. It's a simple example that programmers have used throughout history when learning IDEAScript or another language. Don't worry too much about the specifics of the code

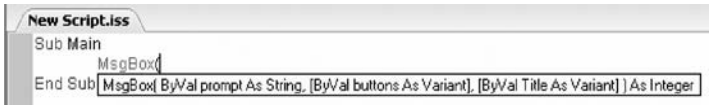


FIGURE 2.7 IDEA Provides Help on What to Type Next When You Begin Typing a Function Name

FIGURE 2.8 The First Macro Displays a Message Box With Hello World in It

for now. The code is a little hard to read as is, so add three spaces in front of `MsgBox` to show that `Sub Main` holds one procedural step that displays a message box. Your code should look like this:

```
Sub Main
   MsgBox(``Hello World``)
End Sub
```

It's time to try your first macro. Select **Debug** > **Run** or simply press **F5**. You see a message box like the one shown in Figure 2.8. Isn't it exciting to see the first macro run? As you can see, it doesn't take very much to boss IDEA around. Click **OK** to remove the dialog box.

Saving Your Macro

The macro you created exists only in memory right now. Just like your other documents, you must save the macro to disk to reuse it. IDEA will remind you to save the macro if you haven't done so already. To save your macro, select **File** > **Save**, click the **Save** toolbar button, or press **Ctrl+S**. In all three cases, you see the **Save As** dialog box shown in Figure 2.9.

Use the **Save in** field to change the location where you save the macro. Type the name of the macro in the **File name** field. The example uses `Hello World` as a file name. Click **Save** to save the file.

Understanding .iss Versus .ise Files

Macros can appear with two different extensions: `.iss` or `.ise`. The `.iss` file is a standard text file that contains the code you write, while the `.ise` file is compiled. Compiling the file turns the text into tokens that IDEA can understand. (There's a third form, an executable file that you can learn about in the "Building Your Application" section, but this isn't strictly a macro file.) So, if the `.iss` file contains words you understand and executes as a macro, just like the `.ise` file, why would you ever use the `.iss` file? The following list describes situations where the `.ise` file excels.

- **Code Hiding:** You put a lot of work into your macros. If you make the code accessible to everyone, someone will almost certainly steal it and claim that they created

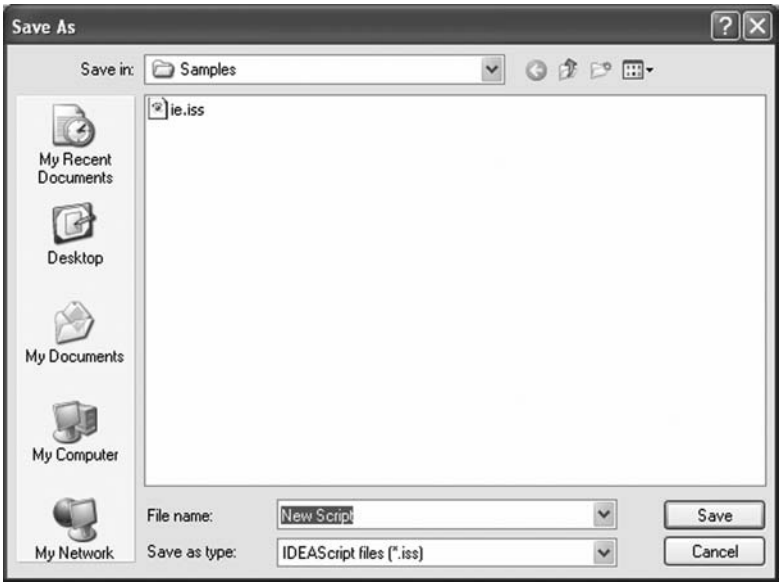


FIGURE 2.9 Save Your Macro so You Can Use It Later

it. If you want to maintain control over the code you create, compiling it hides the code from everyone else, while letting them use it for purposes you specify.

- **Faster Execution:** Theoretically, the compiled file will provide better execution times since it has already been converted into a form that IDEA recognizes. The interpretation phase of working with the macro is taken care of before you actually use the macro.

It's easy to create a compiled file. Select **File** > **Save As**, click the **Save** toolbar button, or press **Ctrl+S**. When you see the **Save As** dialog box shown in Figure 2.9, select the **Compiled files (*.ise)** option in the **Save as type** field. Type a name in the **File name** field as normal and click **Save**. Figure 2.10 shows the `Hello World.ise` file.

As you can see, the file content looks like pure garbage, yet it executes as normal. To execute this file, select **Tools** > **Macros** > **Run** in IDEA. Locate the `Hello World.ise` file and click **Open**. You see precisely the same result as when using the `Hello World.iss` file.

FIGURE 2.10 It's Impossible to Read an `.ise` File, Making It More Secure than an `.iss` File

Sending Your Macro to Someone Else

At this point, you have a lovely new macro. You're justifiably proud of your efforts and want to share it with someone else. IDEA makes it easy to send others the macros you create. In the **IDEAScript Editor**, select **File** > **Send**. IDEA will create an e-mail message using your default e-mail program. The macro is automatically attached to the e-mail message for you. All you need do is supply an address, subject, and message, and send the e-mail to the recipient as normal.

Building Your Application

IDEA supports the creation of executable files from your macros. Using this technique lets you build complete applications that run outside IDEA, unlike a macro that you have to execute from within IDEA. (You still have to have IDEA installed in order to run the application, but the convenience of using an executable file still makes this a valuable option.) This technique is most useful when you create a complete, self-contained process that you don't want others to modify. The following sections describe how to build an executable application.

Creating the Executable File

Before you do anything else, you need to save your macro as described in the "Saving Your Macro" section. It's important to save your macro before you create an executable application from it. The following steps tell how to create the executable:

1. Select **File** > **Build Application Name.exe**. For example, if you've followed the `Hello World` example, you'd select **File** > **Build Hello World.exe**. Interestingly enough, you see a **Save As** dialog box that looks similar to the one in Figure 2.9. However, in this case, the **Save as** type field contains an **Executable files (*.exe)** entry in place of the normal macro entries.
2. Type a name for the executable file in the **File name** field.
3. Select a location for the executable file in the **Save in** field.
4. Click **Save**. You'll see a new file created in the folder you selected with an `.exe` extension.

Warning

Always save a copy of your application code in a safe place. Choose a central location for all of your application code so that you can find it when you need it. For security reasons, keep compiled or executable forms of your application on network drives or the drives of other users. The important thing to remember is that you can't recover your application code if you lose it—the compiled and executable files are unreadable and you can't obtain your application code from them.

Executing Your Application from Windows Explorer

Now that you have a shiny new executable to use, locate it in Windows Explorer. Double-click the icon as you would any other Windows application and you'll see the same results as before for `Hello World`. However, in this case, you don't have to have IDEA running to use the macro—you can double-click the executable file with IDEA closed.

Just as you can add any Windows application to a shortcut or the Start menu, you can place an IDEA executable in these locations as well. You can send the `.exe` file to a friend and they'll have your complete application. In short, creating an executable file has every advantage that you obtain from its Windows counterpart—an `.exe` file is always an executable. Just imagine—your first application and you've already created something that takes programmers weeks to learn!

Note

If you want to run your uncompiled application using the Windows Scheduler, you must do so by specifying the `IDEA.EXE` file and the/`m` command line switch. For example, if you have an application named `MyApplication`, then you would add `IDEA.EXE/m=MyApplication.ISS` to the Windows Scheduler Run field. However, you can get around this requirement by using the executable form of the IDEAScript application.

Summary

This chapter has demonstrated how easy it is to create an application using IDEAScript. You don't need any special skills to perform the task—simply a few instructions. Using the techniques in this chapter, anyone can create an application that will help others work faster and smarter. Of course, you still need to discover more about IDEAScript before you can do anything impressive. The basic ideas in this chapter tell you what you'll do, but not how to do it—that's what the rest of the book is for.

It's important that you understand this chapter completely before you proceed. Make sure you create both the `10K Customers` database and the `Hello World` application as well as understand what you're doing before you proceed. It isn't necessary at this point that you understand the code—simply the process that you'll follow. Don't worry; you'll discover more about IDEAScript as the book progresses.

It's extremely important to learn about the tools you use to work with IDEAScript. Chapter 3 helps you take this next step by describing all of the elements of the IDEAScript Editor in detail. So far, you've simply followed instructions to perform a task. By the time you finish Chapter 3, you'll understand how the IDEAScript Editor works and will be able to complete many tasks on your own. Knowing your tools is extremely important. Just as you expect a carpenter to know how to use a hammer, others will expect you to know how to use the IDEAScript Editor.

CHAPTER 3

Understanding the Basics of the IDEAScript Editor

The IDEAScript Editor is your tool for creating IDEA macros. As with any other tool, understanding how the IDEAScript Editor works is central to your ability to create macros efficiently. Trying to use a hammer to drive home a screw might work, but it's not very efficient and definitely won't produce a very satisfying result. Likewise, your ability to use the IDEAScript Editor helps determine the efficiency with which you work and partly determines the result you receive.

One of the essentials of working with the IDEAScript Editor is to understand the windows it provides. Each of these windows gives you a different perspective of your application. Some windows, such as the Editor window, you use all the time, while others, such as the Watch window, only see use for specific tasks (debugging, or removing errors, in the case of the Watch window). It's important to know how to hide and view these windows as needed.

Along with windows, the IDEAScript Editor includes a number of menus and toolbars that you use to access commands. Locating the correct command quickly will help you work faster. This chapter considers the default configuration of the menus and toolbars first and then looks at ways you can customize the menus and toolbars as needed.

Perhaps the most important tool in your arsenal is the Language Browser. No one can possibly remember every function that you can use to create an application. Consequently, you need well organized help to locate precisely the function you need. You'll use the Language Browser regularly as the book progresses, so this is possibly one of the most important tools to learn about first.

Working with Windows

Think about the windows in the IDEAScript Editor in the same way that you think about windows in your house—they offer a view of your application. You look through the windows to see your application in a certain way. Just as each window in your house offers a different view, so do the windows in the IDEAScript Editor. The following sections describe each of the IDEAScript windows that you use when creating an application.

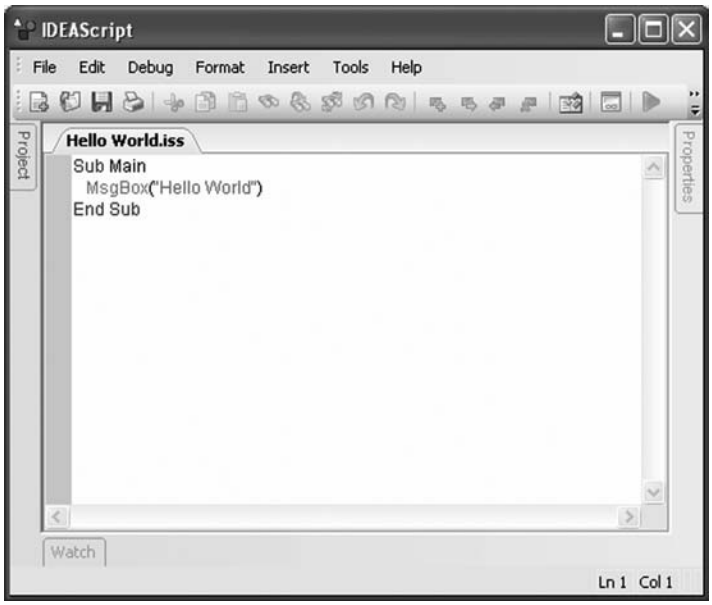


FIGURE 3.1 The Editor Window Provides a Place to Type Code for Your Application

Editor

The Editor window is where you type the commands that you want IDEA to perform. Figure 3.1 shows the Editor window. In this case, you see the `Hello World` application created in Chapter 2. However, the Editor window shows whatever code your application uses.

As noted in Chapter 2, whenever you type a keyword in the Editor window, it displays a tooltip with information about that keyword. For example, if you type the name of a function, you see the input that the function requires to perform useful work.

One of the Editor window features is hidden. Right-click anywhere within the Editor window and you see the context menu shown in Figure 3.2. You can use the options on this context menu to perform the following tasks:

- **Cut:** Removes the text you have highlighted in the Editor window and places it on the Windows Clipboard. You can use the Paste command to place this text in another

FIGURE 3.2 The Editor Window Context Menu Contains Options for Working with Text

location and create multiple copies of it. IDEA only enables the Cut command when you have text highlighted.

- **Copy:** Places a copy of the text you have highlighted in the Editor window onto the Windows Clipboard. You can use the Paste command to create multiple copies of the text anywhere you need it within the Editor window. It's also possible to use this feature to copy code to another application. IDEA only enables the Copy command when you have text highlighted.
- **Paste:** Writes text that currently appears on the Windows Clipboard to the Editor window at the insertion point (the point at which the text cursor appears). IDEA only enables the Paste command when you have text placed on the Windows Clipboard.

Note

The Editor window is the only IDEAScript Editor window that you can't hide. Because you use this particular window for every task, from writing the application to testing it, hiding the Editor window wouldn't serve a useful purpose.

Project

The Project window contains details about the elements in your application. Every time you create a new application element such as a function, subroutine, or dialog box, it appears in the Project window. Figure 3.3 shows the Project window as it appears for the `Hello World` application.

Whenever you double-click an element in the Project window, IDEA presents that element to you. For example, if you double-click a subroutine or a function name, IDEA places the insertion pointer next to the subroutine's or function's name in the Editor window.

Right-clicking the project's file name, such as `Hello World.iss`, presents a context menu with a single entry, Close. Choosing this entry will close the project for you.

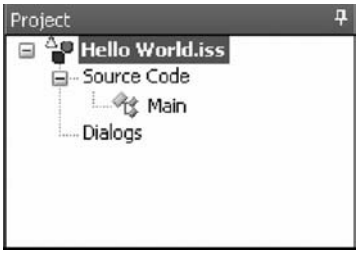


FIGURE 3.3 The Project Window Contains a Complete List of All Application Elements

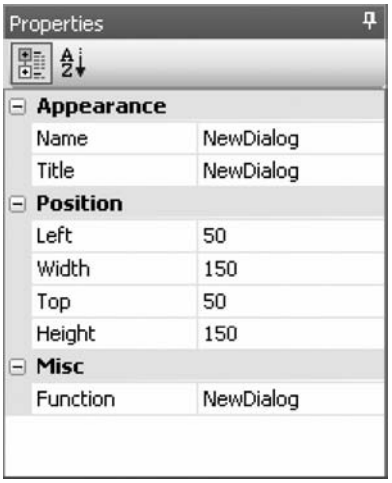


FIGURE 3.4 The Properties Window Helps You Work with Dialog Boxes

Tip

It pays to right-click in the IDEAScript Editor wherever you think you might find a context menu. The IDEAScript Editor provides a number of context menus, all of which are designed to make your work easier. Context menus help you perform tasks quickly by making it unnecessary to search for a command on a menu or toolbar.

To create a new dialog box, right-click the **Dialogs** folder and then select **New Dialog** from the context menu. You see a new dialog box added to the application. Don't worry too much about the actual use of dialog boxes yet, you will learn about them in Chapter 12.

Properties

The Properties window shown in Figure 3.4 helps you configure dialog boxes and the objects they contain using properties. A *property* is basically part of the description of an object, such as the color of an apple, may be red, green, or yellow. Color is the name of the property, and red, green, or yellow are the property values. Don't worry too much about properties now; you'll discover them in Chapters 9 and 12. IDEA only enables this window when you work with dialog boxes. Even though the Properties window can appear at other times, it doesn't have a use outside of modifying dialog boxes.

As with many other windows, the Properties window supports a context menu. When working with a text property, you can right-click the text value and see the context menu shown in Figure 3.5. These options help you perform the following tasks:

- **Undo:** Reverses any changes you make to the property value.
- **Cut, Copy, and Paste:** Performs the same tasks as the Cut, Copy, and Paste commands for the Editor window. Essentially, you can cut, copy, or paste values in the Properties window.
- **Delete:** Removes the highlighted property value text. However, this option doesn't place the text on the Clipboard—the text is simply deleted, so you need to use this option with care.
- **Select All:** Highlights the entire property value. This is an exceptionally useful option when the property value contains a lot of text.

At the top of the Properties window, you see two buttons. Click the first button if you want to see a list of properties for the selected option by category. For example, if you want to see all of the properties that affect the position of a dialog box on screen. This sort order is helpful when you need to change a number of related properties at one time. Click the second button when you want to see the properties listed in alphabetical order. Using alphabetical order is handy when you remember the name of a property you want to change and need to find it quickly.

Dialog Tools

The Dialog Tools window shown in Figure 3.6 contains a palette of controls you can use to create a dialog box. Simply drag a control onto the dialog box window and drop it. (Chapter 12 tells you more about working with dialog boxes.) This window only appears when you work with dialog boxes. The Dialog Tools window doesn't provide any context menus.

Dialog Box Editor

The Dialog Box Editor window shown in Figure 3.7 is where you create the dialog boxes you want to present to the user. IDEA supports a number of controls such as text boxes, buttons, and labels. To draw a dialog box, you simply drag and drop the controls you want onto the dialog box in the Dialog Box Editor. Chapter 12 describes this process completely. For now, all you need to know is that IDEA takes care of many of the details of creating a dialog box for you and all you really need to know is what you want the user to see.

FIGURE 3.5 Use This Context Menu to Manipulate Text Values in the Properties Window

FIGURE 3.6 The Dialog Tools Window Contains the Controls You Use to Create Dialog Boxes

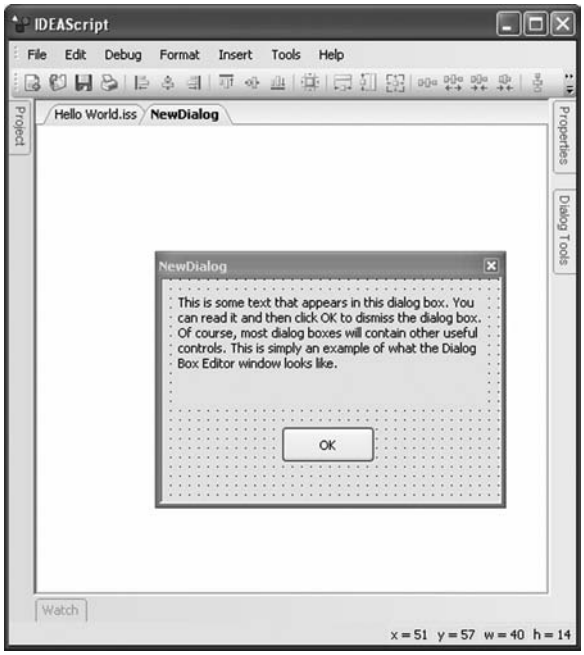


FIGURE 3.7 Place Controls on the Dialog Box to Create a User Interface

Watch

The Watch window shown in Figure 3.8 shows you the values of variables and objects you define as part of your IDEA application. It only becomes available when you place the IDEAScript Editor into debug mode. *Debugging* is the process of removing errors from your code.

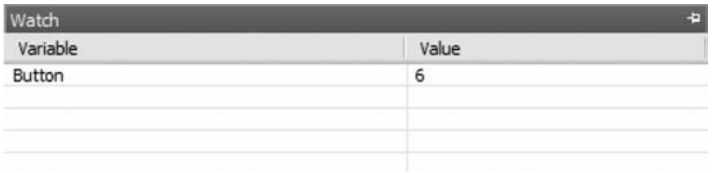


FIGURE 3.8 The Watch Window Shows You the Value of Variables and Objects

You may not think it now, but many programmers spend considerable time debugging their applications because some bugs are both persistent and hard to locate. Of course, before you can debug your application you need to know a lot more about writing code, so don't worry too much about this topic now. Chapter 21 shows you a wealth of useful debugging techniques.

There are many ways to add a variable to the Watch window. For example, you can simply type the variable name into the window. You can also drag and drop the variable onto the Watch window. Finally, you can use the Debug > Quick Watch command or press Ctrl+W. Chapter 21 discusses all of these techniques. For now, all you need to know is that the Watch window lets you see variables you create and determine what they contain.

Hiding and Viewing Windows

Most people have monitors with large displays today, so screen real estate isn't at the premium it once was. Even so, you may not want the IDEAScript Editor to fill your entire display and may want to hide some windows to simplify your view of your application. Fortunately, IDEA makes it easy to show and hide windows as desired.

Note

You can't hide the Editor window. The reason is simple—you use the Editor window for most activities within the IDEAScript Editor, so hiding it doesn't make sense. The Editor window is the only window you can't hide.

When you look at the screenshots of windows in this chapter, you see a little thumbtack in the upper right corner. When you hover the cursor over this thumbtack, the tooltip tells you that it's the **Auto Hide** button. Click this button once and it changes to show that the thumbtack is no longer pushed in. When you move the mouse, the window slides shut. Figure 3.9 shows the IDEAScript Editor with all of the windows hidden. Notice the little tabs positioned around the Editor window—these tabs represent the hidden windows. The left side of the display contains the Project window tab, the right the Properties window, and the bottom the Watch window.

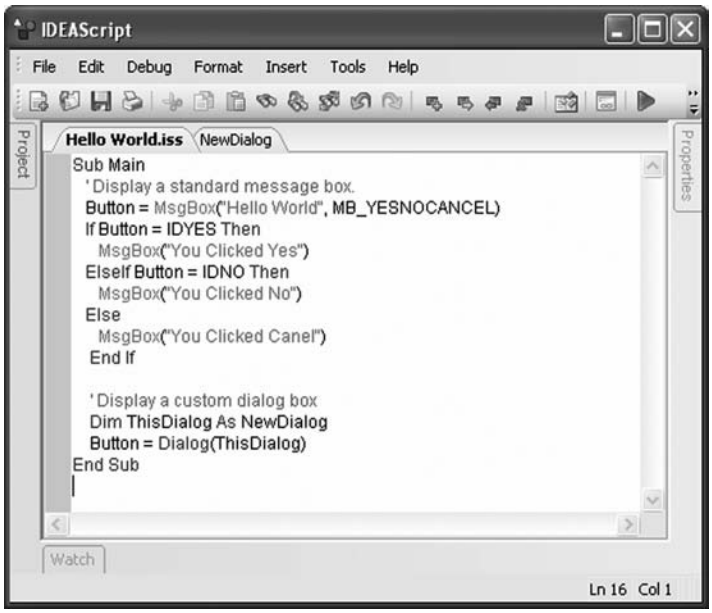


FIGURE 3.9 The IDEAScript Editor Uses Tabs to Represent Hidden Windows

To see a window again, simply move the cursor over its tab. The window will slide out again as shown in Figure 3.10. Because the window is normally hidden, it actually appears over the macro source code. If you want to see the window all the time again, simply click the **Auto Hide** button.

The IDEAScript Editor provides other alternatives for moving windows to a better position. You can use the cursor to grab the window title bar and drag it to a different location as shown in Figure 3.11. Detaching the window means you can move it anywhere on your display and access it only when you need to do so. The detached window is always accessible, but it's less distracting because you only focus on it when you need to work with it.

Eventually, you'll want to put the window back in its original location or place it in another spot on the screen. Let's say you want all of the windows to appear on the right side. You can tell the IDEAScript Editor where to place the window in a process called docking. A *docked* window is one that is attached to a specific location in the IDEAScript Editor.

In order to dock the window, grab its title bar with the mouse cursor. As you drag the window, you see the docking selectors shown in Figure 3.12. Place the mouse cursor over the docking selector you want to use to dock the window.

Depending on which docking selector you choose, windows can appear above, below, to the right, or to the left of other windows. You can even create tabbed windows. Figure 3.13 shows an example of just one window combination you could try. In this case, the Properties and Projects windows are tabbed, while the Dialog Tools window appears below the other two.

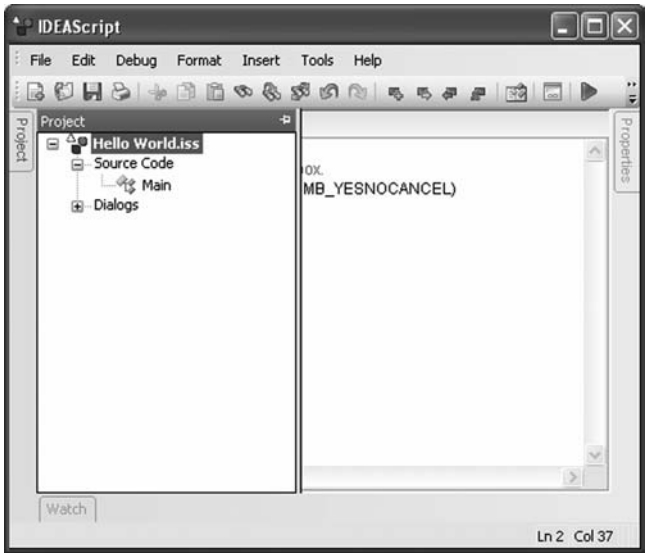


FIGURE 3.10 View Hidden Windows by Hovering the Mouse Cursor Over Their Tabs

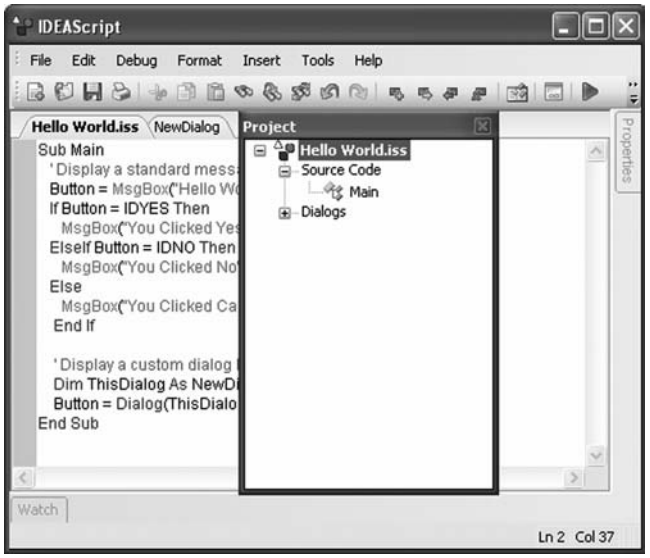


FIGURE 3.11 Detach Windows to Move Them to Another Place on Your Display

Working with Menus and Toolbars

While windows contain data, menus and toolbars contain commands that let you interact with that data. Menus tend to provide organized access to every command that the IDEAScript Editor provides, while toolbars provide quick access to common commands.

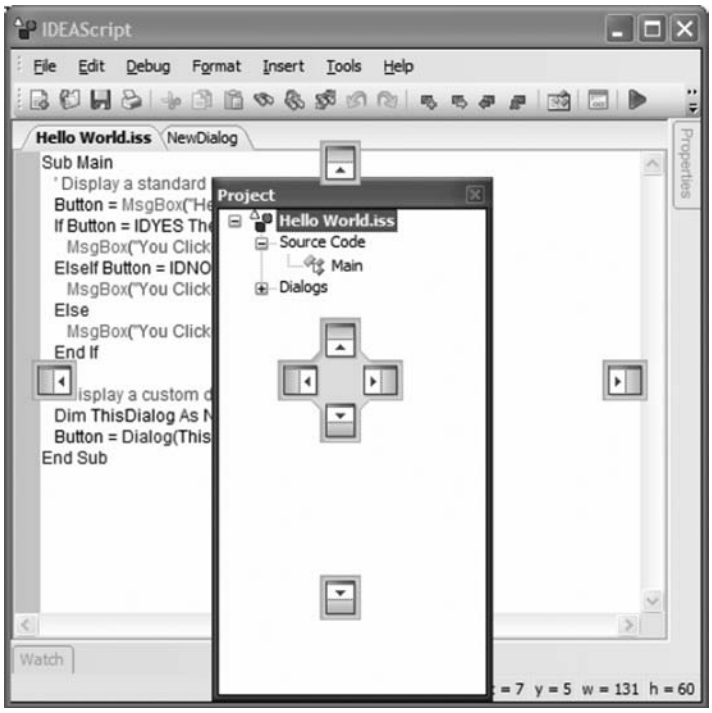


FIGURE 3.12 Choose the Docking Selector You Want to Use for the Window

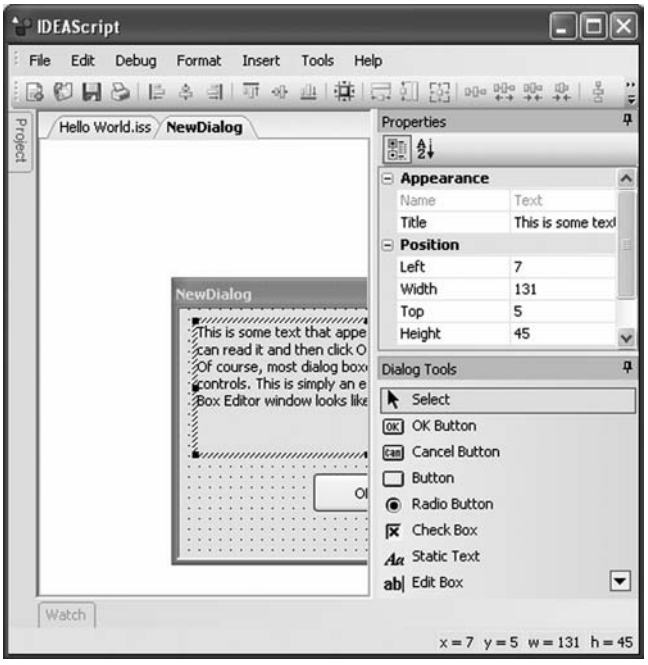


FIGURE 3.13 Create Combinations of Windows that Work Best for You When Editing

Most people use a combination of menus and commands to work with their applications. The following sections describe both menus and toolbars.

Menu Bar

The Menu bar appears at the top of the IDEAScript Editor window and contains a list of all of the commands that you can perform. Of course, you can't always perform every command. For example, debugging commands are only useful when you debug the program. As a result, the IDEAScript Editor disables some commands until you can actually use them (the commands appear grayed out). The following sections describe the various menus.

FILE The File menu shown in Figure 3.14 contains commands you use to work with application files. Most of these commands should look familiar because you use them in just about every application you work with. Just like any other application, you need to create new files, save modified files, close files when you no longer need them, and open existing files. Of course you'll also want to print your source code at times, and there are commands that help you handle this requirement.

Chapter 2 discusses one of the special File menu commands, Send, which lets you send your application code to someone else. Chapter 2 also tells you how to transform your application into an executable file using the Build command.

As with many editors, the File menu also contains a list of the files you recently modified. You also use the File > Exit command to exit the application.

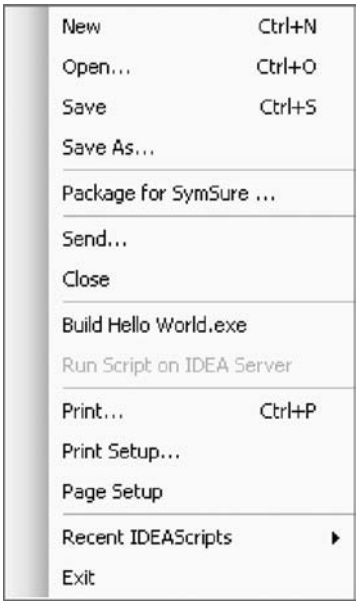


FIGURE 3.14 Use the File Menu Options to Manipulate Application Code and Executable Files

FIGURE 3.15 Use the Edit Menu Options to Interact With the Application Code and Dialog Box Objects

EDIT The Edit menu shown in Figure 3.15 contains commands for working with application code or dialog box objects. When working with a dialog box, you can cut, copy, and paste objects. Of course, you can perform these same tasks when working with application code. As with any good editor, you can also undo and redo actions that you perform on either application code or a dialog box.

When working with application code, you can also perform a number of text-related tasks. For example, you can find a particular piece of text using the Find and Find Next commands or replace occurrences of text with different text using the Replace command. It's also possible to go to a particular line in the application code file, which comes in handy when the editor tells you there's an error.

The IDEAScript Editor also supports bookmarks. Think of bookmarks as you would the physical equivalent used to mark a position in a book. You use bookmarks to make it easy to locate code that you're modifying or simply want to mark for reference purposes. Bookmarks can prove especially helpful as your application increases in size and locating a particular element becomes more difficult.

DEBUG The Debug menu shown in Figure 3.16 contains all of the commands required to check your application for errors. For now, the two important commands are Run, which lets you execute the application, and Stop Debugging, which lets you stop the application even if it isn't finished executing. The book discusses the other commands in Chapter 21.

FORMAT The Format menu shown in Figure 3.17 contains a host of positioning commands. Appearance means a lot when it comes to applications. If your application

FIGURE 3.16 Use the Debug Menu Options to Locate Errors in Your Code

FIGURE 3.17 Use the Format Menu Options to Change the Appearance of Dialog Boxes

looks neat and well constructed, users will enjoy using it more. All of these positioning commands have toolbar counterparts, so you can see a brief description of them in the "Standard Toolbar" section. Chapter 12 shows how to work with dialog boxes in detail.

INSERT The Insert menu shown in Figure 3.18 lets you create new dialog boxes. In addition, it contains a list of controls you can add to dialog boxes you create. You can use this menu in place of the Dialog Tools window if you like—the list of controls is the same. Chapter 12 tells you more about working with dialog boxes.

TOOLS The Tools menu contains two commands: Font and Language Browser. Select the Font command when you want to change the font in either the Editor window or the Dialog Box Editor. The font affects the appearance of your code in the Editor window, while it affects the appearance of the user interface in the Dialog Box Designer.

The Language Browser is a kind of hierarchical help file. You use it to learn more about the commands that you can type into the Editor window. You'll discover more about the Language Browser as you work though the examples in this book.

HELP The Help menu contains a single command, Contents. Click this command and you see the IDEAScript Help file open to the Contents page. The Help file contains basic

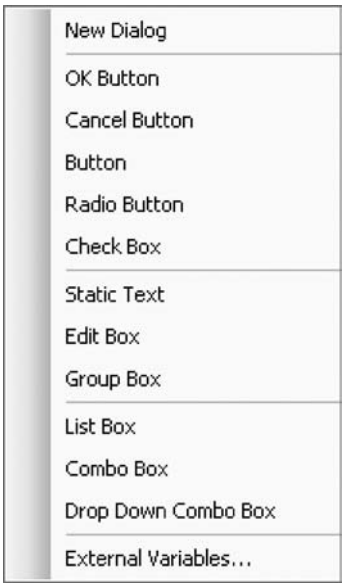


FIGURE 3.18 Use the Insert Menu Options to Add Items to Dialog Boxes

information about creating applications and tells you about IDEAScript. For example, the Help file tells you about new features in the latest version of IDEAScript.

Standard Toolbar

Toolbars provide quick access to common commands. Using a toolbar is faster than using a menu because you don't have to search for the command you want to use through the menu hierarchy. Figure 3.19 shows the Standard toolbar you see when working with text, while Figure 3.20 shows the Standard toolbar you see when working with dialog boxes.

The Standard toolbar contains icons that represent various commands. When you hover the mouse cursor over a particular icon, the IDEAScript Editor displays a tooltip that tells you the purpose of that icon. Clicking the associated button is the same as selecting

FIGURE 3.19 The Standard Toolbar Used for Text

FIGURE 3.20 The Standard Toolbar Used for Dialog Boxes

the associated menu command. As with menu commands, the IDEAScript Editor grays out toolbar buttons you can't use in a particular circumstance. The following list provides an overview of the Standard toolbar commands:

- **New** (Both): Creates a new project file that can contain a combination of macros and dialog boxes.
- **Open** (Both): Opens an existing project file.
- **Save** (Both): Saves the current project file to disk.
- **Print** (Both): Outputs the application code in the current project file to the printer. Many developers prefer viewing their code in printed form because it can be easier to see on paper than on screen.
- **Cut** (Text Only): Removes the highlighted text and places it on the Windows Clipboard.
- **Copy** (Text Only): Copies the highlighted text to the Windows Clipboard.
- **Paste** (Text Only): Places the text currently found on the Windows Clipboard at the insertion point.
- **Find** (Text Only): Displays the Find dialog box used to locate specific text within the application code.
- **Find Next** (Text Only): Repeats the last find.
- **Replace** (Text Only): Displays the Replace dialog box. You use the entries in this dialog box to find particular text instances and replace them with some other text.
- **Undo** (Both): Reverses the previous action in the selected window. For example, if you delete some text and then click Undo, the IDEAScript Editor will restore it.
- **Redo** (Both): Re-performs an action that you reversed in the selected window.
- **Toggle Bookmark** (Text Only): Adds or removes a bookmark at the current insertion point. A bookmark helps you locate a particular position within the application code file.
- **Next Bookmark** (Text Only): Locates the next bookmark in the selected application code file.
- **Previous Bookmark** (Text Only): Locates the previous bookmark (the next one toward the top of the file) in the selected application code file.
- **Delete Bookmark** (Text Only): Removes the bookmark at the current insertion point. Nothing happens if there isn't a bookmark.
- **Language Browser** (Both): Displays the Language Browser window.
- **New Dialog** (Both): Creates a new dialog box and opens the Dialog Box Editor window so you can edit it.
- **Run Script** (Text Only): Runs the currently selected macro (an application code file can contain several macros, so you have to be sure to select the correct macro within the file).
- **Stop Debugging** (Text Only): Stops application execution, even if the application isn't finished performing a task. Stopping an application at the wrong time can damage data, so use this command with care.
- **Toggle Breakpoint** (Text Only): Adds or removes a breakpoint at the current insertion point. Breakpoints stop application execution at a particular place in the code during debugging.

- **Step Into** (Text Only): Executes the next line of code. If the next line of code is a function or subprocedure call, execution continues within the function or subprocedure. This command is used during debugging.
- **Step Over** (Text Only): Executes the next line of code. If the next line of code is a function or subprocedure call, IDEA executes the entire function or subprocedure. This command is used during debugging.
- **Quick Watch** (Text Only): Adds the highlighted text to the Watch window. This command is used during debugging.
- **Clear All Breakpoints** (Text Only): Removes all of the breakpoints from the application file. You need to perform this step after debugging the application.
- **Align Lefts** (Dialog Box Only): Aligns all of the selected dialog box objects on their left edges.
- **Align Centers** (Dialog Box Only): Aligns all of the selected dialog box objects along their centers.
- **Align Rights** (Dialog Box Only): Aligns all of the selected dialog box objects on their right edges.
- **Align Tops** (Dialog Box Only): Aligns all of the selected dialog box objects along their tops.
- **Align Middles** (Dialog Box Only): Aligns all of the selected dialog box objects through their middles.
- **Align Bottoms** (Dialog Box Only): Aligns all of the selected dialog box objects along their bottom edges.
- **Align to Grid** (Dialog Box Only): Places dialog box objects to align with a grid as you create and move them.
- **Make Same Width** (Dialog Box Only): Modifies the selected dialog box objects so they are all the same width. The IDEAScript Editor uses the last object you select as the target width.
- **Make Same Height** (Dialog Box Only): Modifies the selected dialog box objects so they are all the same height. The IDEAScript Editor uses the last object you select as the target height.
- **Make Both Sizes Same** (Dialog Box Only): Modifies the selected dialog box objects so they are all the same width and height. The IDEAScript Editor uses the last object you select as the target width and height.
- **Make Same Horizontal Spacing** (Dialog Box Only): Modifies the spacing of the selected dialog box objects so that they each have the same amount of horizontal space between them.
- **Increase Horizontal Spacing** (Dialog Box Only): Increases the horizontal spacing between the selected dialog box objects by an equal amount for each object.
- **Decrease Horizontal Spacing** (Dialog Box Only): Decreases the horizontal spacing between the selected dialog box objects by an equal amount for each object.
- **Removing Horizontal Spacing** (Dialog Box Only): Removes all the horizontal space between the selected dialog box objects.
- **Make Same Vertical Spacing** (Dialog Box Only): Modifies the spacing of the selected dialog box objects so that they each have the same amount of vertical space between them.

- **Increase Vertical Spacing** (Dialog Box Only): Increases the vertical spacing between the selected dialog box objects by an equal amount for each object.
- **Decrease Vertical Spacing** (Dialog Box Only): Decreases the vertical spacing between the selected dialog box objects by an equal amount for each object.
- **Removing Vertical Spacing** (Dialog Box Only): Removes all the vertical space between the selected dialog box objects.
- **Center in Form Horizontally** (Dialog Box Only): Centers the selected dialog box object horizontally on the form.
- **Center in Form Vertically** (Dialog Box Only): Centers the selected dialog box object vertically on the form.
- **Help** (Both): Displays the IDEAScript Editor Help window.

Summary

This chapter has helped you understand the main tool you use to create IDEAScript applications, the IDEAScript Editor. It's important to know how to work with the IDEAScript Editor so that you can produce applications quickly and efficiently. Fortunately, you get a lot of practice as the book progresses. Make sure you refer back to this chapter as you have questions about the IDEAScript Editor.

One of the best ways to start working with the IDEAScript Editor is to customize it for your personal tastes. Try hiding and showing windows. Move windows around to suit your needs. Check out the various menu options. In short, play around with the IDEAScript Editor a little before moving on to the next chapter. The play time you spend will save you considerable frustration later.

Chapter 4 begins working with applications in earnest. Now that you have a better idea of how to use the IDEAScript Editor, you can use it to begin learning more about how to put applications together. Chapter 4 explores issues such as the general structure of applications. Don't worry about becoming a programmer just yet though. The two main points of Chapter 4 are to look at macros as boxes that you put together to create an application and how to use the IDEAScript Editor to create these boxes. The content of the macro boxes is a discussion in later chapters in this book.

CHAPTER 4

Designing Structured Applications

At one time, in the early days of computer programming, applications didn't have much, if anything, in the way of structure. These older applications were incredibly hard to write and even harder to maintain. In fact, some of them were so hard to maintain that the best thing most companies could do was to throw them out and start from scratch. Using the structured application development techniques of today is important for a number of reasons, but the reasons you need to consider as part of this chapter are that these applications are easier to:

- Write
- Maintain
- Understand

Note

Application is a term used to describe a process—one or more tasks packaged together. An application can be comprised of IDEAScript, Visual Script, or a combination of both.

Fortunately, IDEAScript makes it incredibly easy to write structured applications. In fact, you have already been introduced to some of the basics of designing structured applications. Simply using subroutines and functions adds structure to your application. Of course, you need to know how to use them correctly, which is the purpose of this chapter.

In addition to discussing the basics of structured applications, this chapter also looks at some techniques you can use to create applications quickly and easily. You'll discover tools such as the Macro Recorder and how even using Help can make it easier to write structured applications.

One of the essentials discussed in this chapter is the use of comments. A *comment* is a kind of note—it helps other people to understand why you did things in a certain way. Comments also help jog your memory so that you can remember how your own

applications work—nothing's quite as bad as having to rediscover code that you knew quite well at one point.

Finally, this chapter shows how to attach your application to IDEA through a toolbar or menu command. Adding your application directly to the user interface makes it considerably easier for other people to use.

Understanding the Parts of an Application

An application consists of a number of parts. You won't fully understand all of the parts right now, but it's important that you begin to see these parts and work with them. As you progress through the book, you'll discover more about these parts and begin using them to create complex applications. Here's a listing of the parts and overviews of how you'll use them:

- **Source File:** The source file is the container that holds everything together. Everything you create to complete an application appears within the source file. When you use external data, code, or resources, the reference and manipulation code for these resources appear within the source file.
- **Subroutines:** Subroutines contain code that performs a task. The subroutine is self-contained. It can accept input information, but doesn't provide direct output.
- **Main Subroutine:** Every time you create a new project, it includes `Sub Main`, which is the main subroutine for the application. This subroutine appears first in the source file and the application follows the instructions it contains first.

Note

IDEA always executes the first subroutine or function in the file first—it doesn't matter what the name of the subroutine or function is. It's always a good idea to call this first entry `Main` so that anyone viewing your code will understand that this is the beginning point for the macro. Always add other subroutines and functions after `Sub Main`.

- **Functions:** Like a subroutine, a *function* contains code that tells IDEA how to perform a task. However, a function also returns information to the caller. This feature allows the caller to know something about how the code in the function worked. For example, a return value of "true" might say that the function succeeded, while a value of "false" might say the function failed.
- **Global Variables:** A *variable* is a sort of box used to hold information. For example, you might create a variable to hold the user's name or the current date. A global (or public) variable is one that's available to every subroutine or function in the application. In most cases, global variables hold information that affects every part

of the application, such as application configuration information. Chapter 5 tells you how to work with variables and constants.

- **Constants:** A *constant* is a kind of box that holds information that never changes. For example, the days of the week never change, so you can define them as constants. Most constants are global to the entire application because they don't change and there's a high likelihood that multiple parts of the application will require the constant value at some point.
- **Local Variables:** *Local* (or private) *variables* are only accessible to the subroutine or function that creates them. Using local variables keeps private information private and tends to reduce confusion in your application.
- **Dialog Boxes:** A *dialog box* can contain either a simple message or a group of controls. You use two different techniques for creating dialog boxes. You've already seen the `MsgBox` function in earlier chapters. This chapter will show you how to use more of the `MsgBox` function. Chapter 12 shows the more complex method of creating dialog boxes using controls.
- **External Data:** You can create internal data—the data that only appears within the current application—or *external data*—data that more than one application could use. For example, the databases you use in IDEA are external data. Chapter 5 tells you about internal data while Chapters 7, 8, and 9 begin your journey using external data.
- **External Code:** IDEAScript makes it possible to use code found in other applications. It may seem amazing, but you can actually create a macro that adds data to an application such as Microsoft Word and then use Word to print a report. As mind-boggling as this concept may seem right now, you'll begin to work with *external code* in Chapter 14. Chapter 16 shows you how to manipulate your data using Excel, a truly exciting way to use macros!
- **Other External Resources:** Don't let anyone tell you that IDEAScript isn't a fun language to use. After you know how to use all the features of IDEAScript, you can grab resources from almost anywhere and use them within your application. Chapter 12 shows how to add both pictures and sound to your applications. Chapter 15 discusses how you can interact directly with external files.

Understanding the Methods Used to Create an Application

IDEA provides a number of methods for you to create an application. Of course, you can always start by creating a Visual Script macro and using the techniques in Chapter 23 to create an IDEAScript macro from the Visual Script macro. However, there are several different direct approaches you can use to create macros as described in the following sections.

Recording a Series of IDEA Tasks

Perhaps the easiest method for starting a macro is to let IDEA create it for you. It's possible to use the Macro Recorder to start your macro and then add dialog boxes and

other code to it later. The Macro Recorder is a tool that watches the tasks you perform and records them for you. The "Using the Macro Recorder" section describes this tool in more detail. Use this approach whenever you already know how to perform a task and simply want to automate the process.

Copying from History

The source of code that you're most likely to miss is the History display. Not only does the database's history window show you what you've done to the database, it also contains all the source code so that you can reproduce the steps. As you work with your database, IDEA automatically creates a historical record of your activities for you. If you later decide that you want to automate all or part of the process, you can use the history as a starting point. To use the History feature, follow these steps:

1. Open the database you want to use.
2. Complete the tasks you want to perform.
3. In the **Properties** window, click **History**. You'll see a display similar to the one shown in Figure 4.1. Notice the **IDEAScript Code** entry near the bottom of the figure.

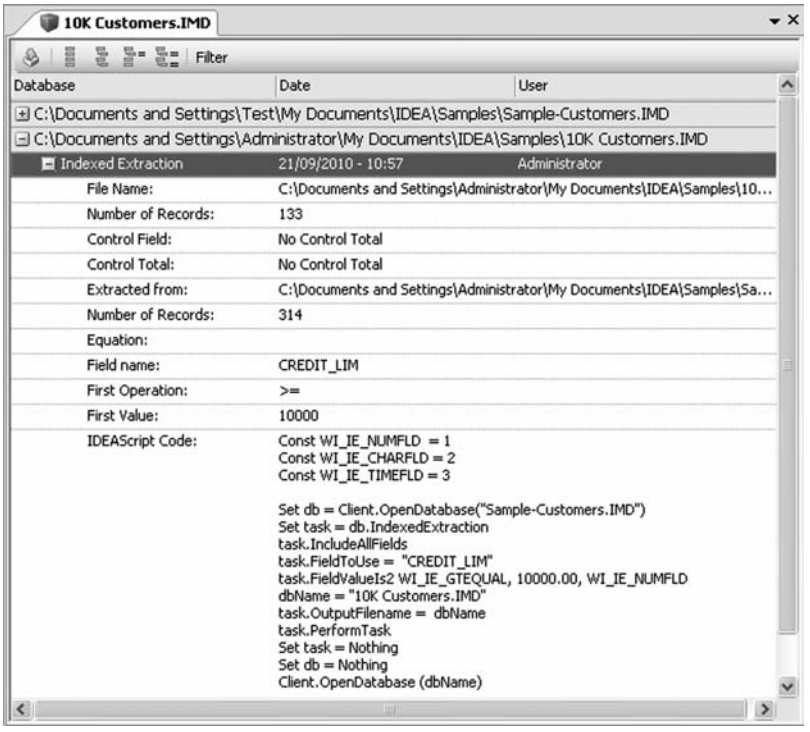


FIGURE 4.1 The History Window Contains IDEAScript Code You Can Use to Repeat a Task

4. Right-click the **IDEAScript Code** entry and select one of the options from the context menu. Normally, you'll select **Copy** for the current task.
5. Create a new macro by selecting **Tools** > **Macro** > **New** > **IDEAScript**.
6. Place the insertion pointer on the line between `Sub Main` and `End Sub`. Click **Paste**. IDEA pastes the code from the Clipboard into the new IDEAScript macro.
7. Save the macro for later use.

Typing into the Editor Window

This is the method you've used so far to create the simple applications in previous chapters. In many cases, typing keywords directly into the Editor window is your only option. Use this option when you've exhausted other methods of creating an application. Normally, you need to type code by hand when the task involves something more than simple database manipulation. For example, you always type code into the Editor window when you need to provide a user interface.

Copying from Help

IDEA provides two forms of help that you can use immediately for locating usable code. The first is the actual IDEAScript Help dialog box shown in Figure 4.2. You access the IDEAScript Help dialog box by selecting Help > Contents in the IDEAScript Editor.

In this case, you see an example heading you could add to your code. Simply highlight the code in the **IDEAScript Help** dialog box and press **Ctrl+C** to copy it to the

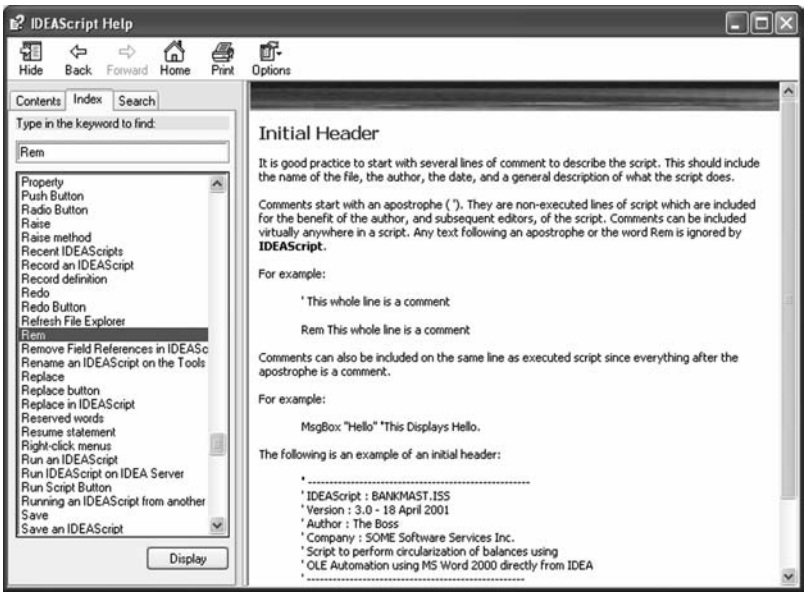


FIGURE 4.2 The IDEAScript Help Dialog Box Contains a Wealth of Helpful Code

Clipboard. Place the insertion point where you want it in the **IDEAScript Editor** and click **Paste**.

Tip

Notice that the code shown in Figure 4.2 appears with the Rem entry, which also includes the shorthand form of the single quote (`'`). It pays to scout around in the Help system whenever you have a few extra minutes. You might be amazed at the kinds of helpful code you find!

The second source of help is the Language Browser shown in Figure 4.3. You display the Language Browser by selecting Tools > Language Browser or by pressing Ctrl+L. As with the IDEAScript Help dialog box, you highlight the code you want to use, copy it to the Clipboard, and paste it into the IDEAScript Editor.

Figure 4.3 shows an example of how to use the `MsgBox` function. Most of the entries in the Language Browser include similar examples that you can simply copy and paste to your application. Use the Language Browser when you need code to work with a particular Basic Script or IDEAScript function.

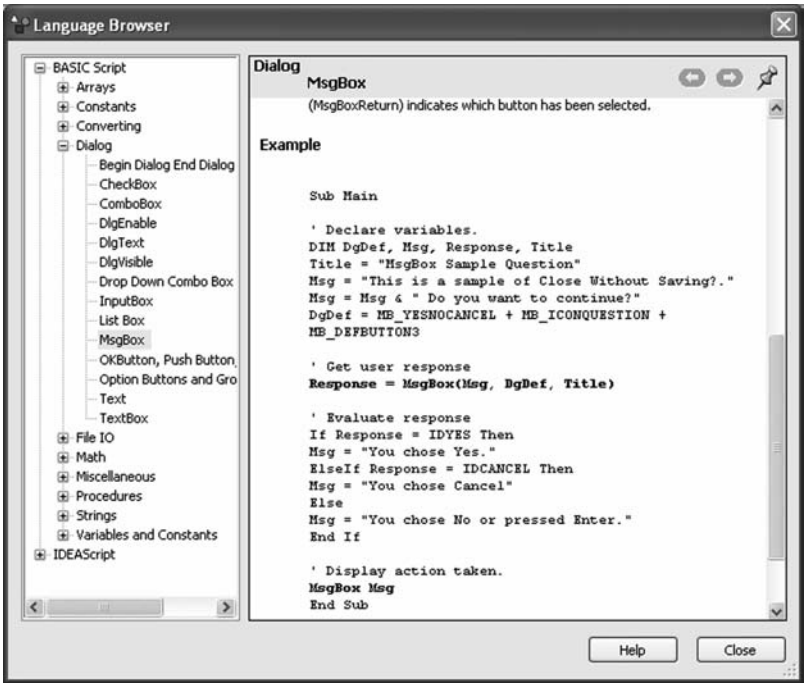


FIGURE 4.3 Locate Code You Want to Use in the Language Browser and Copy it to Your Macro

Copying from Other Macros

As you create applications, you'll build a library of code you can use. That's right, recycling applies to application code. In many cases, you can recycle code you created for another purpose by making a few simple changes for the current application. In fact, some programmers are so good at creating reusable code that they can use many pieces of code without any changes. Use this approach as often as possible once you begin creating your own code.

Tip

Code reuse is just one of many reasons to write the cleanest code possible. As you work through this book, you'll discover a wealth of techniques you can use to create code that's well documented, easy to understand, and easy to reuse. Creating this code may seem like a lot of work at first, but eventually you'll have a large library of fully debugged code that you can simply copy and paste into applications as you need it. The initial effort in coding the application correctly is well worth the results. Imagine the look on your boss's face when you're able to create an application in a week that would normally require an entire month to write!

Using the Macro Recorder

Record Macro is an amazing task. It helps you create an application simply by performing a task or a series of tasks as you would normally in IDEA. IDEA records the steps you perform, creates code for them, and then lets you save them for future use. When working with the Macro Recorder, you follow these simple steps to create an application.

1. From the main **Operations** toolbar in IDEA, click the **Record Macro** button.
2. Perform the steps you normally perform.
3. Click the **Record Macro** button again to stop recording.
4. Save the macro as either an IDEAScript macro or a Visual Script macro.
5. View the results in the **IDEAScript Editor** or **Visual Script Editor**.
6. Make changes such as adding dialog boxes and increasing application flexibility as required.
7. Test the resulting application.

That's it! You don't have to do anything fancy to get this help. The following sections contain all of the details for using the Record Macro task to start your next application. As an example, these sections repeat the 10K Customers example found in the "Opening the Visual Script Editor" section in Chapter 2.

However, instead of building the routine manually in the Visual Script Editor, this example uses the Macro Recorder to perform the same actions.

Tip

Always plan the task you want to perform and carry out the task carefully. Any time you make a mistake in executing the task, you generate extra code that you must remove later. The fewer errors you make in performing the task, the better the output from the Macro Recorder. In addition to getting better code, scripting the task also helps you review the method you use to perform it. Sometimes you can find a more efficient way to perform the task, which reduces the amount of code you have to generate to perform it. Of course, any steps you remove also improve execution speed, making it possible for you to get home and start enjoying that new movie on television!

Starting a Macro

In order to start recording your macro, select **Tools** > **Record Macro** or press **Ctrl+R**. At this point, the Macro Recorder is started. You now perform any steps required to perform the task. Use the following steps to record the `10K Customers` macro.

1. Go to the **File Explorer** window and double-click the `Sample-Customers` database entry. IDEA opens the `Sample-Customers` database for you.
2. Select **Data** > **Extractions** > **Indexed Extraction**. You see the **Indexed Extraction** dialog box shown in Figure 4.4. (Note that Figure 4.4 shows all of the criteria you need defined—the initial dialog box is blank.)
3. In the **Field** box, select **CREDIT_LIM**. In the **Value is** field, select >= and type **10000**. In the **File name** field, type **10K Customers 2**. Each of these actions defines part of the task you want performed on the `Sample-Customers` database. In this case, you're telling IDEA to extract all of the records in the `Sample-Customers` database that have a value equal to or greater than 10000 in the CREDIT_LIM (credit limit) field and place them in the `10K Customers 2` database.

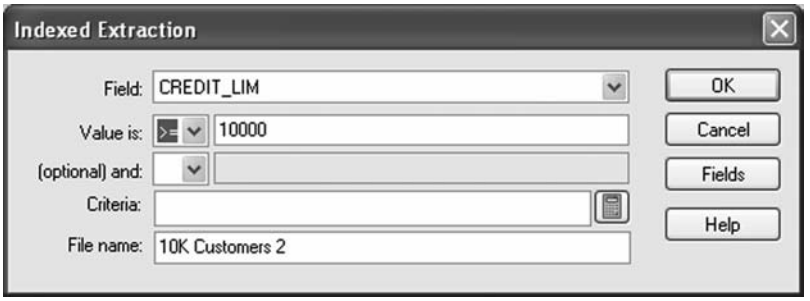


FIGURE 4.4 The Indexed Extraction Dialog Box Lets You Extract Data Based on Key Values

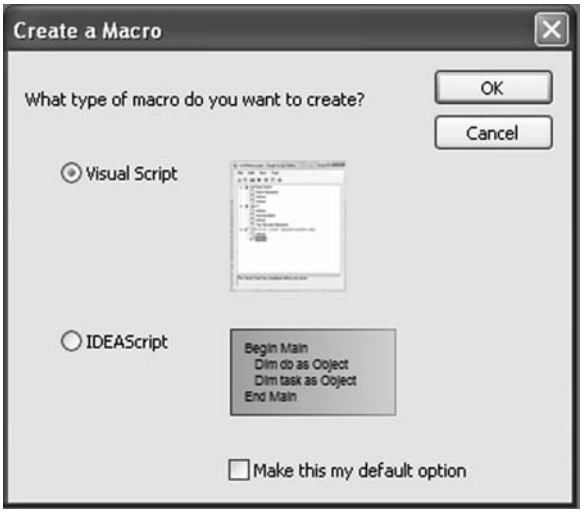


FIGURE 4.5 When You Stop Recording, You Must Decide What Type of Macro to Create

4. Click **OK**. IDEA creates the new database for you. You see tabs for both the `Sample-Customers` and `10K Customers 2` databases—IDEA displays the `10K Customers 2` database.
5. Close both databases.

At this point, you've completed the procedure. Now it's time to work with the macro you created.

Stopping a Macro

Before you do anything else, you need to stop the macro. The following steps tell you how to perform this task.

1. Select **Tools** > **Record Macro**. This action stops the macro recording process and displays the **Create a Macro** dialog box shown in Figure 4.5.
2. Select the **IDEAScript** option. This option creates editable code that you can use to create an application.
3. Optionally, check **Make this my default option**. If you check this option, IDEA will automatically assume you want to create IDEAScript macros when you create a macro.
4. Click **OK**. At this point, IDEA creates the macro shown in Figure 4.6.

Viewing the Results

When you initially see Figure 4.6, you might wonder where the application you created is and how it actually follows the steps you defined. The answer is that you need to view

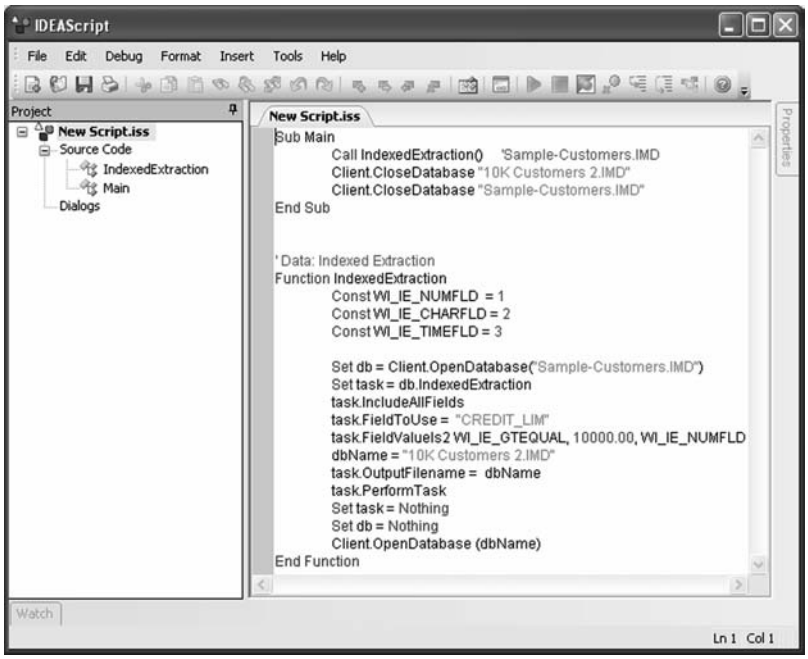


FIGURE 4.6 The IDEAScript Editor Contains the Results of the Task You Performed

the process in a particular way to see your task. (It helps to follow along in the code as you read this section.) `Sub Main` contains these three steps:

1. Perform the Indexed Extraction started in Step 1 of the "Starting a Macro" section and ended at Step 4. In order to perform these steps, the code calls `IndexedExtraction()`. (The term *call* refers to the act of one piece of code executing another piece of code, such as `Sub Main()` executing the contents of the `IndexedExtraction()` function.)
2. Close the `10K Customers 2` database, which is the first part of Step 5 in the "Starting a Macro" section.
3. Close the `Sample-Customers` database, which is the second part of Step 5.

Now you need to look at the `IndexedExtraction()` function. The first step in this function is to open the `Sample-Customers` database, which is actually Step 1 in the "Starting a Macro" section. So, at this point, you have all of the steps accounted for.

In following the code, you'll find that the Data > Extractions > Indexed Extraction action found in Step 2 is accomplished using the `Set task = db.IndexedExtraction` call line of code. In fact, Step 2 requires quite a bit of code to complete, as shown in Listing 4.1.

LISTING 4.1 Performing the Data > Extractions > Indexed Extraction Task

```
Const WI_IE_NUMFLD = 1
Const WI_IE_CHARFLD = 2
Const WI_IE_TIMEFLD = 3

Set db = Client.OpenDatabase("Sample-Customers.imd")
Set task = db.IndexedExtraction
task.IncludeAllFields
task.FieldToUse = "CREDIT_LIM"
task.FieldValueIs2 WI_IE_GTEQUAL, 10000.00, WI_IE_NUMFLD
```

LISTING 4.2 Completing the Data Extraction and Opening the Database

```
dbName = "10K Customers 2.imd"
task.OutputFilename = dbName
task.PerformTask
Set task = Nothing
Set db = Nothing
Client.OpenDatabase (dbName)
```

In addition to performing the actual task, the code must perform a number of tasks to create the extracted database and display it to the user, as shown in Listing 4.2.

Editing the Code

The code that the Macro Recorder generates works just fine, but it doesn't necessarily provide everything you need. At some point, you'll have the skills required to combine various pieces of code to create complex applications based on tasks you normally perform. In addition to combining tasks, programmers often add:

- Informative messages
- A user interface
- *Glue code* (used to combine tasks invisibly—think of it as gluing pieces of other code together)
- Non-IDEA task code, such as interacting with disk files
- Bells and whistles, such as a logo or other decorations

Let's keep the editing simple this time. This section will show you how to add an informative message to the code you created with the Macro Recorder. In this case, you add the `MsgBox()` function to tell the user that the macro is finished. Listing 4.3 shows the `MsgBox()` code you should add to `Sub Main` in bold.

This piece of example code uses the `MsgBox()` function to display the message box shown in Figure 4.7. This example uses three arguments. An *argument* is a piece of information that the function requires to execute. Arguments are either mandatory or optional. The first `MsgBox()` argument is mandatory—you must provide a message of some sort to display. The second, optional, argument contains special values that determine what buttons the message box displays, the default button, any icons it provides,

LISTING 4.3 Editing the 10K Customers 2.ISS Macro

```
Sub Main
   Call IndexedExtraction()    'Sample-Customers.imd
   Client.CloseDatabase "10K Customers 2.imd"
   Client.CloseDatabase "Sample-Customers.imd"
   MsgBox "10K Customers 2 Successfully Created!", _
          MB_OK + MB_ICONINFORMATION, _
          "Success"
End Sub
```

and when the user must respond to the message box. The third, optional, argument contains the message box title.

FIGURE 4.7 The MsgBox() Function Can Display Status Messages

The second `MsgBox()` function argument requires a little more explanation. Some arguments are constant values. A *constant* is a kind of variable that never changes value. Notice that the second argument adds two message box (`MB`) constants together to provide both a button definition and an icon definition. The following list contains the `MsgBoxType` constants you can use for the second argument:

- **MB_OK:** Display only the OK button.
- **MB_OKCANCEL:** Display the OK and Cancel buttons.
- **MB_ABORTRETRYIGNORE:** Display the Abort, Retry, and Ignore buttons.
- **MB_YESNOCANCEL:** Display the Yes, No, and Cancel buttons.
- **MB_YESNO:** Display the Yes and No buttons.
- **MB_RETRYCANCEL:** Display the Retry and Cancel buttons.
- **MB_ICONSTOP:** Display a critical message.
- **MB_ICONQUESTION:** Display a warning query.
- **MB_ICONEXCLAMATION:** Display a warning message.
- **MB_ICONINFORMATION:** Display an information message.
- **MB_DEFBUTTON1:** Set the first button as the default.
- **MB_DEFBUTTON2:** Set the second button as the default.
- **MB_DEFBUTTON3:** Set the third button as the default.
- **MB_APPLMODAL:** Application modal. The user must respond to the message box before continuing work in the current application.

- **MB_SYSTEMMODAL:** System modal. All applications are suspended until the user responds to the message box.

You should notice one additional feature of the example code. The space/underline combination (_) at the end of each line is called a *line continuation character*. Sometimes a function has so many arguments that it would be hard to read the code on a single line. Unfortunately, IDEAScript requires that you have the whole function call on a single line. The line continuation character makes the IDEAScript Editor think that the entire function appears on one line when it really appears on several. Consequently, you see the three lines in bold in Listing 4.3 as three lines, while the IDEAScript Editor sees them as a single line. Use line continuation characters to make your code more readable.

Testing the Results

Testing is a necessary part of every application development strategy. Of course, testing provides you with personal feedback that you're accomplishing something. Nothing's quite as motivating as seeing your application grow and become everything you thought it should be. The testing process provides part of the encouragement needed to complete the application, which can sometimes seem to drag on without the required encouragement.

However, testing's also practical. You begin by testing the code that the Macro Recorder created for you. It's important to perform this first step because you want to be sure that your scripted process actually accomplishes the task that you think it should. It's possible to create a macro that contains bugs and you may not see these bugs until you actually test the Macro Recorder created code.

Test as you add each new piece of code. The earlier you find a bug, the easier it is to fix. If you have to go back several steps to find a bug, you might find it hard to find. In addition, the bug could interfere with code you've added afterward, which would mean a lot of rework on your part. Testing early and often has significant benefits that you shouldn't discount.

Warning

Testing without saving your code first is a recipe for disaster. If the code somehow freezes IDEA or the system, then you'll lose all of your changes. Save often to reduce your risk of loss!

In some cases, testing also points out potential application needs. For example, when you run the `10K Customers 2.iss` macro a second time, you'll notice the message box shown in Figure 4.8. IDEA is asking whether you want to overwrite the file. To avoid potential user confusion, you might want to add code to your macro that erases the old file before it creates the new one. The user never sees the message box in Figure 4.8 because there's never an old file to overwrite.

This section only describes a few basic reasons for testing. As the book progresses, you'll discover other reasons to test your application. In addition, Chapter 21 describes

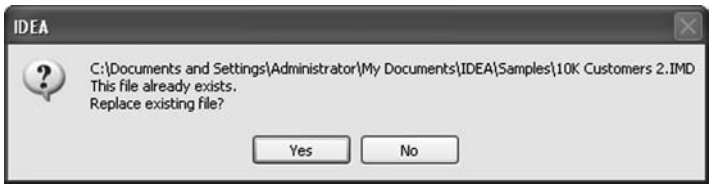


FIGURE 4.8 Use Testing to Locate Potential Application Additions

testing in detail in the form of debugging. That's right, debugging is a kind of in-depth testing that helps you locate errors in your application code.

Working with Subroutines and Functions

You can create applications using a combination of subroutines and functions. In both cases, the subroutine or function acts as a container for your code. Both can receive input in the form of arguments. However, the difference between a subroutine and a function is that a function returns a value to the caller and a subroutine doesn't. You use a subroutine when you want to perform a task without returning a value. The following sections show you subroutines and functions in use.

Creating a Subroutine

When you create a new application, IDEA automatically creates a subroutine named `Main` for you. However, you don't have to keep that name. You can name the subroutines and functions in your application nearly anything you want. Subroutine and function names do have certain requirements:

- They must start with a letter.
- They can only be made up of letters, numbers, and the underscore character.
- They must not conflict with any of the IDEAScript reserved words, such as active code commands.

These rules apply to variable and constant names. Listing 4.4 shows the main subroutine for the example application, `Hello1`.

`Hello1()` is the first subroutine that shows variables. Both `HelloAnswer` and `HelloResult` are variables that act as containers for `Variant` data. Don't worry too much about what `Variant` data is for now. All you really need to know is that the variables act as containers.

After creating the two variables, `Hello1()` adds information to another variable called `MyHello`. `MyHello` is a String variable that you'll learn more about in the "Understanding Scope" section.

LISTING 4.4 Creating a Subroutine That Calls Other Subroutines and Functions

```
Sub Hello1
   Dim HelloAnswer As Variant
   Dim HelloResult As Variant

   MyHello = "Do you feel well today?"
   HelloAnswer = MsgBox(MyHello, _
      MB_YESNO + MB_ICONQUESTION, _
      "Your Status")

   HelloResult = Hello2(HelloAnswer)

   MyHello = "OK, then your final answer is " + _
      CheckAnswer(HelloResult)
   MsgBox MyHello, _
      MB_OK + MB_ICONINFORMATION,
      "Final Status"
End Sub
```

The next line of code shows a different kind of `MsgBox()` function call. (For many developers, this use of the `MsgBox()` function marks the difference between a simple statement that displays a message and a function that returns data to the application.) The `MsgBox()` function can actually tell you which button the user clicked. When the code makes this call, `HelloAnswer` will contain the selection the user made. Notice that when you use `MsgBox` as a function to return a value, you must enclose the arguments in parentheses. If you don't want to return a value, then you don't need the parentheses. When the application calls `MsgBox()` you see the dialog box shown in Figure 4.9.

Now that `HelloAnswer` has a value in it, you can use it as an argument to call `Hello2`. Don't worry about this code for now, it's explained in the "Calling a Function from a Subroutine" section of this chapter. However, at this point, let me explain that the code uses `HelloAnswer` as input to `Hello2` and it receives something back from `Hello2`, which is placed in `HelloResult`.

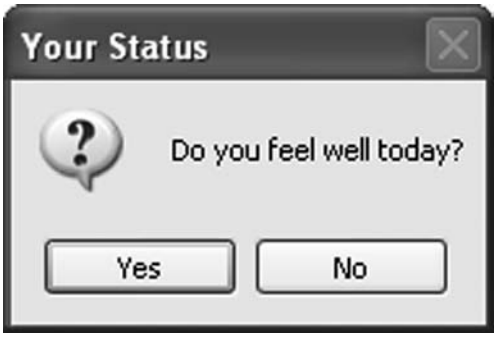


FIGURE 4.9 The First Hello Message Asks How You Feel

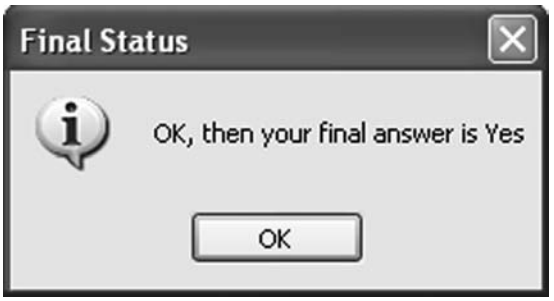


FIGURE 4.10 The Final Hello Message Tells What You Decided

At this point, the code displays the final answer from this example. It tells you what you've finally decided. Notice that the resulting answer is complicated because you don't know in advance what the user will answer. So, the code calls another function, `CheckAnswer()`, to create a string suitable for display, from `HelloResult`. The code uses `MyHello` to display another dialog box that will look similar (depending on the user's answer) to the one shown in Figure 4.10.

This example is obviously more complicated than earlier examples you typed, but one thing holds true: All this code really does is follow a set of steps:

1. Create some containers to hold information.
2. Ask how the user feels and place this information in one of the containers.
3. Verify that the user actually feels this way and place this information in another container.
4. Convert the answer to a human-readable string.
5. Display the final result on screen.

Calling a Function from a Subroutine

Functions are commonly used to accept input, do something with that input, and then produce a result. The example application has two functions: `Hello2()` and `CheckAnswer()`. The purpose of `Hello2` is to verify the user response obtained in `Hello1`. The purpose of `CheckAnswer()` is to convert `MsgBox()` function output into something that a human can understand. Listing 4.5 shows the `Hello2()` code.

A function begins with a declaration that includes the word `Function` and the function name. You tell IDEA what arguments the function requires by placing them in parenthesis. Notice that it looks like a variable declaration. The arguments consist of a name and a data type, just like a variable. You can define multiple arguments by separating them with commas. The function declaration ends with the output type of the function—`Variant` in this case.

The code begins by creating a message for `MyHello`. This message asks the user to verify their previous answer and includes that answer as part of the message. Notice that this is the second time that `CheckAnswer()` has appeared. You can call functions

LISTING 4.5 Verifying the User Response

```
Function Hello2 (HelloValue As Variant) As Variant
   Dim Hello2Answer As Variant

   MyHello = "You Clicked: " + CheckAnswer(HelloValue) + _
      ". Are you sure you feel this way?"

   Hello2Answer = MsgBox(MyHello, _
      MB_YESNO + MB_ICONQUESTION, _
         "Status Check Verification")

   Hello2 = Hello2Answer
End Function
```

FIGURE 4.11 This Message Box Verifies the User Input

whenever and wherever you need to call them. Even though you write the function once, you can use it as many times as needed. Consequently, writing well-structured code means that you only do things once.

The code calls `MsgBox()` with `MyHello` as the message. Figure 4.11 shows a typical example of this message box. When the user clicks one of the buttons, `MsgBox()` places the result in `Hello2Answer`.

Now you get to that last line of code. In order to return a value from a function, you make the function, `Hello2`, equal to whatever you want to return. In this case, the code returns the value of the button that the user clicked. The caller gets this value as an output.

Now it's time to look at `CheckAnswer()`. Listing 4.6 shows the code for this function.

LISTING 4.6 Converting the MsgBox() Output

```
Function CheckAnswer(Answer As Variant) As String
   CheckAnswer = "No Answer"

   If Answer = IDYES Then
       CheckAnswer = "Yes"
   ElseIf Answer = IDNO Then
      CheckAnswer = "No"
   End If
End Function
```

As with `Hello2`, `CheckAnswer` accepts a single argument of type `Variant`. However, it outputs a String. You don't have to understand the specifics of this code just yet. However, what this code does is check the input against each of the possible input values, `Yes` or `No`. If the input doesn't match either of these values, `CheckAnswer` tells you that it doesn't know what the answer is by providing the `"No Answer"` string. Chapter 6 describes how to use the `If...Then` structure.

Understanding Scope

You've seen both subroutines and functions that contain variables. These variables are only accessible within the subroutine or function—another subroutine or function can't access them. Variables within a subroutine or function are called local (or private) variables—they have a local scope. *Scope* defines who can see a particular variable.

In general, it's a good idea to use local variables whenever you can because they reduce the risk of data contamination and make your code easier to understand. However, what do you do if you simply must create a variable that everyone can access? That's where global (or public) variables come into play. You define these variables outside of the subroutines and functions so that everyone can access them. In short, global variables have a global scope.

The example application has a single global variable, `MyHello`. It appears at the top of the example code listing like this:

```
Dim MyHello As String
```

Notice that you don't declare global variables any differently than you do local variables. The only difference is their location in the file. Anything within a subroutine or a function is always a local variable, while anything outside a subroutine or function is a global variable.

It's important to define how you want IDEA to interpret the variables you create. Otherwise, you get the default definition of a global variant, which isn't always what you want.

Making Your Code Easy to Read

Code will never be as easy to read as the text in a book. However, you can easily make code more readable by following a few simple guidelines.

First, use white space to format the code (simply press Tab to create the white space). You may have noticed the use of indentation and spacing in the examples in this chapter. The indentation and spacing provide clues as to how the code is related and makes it easier to determine just where one task ends and another begins. As you progress through the book, you'll see more and more examples of using formatting to create readable code.

Second, comments are also a helpful way to make code easier to read. You create a comment by starting a line with a single quote (`'`). Listing 4.4 shows good use of white

LISTING 4.7 Hello1 Using Comments

```
Sub Hello1
   ' Create some containers to hold information.
   Dim HelloAnswer As Variant
   Dim HelloResult As Variant

   ' Ask how the user feels and place this information
   ' in one of the containers.
   MyHello = "Do you feel well today?"
   HelloAnswer = MsgBox(MyHello, _
      MB_YESNO + MB_ICONQUESTION, _
       "Your Status")

   ' Verify that the user actually feels this way and place
   ' this information in another container.
   HelloResult = Hello2(HelloAnswer)

   ' Convert the answer to a human-readable string.
   MyHello = "OK, then your final answer is " + _
      CheckAnswer(HelloResult)

   ' Display the final result on screen.
   MsgBox MyHello, _
      MB_OK + MB_ICONINFORMATION, _
         "Final Status"
End Sub
```

space to make the code easier to read, but it doesn't include any comments. Listing 4.7 shows the same code as Listing 4.4, but this time it includes comments. Notice how much easier this code is to read.

Third, always use descriptive names for variables, function names, subroutines, and other elements. The simple examples so far aren't very confusing. However, as the book progresses, the examples will become more complex and you'll see how naming comes into play.

Adding Your Application to a Toolbar or Menu

You won't want users to access your applications using the Tools > Macros > Open command in most cases. Doing so would lead to all kinds of errors. In fact, many users would refuse to use your finely wrought application at all. Consequently, you'll have to bind your application to either the Tools menu or a toolbar in most cases. *Binding* is the act of making the application accessible using a menu command or a toolbar button. The following sections describe the process of binding your application to a menu or toolbar.

Best Practices for Binding an Application

Generally, you do want to make the applications you create available to other people. However, there are times where you won't want to bind the application. The following

list gives you an idea of when binding might not be the best answer to application availability.

- The application is a utility that only you plan to use.
- Only administrators or advanced users require the application and you want to keep it hidden from less skilled users.
- You've already made the application available as an executable from the Start menu.
- Automation is the key goal for the task that the application performs, so you never start it manually.

Once you do decide to bind the application, you must decide how to bind it. When binding to a menu, you only get one choice—the Tools menu. The application will appear in its assigned position on the Tools menu, which may actually hide it from view for some people because they won't think to look on the Tools menu. Use the Tools menu when the user will require the application occasionally and possibly during a scripted process.

Using the toolbar is a little more flexible because you can place the application on any toolbar. The toolbar is also more accessible and users will generally find your application with less effort. However, the downside to using a toolbar is that adding too many icons can quickly clutter the display and actually interfere with the user's ability to perform useful tasks. Use the toolbar for those very few situations where a user will require access to the application every day.

Binding the Application to a Menu

IDEA makes it easy for you to add your application to a menu. These steps show you how.

1. Select **Tools** > **Bind Macro to Tools Menu**. You'll see the **Menu Editor** dialog box shown in Figure 4.12.
2. Click **New**. You'll see an **Open** dialog box where you can select the application you want to add to the Tools menu.
3. Locate and select the `.iss` (IDEAScript), `.ise` (compiled IDEAScript), `.vscript` (Visual Script), or `.exe` (executable) file that you want to add to the Tools menu. The example uses the `CallingSubsAndFunctions.iss` macro file described in the "Working with Subroutines and Functions" section.
4. Click **Open**. IDEA adds the macro or executable file to the Menu Editor dialog box and fills out a number of the fields for you.
5. Type a meaningful name for your application in the **Menu Name** field (such as **Calling Subs and Functions**).
6. Repeat steps 2 through 5 for as many applications you want to add to the Tools menu. If you want to add separators between entries (the line that separates items on the menu), then click **Separator**. You can also move items up and down using

FIGURE 4.12 The Menu Editor Dialog Box Lets You Add Applications to the Tools Menu

the **Move Up** and **Move Down** buttons. If you make a mistake, click the entry and click **Delete**.

7. Click **OK**. You can see the new entries on the Tools menu.

Binding the Application to a Toolbar

It's also easy to bind your application to a toolbar. These steps show you how.

1. Click the **Toolbar Options** button (down arrow on the toolbar) and then select **Add or Remove Buttons** > **Customize**. Select the **Macros** tab. You'll see the **Customize** dialog box shown in Figure 4.13.
2. Click the ellipses next to the **Command** field. You see an **Open** dialog box.
3. Locate and select the `.iss`, .`ise`, .`vscript`, or `.exe` file that you want to add to the toolbar. The example uses the `CallingSubsAndFunctions.iss` macro file described in the "Working with Subroutines and Functions" section.
4. Click **Open**. IDEA adds the macro or executable file to the **Command** field.
5. In the **ToolTip** field, type a descriptive tooltip for the macro or executable, such as **Try Calling Subs and Functions**.
6. Drag one of the icons from the **Buttons** group (in the upper half of the Macros tab) to the toolbar that should have the button added to it. You'll see the button appear on the toolbar.
7. Click **Close**. Your application is ready to access from the toolbar.

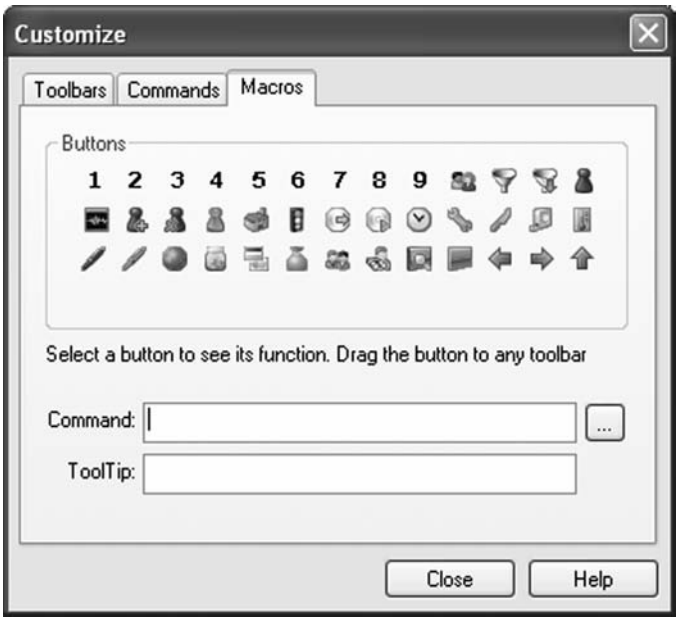


FIGURE 4.13 Use the Macros Tab of the Customize Dialog Box to Add Macros to a Toolbar

Summary

This chapter has introduced you to some real world application design techniques and strategies. Adding structure to your applications is essential. The Macro Recorder automatically produces structured code for you, which gives you a very good start on the development process. In addition, you see good examples of structured programming in both the Help system and the Language Browser. Of course, this book will continue to demonstrate structured programming techniques that you can use when creating your own application.

It's important to work through the examples in this chapter, even if they are relatively simple, because the techniques you learn here will play an important part in helping you create great applications. Make sure you begin with the example in the "Using the Macro Recorder" section. When you get to the "Testing the Results" section, make sure you actually test the `10K Customers 2.iss` macro. There's a real thrill in seeing this first application work because you actually modified it to display a success message!

Although the example in the "Working with Subroutines and Functions" section isn't as complex as the one in the "Testing the Results" section, it's the first application you write completely from scratch. Make sure you test it as well. Of course, this example also provides a perfect opportunity to play. Try various messages, icon changes, button changes, and so on. Playing with the example code is an extremely important part in learning to program. Have some fun while you're doing it!

Up until this chapter, you've played with applications without really knowing what it all means. Chapter 5 begins the process of discovering what the code words in IDEAScript mean. In Chapter 5, you discover internal application data. Knowing how to use variables is an essential first step in learning how to perform other tasks with IDEAScript. Variables hold particular kinds of data—think of the specialized containers you use to hold kitchen goods, shoes, valuables, and other items. Just as you wouldn't stick your shoes in your home safe, you can't stick a string in a box meant for a number. Chapter 5 helps you understand data and the variables used to hold it.

CHAPTER 5

Working with Data

Data—it's the reason you work with applications, write applications, and interact with computers at all. The biggest investment for any company isn't the hardware or the software, but the data manipulated by computers. Consequently, understanding data is potentially the most important part of learning how to write IDEA applications. Without an appreciation of data in all its forms, you won't be able to gain a complete understanding of how to manipulate it and your efforts in writing an application will prove frustrating at best—economically disastrous at worst.

This chapter helps you understand data from several perspectives. First, you look at data from the computer's point of view and then you view it from the IDEAScript perspective.

These two viewpoints will serve to help you understand the way your application works with data and avoid making mistakes in performing data manipulation.

Data comes in a number of types in IDEAScript. *Type* describes the kind of data—how it's viewed by IDEAScript. These types help you categorize the data and choose the right type for a particular need. In addition, you discover how to convert one data type into another data type. Data conversion is important because it ensures that your application handles the data correctly.

The final part of the chapter begins your journey in data manipulation. You discover how to use operators to modify the data. For example, when you add two numbers together, you use the + operator to tell IDEAScript what task to perform. Data manipulation also involves formatting the data for display and creating your own data types as necessary to make it easier to work with some types of data.

Understanding Variables and Constants

As humans, we look at data of all sorts without truly considering its type. We have a different understanding of data than the computer does. Ask someone their age and they'll give you a number that indicates how many years they've lived. Ask them their name and they'll say a word composed of letters that contains their name. A computer

has no such understanding of data. For a computer, data is simply numbers. Even the letters in your name are simply numbers. Computers have no concept of data as anything other than numeric values manipulated in a certain way to obtain the result you specify. It may surprise you to find out that computers don't even place any value on the data—the data simply exists as electrical signals.

Not many people can relate to data as electrical signals. Certainly, the early computer professionals did, but even they found the task difficult, which is why they started creating applications to manipulate the data. How IDEAScript views data differs from how the computer views data, but it must view the data within the context of the computer. In short, IDEAScript acts as a kind of interpreter between you and the computer. IDEAScript provides convenient methods for humans to describe data and then interacts with that data in a way that the computer can understand.

With this in mind, it's time to discuss the first of several ways to view data: variables and constants. The following sections provide you with details of using both variables and constants in your application.

Defining a Variable

As discussed in other chapters, a variable is essentially a storage container for data. When you create a variable, you use the `Dim` keyword, which is short for dimension. At one time, you'd dimension a variable—essentially construct it. The `Dim` keyword is followed by the variable name like this:

```
Sub Main
   ' Create an empty variable and display it.
   Dim StringVar
   MsgBox StringVar
End Sub
```

If you use the `MsgBox()` function to display the content of the variable, `StringVar`, as shown in the example, you won't see anything. The resulting message box is empty as shown in Figure 5.1.

A variable that doesn't contain anything is said to be *empty*. In fact, you can use the `IsEmpty()` function to determine whether the variable is empty like this:

```
' Check to see if the variable is empty.
MsgBox IsEmpty(StringVar)
```

If you simply declare the variable, the variable doesn't have a type until you fill it with something. The type defines the kind of data that the variable can contain. Unfortunately, using untyped variables makes your code hard to read, so you normally give the variable a type using the `As` keyword and a type name (see the "Choosing a Data Type" section for details on types). If you want to create a variable that can hold text (strings), you use the String type as shown here.

```
' Create a string variable and fill it with data.
Dim StringVar2 As String
StringVar2 = "Hello"
If IsEmpty(StringVar2) Then
   MsgBox "StringVar2 is Empty"
Else
   MsgBox "StringVar2 Contains: " + StringVar2
End If
```

In this case, you see the message box that tells you the content of `StringVar2` because it does contain a value. If the string variable were empty, you'd see `"StringVar2 is Empty"` instead. The `IsEmpty()` function is extremely useful in determining what to do with a particular variable.

This section tells you about declaring general variables. There are some special variable types that you learn about later, such as the Variant described in the "Working with Variant Data" section. Chapter 11 tells you about two other special variable types, arrays and collections. You'll also discover objects in Chapter 9. However, at this point, you do know enough to begin using variables in simple applications.

Defining a Constant

Constants are a special kind of data storage box. When you think about constants, think about a sealed box. Once you place something in the box, you can never change it. Constants are valuable precisely because they don't change—you often use them for comparison purposes. For example, you can use a constant to check whether a function call return value is what you expect.

In order to declare a constant, you use the `Const` keyword, followed by the constant name and its value. You must assign a value to the constant when you create it because you can't change a constant later. Here's an example of a constant declaration.

```
' Create and display a constant value.
Const MyConstValue = "Goodbye"
MsgBox MyConstValue
```

FIGURE 5.1 Variables are Empty Until You Put Something in Them

As with variables, constants can be of any type. You can also make constants local or global in scope. However, unlike variables, constants more often than not are global in scope. There's no security penalty for making a constant global because you can't change it.

There's a small but important benefit for using constants. Because of the way IDEAScript handles constants, they provide a slight performance boost. In addition, constants tend to require a little less memory than a variable of the same type.

IDEAScript defines two constants for you. In fact, you've already seen one of these constants at work in the "Editing the Code" section in Chapter 4. Here are the two constants and a description of how you use them.

- **MsgBoxReturn:** Returns a value to determine which button the user clicked in the message box.
- **MsgBoxType:** Specifies the type of buttons in the message box and the icon that's displayed.

Using Option Explicit

Normally, you don't even have to declare your variables in IDEAScript. You can simply type a variable name and assign a value to it. This is an extremely bad practice to use because typos can cause all kinds of hard-to-find bugs and you'll find yourself pulling your hair out—the makers of IDEA don't want you to be bald any more than you want to be bald. Consequently, there's a solution that makes it possible to ensure you declare every variable you create, `Option Explicit`.

To use this feature, simply type `Option Explicit` as the first line in your application code file. Now, whenever you attempt to use a variable without declaring it first, IDEA displays the error message shown in Figure 5.2 during run time (using `Option Explicit` doesn't change anything while you type your code).

With `Option Explicit` in place, there's far less chance that you'll have typos in your application. No typos means there's one less potential source of bugs, which is a very good thing indeed.

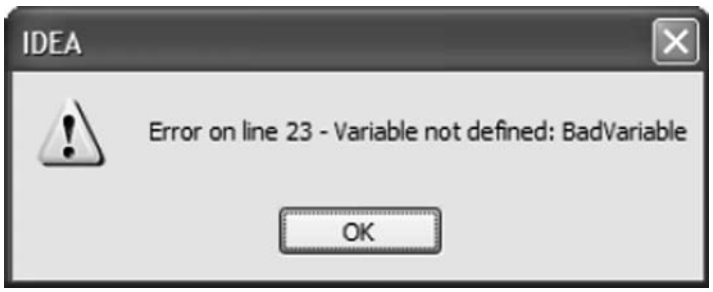


FIGURE 5.2 Use Option Explicit to Remove Some Types of Bugs

Choosing a Data Type

The "Understanding Variables and Constants" section of this chapter describes one of several ways to view data in IDEAScript. This section describes a second way to view data—the data type. When you tell someone your age, you naturally know that your age is a kind of number. Likewise, when you give someone your name, you naturally know that your name is a series of characters. This information is apparent to you, but the computer knows nothing about it and it doesn't care to know either.

Over the years, computer scientists have created precise definitions of data type. The reason you need such precise definitions when working with data in an application is that the difference between data types is artificial. These data types are created for your use and IDEAScript helps you follow rules in working with the data types. Otherwise, chaos would result from no one knowing what kind of data the application is manipulating. The following sections provide more information about data types.

Understanding How Data Type Affects Code

The kind of container you create is called a *data type*. IDEA treats numbers differently from characters. It also treats numbers with a decimal portion differently from those that contain only an *integer* (only whole numbers without a decimal portion). In addition, there's a special type for logical (truth) answers and even one for dates. In short, you can probably find a data type for every need, so it's important to know what kind of data type to use for a variable.

IDEAScript supports the data types shown in Table 5.1.

Some data types shown in Table 5.1 may seem repetitive, but you have to look at the entire entry. For example, the Byte, Integer, and Long all deal with integer numbers. However, a Byte data type requires only 1 byte of storage space, while a Long requires 4 bytes. Spending more on storage space does net an equivalent boost in integer size, however, a Byte can only store values from 0 to 255, while a Long can store values from −2,147,483,648 to 2,147,483,647. Choosing the right data type means looking at both the kind of storage provided (integer in this case) and the storage box size.

A few of the data types also have a special suffix character associated with them. You can use this character to tell IDEAScript to create a variable of that type without actually using the data type name. For example, if you want to create a String data type, you use the dollar sign (`$`) as a suffix. Listing 5.1 shows an example of the suffix character in action.

In this example, both variables rely on a suffix. However, the first variable is a Long, while the second is a Double. The `VarType()` function helps you see the difference. This function reports the data type of the variable as a number. When you run the application, it reports that the data type of the first variable is 3 (Long), while the second variable has a data type of 5 (Double). (See the "Performing Data Conversion" section for more details about the output of the `VarType()` function.)

Note

You can find a complete discussion of the Object data type in Chapter 9.

TABLE 5.1 IDEASCript Data Types

Type	Suffix	Declaration	Size	Range
Boolean	None	Dim BoolVar As Boolean	1 byte	True or False
Byte	None	Dim BVar As Byte	1 byte	0 to 255
Currency	None	Dim CVar As Currency	8 bytes	−922,337,203,685,477.5808 to +922,337,203,685,477.5807 (assuming a 4-digit fraction)
Date	None	Dim DVar As Date	8 bytes	Any day between the years 100 and 9999
Double	#	Dim DblVar As Double	8 bytes	−179e308 to −4.94e-324 for negative values 4.94e-324 to 1.80e308 for positive values
Integer	%	Dim IntVar As Integer	2 bytes	−32,768 to 32,767
Long	&	Dim LongVar As Long	4 bytes	−2,147,483,648 to 2,147,483,647
Object	None	Dim X As Object	4 bytes	Any object
Single	!	Dim SnglVar As Single	4 bytes	−3.4e38 to −1.40e-45 for negative numbers 1.40e-45 to 3.40e38 for positive numbers
String	$	Dim StrVar As String	0 to 65,500 characters	Up to 65,500 characters
User-Defined Type	None	Created using the Type keyword	Size of each element	Anything the developer wants to define
Variant	None	Dim X As Variant	Determined by variable stored	Any type of variable

Working with Strings

The String data type provides the means for working with character data of all sorts. Whenever you work with character information, you use a string to do it. Strings are somewhat complicated because the computer sees them only as numeric data. IDEA interprets these numbers as letters for you. However, the numeric nature of the data can work to your advantage. You can do things such as changing a string to uppercase using a simple function. Table 5.2 provides a complete list of the functions you can use to modify strings using IDEAScript.

LISTING 5.1 Using a Suffix to Define a Variable

```
Sub Main
   ' Create the variables using a suffix.

   MyLong& = 1
   MyDouble# = 1

   ' Demonstrate the variables are different types.
   MsgBox VarType(MyLong)
   MsgBox VarType(MyDouble)
End Sub
```

TABLE 5.2 String Manipulation Functions

Function	Description
Chr, Chr$	Returns a one-character string whose ASCII number is the argument.
InStr	Returns the character position of the first occurrence of string2 within string1.
LCase	Returns a string in which all letters of the string parameter have been converted to lowercase.
Left	Returns from the left, the specified number of characters from a string parameter.
Len	Returns the number of characters in a string.
Let	Assigns a value to a variable.
LTrim	Removes the leading spaces of a string.
Mid	Returns a substring within a string.
Right	Returns from the right, the specified number of characters from a string parameter.
RTrim	Removes the trailing spaces of a string.
Space	Inserts a string of a specified number of spaces in a statement.
StrComp	Returns the value indicating the result of a string comparison.
String	Returns a string consisting of one character charcode repeated num times.
Trim	Removes the leading and trailing spaces of a string.
Ucase	Returns a string in which all letters of the string parameter have been converted to uppercase.

While Table 5.2 is interesting, it really doesn't show you the string-related functions in action. Listing 5.2 shows how you can use these functions to perform tasks. Of course, exactly how you use the string functions depends on your application and imagination.

Listing 5.2 begins by showing you how to create a variable called `vbCrLf`. When you think about this variable, think of a typewriter (as antiquated as such a reference is, it really does work). When you pressed the return bar on the typewriter, it returned the carriage to the beginning and also advanced the roller one line so that you could begin typing the next line. The same thing happens with this variable—it places the cursor on

LISTING 5.2 String-Related Tasks

```
Sub Main
   ' Create a handy variable for adding lines to the output.
   Dim vbCrLf As String
   vbCrLf = Chr(13) + Chr(10)

   ' Define the strings.
   Dim StrOne As String
   Dim StrTwo As String
   Dim StrThree As String
   StrOne = "Hello"
   StrTwo = "World"
   StrThree = "Goodbye"

   ' Show uppercase and lowercase
   MsgBox "Uppercase: " + UCase(StrOne) + _
      vbCrLf  + "Lowercase: " + LCase(StrOne)

   ' Find the middle of the string.
   Dim Middle As Long
   Middle = Len(StrOne) / 2

   ' Display string parts.
   MsgBox "The 'o' is at : " + CStr(InStr(1, StrOne, "o")) + _
      vbCrLf + "The left two characters are: " + _
      Left(StrOne, 2) + vbCrLf + _
      "The right two characters are: " + _
      Right(StrOne, 2) + vbCrLf + _
      "The middle three characters are: " + _
      Mid(StrOne, Middle - 1, 3)

   ' Create composite strings.
   MsgBox StrOne & _
      Space(2) & String(3, "-") & Space(2) & _
      StrTwo
   ' Remove excess space from strings.
   MsgBox StrThree + Chr(33) + vbCrLf + _
      LTrim(StrThree) + Chr(33) + vbCrLf + _
      RTrim(StrThree) + Chr(33) + vbCrLf + _
      Trim(StrThree) + Chr(33)

   ' Compare two strings.
   If StrComp(StrOne, StrTwo) = 0 Then
      MsgBox "The strings are equal."
   ElseIf StrComp(StrOne, StrTwo) = 1 Then
      MsgBox "String one is greater than string two."
   Else
      MsgBox "String one is less than string two."
   End If
End Sub
```

FIGURE 5.3 Using a Simple Function You Can Make Text Uppercase or Lowercase

the next line so you can display the next line of text. To create this special variable, you use the `Chr()` function. In this case, you create control characters—special characters that control the display or application in some way. You can find a list of these special characters at `http://www.cs.tut.fi/~jkorpela/chars/c0.html`.

After the code creates some strings to work with, it begins showing the kinds of tasks you can perform using the string functions. The first step is to show the effect of `UCase()` and `LCase()`, as shown in Figure 5.3. Even though the string originally contains mixed case characters, it now contains either uppercase or lowercase characters. Notice also how the use of `vbCrLf` displays the text on two lines.

One of the tasks you perform relatively often is finding the middle of a string so that you can display it centered. The next bit of code shows how to perform this task. You use the `Len()` function to determine the total length of the string and then simply divide by 2 to find the middle.

The next bit of code uses the `Middle` variable to display parts of the `StrOne` string as shown in Figure 5.4. There are many situations where you need to locate a particular bit of information or display only part of the available information. The `Instr()`, `Left()`, `Right()`, and `Mid()` functions help you perform this task.

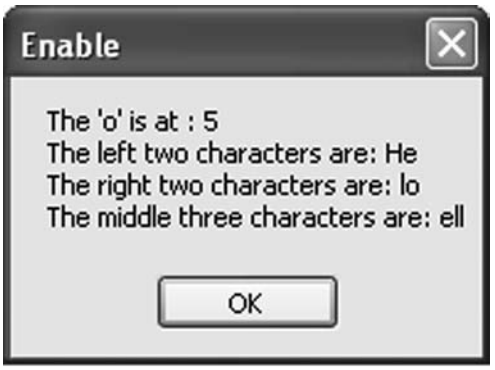


FIGURE 5.4 String Functions Make it Easy to Locate a Particular Piece of Information within a String.

FIGURE 5.5 Define Composite Strings as Needed and Use Repetitive Elements for Decorative Purposes

It's possible to create strings by combining pieces together or using the `String()` and `Space()` functions to define repetitive strings. Figure 5.5 shows a combination of all of these techniques. In this case, using `StrOne`, `StrTwo`, the concatenation operator, `Space()`, and `String()` all work together to create the output shown.

Sometimes strings will have excess space at either end. You use the `LTrim()` function to remove excess space from the left, the `RTrim()` function to remove excess space from the right, and the `Trim()` function to remove excess space from both ends. `StrThree` contains excess space on both ends. Figure 5.6 shows how this string appears initially, and then after using the `LTrim()`, `RTrim()`, and `Trim()` functions. The exclamation mark (`chr (33)`) shows the true end of each version of the string.

The final function, `StrComp()`, lets you compare two strings to determine how one differs from another. The comparison looks at the numeric values of each character in each string. When it finds an inequality, the `StrComp()` function determines which string is greater than the other and outputs a result as shown in Figure 5.7. In this case, `Hello` is definitely less than `World` because the letter H has a lower numeric value than W. Remember that Chapter 6 shows how to use conditional statements, so you don't have to worry too much about the logic of this last bit of code for now. The important thing to remember is that you can compare strings using the `StrComp()` function.

FIGURE 5.6 Remove Excess Space from Strings as Needed

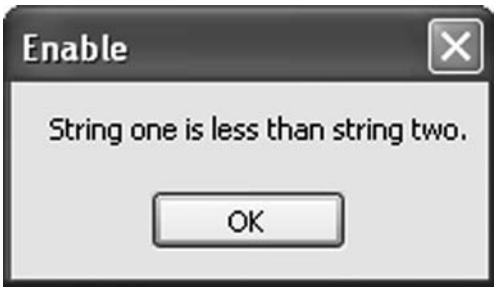


FIGURE 5.7 Compare Strings to Determine Which One Is Greater

Working with Numbers

Besides strings, most people relate best to numbers. After all, one of the first things that most people learn is to count, and counting is simply working with numbers at a very simple level. Computers see numbers as not having a decimal part. The reason for the omission of the decimal is buried in the early hardware of PCs, but these pure integer values receive special emphasis in your computer. IDEAScript provides access to three different sizes of numbers: Byte, Integer, and Long. Listing 5.3 shows an example of all three numeric types in use.

This example defines three numbers of different sizes. It then adds the numbers together, first as a Long and then as an Integer. Notice that you must convert the variables so that they're all the same size. The "Performing Data Conversion" section describes data conversion in more detail.

The next part of this example begins by asking the user for input using `InputBox`. Like the `MsgBox` function, you can provide a message (called a prompt in this case)

LISTING 5.3 Using Numbers

```
Sub Main
   ' Define some numbers.
   Dim MyByte As Byte
   MyByte = 120
   Dim MyInt As Integer
   MyInt = 120
   Dim MyLong As Long
   MyLong = 120
      ' Display sum of the numbers on screen.
   MsgBox CLng(MyByte) + CLng(MyInt) + MyLong
   MsgBox CInt(MyByte) + MyInt + CInt(MyLong)
      ' Ask the user for input and then display it as a number.
   Dim InputNum As Integer
   InputNum = InputBox("Type a number")
   MsgBox "InputNum + MyInt = " + CStr(InputNum + MyInt)
End Sub
```

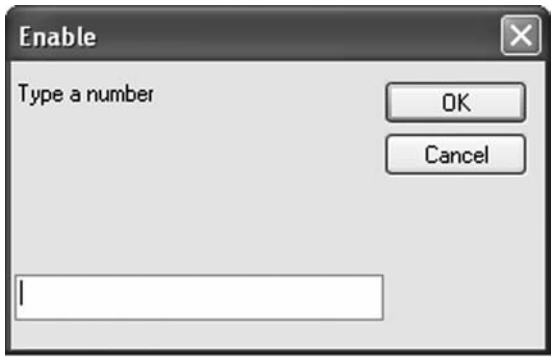


FIGURE 5.8 Obtain Information from the User with InputBox()

and a title for `InputBox()`. It's also possible to provide a default value if you want. Figure 5.8 shows `InputBox()` in use.

Once the user provides input, IDEAScript automatically places the value in `InputNum`. The application adds `InputNum` with `MyInt` to produce a sum of the two values. If the user provides a number with a decimal part, such as 15.6, IDEAScript automatically rounds the input to the next nearest integer value, which is 16 in this case. If the user inputs something unexpected, such as "Hello," IDEAScript displays a type mismatch error message. Chapter 6 tells you how to avoid errors of this sort.

It's important to remember that the size differences are real from IDEAScript's perspective. You can't mix data types together in one line of code without conversion. Mixing data types would result in errors because IDEAScript won't know what data type to use as output. Such errors result in data loss—something you always want to avoid. Consequently, the following code produces a type mismatch error:

```
' Not converting the numbers produces an error.
MsgBox MyByte + MyInt + MyLong
```

IDEAScript will also protect you from making errors such as trying to pour the content of a large box (for example, a Long) into a small box (for example, a Byte). As with mixing data types, trying to put too much data into too small a container will result in data loss. For example, the following code displays an overflow error (see Table 5.1 for a listing of the size of the various data type containers).

```
' Try to stuff too much into a number.
MyByte = 300
MsgBox MyByte
```

Working with Boolean Values

Boolean values simply say yes or no, true or false. The black and white values provided by a `Boolean` variable are important for decision making in IDEAScript. As shown in Listing 5.4, you can use the Boolean data type for all kinds of evaluations.

LISTING 5.4 Performing Comparisons Using a Boolean

```
Sub Main
   Dim MyBool As Boolean

   ' Compare two strings.
   MyBool = "Hello" >  "Goodbye"
   MsgBox MyBool

   ' Convert a number to a Boolean.
   MyBool = CBool(1)
   MsgBox MyBool
End Sub
```

In this case, the letter H is greater than the letter G, so the first comparison is true. In the second case, you also obtain a true result. As far as the Boolean is concerned, any non-zero number (including negative numbers) is true. Chapter 6 uses Boolean values extensively.

Working with Scientific Values

Scientific values are simply another kind of number. However, in this case, the numbers have decimal values. At one time, computers included a special chip to handle scientific values directly, but now your CPU contains this feature. Even so, applications still represent scientific values differently from integer values. You don't need to know how these numbers appear to the computer, simply that they're different. Scientific data, like integer values, can appear in two different sized containers: Single and Double.

Working with Currency Values

Computers work with switches. Everything is either on or off. As a result, a computer normally relies on binary (two value) data. Because it's nearly impossible for humans to work in binary, the early developers of the computer decided to group these switches together. Octal data uses the numbers 0 through 7 and hexadecimal data uses the numbers 0 through 9 along with the letters A through F. Because your application automatically converts these values into something you can understand, you won't normally see any difference between the numbers that you use and the numbers that the computer uses.

Unfortunately, when you convert between binary, octal, or hexadecimal and decimal—the numbering system humans understand—you can incur rounding errors. The errors are small, but still noticeable. Because you don't want rounding errors to affect currency data, IDEAScript provides a special data type, Currency, that mimics the decimal system. The Currency data type eliminates the rounding errors that vex other data types mentioned in this chapter. However, using this data type comes at a price—it affects application performance. Consequently, you only use the Currency data type when you need the extra accuracy it provides.

Working with Date/Time Values

Most businesses need to track the date and time at which events occur—when transactions take place. During an audit, the matter of a single day can mean the difference between a business that has followed the rules and one that hasn't. Consequently, working with date and time values in your application is important. IDEAScript provides the Date data type for this purpose. Listing 5.5 shows just a few of the ways in which you can work with dates and times (you'll see a number of other examples as the book progresses).

The example begins by creating a `Date` variable, `Today`, and filling it with the current date and time using the `Now()` function. If you want just the date in the variable, use the `Date()` function. Likewise, using the `Time()` function gives you just the current time. Figure 5.9 shows the content of `Today`. Your display might vary because the display automatically uses the short date format defined by your computer. Fortunately, you can use the IDEAScript formatting features (described in the "Formatting Data" section) to present the date and time any way you want.

LISTING 5.5 Using Dates

```
Sub Main
   ' Create the required variables.
   Dim Today As Date

   ' Obtain the current date and time.
   Today = Now

   ' Display the date and time.
   MsgBox "The current date and time is: " & Today

   ' Display the date using individual numbers.
   MsgBox "The year is: " & Year(Today) & _
      " The month is: " & Month(Today) & _
      " The day is: " & Day(Today)

   ' Convert the date and time to a number.
   Dim TodayValue As Double
   TodayValue = CDbl(Today)
   MsgBox "The date and time as a number is: " & TodayValue

   ' Add a day to the current date.
   TodayValue = TodayValue + 1
   ' Add 2 minutes to the current time.
   TodayValue = TodayValue + (120 / 86400)

   ' Update Today with the new date and time.
   Today = CDate(TodayValue)
   MsgBox "The updated date and time is: " & Today
End Sub
```

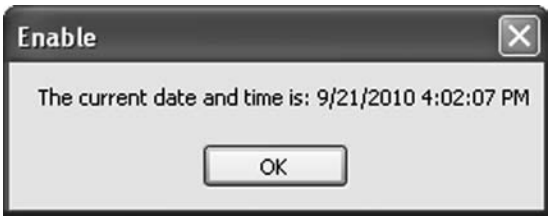


FIGURE 5.9 Date and Time Displayed Using the Short Date Format

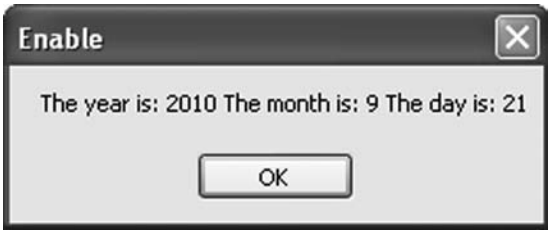


FIGURE 5.10 Displaying the Date as Individual Elements

You can also work with individual date and time elements using the `Year()`, `Month()`, `Day()`, `Weekday()`, `Hour()`, `Minute()`, and `Second()` functions. The example demonstrates the result of obtaining the year, month, and day as shown in Figure 5.10.

It's also possible to convert the date to a number using the `CDbl()` function. Make sure you use a Double to ensure accuracy. Using a number makes it possible to perform date arithmetic. The whole portion of the value is the number of days since 12/31/1899 (the value `1` produces this result). To add another day, you simply add `1` to the whole portion of the number.

The fractional portion of the value is the time. Computing the time is a little more difficult, but it can be done. There are 24 hours in a day, 60 minutes in an hour, and 60 seconds in a minute. When you multiply them all together, you get $24 * 60 * 60 =$ 86,400 seconds in a day. Now, let's say you want to add 2 minutes or 120 seconds to the current time. You'd need to add 120/86400 or 0.0013888888888888888888888888888888889 to the current time.

Tip

You can build dates and times using easily recognized values. Simply use the `DateSerial()` function to build a date or the `TimeSerial()` function to build a time. Both functions accept three values that represent the current date (year, month, and day) or time (hour, minute, and second). The output in both cases is a `Date`, which means you might have to perform some fancy coding to obtain a complete date and

(*Continued*)
time. Let's say you want to create a `Date` for May 6, 2010 at 12:01. You'd use the following code to do it:

```
CDate(CDbl(DateSerial(2010, 5, 6)) + CDbl(TimeSerial(12, 01, 0))).
```

This code looks complicated, but just take it apart a bit at a time and you'll see it isn't. The code begins by creating two `Date` variables—one of which contains the date and the other contains the time. Because you want a single date and you can't add `Date` variables together, you must convert each value to a Double, add them together, and then convert them back into a Date.

The next part of the example performs some date and time arithmetic for you. It adds 1 day and 2 minutes to the current time and displays them on screen. Of course, before you can display anything on screen, you have to convert the numeric value to a Date. There are three options for performing this task. Use the `CDate()` function if you want to convert both date and time. Use the `DateValue()` function for just the date and the `TimeValue()` function for just the time.

Working with Variant Data

The Variant data type is a container that can contain any kind of data. You can use it when you truly don't know what kind of data that a variable will hold. The fact that a Variant is a kind of wildcard is appealing for many developers because using a Variant can solve a number of problems. However, the Variant data type extracts a heavy price because it has the following disadvantages.

- A Variant consumes more memory, which can bloat your application size and make it use resources inefficiently.
- IDEAScript runs slower because it must perform special processing on the Variant data type.
- Your code is harder to read because no one knows what kind of data the variable holds.

A Variant has another special property that other data types don't possess. When most variables are empty, they don't contain any data, but they still have a value of empty. However, there's a special condition where a Variant variable contains nothing. Such a variable is *null*—nothing. The Variant data type is null when you don't assign a value to it or when you specifically set it to `Null`. Fortunately, there's a special function, `IsNull()`, that you can use to detect empty variant data types as shown in Listing 5.6.

The concept of `Null` is important. If you create a string variable, it contains a zero-length string. Yes, it's empty, but it still contains something. A Variant that's `Null` contains

LISTING 5.6 Assigning a Null Value to a Variant

```
Sub Main
   ' Create the variable and make it Null.
   Dim MyVar As Variant
   MyVar = Null

   ' Check for a Null value.
   If IsNull(MyVar) Then
      MsgBox "MyVar is Null"
   End If
End Sub
```

nothing at all, which is something you can check for in your application when you're expecting something from an external source. The Variant data type appears often in the more advanced sections of this book because it's so versatile.

Performing Data Conversion

It's nearly impossible to work with variables without performing some type of data conversion. In fact, many of the examples in this chapter contain data conversion functions. *Data conversion* is simply the act of removing data from one container, making it acceptable for insertion into another container, and then inserting it into that new container. The most common use of data conversion is to make data work in a different way or appear in a different manner. Table 5.3 contains a listing of all of the data conversion functions that IDEAScript supports.

Of course, before you can convert something, you need to know its data type. In most cases, you simply read the code and make your choice based on what you see. However, there are situations where you won't know the data type of a particular variable, especially when you use the Variant type. In this case, you can detect the variable type using the `VarType()` function. Unfortunately, `VarType()` outputs a numeric value that doesn't really say much to humans, so Listing 5.7 shows one way to convert the output value to a string that you can read.

In this example, the code begins by creating an array and then assigning values to the individual members of that array. You'll find that arrays are extremely useful and Chapter 11 describes them in depth. However, for now, all that you really need to know is that there's a single obtain in the code that contains a list of values. Because `Init()` appears as the first subprocedure, IDEAScript executes it first and creates the array.

Eventually, `Init()` calls `Main()`. The `Main()` subprocedure creates a string variable named `X` that contains a string, `Hello`. At this point, the code calls `VarType()` to identify the variable type of `X`, which you already know is a string. `VarType()` returns a number, which isn't particularly useful. However, that number can select one of the strings in `VarTypes`. `VarTypes` supplies this string to `MsgBox` and you see the variable type as shown in Figure 5.11. Try this with a number of other variable types and you'll see that the technique works every time.

TABLE 5.3 IDEASCript Conversion Functions

Conversion Function	Description
Asc	Returns a numeric value that's the ASCII code for the first character in a string.
CBool	Converts a valid numeric expression to a Boolean.
CDate	Converts a valid string date expression to a Date. The string expression may include just the date or both the date and time. Valid string expressions will be dependent upon the user's regional data settings.
CDbl	Converts numeric expressions from one data type to a Double.
CInt	Converts any valid numeric expression to an Integer.
CLng	Converts any valid numeric expression to a Long.
CSng	Converts any valid numeric expression to a Single
CStr	Converts any valid expression to a String.
CVar	Converts any valid expression to a Variant.
Date	Returns a String containing the current system date.
DateSerial	Returns a variant (Date) corresponding to the year, month, and day parameters that were passed to the function. All three parameters are required and must be valid numeric expressions.
DateValue	Returns a variant (Date) corresponding to the string date expression that was passed to the function. The DateExpression can be a string or any other expression that represents a valid date, time, or both date and time.
Day	Returns the day of the month for a specified date expression.
Fix	Returns the integer portion of a number. For negative numbers, Fix returns the first integer greater than or equal to the number.
Format, Format$	Formats a string, number, or variant (date/time) data type to a format expression. Format returns a variant. Format$ returns a string.
Hex, Hex$	Returns the hexadecimal value of a decimal parameter. Hex returns a variant. Hex$ returns a string.
Hour	Returns an integer representing the hour of the day for the time value in the parameter.
Int	Returns the integer portion of a number. For negative numbers, Int returns the first integer less than or equal to the number.
Minute	Returns an integer between 0 and 59 representing the minutes in the time value parameter.
Month	Returns an integer between 1 and 12, inclusive, that represents the month of the year.
Now	Returns a date that represents the current date and time according to the setting of the computer system's date and time.
Oct	Returns the octal value of a decimal parameter.
Second	Returns an integer between 0 and 59 representing the seconds portion of the time value in the parameter.
Str	Returns the string value of a numeric expression.
Time	Returns the current system time.

Conversion Function	Description
TimeSerial	Returns a variant (Time) for the supplied parameters hour, minute, and second. If the minute or second parameters aren't provided, the function will display 0 for those values.
TimeValue	Returns a variant (Time) representing a time, based on the parameter TimeString.
Val	Returns the numeric value of a string of numbers. To obtain a valid result, the string cannot contain nonnumeric characters.
Weekday	Returns an integer from 1 to 7 indicating the day of the week for the parameter date.
Year	Returns an integer representing the year in the date expression.

LISTING 5.7 Determining the Data Type of a Variable

```
Dim VariableTypeArray(0 To 13) As String

Sub Init
   ' Fill the VariableTypeArray array.
   VariableTypeArray(0) = "Empty"
   VariableTypeArray(1) = "Null"
   VariableTypeArray(2) = "Integer"
   VariableTypeArray(3) = "Long"
   VariableTypeArray(4) = "Single"
   VariableTypeArray(5) = "Double"
   VariableTypeArray(6) = "Currency"
   VariableTypeArray(7) = "Date"
   VariableTypeArray(8) = "String"
   VariableTypeArray(9) = "Object"
   VariableTypeArray(10) = "Error"
   VariableTypeArray(11) = "Boolean"
   VariableTypeArray(12) = "Variant"
   VariableTypeArray(13) = "DataObject"

   ' Call the Main subprocedure.
   Main
End Sub

Sub Main
   Dim X As String
   X = "Hello"

   MsgBox "Type X: " + VariableTypeArray(VarType(X))
End Sub
```

FIGURE 5.11 Show the Data Type of Variable X

The interesting part of this example is that you've just created a custom data conversion. As you write more applications, you'll find that you require customized data conversions to make your application work properly. Even though Table 5.3 contains all the generic data conversions you should need, make sure you consider any custom requirements as well.

Employing Operators

An *operator* simply tells IDEAScript what operation or task to perform. For example, you may want IDEAScript to add two numbers together, so you use the + operator to signify that task.

Most operators, such as +, −, *, and /, affect just two variables. These operators are called *binary* operators because they affect two entities. A few operators, such as negation (which also uses the – operator) affect just one variable. These operators are called *unary* operators. For example, when you type -5, you apply the unary negation operator to the number 5.

Operator sequences can become very complex, especially when performing math-related tasks. Because you don't want IDEAScript to work in a chaotic environment, it relies on operator precedence to determine which tasks to perform first. For example, IDEAScript will perform all multiplication and division before it performs addition and subtraction. There's also a special case of the parenthesis. The parenthesis is a grouping operator and signifies what task to perform first. If you want to perform addition before multiplication, then you can use the parenthesis to tell IDEAScript to perform tasks in that order. Table 5.4 shows all of the IDEAScript operators in order of precedence.

Most of the operators in Table 5.4 should be familiar if you've read through this chapter. In fact, most of the examples so far in this book have relied on operators to some extent. You'll make extensive use of the comparison operators in Chapter 6, but even so, you've already seen some of them in the examples shown so far.

There's a special operator case that you need to know about when working with strings. If you use the + operator, IDEAScript expects you to perform an operation or task with two strings, and nothing else. For example, the following code displays a type mismatch error:

```
' Create the required variables.
Dim Today As Date
' Obtain the current date and time.
Today = Now
' Display the date and time.
MsgBox "The current date and time is: " + Today
```

Now, look back at Listing 5.5. You'll notice that this listing uses the & operator. When you use the & operator, IDEAScript will convert the elements of the string you want to create into strings. Of course, the & operator assumes that there's a generic conversion or it will also display the type mismatch error. Whenever possible, use the + operator to

TABLE 5.4 IDEASCript Operators in Order of Precedence

Operator	Description	Examples
()	Groups a part of the equation to override the normal order of precedence.	(5 + 2) * 2 = 14 5 + (2 * 2) = 9
^	Exponentiation—raises a variable to a power.	2 ^ 2 = 4 2 ^ 3 = 8
+, −	Performs unary identity and negation.	−3 + 4 = 1
*, /	Multiplies or divides two variables.	2 * 2 = 4 4/2 = 2
Mod	Obtains the remainder from integer division.	5 Mod 2 = 1
+, −	Adds or subtracts two variables.	5 + 2 = 7 5 – 2 = 3
+, &	String concatenation—combines two strings together.	"Hello" + "World" = "Hello World"
=, <>, <, <=, >, >=	Comparison operators—determines the relationship between the values of two variables.	1 < 2 = True 2 < 1 = False 2 <= 2 = True
Not, And, Or, Xor	These logical comparison and bitwise operators help you compare two values or modify a single value in a logical way. The comparison operators include: conjunction (And), inclusive disjunction (Or), and exclusive disjunction (Xor). The bitwise operator is Not.	Not 5 = −6 5 And 4 = 4 5 Or 4 = 5 5 Xor 4 = 1
Eqv, Imp	Logical equivalence and logical implication (see how these operators are used later in this section).	False Eqv True = False False Imp True = True

TABLE 5.5 Logical Implication

P	Q	PQ
True	True	True
True	False	False
False	True	True
False	False	True

concatenate strings because it provides greater speed since IDEA doesn't have to perform as many checks.

There are a few operators that will likely be unfamiliar to even seasoned developers because they simply aren't used very often in standard coding. The `Eqv` and `Imp` operators fall into this category. Logical equivalence and implication are somewhat hard to grasp, but they're sometimes used for complex logic scenarios. If you don't ever engage in such complex logic, you can probably skip the rest of this section.

Implication means that one proposition implies another. In short, the only time implication is false is when the first proposition is true and the second is false. In this case, the true proposition didn't imply the second false proposition. Table 5.5 shows the truth table for implication.

Implication can produce some very odd results until you really consider how it works. Let's say you use the following code:

```
' Create the variables
Dim P As Integer
P = 12
Dim Q As Integer
Q = 10
Dim PQ As Integer

' Determine Implication
PQ = P Imp Q
MsgBox PQ
```

Using implication, you're really comparing the bit logic of the two numbers. Consequently, what you're really doing here is saying:

```
 P = 0000 1100
 Q = 0000 1010
-------------
PQ = 1111 1011
```

When you run the code, you see the output of –5, which is precisely what you would expect after viewing the binary values (a leading 1 always translates to a negative number when working with signed variables such as an Integer). Using the Windows Calculator in scientific mode can really help seeing how these outputs work. Put the calculator into binary mode, set the input type to Byte, type the value shown for `PQ`,

TABLE 5.6 Logical Equivalence

P	Q	PQ
True	True	True
True	False	False
False	True	False
False	False	True

click +/−, and then revert to decimal mode. You'll see the output is `5` (which is the negation of the `-5` output).

Equivalence also works at the binary level, but the truth table is different from implication. In this case, you're saying that logically, two propositions are equivalent (not true). If they're both true or both false, the output is true because the propositions are equivalent to each other. Table 5.6 shows logical equivalence.

To understand how `Eqv` affects the output of your application, it helps to look at the binary math again. Using the same numbers as before, here's the math you see when you try `PQ = P Eqv Q`.

```
 P = 0000 1100
 Q = 0000 1010
-----------
PQ = 1111 1001
```

In this case, the output is -7, as shown by the binary math above. Again, it's a good idea to try the numbers out on your Windows Calculator.

Formatting Data

Formatting your data may not seem very important as you experiment with IDEAScript, but it becomes very important when you create reports or interact with other applications. Fortunately, IDEAScript provides a number of functions for formatting your data: `Format()`, `Format$()`, `Hex()`, `Hex$()`, `Oct()`, and `Oct$()`. In all cases, the default version of the function, such as `Format()`, returns a variant, while the $ version of the function, such as `Format$()` returns a string.

The `Hex()`, `Hex$()`, `Oct()`, and `Oct$()` functions all perform essentially the same task—they convert your decimal number to either a hexadecimal or octal equivalent. You can test this functionality on the Windows Calculator in scientific mode if you like. For example, type `16` in decimal mode (the `Dec` option is highlighted) and then click `Hex`. You'll see that `16` decimal is equal to `10` hexadecimal. Now click `Oct` and you'll see that the same number is also equal to `20` octal. In most cases, you won't need this functionality unless you're interacting with another application—a task discussed later in this book.

The `Format()` and `Format$()` functions are the two that receive the attention in this chapter. These two functions accept your unformatted input and create a formatted equivalent using the formatting characters and strings shown in Table 5.7.

TABLE 5.7 IDEASCript Formatting Strings

Formatting Character or String	Type	Description
−+ $ ( )	Numeric	Displays the literal character. To display a character other than one of those listed, precede it with a backslash (\).
!	General	Displays a character or a space. Placeholders are filled from right to left unless there's an ! character in the format string. When there's an ! character in the format string, it forces placeholders to fill from left to right instead of right to left.
#	Numeric	A digit placeholder that displays a digit or nothing. If there's a digit in the expression being formatted in the position where the # appears in the format string, it's displayed; otherwise, nothing's displayed.
%	Numeric	A percentage placeholder that's inserted in the position where it appears in the format string. The expression is multiplied by 100.
&	General	A character placeholder. Display a character or nothing.
,	Numeric	A thousands separator that separates thousands from hundreds within a number that has four or more places to the left of the decimal separator. Use of this separator as specified if the format statement contains a comma surrounded by digit placeholders (0 or #). Two adjacent commas or a comma immediately to the left of the decimal separator (whether or not a decimal is specified) means "scale the number by dividing it by 1000, rounding as needed."
.	Numeric	A decimal placeholder that determines how many digits are displayed to the left and right of the decimal separator.
/	Numeric	The date separator. The actual character used depends on the date format specified in the Regional and Language Options area of the Windows Control Panel.
:	Numeric	The time separator. The actual character used as the time separator depends on the time format specified in the Regional and Language Options area of the Windows Control Panel.
@	General	A character placeholder.
\	Numeric	Displays the next character in the format string. The backslash itself isn't displayed. To display a backslash, use two backslashes (\\).
<	General	Forces lowercase.
>	General	Forces uppercase.
0	Numeric	A digit placeholder that displays a digit or a zero. If the number being formatted has fewer digits than there are zeros (on either side of the decimal) in the format expression, leading or trailing zeros are displayed. If the number has more digits to the right of the decimal separator than there are zeros to the right of the decimal

Formatting Character or String	Type	Description
		separator in the format expression, the number's rounded to as many decimal places as there are zeros. If the number has more digits to the left of the decimal separator than there are zeros to the left of the decimal separator in the format expression, the extra digits are displayed without modification.
a/p	Date/Time	Uses the 12-hour clock and displays a lowercase a (for am) or p (for pm).
A/P	Date/Time	Uses the 12-hour clock and displays an uppercase A (for AM) or P (for PM).
am/pm	Date/Time	Uses the 12-hour clock and displays a lowercase am or pm.
AM/PM	Date/Time	Uses the 12-hour clock and displays an uppercase AM or PM.
AMPM	Date/Time	Uses the 12-hour clock and displays the contents of the 11:59 string (s1159) in the WIN.INI file with any hour before noon. Displays the contents of the 2359 string (s2359) with any hour between noon and 11:59 PM. AMPM can be either uppercase or lowercase, but the case of the string displayed matches the string as it exists in the WIN.INI file. The default format is AM/PM.
c	Date/Time	Displays the date in dddd format and displays the time in ttttt format.
d	Date/Time	Displays the day as a number without a leading zero (1–31).
dd	Date/Time	Displays the day as a number with a leading zero (01–31).
ddd	Date/Time	Displays the day as an abbreviation (Sun-Sat).
ddddd	Date/Time	Displays a date serial number as a complete date (including day, month, and year).
E-, E+,e-, e+	Numeric	The scientific format. It contains at least one digit placeholder (0 or #) to the right of E−, E+, e−, or e+. The number is displayed in scientific format with E or e inserted between the number and its exponent. The number of digit placeholders to the right determines the number of digits in the exponent. Use E− or e− to place a minus sign next to negative exponents. Use E+ or e+ to place a plus sign next to positive exponents.
Fixed	Numeric	Displays at least one digit to the left and two digits to the right of the decimal separator.
General	Date/Time	Displays a date and/or time. For real numbers, displays a date and time. For example, 4/3/93 03:34 PM. If there's no fractional part, displays only a date. For example, 4/3/93. If there's no integer part, displays time only. For example, 03:34 PM.

(*continued*)

TABLE 5.7 (*Continued*)

Formatting Character or String	Type	Description
General	Numeric	Displays the numbers as is, with no thousands separators.
h	Date/Time	Displays the hour as a number without leading zeros (0–23).
hh	Date/Time	Displays the hour as a number with leading zeros (00–23).
Long Date	Date/Time	Displays a long date as defined in the Regional and Language Options area of the Windows Control Panel. For example, February 21, 2006.
Long Time	Date/Time	Displays a long time, as defined in the Regional and Language Options area of the Windows Control panel. Long Time includes hours, minutes, seconds, and the AM/PM designator. For example, 9:55:24 AM.
Medium Date	Date/Time	Displays a date in a similar form as the short date, as defined in the Regional and Language Options area of the Windows Control Panel, except the month is abbreviated. For example, 21/Feb/2007.
Medium Time	Date/Time	Displays time in the 12-hour format using hours, minutes, and the AM/PM designator. For example, 9:55 AM.
m	Date/Time	Displays the month as a number without a leading zero (1–12). If m immediately follows h or hh, the minute rather than the month is displayed.
mm	Date/Time	Displays the month as a number with a leading zero (01–12). If mm immediately follows h or hh, the minute rather than the month is displayed.
mmm	Date/Time	Displays the month as an abbreviation (Jan-Dec).
mmmm	Date/Time	Displays the full month name (January-December).
n	Date/Time	Displays the minutes as a number without leading zeros (0–59).
nn	Date/Time	Displays the minutes as a number with leading zeros (00–59).
Null string	Numeric	Displays the number with no formatting.
Percent	Numeric	Displays numbers multiplied by 100 with a percent sign (%) appended to the right. Displays two digits to the right of the decimal separator.
q	Date/Time	Displays the quarter of the year as a number (1–4).
s	Date/Time	Displays the seconds as a number without leading zeros (0–59).
Scientific	Numeric	Uses standard scientific notation.
Short Date	Date/Time	Displays a short date, as defined in the Regional and Language Options of the Windows Control Panel. For example, 21/02/2007.
Short Time	Date/Time	Displays a time using the 24-hour format. For example, 09:55.
ss	Date/Time	Displays the seconds as a number with leading zeros (00–59).
Standard	Numeric	Displays numbers with a thousands separator, and if appropriate, displays two digits to the right of the decimal separator.

Formatting Character or String	Type	Description
True/False	Numeric	Displays False if a number is 0, otherwise displays True.
ttttt	Date/Time	Displays a time serial number as a complete time (including hour, minute, and second) formatted using the time separator defined in the Regional and Language Options area of the Windows Control Panel. A leading zero is displayed if the Leading Zero option is selected and the time is before 10:00 AM or PM. The default time format is h:mm:ss.
w	Date/Time	Displays the day of the week as a number (1–7).
ww	Date/Time	Displays the week of the year as a number (1–52).
y	Date/Time	Displays the day of the year as a number (1–366).
yy	Date/Time	Displays the year as a two-digit number (00–99).
yyyy	Date/Time	Displays the year as a four-digit number (100–9999).

The formatting strings or characters you use depend on the kind of data you want to format. Using strings tends to provide a single solution for common formatting tasks, while using characters tends to provide a customizable approach. Later chapters in this book will show you how to work with most of the formatting features IDEAScript provides. However, Listing 5.8 provides you with some samples you can use to get started.

The code begins by defining a Date variable, `Today`, and giving it today's date. The first dialog box uses formatting strings to produce the common formats shown in Figure 5.12. The same variable outputs all of this information by using simple formatting changes.

Of course, you may find that none of the string formats meet your needs. In this case, you can create a custom format using the characters described in Table 5.7 to define a string like the one shown for the second message box. Figure 5.13 shows the results of this custom format. As you can see, formatting lets you output the same data in many different ways.

Creating Custom Data Types

In some cases, you must model complex data. Trying to work with complex data when all you have are simple data types is hard, sometimes impossible. A user-defined type can help you model complex data to make application coding easier.

Let's say you want to create a complex type for a call record that includes the name of the caller, the time and date of the call, the telephone number, and the message. You'd use the Type data type as shown here to perform the task:

```
Type PhoneMessage
   CallerName As String
   TimeCalled As Date
   ContactNumber As String
   Message As String
End Type
```

LISTING 5.8 Using Formatting with Dates

```
Sub Main
   ' Create a handy variable for adding lines to the output.
   Dim vbCrLf As String
   vbCrLf = Chr(13)  + Chr(10)

   ' Create the required variables.
   Dim Today As Date

   ' Obtain the current date and time.
   Today = Now

   ' Display the date and time.
   MsgBox "The current date and time is: " & _
      vbCrLf & "Long Date: " & Format$(Today, "Long Date") & _
      vbCrLf & "Medium Date: " & Format$(Today, "Medium Date") & _
      vbCrLf & "Short Date: " & Format$(Today, "Short Date") & _
      vbCrLf & "Long Time: " & Format$(Today, "Long Time") & _
      vbCrLf & "Medium Time: " & Format$(Today, "Medium Time") & _
      vbCrLf & "Short Time: " & Format$(Today, "Short Time")

' Display a custom time and date.
   MsgBox "Custom Time and Date: " & _
      Format$(Today, "hh:mm ddd, dd mmm yyyy")
End Sub
```

FIGURE 5.12 Using the Common Formats for Date and Time

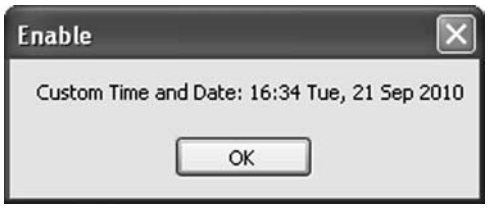


FIGURE 5.13 Rely on Custom Formatting When None of the Standard Formats Work

In this case, you create a `PhoneMessage type`. The content of this complex type isn't anything new—you've already used these simple types in other places in this chapter. All you're really doing is combining them to create something new. You'd use this new type as shown here:

```
Sub Main
   ' Create and fill the user defined type.
   Dim MyCall As PhoneMessage
   MyCall.CallerName = "Sam"
   MyCall.TimeCalled = Now
   MyCall.ContactNumber = "1(555)555-1212"
   MyCall.Message = "Call me back!"

   ' Display the information.
   MsgBox MyCall.CallerName & " called at " & _
      MyCall.TimeCalled & " and says "' & _
      MyCall.Message & "'. You can reach the party at " & _
      MyCall.ContactNumber & "."
End Sub
```

Declaring a variable using the new type is no different from any other variable. However, notice that there's a difference in filling the variable with information. You specify the variable name, followed by a period (`.`), followed by the internal variable name. This is called *dot syntax* and you'll encounter it a number of times in this book. Except for this little difference, using your user-defined type isn't any different from using any other type described in this chapter.

Summary

This chapter helps you discover the value of internal data—the data your application uses to perform tasks. As you know by now, IDEAScript can work with a wide variety of data types, each of which has its place in your application. By combining various data types with operators, you can create composite information. This composite information represents a new kind of data that didn't exist previously. It's also possible to format data to present it nicely on screen. When you find that you don't have the kind of data you

need, a kind of complex data that most businesses require, you can create a user-defined type to meet the need.

For many readers, this chapter is going to become a reference chapter. You'll use it quite often as you build applications using IDEAScript. Don't try to memorize everything in this chapter, but instead, become familiar with the content so that you know where to find the information you need. Try out all of the examples, and then modify the examples to see how changes affect the output. The more you work with various data types, the better you become at choosing a data type for a particular application need.

Now is also a good time to begin discussing custom data in your application. Think about things like a customer record or report results. You don't quite have the skills required to create truly complex data types yet, but you do have enough knowledge to begin thinking about these complex data types. The more you think about them, the better your chances of creating the right complex data type when the time comes. Make sure you get input from others in your organization as to the content of the complex data type.

Chapter 6 examines the next rung on the programming ladder. Now that you know about data types, operators, formatting, and complex data types, it's time to begin using that information to create flexible applications. You've already seen the use of conditional statements and loops in this book. Now it's time to see how they work and what you can use them for within your application.

CHAPTER 6

Using Conditional Statements and Loops

This chapter is all about controlling your application. For the most part, the examples so far have run straight through a procedure—a particular set of steps for performing a task. However, the real world doesn't work that way and neither can your application. As you perform tasks, you make decisions and sometimes need to perform a task more than once—such as write the check, place it in the envelope, seal the envelope, add postage, repeat until bills are paid. The decision-making part of an application is called a *conditional statement*, while the repetitive part is called a *loop*.

Both conditional statements and loops provide organization for your application. In fact, most developers call them *structures*. For example, you use an `If...Then` structure to hold code that conditionally executes based on the condition you provide. Structures are an essential part of applications and they come in many forms, as you'll discover as the book progresses.

Making Decisions Using the If. . .Then. . .Else Statement

The bread and butter of decision making for the developer is the `If...Then` statement. In fact, you've already seen this statement in use in some of the examples in previous chapters. The following sections describe the various forms of the `If...Then` statement.

Using If. . .Then Alone

The `If...Then` statement is the most basic form of conditional statement that you can use. You use it to optionally execute code within your application. It relies on the following form:

```
If  <Condition>  Then
    <Statements>
End If
```

The `<Condition>` provides a True/False statement to analyze. When the statement is `True`, the `<Condition>` passes, and IDEA executes the `<Statements>` within the `If...Then` structure. Here's an example of an `If...Then` conditional statement:

```
' Create a variable that indicates happiness.
Dim Happy As String

'Obtain the happiness factor of the user.
Happy = InputBox$("Are you happy? (Yes/No)")

' Make a decision.
If LCase(Happy) = "yes" Then
   MsgBox "You're happy! Great!"
End If
```

In this case, the code relies on the `InputBox$()` function to obtain a string value from the user. Remember that the `InputBox()` function obtains a numeric value from the user. After the user enters a value, the code relies on the `LCase()` function to change it to lowercase. The code then evaluates the `LCase(Happy) = "yes"` condition. When the user supplies a value of `"yes,"` the code executes a message box congratulating the user.

Using the If. . .Then. . .Else Combination

The `If...Then` statement has a limitation. If the condition evaluates to `False`, the application doesn't execute any code at all. The `If...Then...Else` statement overcomes this problem. In this case, the code performs one task or another, but not both, based on the truth of the condition you provide. The `If...Then...Else` statement takes the following form:

```
If  <Condition>  Then
    <Statements>
Else
    <Statements>
End If
```

Using the `If...Then...Else` statement brings additional flexibility because you can execute one of two choices. Here's the previous example with a new twist added. The code can now react when the user provides yes or no as an answer.

```
' Choose one option or the other.
If LCase(Happy) = "yes" Then
   MsgBox "You're happy! Great!"
Else
   MsgBox "Sorry to hear you aren't happy."
End If
```

Using the If. . .Then. . .ElseIf Combination

A single conditional statement can only provide two answers, True or False. Some real world situations may not react well to the True/False situation. You may have to look for

three or even four potential solutions. Of course, in order to provide this functionality, you must provide more than one condition. In fact, you need one less condition than the number of potential choices. The `If...Then...ElseIf` structure takes the form shown here.

```
If <Condition> Then
   <Statements>
Else If <Condition> Then
   <Statements>
Else
   <Statements>
End If
```

The `Else` portion of the structure is optional. You don't have to provide it unless there really is a catchall answer for situations that the conditions can't handle. In the previous example, a user could enter yes, no, or something else—something not an answer. The `If...Then...ElseIf` structure handles this problem with aplomb as shown here.

```
' Handle non-answers as well as good answers.
If LCase(Happy) = "yes" Then
   MsgBox "You're happy! Great!"
ElseIf LCase(Happy) = "no" Then
   MsgBox "Sorry to hear you aren't happy."
Else
   MsgBox "Please provide a valid answer!"
End If
```

In this case, the code checks for two different possible `Happy` contents. These are the two expected answers. Of course, the user could type anything—maybe, the number 1, or even their name. The `Else` clause handles these unexpected answers and provides the user with an error message (albeit, a limited error message in this case).

Choosing between Options Using Select Case

In some cases, you need to check for a number of answers found in a single variable. When you have two or three potential answers, you can probably get by using an `If...Then...ElseIf` structure. However, as the number of potential answers increases, you need something better—something easier to use and read. The `Select Case` structure handles this requirement.

Don't get the idea that the `Select Case` structure is a complete replacement for the `If...Then...ElseIf` structure. When working with a `Select Case` structure you must provide just one variable, which greatly decreases the potential conditions you can check within one structure. Consequently, when working with the `Select Case` structure, you give up some amount of flexibility.

Tip

Using the `Select Case` structure does provide some performance benefits in addition to easier-to-read code. The IDEA interpreter can optimize access to a `Select Case` structure in a way that it can't for the `If...Then...ElseIf` structure. The result is a very small, but noticeable, increase in application speed.

Using Select Case

Think of the `Select Case` statement as a kind of menu and you'll be ahead of everyone else in learning to use it. The `Select Case` statement chooses one item from a potential list of items, based on a variable you define. The basic `Select Case` structure takes the form shown here.

```
Select Case <Variable>
   Case <VariableValue1>
      Statements
   Case <VariableValue2>
      Statements
...
End Select
```

Using the `Select Case` structure is straightforward. It's important to provide enough indentation to keep the options separate from the tasks you want to perform. The following example shows a basic `Select Case` structure in operation.

```
' Create a handy variable for adding lines to the output.
Dim vbCrLf As String
vbCrLf = Chr(13)  + Chr(10)

' Create a handy variable for adding tabs to the output.
Dim vbTab As String
vbTab = Chr(9)

' Obtain the user's food selection.
Dim MenuOption As Integer
MenuOption = InputBox( "Please select a food option:" _
   & vbCrLf & vbTab & "1. American" _
   & vbCrLf & vbTab & "2. Chinese" _
   & vbCrLf & vbTab & "3. Mexican" _
   & vbCrLf & vbTab & "4. German")

' Display the selection on screen.
Select Case MenuOption
   Case 1
      MsgBox "You chose American."
```

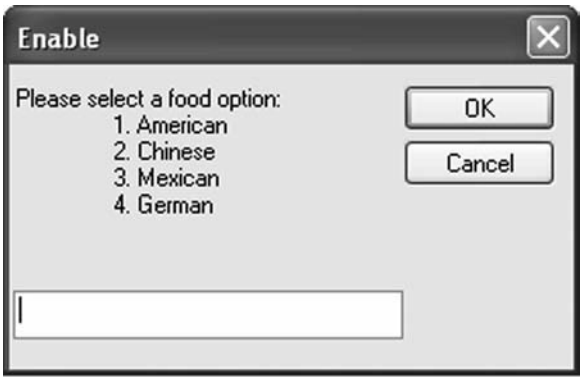


FIGURE 6.1 Use Line Feeds and Tabs to Create an Attractive InputBox() for the User

```
   Case 2
      MsgBox "You chose Chinese."
   Case 3
      MsgBox "You chose Mexican."
   Case 4
      MsgBox "You chose German."
End Select
```

The code begins by defining a new constant that you'll want to add to your collection, `vbTab`. The `InputBox()` function uses this new constant as part of the display processing. The resulting dialog box is quite usable despite being so simple, as shown in Figure 6.1.

After the user selects one of the numeric options (a text choice won't work), the code uses a `Select Case` to choose one of the predefined options and display a message box on screen. Notice that the `Select Case` will continue processing statements until it comes to the next `Case` clause in the list. If none of the `Case` clauses match the variable content, then the `Select Case` statement doesn't do anything.

Defining a Default Action Using Case Else

Normally, you use the `Select Case` statement to handle a well-defined list of choices. However, there are times when you have a well-defined list of choices and you still want to handle errant input, such as a user making the wrong choice. After all, it's frustrating to provide input and then have the application ignore you. The `Case Else` clause handles this situation. Here's the form of a `Select Case` statement with the `Case Else` clause added.

```
Select Case <Variable>
   Case <VariableValue1>
      Statements
   Case <VariableValue2>
      Statements
```

```
...
   Case Else
      Statements
End Select
```

The previous `Select Case` example presented four choices. As long as the user selects one of the four choices, the application displays the menu selection. However, if the user provides some other answer, the application simply ignores the input. The following version of the example provides an error message that will help the user make the correct choice the next time:

```
' Handle incorrect choices.
Select Case MenuOption
   Case 1
      MsgBox "You chose American"
   Case 2
      MsgBox "You chose Chinese"
   Case 3
      MsgBox "You chose Mexican"
   Case 4
      MsgBox "You chose German"
   Case Else
      MsgBox "You must choose an option between 1 and 4."
End Select
```

Using a Select Case Alternative, Choose

IDEAScript provides a concise alternative to the `Select Case` statement, the `Choose()` function. This function provides an output based on the value of the input provided. You can only use it with numeric input because the number selects the output. In addition, `Choose()` only works when you're expecting a particular text output. It won't work when you need to write a lot of code between each `Case` clause. Even so, you can use `Choose()` in a number of ways to reduce the overall size of your code and potentially make it easier to read. The `Choose()` function takes the following form:

```
Choose(<Variable>, <Output1>, <Output2>, ...)
```

The `Choose()` function works fine with the previous `Select Case` examples. In addition, it's much shorter and easier to read. Here's the `Choose()` version of the first `Select Case` example.

```
' Display the selection.
MsgBox "You chose " & _
   Choose(MenuOption, "American", "Chinese", "Mexican", "German")
```

That's it, just one line of code, in place of the ten lines of code required for an equivalent `Select Case` statement. It's also important to note that there isn't a `Case`

`Else` clause available when using the `Choose()` function, so you can't tell the reader about incorrect choices unless you add other error handling. Still, you can see that `Choose()` provides a valuable alternative to `Select Case` when you need it.

Performing Tasks a Specific Number of Times with For. . .Next

IDEAScript provides two kinds of loop structures, each with a specific purpose. The first kind is `For...Next`. You use a `For...Next` loop when you want to perform a task a specific number of times. The `For Each...Next` structure also performs a task a set number of times, but it relies on the content of an array or collection to determine the number of times to perform the task (see Chapter 11 for more information on working with arrays and collections). The following sections describe both the `For...Next` and `For Each...Next` loops.

Using For. . .Next

The `For...Next` loop performs a task a set number of times. You tell the `For...Next` loop where to start counting and where to end counting. A counter variable keeps track of the count and you can even use the counter variable for calculations within the loop. The `For...Next` loop structure takes the following form:

```
For <Counter> = <StartValue> To <EndValue> [Step <StepValue>]
   <Statements>
Next
```

The `<StartValue>` and `<EndValue>` can be any integer value you want. If you specify a `<StartValue>` of `3` and an `<EndValue>` of `4`, then the code will loop twice, once for `3` and once for `4`.

Tip

You can cause the loop to run backward values by selecting a larger `<StartValue>` than `<EndValue>` and including the `Step` argument with a negative `<StepValue>`, such as `–1`. For example, if the `<StartValue>` is `3`, the `<EndValue>` is `1`, and the `<StepValue>` is `–1`, then the loop will execute backward from `3` to `1`.

The optional `Step` clause determines the amount the loop increments the `<Counter>` for each loop. The default increment value is `1`. However, let's say you wanted to add all of the even values from `2` to `10`. You'd use a `<StartValue>` of `2`, an `<EndValue>` of `10`, and a `<StepValue>` of `2`. Here's an example of using the `For...Next` to calculate `n!` (factorial) for the value of `5`:

```
Sub Main
   ' Create a variable to hold the result.
   Dim Total As Integer
   Total =  1

   ' Define a counting variable.
   Dim Count As Integer

   ' Compute n! (factorial) for 5.
   For Count = 1 To 5
      Total = Total * Count
   Next

   ' Display the result.
   MsgBox "The total is: " & Total
End Sub
```

What this loop is really doing is calculating the value of `1 * 2 * 3 * 4 * 5`. The result is an output of `120`. If you use the Windows Calculator in scientific mode, you'll find that clicking `5` and then `n!` results in an output of `120` as well.

Using For Each. . .Next

While this book hasn't discussed arrays and collections yet, it's important to know about the `For Each...Next` loop. This loop examines each element of an array or collection and does whatever you want with it. Don't worry too much about the specifics just yet, but do follow along with the example because it's a lot of fun. The `For Each...Next` loop takes the following form.

```
For Each <Item> In <CollectionOrArray>
   <Statements>
   [Exit For]
   <Statements>
Next
```

As you can see, the `For Each...Next` loop doesn't include a counter variable or a start or end value. What this loop does is place one element from `<CollectionOrArray>` on each loop in `<Item>`. You can then process `<Item>` any way you like. The loop ends when every element in `<CollectionOrArray>` is processed.

Of course, you may not always want to process every element in `<CollectionOrArray>`. The optional `Exit For` clause lets the code exit the loop prematurely. Execution resumes at the next statement after the `For Each...Next` loop. Normally, you place the `Exit For` clause within an `If...Then` conditional statement. When an exit condition becomes True, the loop exits. Here's an example of the `For Each...Next` loop in action (you may not understand everything just yet—make sure you come back to this example after you read Chapter 11 later).

```
Sub Main
   ' Create an array of values.
   Dim MyMessage(5) As String

   ' Define a counter.
   Dim Counter As Integer

   ' Define the output variable.
   Dim Result As String

   ' Fill the array with data.
   For Counter = 1 To 5
      MyMessage(Counter) = Chr(64 + Counter)
   Next

   ' Define the result.
   For Each Letter In MyMessage
     Result = Result + Letter
   Next

   ' Display the array content.
   MsgBox Result
End Sub
```

The example begins by creating an array of strings. Think of this array in the same way as you think of an apartment mailbox. Each individual box holds the mail for one apartment, but the mailbox as a whole holds everyone's mail. `MyMessage` is the mailbox and it contains five boxes for individual strings.

After the code creates `MyMessage`, it fills it with some letters. The letter A is `Chr(65)`. The `For...Next` loop fills each of the individual elements of `MyMessage` with a different letter.

At this point, the code has an array filled with data to process. It uses the For `Each...Next` loop to look at each element in `MyMessage`. During each loop, `Letter` contains a different letter from `MyMessage` and places it in `Result`. Finally, the code displays `Result` to show you the letter string found in Figure 6.2.

FIGURE 6.2 This Single String Started as Individual Letters

Performing Tasks Using Conditions with Do Loop

The `Do Loop` group of statements are the ones to use when you don't know how long a loop should run. These loops basically say that IDEA should keep working until the task is done. Of course, you specify task completion as part of a condition. The main differences between the four kinds of `Do Loop` statements are when the loop evaluates the condition and how IDEA evaluates the condition, which can have a big impact on how your application runs. The `Do Loop` statements include:

- `Do While...Loop`
- `Do...Loop While`
- `Do Until...Loop`
- `Do...Loop Until`

To make this example easier to understand, `Main()` will call four different functions—each of which demonstrates a different `Do Loop` statement. (These functions appear in the sections that follow and you must type both `Main()` and the functions it calls into a single file.) Here's the `Main()` code for this example:

```
Sub Main
   ' Create a result variable.
   Dim Result As Integer

   ' Create an ending condition.
   Dim EndValue As Integer
   EndValue = 1

   ' Obtain the result of Do While...Loop.
   Result = DoWhile(EndValue)
   MsgBox "The result of Do While...Loop is: " & Result

   ' Obtain the result of Do...Loop While.
   Result = DoLoopWhile(EndValue)
   MsgBox "The result of Do...Loop While is: " & Result

   'Obtain the result of Do Until...Loop
   Result = DoUntil(EndValue)
   MsgBox "The result of Do Until...Loop is: " & Result

   'Obtain the result of Do...Loop Until
   Result = DoLoopUntil(EndValue)
   MsgBox "The result of Do...Loop Until is: " & Result
End Sub
```

In this case, `Main()` essentially functions to call the four functions and display the results from each of them. It passes `EndValue` to each of the functions and changing `EndValue` makes a difference in the results you obtain (try different values just to

see what happens). The following sections describe each of the `Do While...Loop` statements.

Using Do While. . .Loop

The `Do While...Loop` statement checks a condition before it begins performing the tasks that you require. As a consequence, the `Do While...Loop` may not even execute once if the conditions aren't right. The `Do While...Loop` takes the following form.

```
Do While <Condition>
   <Statements>
   [Exit Do]
   <Statements>
Loop
```

All of the `Do Loop` forms provide an optional `Exit Do` clause. You can place this clause within a conditional statement to ensure the loop exits should certain conditions arise. Here's an example of the `Do While...Loop` in action.

```
Function DoWhile(EndValue As Integer) As Integer
   ' Create the condition variable.
   Dim CurrentValue As Integer
   CurrentValue = 1

   ' Set result to a specific value.
   DoWhile = 0

   ' Perform the task.
   Do While CurrentValue < EndValue
      ' Add CurrentValue to the result.
      DoWhile = DoWhile + CurrentValue

   ' Update the condition variable.
   CurrentValue = CurrentValue + 1
   Loop
End Function
```

The code begins by creating a variable to hold the current value of the loop. In this case, `CurrentValue` begins at `1`, but you can begin it at any value you wish. The function result is set to `0` to ensure it has an output value. Setting your function to a value immediately upon starting it is always a good way to prevent errors.

The `Do While...Loop` executes while `CurrentValue` is less than `EndValue`. In this respect, the `Do While...Loop` executes until a condition is no longer true. Each time the loop executes, the code updates the output value of the function.

Tip

Notice that the code increments `CurrentValue` during each loop. If you don't modify the condition variable during each loop, the loop can become endless—IDEA will never complete the work you assign to it. Endless loops often require you to force a closure of IDEA and could mean the loss of work and data. Be very careful when using `Do Loops` to update the condition variable as needed.

If you do find yourself in an endless loop, you can try several methods to force IDEA to close. First, you can try stopping the macro from executing by pressing Ctrl+Break. If this technique doesn't work, try clicking the Close button in the upper left corner of IDEA. If you still can't get IDEA to stop, right-click the Windows Task bar and select Task Manager from the context menu. In the Windows Task Manager window, under the Applications tab, highlight the IDEA entry and click End Task. It may take several seconds for Windows to end the task. No matter how you force IDEA to close, you'll very likely lose any unsaved data, so saving your data before you run an application is essential.

Using Do. . .Loop While

The `Do...Loop While` statement checks the condition after the code in the loop executes. This statement ensures that the code executes at least one time, even if the condition isn't true. You can use it in situations where you know you must perform the task no matter what else may be happening, but want to perform the task only one time unless conditions warrant. The `Do...Loop While` statement takes the following form.

```
Do
   <Statements>
   [Exit Do]
   <Statements>
Loop While <Condition>
```

Everything works as it does for the `Do While...Loop` statement, except that you execute the code first. Given the example, the `Do While...Loop` statement doesn't execute the code even once when `EndValue` equals `1` because the loop condition is never true. However, the `Do...Loop While` statement does execute the code, so it outputs a value of `1`, rather than `0`. Here's an example of how to use the `Do...Loop While` statement.

```
Function DoLoopWhile(EndValue As Integer) As Integer
   ' Create the condition variable.
   Dim CurrentValue As Integer
   CurrentValue = 1

   ' Set result to a specific value.
   DoLoopWhile = 0
```

```
   ' Perform the task.
   Do
       ' Add CurrentValue to the result.
       DoLoopWhile = DoLoopWhile + CurrentValue

       ' Update the condition variable.
       CurrentValue = CurrentValue + 1
   Loop While CurrentValue <  EndValue
End Function
```

Using Do Until. . .Loop

Many developers become confused between the `Do While...Loop` statement and the `Do Until...Loop` statement. The difference is that you do something until you've finished it. In other words, you perform the task until some condition becomes true. The `Do Until...Loop` statement takes the following form.

```
Do Until <Condition>
   <Statements>
   [Exit Do]
   <Statements>
Loop
```

As with the `Do While...Loop` statement, you can use the optional `Exit Do` clause within a conditional statement to exit the loop early when conditions require you to do so. The main difference is the condition. The following example looks almost precisely the same as the `Do While...Loop` statement example, except for the condition. In this case, you must create a condition that stops loop execution when it becomes true.

```
Function DoUntil(EndValue As Integer) As Integer
   ' Create the condition variable.
   Dim CurrentValue As Integer
   CurrentValue = 1

   ' Set result to a specific value.
   DoUntil = 0

   ' Perform the task.
   Do Until CurrentValue >=  EndValue
       ' Add CurrentValue to the result.
       DoUntil = DoUntil + CurrentValue

       ' Update the condition variable.
       CurrentValue = CurrentValue + 1
   Loop
End Function
```

Using Do. . .Loop Until

The `Do...Loop While` statement also has a counterpart in the `Do...Loop Until` statement. The `Do...Loop Until` statement takes the following form.

```
Do
   <Statements>
   [Exit Do]
   <Statements>
Loop Until <Condition>
```

The difference between the `Do...Loop While` statement and the `Do...Loop Until` statement is in the condition. Notice that the following code shows that you must create a condition that becomes true when you want to stop the loop from executing.

```
Function DoLoopUntil(EndValue As Integer) As Integer
   ' Create the condition variable.
   Dim CurrentValue As Integer
   CurrentValue = 1

   ' Set result to a specific value.
   DoLoopUntil = 0

   ' Perform the task.
   Do
        ' Add CurrentValue to the result.
        DoLoopUntil = DoLoopUntil + CurrentValue

        ' Update the condition variable.
        CurrentValue = CurrentValue + 1
   Loop Until CurrentValue >= EndValue
End Function
```

Adding Error Trapping to Your Application

No matter how hard you try, your application will have errors at some point. In many cases, the compiler will tell you that the errors exist. For example, if you write a statement incorrectly, the compiler will catch the error before you run the application. Likewise, if there's a type mismatch in your application, IDEA will catch the error and tell you about it when you run the application. However, IDEA can't catch some errors. For example, if you add too much to a variable or run a loop too many times, you'll get the wrong output, but IDEA can't catch that sort of error. The following sections provide some information on error handling in your application. Of course, you'll see other error handling examples as the book progresses. Chapter 21 is especially important because it shows how to locate and fix errors once you discover they exist (part of which is to add more error handling).

Avoiding Errors

The best way to avoid errors is to assume absolutely nothing about the person using your application. After all, you don't know anything about this person—you have no idea of whether the person has received any training or even knows anything about your application except that it performs a specific task. It's also important to assume that the user has no desire to admire your fantastic code (as horrifying as that may sound). The user wants to get whatever task your application is designed to do accomplished as quickly as possible and then go home—possibly to play solitaire on their computer (see, you've already learned something new about the prospective user). With this in mind, here are the top ways to avoid errors in your application:

- **Understand the Problem:** The worst mistakes are made when the person developing the application doesn't fully understand the problem the application is designed to solve. When this happens, users look for workarounds, most of which don't work, but do cause application errors.
- **Create a Helpful Interface:** Your application should always include well-written prompts that provide an example of what the user should type or select. Whenever possible, provide the most common answer as the default entry in a field. Take time to work with the people who will use your application to ensure they actually understand the prompts you provide (rewrite as necessary).
- **Check All Input:** Even the best users make mistakes. A simple typo can cause problems for the best application. Checking every input means that your application can catch errors before they become a problem and ask the user for corrected input.
- **Verify All Resources before Using Them:** As this book progresses, you'll discover methods of verifying the availability of resources such as disk files. Never assume that a resource is available, always assume that someone's deleted it before the application was able to use it.
- **Initialize Everything before You Use It:** Many developers don't initialize variables. As a result, the variable could contain just about any value, which makes it possible for random errors to occur. Random errors are the hardest to find and fix because you can't repeat them (making debugging horribly difficult).

Using On Error

At some point, your application will be ready for use by others—at least, for the most part. The application runs and you've discovered all of the major errors in it. However, your application isn't quite ready yet. You need to add some type of error handling to it, which means adding the `On Error` statement at the beginning of the subprocedure or function where you want to perform error handling. There are three basic forms of `On Error`:

- `On Error Resume Next`
- `On Error GoTo 0`
- `On Error GoTo <Label>`

The least useful of the three forms is `On Error Resume Next`. This form simply ignores the error and tells the application to continue running as shown here:

```
Sub Main
   ' Simply ignore the error.
   On Error Resume Next

   ' Create a variable.
   Dim MyErr As Integer
   MyErr = 0

   ' Define an error condition.
   MyErr = 1 / 0

   ' Display the current value of MyErr.
   MsgBox "MyErr contains: " & MyErr
End Sub
```

In this case, the application simply ignores the divide by zero error and displays the message box. The message box will tell you that `MyErr` still contains `0`. This option isn't very good because no one will know an error has occurred. However, it can come in handy in situations where you know an error could occur, but that the error won't affect the application as a whole.

The second form, `On Error GoTo 0`, simply returns the application to the default IDEA error handling. You use it after you set some other form of error handling, such as `On Error Resume Next`.

The third form of error handling, `On Error GoTo <Label>` is probably the most useful. This form let's you provide custom error handling for your application, even if that error handling consists solely of a custom dialog box. Here's an example of `On Error GoTo <Label>`.

```
Sub Main
   ' Create a variable.
   Dim MyErr As Integer
   MyErr = 0

   ' Define the error handler
   On Error GoTo ErrHandler

   ' Define an error condition.
   MyErr = 1 / 0

   ' Display the current value of MyErr.
   MsgBox "MyErr contains: " & MyErr

   ' Exit the application.
   Exit Sub

   ' Create an error handler label.
   ErrHandler:
```

```
      ' Check the Err object for information.
      MsgBox "Error number: " & Err.Number _
         & " is: " & Err.Description

      ' Resume execution after fixing the error.
      Resume Next
End Sub
```

This example has a few unique additions to it. First, the code calls `On Error GoTo ErrHandler`. `ErrHandler` is a label that's defined later in `Main()`. You define a label by typing the label name followed by a colon (`:`). It pays to outdent labels (make the label appear tight to the left-hand margin of the code window) so that you can see them easily as shown in the example. Immediately before the `ErrHandler` label, you see `Exit Sub`. This clause makes it easy to exit a subroutine whenever you need to, such as to prevent damage or simply because the application has reached the end of the active code (as in this case). Make sure you use an `Exit Sub` clause when you construct your error handler as shown in the example; otherwise, the error handling code will be executed without an associated error. When working with a function, you use `Exit Function` instead of `Exit Sub`.

The error handling section also ends with a `Resume Next` statement. You can use this statement to continue application execution when you feel that error handling efforts are successful. When you run this application, you'll see two message boxes. The first message box will tell you about the error using the custom message and the second message box will contain the current value of `MyErr`.

Raising Your Own Errors

Sometimes your code will encounter an error that IDEA can't detect. Your application might perform some manipulation that makes it hard for IDEA to see the error. For example, the code might detect an error in data that will eventually cause an error. By being proactive about the error, you can usually prevent data damage. In addition, the application has a better chance of recovering from the error. IDEAScript supports the error codes shown in Table 6.1.

You raise errors using the `Err.Raise()` function. All you need to do is supply one of the error numbers in Table 6.1 with the call. Here's an example of raising an error.

```
Sub Main
   ' Define an error handler.
   On Error GoTo ErrHandler

   ' Raise an error.
   Err.Raise 11


   ' Create an error handler label.
   ErrHandler:
```

```
      ' Check the Err object for information.
      MsgBox "Error number: " & Err.Number _
        & " is: " & Err.Description
End Sub
```

TABLE 6.1 Error Codes Supported by IDEASCript

Code	Message
3	Return without GoSub
5	Invalid procedure call
6	Overflow
7	Out of memory
9	Subscript out of range
10	This array is fixed or temporarily locked
11	Division by zero
13	Type mismatch
14	Out of string space
16	Expression too complex
17	Cannot perform requested operation
18	User interrupt occurred
20	Resume without error
28	Out of stack space
35	Sub, Function, or Property not defined
47	Too many DLL application clients
48	Error in loading DLL
49	Bad DLL calling convention
51	Internal error
52	Bad file name or number
53	File not found
54	Bad file mode
55	File already open
57	Device I/O error
58	File already exists
59	Bad record length
61	Disk full
62	Input past end of file
63	Bad record number
67	Too many files
68	Device unavailable
70	Permission denied
71	Disk not ready
74	Cannot rename with different drive
75	Path/File access error
76	Path not found
91	Object variable or With block variable not set
92	For loop not initialized
93	Invalid pattern string
94	Invalid use of Null
97	Cannot call Friend procedure on an object that's not an instance of the defining class

Code	Message
98	A property or method call cannot include a reference to a private object, either as an argument or as a return value
321	Invalid file format
322	Cannot create necessary temporary file
325	Invalid format in resource file
380	Invalid property value
381	Invalid property-array index
382	Property Set cannot be executed at run time
383	Property Set cannot be used with a read-only property
385	Need property-array index
387	Property Set not permitted
393	Property Get cannot be executed at run time
394	Property Get cannot be executed on write-only property
422	Property not found
423	Property or method not found
424	Object required
429	Component cannot create object or return reference to this object
430	Class does not support Automation
432	File name or class name not found during Automation operation
438	Object does not support this property or method
440	Automation error
442	Connection to type library or object library for remote process has been lost
443	Automation object does not have a default value
445	Object does not support this action
446	Object does not support named arguments
447	Object does not support current locale setting
448	Named argument not found
449	Argument not optional or invalid property assignment
450	Wrong number of arguments or invalid property assignment
451	Object not a collection
452	Invalid ordinal
453	Specified not found
454	Code resource not found
455	Code resource lock error
457	This key is already associated with an element of this collection
458	Variable uses a type not supported in Visual Basic
459	This component does not support the set of events
460	Invalid Clipboard format
461	Method or data member not found
462	The remote server machine does not exist or is unavailable
463	Class not registered on local machine
480	ByRef argument mismatch
481	Invalid picture
482	Printer error
735	Cannot save file to TEMP directory
744	Search text not found
746	Replacements too long

In this case, the example also includes a custom error handler. The `Err.Raise()` function can appear anywhere, including inside your error handling code. Consequently, if you find that your error handler can't fix the error, you can use `Err.Raise()` to pass the error back to a caller or simply alert the user to the problem.

Redirecting Macro Flow Using GoTo

Using a `GoTo` statement in your code will redirect macro flow to whatever label you define. When you use this statement, you tell IDEA to go from the current location to the location specified by a tag. In general, you won't use the `GoTo` statement very often, so this section doesn't provide much coverage for it.

Standard Uses for GoTo

The only current standard use for the `GoTo` statement is for error handling. When an error occurs, the `GoTo` statement tells IDEA where to find error handling code. Theoretically, you could also use a `GoTo` within a conditional statement to redirect macro flow in an emergency, but generally, most developers frown on any use of `GoTo` other than for error handling tasks (see the "Using On Error" section for details).

GoTo Uses That Cause Problems

Many developers refuse to use the `GoTo` statement at all. This may sound like an absurd stance, but it does have a basis in fact. At one time, developers used the `GoTo` statement incorrectly. Instead of writing well-structured applications, the developer would use the `GoTo` statement to fix all kinds of code flow problems. The result was something called *spaghetti code*. It was called spaghetti code because finding the beginning or end (or even the middle) was nearly impossible.

Spaghetti code presents a number of problems. First, it's very hard to read. Second, it doesn't execute well once the code gets to a certain size. Third, it's hard to maintain because even the originator often loses track of what the code is doing. This last problem is particularly terrible because companies often had to throw out applications containing spaghetti code, rather than try to update them. Companies lost a lot of time and money this way and now the use of the `GoTo` statement is very much frowned upon.

The only time you should use the `GoTo` statement is when there's absolutely no other choice. For example, error handling requires that you use a `GoTo` statement in many cases—you don't have any other alternative. If you're writing certain kinds of very complex code, you might find need for a `GoTo` statement. However, in most cases, you can easily avoid using the `GoTo` statement and should do so.

Summary

This chapter has shown you the various conditional statements and loops that IDEAScript supports. You also know when to use particular conditional statements or loops to meet

a specific need. Of course, the use of conditional statements and loops is basic for just about any application. Most applications need to make decisions and perform some tasks repetitively. In many cases, the application has to perform tasks repetitively, but only when the conditions are right. In short, conditional statements and loops give your applications a kind of intelligence.

Many people find it hard to figure out conditional statements and loops until they begin to analyze how they perform tasks. It doesn't matter whether you're at home or at work—you need to make thousands of decisions each day and perform some tasks repetitively. It helps to write down some of the tasks you perform and then analyze them as if you were writing code. You don't need to create usable code, simply work through the task as if you were going to write code. Performing this exercise can help you understand conditional statements and loops with greater ease.

Chapter 7 begins a new kind of development study: databases. In Chapter 5 you discovered data types and the use of variables. Variables represent a kind of internal application storage. Databases provide external data storage. You use databases to store information permanently and in such a way that other people can access the data. As with variables, you use conditional statements and loops to process database content. The only difference is the location of data storage.

CHAPTER 7

Understanding IDEA Databases

Databases are an essential part of every business today. Even a business that lacks computers has a database of some sort. Think about it, a database is simply an organized method of storing related data. A Rolodex can be considered a kind of database. Consequently, all of the terrifying things you've heard about database programming being hard to understand are completely untrue—you've been working with databases from an early age and didn't even know it.

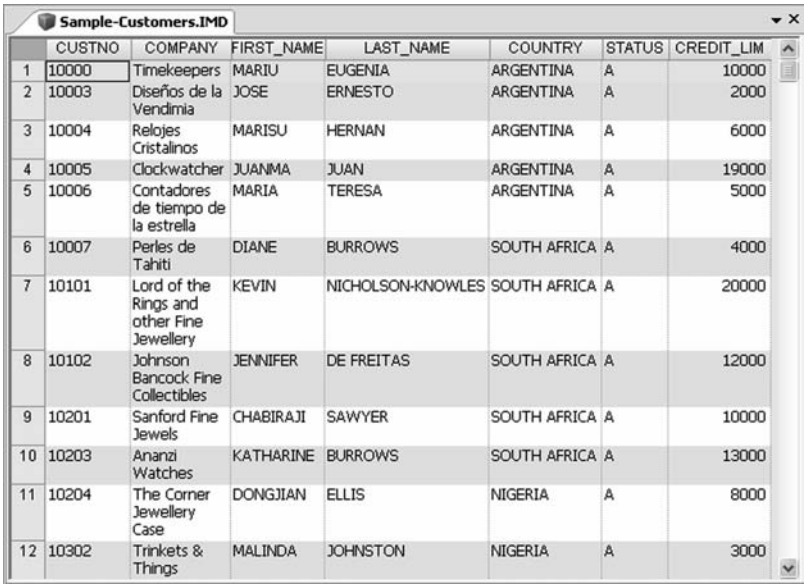
Databases on a computer aren't quite the same as the table you create on paper—they have very specific rules because computers require specific rules. See, the computer isn't nearly as smart as you are, so you need to structure things specifically. This chapter discusses how computer databases are structured at an overview level. You really won't need to know every detail of database design to create good macros in IDEAScript.

Once you understand databases in general, the chapter will review some specifics about IDEA databases. Again, you won't need to know every detail. Chapters 8 and 9 provide additional details about IDEA databases—this chapter is about the basics, so don't worry too much if you don't understand the details just yet.

The rest of the chapter shows you how to perform basic tasks using IDEA databases. This chapter works with the `Sample-Customers` database, but the principles apply to any IDEA database you create.

Considering the Parts of a Database

Computer databases have some very specific parts to them. These parts are necessary because computers can't visualize a table or list the same way you can. When you write entries in an address book, you understand what those entries mean—the computer doesn't understand anything. All it provides is structured data storage for your data. Consequently, you must provide the computer with additional help so it can perform data storage tasks for you. The following sections provide more information on the basic database structure created for any common computer application.



Sample-Customers.IMD

	CUSTNO	COMPANY	FIRST_NAME	LAST_NAME	COUNTRY	STATUS	CREDIT_LIM
1	10000	Timekeepers	MARIU	EUGENIA	ARGENTINA	A	10000
2	10003	Diseños de la Vendimia	JOSE	ERNESTO	ARGENTINA	A	2000
3	10004	Relojes Cristalinos	MARISU	HERNAN	ARGENTINA	A	6000
4	10005	Clockwatcher	JUANMA	JUAN	ARGENTINA	A	19000
5	10006	Contadores de tiempo de la estrella	MARIA	TERESA	ARGENTINA	A	5000
6	10007	Perles de Tahiti	DIANE	BURROWS	SOUTH AFRICA	A	4000
7	10101	Lord of the Rings and other Fine Jewellery	KEVIN	NICHOLSON-KNOWLES	SOUTH AFRICA	A	20000
8	10102	Johnson Bancock Fine Collectibles	JENNIFER	DE FREITAS	SOUTH AFRICA	A	12000
9	10201	Sanford Fine Jewels	CHABIRAJI	SAWYER	SOUTH AFRICA	A	10000
10	10203	Ananzi Watches	KATHARINE	BURROWS	SOUTH AFRICA	A	13000
11	10204	The Corner Jewellery Case	DONGJIAN	ELLIS	NIGERIA	A	8000
12	10302	Trinkets & Things	MALINDA	JOHNSTON	NIGERIA	A	3000

FIGURE 7.1 An IDEA Database Is a Table Consisting of Rows, Columns, Row Numbers, Column Headings, a Name, and Data

Tables

There are many ways to create a database. However, the method used for IDEA is a simple table. Just as you create tables of information on paper, you can also create them inside the computer. In fact, when you look at an IDEA database, such as the `Sample-Customers` database shown in Figure 7.1, you see a table consisting of the following elements:

- Rows
- Columns
- Row Numbers
- Column Headings
- Table Name
- Data

Fields

Fields is the name given to the organizational aid for columns of data. Look again at Figure 7.1. The field names, such as CUSTNO, appear along the top of the table. Under CUSTNO is all of the data associated with the customer number. Each entry is associated with a particular record, but each entry is a CUSTNO field. Using fields tends to enforce the idea of each entry containing just one sort of data. Consequently, when you query the table and ask for Timekeepers CUSTNO, you get the customer number back, and no other information.

Records

Records is the name given to a collection of fields. A single record contains one instance of each field in a table. Not every field will have a value, but every record contains every field. You normally access a database one record at a time. The code you write checks specified fields for the data you want and will retrieve it when necessary. Many of the tasks you perform with IDEA are record-based.

IDEA interacts with records based on the content of one or more fields. When you look at Figure 7.1, you see numbers next to each row. Those numbers are there for your convenience. Unlike the field names across the top of the table, the database doesn't actually store records based on a numeric value (unless there's a field that specifically provides this number). In fact, the database doesn't guarantee that the records will appear in any particular order and you can reorder the records as needed for your application using sorting and indexing (tasks described later in this chapter).

Tip

Never depend on a particular record order. Instead, write the application so that it can retrieve data based on any order. However, you can create indexes to improve application performance and make data easier to access. Careful use of indexes will make working with an IDEA database considerably easier.

Indexes

An *index* is an ordered view of the records in a database—it works much like the card catalogs in the library. When looking for a book in the library, you first locate its entry in the card catalog. The index card from the card catalog will provide a pointer to the book you want. Likewise, an index provides a pointer to the record that contains the data you need.

It's important to remember that an index doesn't change the order of the records in the database—it only provides a pointer to the correct record as shown in Figure 7.2. As you can see, the index is in last name/first name order, the database isn't. In this way, an index works precisely the same as the library's card catalog. An index also doesn't

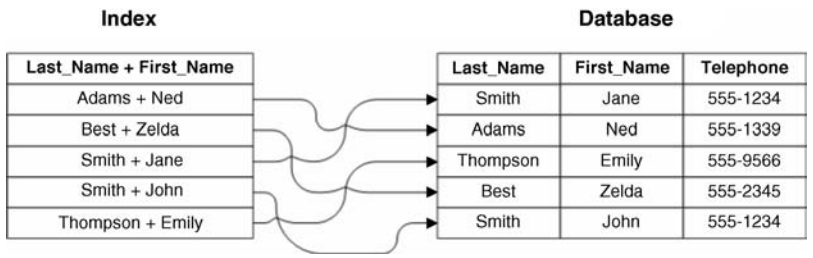


FIGURE 7.2 Indexes Order Information without Changing the Database Content

contain all of the data in the record, no more than an index card found in the card catalog contains all of the content for the book (notice that the telephone number is missing in the figure). Rather, the index orders the data by specific fields to make it easier to find a record based on the field content.

Tip

In most cases, you keep the number of fields used to create an index as small as possible. A large number of fields makes the index perform poorly, increases use of processor and hard drive resources, and reduces your ability to search for data effectively. Small, lean indexes almost always work best for moderately sized data sets. For large data sets (50 MB or larger) that contain highly disorganized data, sorting the data actually produces better results than indexing. This is because an index still requires the database to locate the information on disk, and looking for information can require a significant amount of time.

Introducing the IDEA Database System

The IDEA database system, as with any database system, has specific functionality designed to make it easier for you to work with and to help you maintain database integrity. IDEAScript focuses on data analysis tasks rather than general database management tasks. Consequently, you find that you can't make certain kinds of modifications to an IDEA database that you could make to any general purpose database. There are many reasons for these restrictions, but the most important is that not allowing certain changes makes it less likely that your analysis will be challenged when reviewed in court or in other venues. In short, these seeming restrictions are actually protections put in place for your benefit.

As the chapter progresses, the examples demonstrate some of the tasks you can perform using IDEAScript with an IDEA database. This chapter actually goes through a sort of process by beginning with the first task you normally perform—opening the database—and ending with the last task you normally perform—closing the database. In between, you'll discover some special IDEA features, such as the database history that records actions you take when interacting with the database. You can even create custom history entries to record special events. However, you can't modify existing entries—doing so would damage their value to anyone who wanted to determine what actions you've taken and make it far harder for you to prove your case.

Of course, you have full access to all of the analysis features that IDEA provides. In fact, that's a major purpose for using IDEAScript—to automate analysis that might require days of back breaking manual labor otherwise. In addition, you can create special fields in a database that you can use to store analysis results and calculations you perform. As you progress through the database chapters of the book, you discover just how special the IDEA database is and how it makes your work significantly easier.

LISTING 7.1 Opening a Database

```
Sub Main
   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase("Sample-Customers.imd")

   ' Clear the memory used by db.
   Set db = Nothing
End Sub
```

Opening a Database for Use

You must open a database before you can use it. This requirement isn't any different from opening a word processor document before you work with it or opening a spreadsheet file before you view it. Listing 7.1 shows the technique used to open a database in IDEA.

This example shows the three basic steps in opening a file. First, you create a `Database` object to hold a reference to the file. The `Database` object lets you work with the database in certain ways, such as creating a history entry.

Note

This chapter uses some terms that you probably won't recognize, including object, method, property, and reference. This is one of those times where it's helpful to understand something from a real-world perspective before you learn about the abstract nature of that technology. Chapter 9 describes these four terms (and many others) for you. For now, just think of an object as you would any real-world object, such as an apple. An apple has properties, such as a color, and things you can do to it (a method), such as eating it. Otherwise, the precise meaning of these terms is unimportant for right now.

Second, you use the `Client.OpenDatabase()` method to actually open the database file. You must know the location of the database file on your hard drive. If the file appears in the `IDEA\Samples` folder found in your `My Documents` folder, then all you need to provide is the name of the database, which is `Sample-Customers.imd` in this case.

You must provide the `.imd` file extension with all IDEA databases, even if you don't see it when using Windows Explorer. When you look at the list of databases in the File Explorer window, you see the database name, but not the `.imd` extension. Always add the `.imd` extension to the name you see in the File Explorer window.

When a file doesn't appear in the `IDEA\Samples` folder, you must provide a complete path to the file. The *path* is a special way of telling IDEA where a file is located on your hard drive. It begins with the drive letter, such as `C:`, and includes every folder

starting from the top level folder to the actual location of the database file. For example, if you store your databases in `MyData`, then you must provide a path of `C:\MyData`. The backslash tells IDEA about each level in the folder hierarchy. Consequently, if you want to open `TheData.imd` located in `C:\MyData`, you'd change the string in this example to `C:\MyData\TheData.imd`.

Of course, you might want to share data with another user on a common network drive. In this case, you might have the location mapped as `I:\AuditData`, while the other user has it mapped as `Z:\MyAuditData`. If you hard code the path in your macro, then the other user will need to change every path entry to use your macro, which is time consuming and error prone. IDEAScript provides the `Client.WorkingDirectory` property to make this task easier. Simply set the working directory at the beginning of your macro and IDEA automatically uses that path to locate the files you want to work with. When someone else wants to use your macro, they simply change this one property to make the macro work correctly on their machine. You could even use an input box to ask the user where they have mapped the data. This way, the user wouldn't even have to remember to change the macro.

Third, you always set the `Database` object to `Nothing` when you're done using it. If you don't perform this step, the macro could end without releasing memory it has used to create the `Database` object and your application could create a memory leak. A *memory leak* is something that occurs when an application uses memory without releasing it. Windows tends to lose track of the memory and the memory becomes unavailable until after you reboot your machine. If applications create enough memory leaks, then your machine could literally run out of memory to perform tasks, even if it would normally have enough memory to do so.

Checking the Database History

The database history is important because it records your steps in performing tasks. However, it might not always provide every detail you want. The tasks your macro performs are important and might require custom entries to provide a full explanation of your activities. Listing 7.2 shows how to create a custom entry in the database history.

Working with the database history begins with the `History` object, `ThisHistory`. You obtain a reference to the history of a database using the `History` property of the `Database` object. Notice the use of the `Set` keyword. You must set `ThisHistory` equal to the `History` property in the `Database` object.

The next step is to create a task. Creating the `History` task tells IDEA that you want to do something to the `History` object, so you must create a task to do it using the `NewTask()` method. In this case, you want to tell the world that you opened the database for use. The task could be anything.

The `AppendDatabaseInfo()` method actually adds the history record. The information added includes: file name, description, and number of records. Of course, you may want more information than the default call provides. Calling the `DateStamp()` method adds a date to the entry. Use the `AppendText()` method to create a specific description of the task performed. Figure 7.3 shows how this history entry appears. To see this entry after running the example, open the `Sample-Customers` database and select History in the Properties window.

LISTING 7.2 Modifying the Database History

```
Sub Main
   ' Create the history object.
   Dim ThisHistory As History

   ' Open the database.
   Dim db As Database

   Set db = OpenDB ("Sample-Customers.imd")

   ' Get the history for the current database.
   Set ThisHistory = db.History

   ' Create a new entry.
   ThisHistory.NewTask "Opened Database"

   ' Add the database information.
   ThisHistory.AppendDatabaseInfo

   ' Add a timestamp.
   ThisHistory.DateStamp

   ' Add some additional information.
   ThisHistory.AppendText "Description", "Opened the database for viewing."

   ' Clear the memory.
   Set db = Nothing
   Set ThisHistory = Nothing
End Sub

Function OpenDB(DBPath As String) As Database
   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

You may have noticed the `OpenDB()` function call at the beginning of Listing 7.2. This is actually a separate function within the example. It helps to break applications into smaller pieces whenever possible to make them easier to understand and maintain. The `OpenDB()` function appears in Listing 7.3.

As you can see, the code in this listing mirrors the code used for opening the database in Listing 7.1. Of course, the `OpenDB()` function in your code would be a little more complicated than the code shown in Listing 7.3. You'd want to include error handling

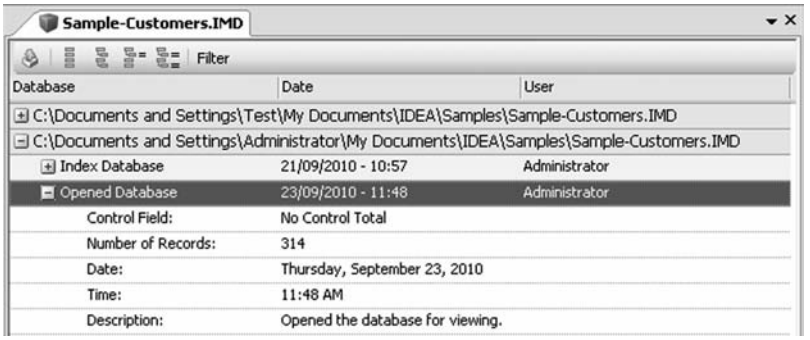


FIGURE 7.3 You Can Create Custom History Entries As Needed

LISTING 7.3 Placing Common Code in a Separate Function

```
Function OpenDB(DBPath As String) As Database
   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

code as a minimum. It would also be helpful to verify that the file actually exists before you try to open it. In addition, you might include some security checks. The list is endless, but the idea is that the `Main()` function isn't encumbered with all this code—looking at `Main()`, all the developer sees is one easily identified call, `OpenDB()`. The `OpenDB()` function appears in the rest of the examples in this chapter.

Obtaining Field Statistics

One of the items many developers want to analyze is field statistics. A field statistic answers the question of how many items are in a field, the average value of those items, and the minimum and maximum value of the items. All of these statistics help the viewer understand the data and how to proceed in an interrogation of the data. What your code is doing is automating part of the auditing process that the user normally has to perform manually. IDEA makes this kind of analysis available through the `FieldStats` object demonstrated in Listing 7.4.

LISTING 7.4 Obtaining Field Statistics

```
Sub Main
   ' Create a handy variable for adding lines to the output.
   Dim vbCrLf As String
   vbCrLf = Chr(13)  + Chr(10)

   ' Create a handy variable for adding tabs to the output.
   Dim vbTab As String
   vbTab = Chr(9)

   ' Open the database.
   Dim db As Database
   Set db = OpenDB("Sample-Customers.imd")

   ' Access the CREDIT_LIM field.
   Dim CredLim As Fieldstats
   Set CredLim = db.FieldStats("CREDIT_LIM")

   ' Obtain statistics about the CREDIT_LIM field.
   MsgBox "The minimum value is: " & _
      vbTab & Format$(CredLim.MinValue, "$###,###.00") & _
      vbCrLf & "The average value is: " & _
      vbTab & Format$(CredLim.AvgValue, "$###,###.00") & _
      vbCrLf & "The maximum value is: " & _
      vbTab & Format$(CredLim.MaxValue, "$###,###.00"), _
      MB_ICONINFORMATION, _
      "The CREDIT_LIM Field has " & CredLim.NumRecords & _
      " Records"
End Sub

Function OpenDB(DBPath As String) As Database
   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

The code begins by opening the database for use. It then uses the `FieldStats` property to obtain the field statistics for the CREDIT_LIM field and place them in `CredLim`. Now you can begin working with any of the available statistics. The example shows the number of records, minimum value, average value, and maximum value of the CREDIT_LIM field, as shown in Figure 7.4.

TABLE 7.1 FieldStats Methods

Method Name	Description
AbsValue	Returns the absolute value for a numeric field.
AverageTime	Gets the average for a time field.
AvgValue	Gets the average value of a numeric field.
Computed	Determines whether the field statistics have been computed.
ComputeStats	Computes the field statistics.
CrValue	Returns the sum of all the negative field values.
DrValue	Returns the sum of all the positive field values.
EarliestDate	Gets the earliest date in a date field.
EarliestTime	Gets the earliest time in a time field.
GetNumber	Gets a count statistic on a numeric field.
GetValue	Gets a statistic on a number type field in number format.
GetValueFmt	Gets a statistic on a numeric type field in a string format.
ItemsInDay	Gets the number of records for each day of the week.
ItemsInMonth	Gets the number of records for the specified month.
Kurtosis	Gets the kurtosis for a numeric field.
LatestDate	Gets the latest date encountered.
LatestTime	Gets the latest time in a time field.
MaxValue	Gets the maximum value for a numeric field.
MinValue	Gets the minimum value for a numeric field.
MostCommonDay	Gets the most common day of the week in a date field.
MostCommonHour	Gets the most common hour for a time field.
MostCommonMinute	Gets the most common minute for a time field.
MostCommonMonth	Gets the most common month in a date field.
MostCommonSecond	Gets the most common second for a time field.
NetValue	Returns the net value for the numeric field.
NumCRRec	Gets the number of records with negative values for this field.
NumDataErrors	Gets the number of invalid date, numeric, and time fields.
NumDRRec	Gets the number of records with positive values for this field.
NumRecords	Gets the number of records in the database.
NumRecsAfter6PM	Gets the number of records in the database that have a time after 6 o'clock at night and before midnight.
NumRecsBefore6AM	Gets the number of records in the database that have a time after midnight and before 6 o'clock in the morning.
NumRecsInAM	Gets the number of records in the database that have a time value in the AM.
NumRecsInPM	Gets the number of records in the database that have a time value in the PM.
NumRecsLessThanADay	Gets the number of records that are less than 24 hours.
NumRecsMoreThanADay	Gets the number of records that are more than 24 hours.
NumZeroItems	Gets the number of records that have zero values.
PopStdDev	Gets the standard deviation for a numeric field.
PopVar	Gets the population variance.
RecNumMax	Gets the record with the maximum value.
RecNumMin	Gets the record with the minimum value.
Reset	Resets field statistics.
SampleStdDev	Gets the standard deviation for a numeric field.
SampleVar	Gets the variance for a numeric field.
Skewness	Gets the skewness for a numeric field.

FIGURE 7.4 Direct Field Access Is One Way to Perform Analysis

The `FieldStats` object is extremely useful. You can perform a wealth of analysis using it. Table 7.1 contains a list of the statistics you can obtain using this object. Simply choose the appropriate method using the same technique as shown in Listing 7.4.

Setting Database Criteria

Database criteria help you filter information in the database by choosing a condition records must meet before IDEA selects them. The basic reason to use database criteria is to reduce clutter. Imagine the time saved if you only have to review 20 records that match specific criteria, rather than 200,000 that don't. Filtering also helps you produce targeted lists and other reports required to conduct an audit. Listing 7.5 shows just one of many uses for database criteria.

In this case, the code uses the `Criteria` property to tell IDEA that you only want to work with records that have a CREDIT_LIM field value equal to or greater than $120,000.00.

After the code applies the criteria, it's possible to begin working with only the matching data. The `ToFirst()` method moves to the beginning of the `RecordSet` object. However, in order to see the first record, you must use the `Next()` method to move to it.

Notice that `Count` is of type Long. Record numbers can become quite large, so you want to be sure that any counter you create will be able to handle the largest number of records. Otherwise, the counter will become overwhelmed and your application will experience an overflow error (a kind of error when you try to fill a small variable with too much data—it overflows).

At this point, the code does something that looks a little odd—it sets up an error handler. When you process records using criteria, IDEA signals you've reached the end of the records by raising an error. As you can see, the next step is to process the information in the `RecordSet` using a `For...Next` loop. The output is `Result`, which contains a list of companies that match the specified criteria. Figure 7.5 shows the output from this example.

Indexing a Database

While the `Criteria` property of a `RecordSet` can filter the data in a database, it doesn't guarantee the data will appear in a particular order. Indexing the database places the information in a particular order and makes it easier to access. Imagine, for a moment,

LISTING 7.5 Performing an Analysis Using Criteria

```
Sub Main
   ' Create a handy variable for adding lines to the output.
   Dim vbCrLf As String
   vbCrLf = Chr(13)  + Chr(10)

   ' Open the database.
   Dim db As Database
   Set db = OpenDB("Sample-Customers.imd")

   ' Create a recordset.
   Dim rs As RecordSet
   Set rs = db.RecordSet

   ' Define a criteria for the recordset.
   rs.Criteria = "CREDIT_LIM >= 120000"

   ' Move to the first record.
   rs.ToFirst
   rs.Next

   ' Set up an error handler.
   On Error GoTo EndOfRecordset

   ' Process all of the records.
   Dim Result As String
   Dim Count As Long

   For Count = 1 To rs.Count
      Result = Result & rs.ActiveRecord.GetCharValue("Company") _
         & vbCrLf
      rs.Next
   Next

   ' Exit if there are no matching records.
   Exit Sub

EndOfRecordset:
   ' Display the results.
   MsgBox Result, MB_ICONINFORMATION, "High Credit Limit Companies"
End Sub

Function OpenDB(DBPath As String) As Database
   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

FIGURE 7.5 Using Criteria, You Can Locate Specific Records in the Recordset

how hard the telephone book would be to use if the names appeared in a random order. The folks at information would suddenly find themselves swamped. Listing 7.6 shows how to create a new index for your database.

It's important to remember that indexing a database doesn't change any of the information in the database. Rather, IDEA builds a separate listing of the data elements you want to index and reorders them. The whole setup works like a card catalog in a library.

In this example, the code begins by opening the database and then creating an `Index` object for it. The code adds a key—reordering on the COMPANY field in ascending order. You can add as many as eight keys as required to obtain the sort order you want. In addition, keys can reorder the data in ascending or descending order as needed.

IDEA doesn't actually change the appearance of the data on screen. All it does is create the index for you. (Indexes continue to exist after you run a macro, unlike criteria which go away once the macro is complete.) It's up to you to apply the index by selecting it in the Properties window. To see the results of this example, choose the COMPANY/A entry in the Indices area of the Properties window. You'll see the order of the entries change to show the companies in alphabetical order.

Sorting a Database

Unlike indexing, which creates an ordered list of references, sorting a database actually changes the order of the records. Of course, since you can't change the content of an IDEA database, you must place the result in a new database.

You use sorting when working with a large number of records and the database is largely unordered—or at least, not ordered in the way you need it ordered. Indexes fall

LISTING 7.6 Creating a Database Index

```
Sub Main
    ' Create a handy variable for adding lines to the output.
   Dim vbCrLf As String
   vbCrLf = Chr(13)  + Chr(10)

   ' Open the database.
   Dim db As Database
   Set db = OpenDB("Sample-Customers.imd")

   ' Define the index object.
   Dim IndexIt As Index
   Set IndexIt = db.Index

   ' Set the index criteria.
   IndexIt.AddKey "Company", "A"

   ' Perform the indexing.
   IndexIt.Index True

   ' Clear the memory.
   Set IndexIt = Nothing
   Set db = Nothing
End Sub

Function OpenDB(DBPath As String) As Database
   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

short at times because the computer must look for each reference individually. Sorting places the records one after another, so that the computer can optimize hard drive access.

Sorting does require more time up front to create the new database with the records sorted in the way you want, but you usually make up the up-front time with better access time during your analysis. Listing 7.7 shows how to sort a database using the same criteria as the example in Listing 7.6. In fact, you'll want to compare these two listings to see how indexing and sorting differ.

The code begins by opening the database. It then creates a `Sort` task. As with indexing, you define one or more sort keys using the `AddKey()` method.

It's at this point that things differ between indexing and sorting. The code calls `PerformTask()` to create the output database. You must supply the name of the new database. The code opens the sorted database at this point so you can see it.

LISTING 7.7 Sorting a Database

```
Sub Main
   ' Open the database.
   Dim db As Database
   Set db = OpenDB("Sample-Customers.imd")

   ' Define the sort object.
   Dim SortIt As Sort
   Set SortIt = db.Sort

   ' Set the sorting criteria.
   SortIt.AddKey "Company", "A"

   ' Perform the sorting.
   SortIt.PerformTask "SortedDatabase.imd"

   ' Open the sorted database.
   OpenDB "SortedDatabase.imd"

   ' Clear the memory.
   Set SortIt = Nothing
   Set db = Nothing
End Sub

Function OpenDB(DBPath As String) As Database
   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

Always remember to set the `Sort` task and `Database` objects to `Nothing`. Otherwise, your code will create a memory leak and you might experience other problems.

Modifying Database Comments

You use the database history to describe actions you've already performed. Database comments often act as a method of reminding you of actions you still need to perform or help you communicate with other people who are working with the same database. Comments appear in the Properties window. A new comment appears in bold type so that you know that it's new, rather than an existing comment. When you open a comment, you see the comment, a link to the affected database, the comment priority, when the comment was made, and who made the comment. Listing 7.8 shows how to create a comment.

LISTING 7.8 Creating a Database Comment

```
Sub Main
   ' Create a handy variable for adding lines to the output.
   Dim vbCrLf As String
   vbCrLf = Chr(13)  + Chr(10)

   ' Open the database.
   Dim db As Database
   Set db = OpenDB("Sample-Customers.imd")

   ' Create a link string.
   Dim LinkString As String
   LinkString = Client.WorkingDirectory + "Sample-Customers.imd"

   ' Define the comment.
   db.AddComment "This is a test comment.", _
      1, _
      Now, _
      "Smith", _
      LinkString, _
      1, _
      LinkString, _
      "", "", ""
End Sub

Function OpenDB(DBPath As String) As Database
   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

As with all of the other examples in this chapter, the code begins by opening the database. Once the database is open, the code creates a `LinkString`, which contains the location of the database. Of course, we've been using the default folder for the examples, so you don't necessarily know where the folder is. Fortunately, you can use the `Client.WorkingDirectory` property to obtain the default location as long as you haven't set this property to another value to work with a database in another location. To create a complete link, you simply add the name of the database you want to use for the link.

At this point, the code creates the comment using the `AddComment()` method. This method takes a number of arguments as follows:

- **Comment:** The comment you want to appear in the Comments window and the Properties window.
- **Importance:** The priority of this comment. The valid values include: `0` – low, `1` – medium, and `2` – high.
- **Date/Time of Comment:** The time the comment was created. Using the `Now()` function ensures that the timestamp includes both date and time.
- **User Name:** The name of the person making the comment.
- **Link String:** The location of the database you want to use for the comment link.
- **IDEAScript Command:** The command you want to execute when the user clicks the link. You have a choice of three numbers: `0` – none, `1` – open database file, and `2` – open other file.
- **File Path:** Tells IDEAScript where to locate the database or file resource.
- **Argument 2:** Saved for future use. Always set this argument to an empty string, `""`.
- **Argument 3:** Saved for future use. Always set this argument to an empty string, `""`.
- **Argument 4:** Saved for future use. Always set this argument to an empty string, `""`.

The comment becomes instantly available. You can see the output of this example in Figure 7.6.

Committing the Database

When you make a number of changes to a database, those changes may not appear immediately. In some cases, those changes remain in memory until you commit them. Committing the database tells IDEA that the changes you've made are complete and you don't plan to change them during this session. Listing 7.9 shows a technique for adding a comment to a database field and then committing that change.

The code begins by opening the `10K Customers 2` database you created earlier in this book. It then obtains a definition of the table—essentially, all of the characteristics used to create the table. You can perform a number of tasks using a `TableDef` object and you'll see many of these tasks as the book progresses. However, for this example, the code adds a description to the COMPANY field. You can see this description by selecting Data > Field Manipulation to display the Field Manipulation window shown in Figure 7.7.

After the code creates the description, it calls the `CommitDatabase()` method to make the change permanent. At this point, the `db` object is no longer valid—you can't

FIGURE 7.6 Add Comments to the Database to Provide Reminders or Notes to Others

LISTING 7.9 Committing a Database Change

```
Sub Main
   ' Open the database.
   Dim db As Database
   Set db = OpenDB("10K Customers 2.imd")

   ' Obtain the table definition.
   Dim ThisTable As TableDef
   Set ThisTable = db.TableDef

   ' Obtain the Company name field.
   Dim CNField As Field
   Set CNField = ThisTable.GetField("COMPANY")

   ' Add a comment to the Company name field.
   CNField.Description = "This is the company name field."

   ' Commit the database change.
   db.CommitDatabase

   ' Clean up memory.
   Set CNField = Nothing
   Set ThisTable = Nothing
   Set db = Nothing
End Sub

Function OpenDB(DBPath As String) As Database
   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

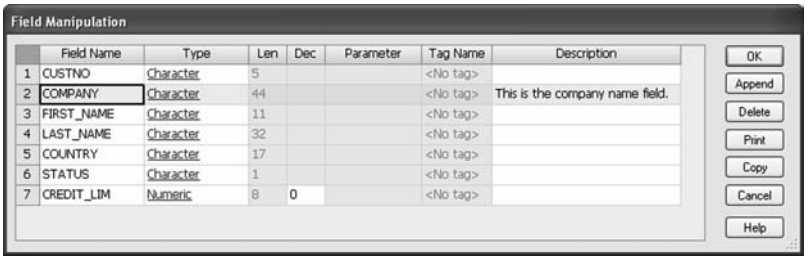


FIGURE 7.7 Commit a Database After You Make Changes to Its Content

LISTING 7.10 Closing a Database

```
Sub Main
   ' Open the database.
   Dim db As Database
   Set db = OpenDB("Sample-Customers.imd")

   ' Close the database.
   db.Close

   ' Free memory.
   Set db = Nothing
End Sub

Function OpenDB(DBPath As String) As Database
   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

use it to perform other tasks. As usual, make sure you clean up memory before the macro completes.

Closing a Database

Closing the database is the last act you perform with it before you go on to something new. Always close the database to ensure that every change you made actually gets onto the hard drive and that there aren't any data errors to cause problems. When you close the database, you also free up room in IDEA for other tasks. Listing 7.10 shows the code used to close a database.

As you can see, closing a database is as easy as calling the `Close()` method. Make sure you free the memory used by the `Database` object after you close the database.

Summary

This chapter has helped you discover database basics as they apply to computers. You already know a lot about databases because you create them every day. From the shopping list you write before you go to the store to the address book you use to call friends, they all qualify as kinds of databases. The main purpose of this chapter is to help you understand the structure that computers require to work with databases.

Now that you understand some of the basics, you're going to want to work with them a bit. Try the tasks that you performed in this chapter on other sample IDEA databases. For example, try making a query on the `Sample-Employees` database. It's important to work with a number of databases because many of the principles aren't easy to understand if you don't practice. Don't worry about damaging the samples; they don't contain any real data and you won't ruin anyone's day by working with them.

Chapter 8 takes what you learned in this chapter and goes to the next step. You'll begin performing more complicated tasks using databases. This chapter still doesn't fill in all of the details—that's the purpose of Chapter 9. Chapter 8 is there to help you understand some of the more complicated tasks you can perform with databases—things you might not do every day.

CHAPTER 8

Working with Databases

This chapter begins looking at databases in detail. Chapter 7 showed you some of the basics, and this chapter takes the next step. In this chapter, you begin working with multiple databases. Two common tasks are adding two databases together so that you can analyze the result as a single database and comparing the content of two databases. This second form is a kind of analysis that helps you locate anomalies—the kind of information that an audit commonly requires as a starting point. Database comparison can be time consuming, so it's an especially appropriate task for IDEAScript.

The next step is to start looking for errors in databases. In this case, you begin working with *keys*—the fields of a database that define the uniqueness of each record. Again, this kind of analysis points out anomalies that you need to consider during an audit. Even more important, these checks can point out database errors that could hinder other kinds of analysis that you need to perform later.

This chapter also introduces you to exporting the database. You can export databases for a number of reasons. For example, you might want to share the details of an analysis with a colleague. In some cases, you simply need the data in another format to create reports and perform additional analysis. Whatever the reason for exporting the data, IDEAScript supports a number of convenient export formats you can use.

Finally, you'll find more details on using fields, records, and tables in this chapter. The examples in Chapter 7 cover just the basics—the examples in this chapter are more detailed and you'll discover the reasons for working with fields, records, and tables in a particular way.

Adding One Database to Another Using AppendDatabase

IDEA makes it possible to create a single database out of two or more existing databases. There are many situations where you might have to perform this task. For example, you might want to combine the sales from several years into a single database so that you can perform analysis on that single database, rather than perform the task year-by-year. In addition, combining databases often lets you audit data over a significant time frame and expose potential oddities in that way.

Another use for combining databases is to create specialized subsets of information. For example, looking again at the `Sample-Customers` database, you might not want

to review the entire database. Perhaps you want all of the customers with credit limits greater than or equal to $10,000 and those with credit limits less than or equal to $4,000. In order to create a single database with these two ranges, you'd need to extract the two ranges into two databases and then combine them into a whole.

You already have the $10,000 database in `10K Customers 2.imd` that you created earlier in this book. Your first task is to modify that macro so that you can extract the less than $4,000 records to the `4K Customers.imd` database. As a hint, you need to change these two lines of code in the `IndexedExtraction()` function:

```
task.FieldValueIs2
 WI_IE_LTEQUAL, 4000.00, WI_IE_NUMFLD
dbName = "4K Customers.imd"
```

Now that you have two databases to use, it's time to look at how you combine them into a single database. Listing 8.1 shows how to use the `AppendDatabase` task.

The code begins by opening the `10K Customers 2` database. It then creates an `AppendDatabase` task for the open database.

At this point, you might think that you need to open the `4K Customers` database you just created as well, but you don't. Instead, the code uses the `AddDatabase()` method to tell IDEA which database to use as the second database.

The output phase comes next. The code defines a name for the output database and then calls `PerformTask` with the name of the new database. Notice that the second

LISTING 8.1 Appending Two Databases Together

```
Sub Main
   ' Open the first database.
   Dim db1 As Database
   Set db1 = OpenDB("10K Customers 2.imd")

   ' Define the Append Database task.
   Dim AppendDB As Task
   Set AppendDB = db1.AppendDatabase

   ' Add the second database.
   AppendDB.AddDatabase "4K Customers.imd"

   ' Define a name for the result database.
   Dim Result As String
   Result = "10K And 4K Customers.imd"

   ' Perform the task.
   AppendDB.PerformTask Result, ""

   ' Clean up memory.
   Set AppendDB = Nothing
   Set db1 = Nothing

   ' Close the database.
   Client.CloseDatabase "10K Customers 2.imd"
End Sub
```

```
Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

argument is an empty string—it isn't used any longer. The example ends by cleaning up memory and closing the `10K Customers 2` database.

You'll want to check the results from here to make sure the example actually works. As shown in Figure 8.1, you can open the File Explorer in IDEA to see the results. The

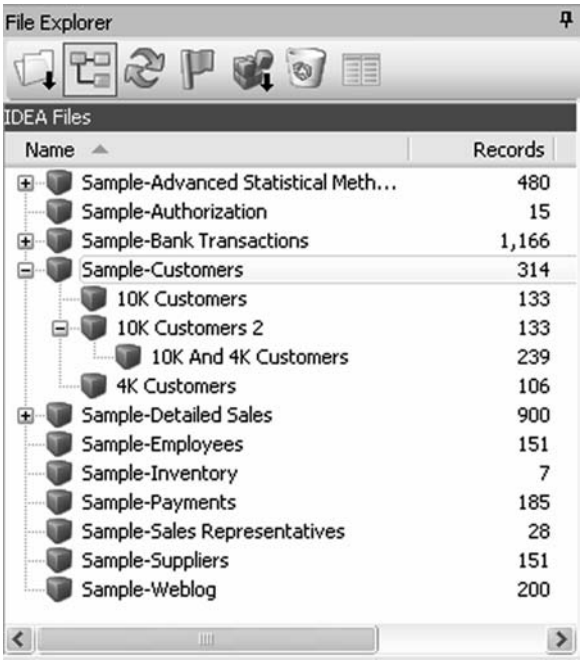


FIGURE 8.1 The File Explorer Shows That the 10K and 4K Customers Database Contains the Records of Both Databases

`10K Customers 2` database has 133 records and the `4K Customers` database has 106 records for a total of 239 records.

Comparing Two Databases Using CompareDB

What would happen if you received two databases that are supposedly the same, but aren't? It happens all the time. In many cases, it's up to the data detective to figure out which database is wrong. Sometimes, both databases are wrong, but in different ways. The `CompareDB` task makes it considerably easier to compare two databases. In this example, the `CompareDB` task examines the `Sample-Customers` database against the `10K Customers 2` database, as shown in Listing 8.2.

The code begins by opening the primary database. You normally select the database that you feel is most accurate for the primary database to make the output report easier to read and understand. However, there really isn't a wrong or right choice.

The next step is to define a key for matching the two databases. A key in this case is the means of comparing the two databases. The unique value in one database is compared to the unique value in the second database. The choice of key is extremely important and you may actually find that you need to perform several comparisons using different keys to locate precisely what you want. Figure 8.2 shows the output of the example using the CUSTNO field as the key. Figure 8.3 shows the change that occurs when you use the CREDIT_LIM field as the key.

In the first case, because CUSTNO is unique, one record in the primary database matches precisely one record in the secondary database. Using this view, you can locate which records are missing from the secondary database. However, unless you're very

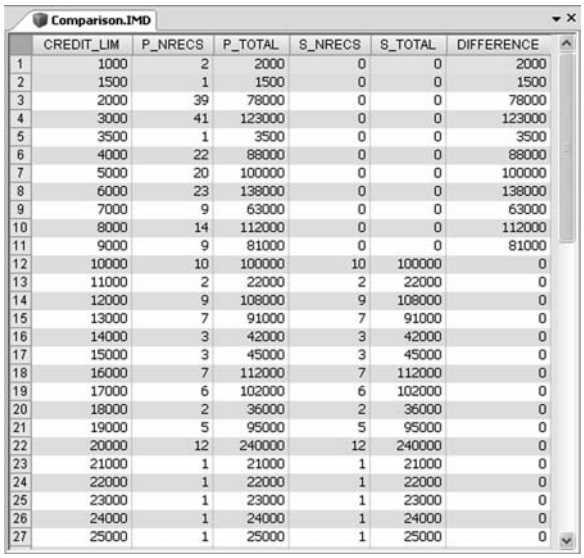
Comparison.IMD

	CUSTNO	P_NRECS	P_TOTAL	S_NRECS	S_TOTAL	DIFFERENCE
1	10000	1	10000	1	10000	0
2	10003	1	2000	0	0	2000
3	10004	1	6000	0	0	6000
4	10005	1	19000	1	19000	0
5	10006	1	5000	0	0	5000
6	10007	1	4000	0	0	4000
7	10101	1	20000	1	20000	0
8	10102	1	12000	1	12000	0
9	10201	1	10000	1	10000	0
10	10203	1	13000	1	13000	0
11	10204	1	8000	0	0	8000
12	10302	1	3000	0	0	3000
13	10400	1	7000	0	0	7000
14	10500	1	5000	0	0	5000
15	10801	1	19000	1	19000	0
16	10900	1	3000	0	0	3000
17	11100	1	23000	1	23000	0
18	11207	1	2000	0	0	2000
19	11300	1	15000	1	15000	0
20	11301	1	2000	0	0	2000
21	11400	1	6000	0	0	6000
22	11600	1	20000	1	20000	0
23	11702	1	9000	0	0	9000
24	11704	1	7000	0	0	7000
25	11805	1	68000	1	68000	0
26	11806	1	30000	1	30000	0
27	11809	1	4000	0	0	4000

FIGURE 8.2 Performing a Comparison Can Tell You a Lot About Two Databases

LISTING 8.2 Comparing Two Databases

```
Sub Main
   ' Open the first database.
   Dim db1 As Database
   Set db1 = OpenDB("Sample-Customers.imd")

   ' Create the task.
   Dim DoCompare As Task
   Set DoCompare = db1.CompareDB

   ' Specify the key used to compare the two databases.
   DoCompare.AddMatchKey "CUSTNO", "CUSTNO", "A"

   ' Perform the comparison.
   DoCompare.PerformTask _
      "Comparison.imd", _
      "", _
      "CREDIT_LIM", _
      "CREDIT_LIM", _
      "10K Customers 2.imd"

   ' Clear memory.
   Set DoCompare = Nothing
   Set db1 = Nothing

   ' Close the first database and open the comparison.
   Client.CloseDatabase "Sample-Customers.imd"
   Client.OpenDatabase "Comparison.imd"
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```



Comparison.IMD

	CREDIT_LIM	P_NRECS	P_TOTAL	S_NRECS	S_TOTAL	DIFFERENCE
1	1000	2	2000	0	0	2000
2	1500	1	1500	0	0	1500
3	2000	39	78000	0	0	78000
4	3000	41	123000	0	0	123000
5	3500	1	3500	0	0	3500
6	4000	22	88000	0	0	88000
7	5000	20	100000	0	0	100000
8	6000	23	138000	0	0	138000
9	7000	9	63000	0	0	63000
10	8000	14	112000	0	0	112000
11	9000	9	81000	0	0	81000
12	10000	10	100000	10	100000	0
13	11000	2	22000	2	22000	0
14	12000	9	108000	9	108000	0
15	13000	7	91000	7	91000	0
16	14000	3	42000	3	42000	0
17	15000	3	45000	3	45000	0
18	16000	7	112000	7	112000	0
19	17000	6	102000	6	102000	0
20	18000	2	36000	2	36000	0
21	19000	5	95000	5	95000	0
22	20000	12	240000	12	240000	0
23	21000	1	21000	1	21000	0
24	22000	1	22000	1	22000	0
25	23000	1	23000	1	23000	0
26	24000	1	24000	1	24000	0
27	25000	1	25000	1	25000	0

FIGURE 8.3 Using a Different Key Provides a Completely Different Output

careful in performing your comparison, you might not notice the pattern shown in Figure 8.3. By looking at this view, you can see that the secondary database is missing all of the records with CREDIT_LIM field values less than $10,000.

After the code sets the key, it calls `PerformTask()`. This method accepts a number of values as input as listed here:

- **Output Database Name:** The name of the database that should receive the comparison report.
- **Not Used:** Set this argument to `""`.
- **Primary Comparison Field:** The name of the field to use in the primary database for comparison purposes. This field must be numeric.
- **Secondary Comparison Field:** The name of the field to use in the secondary database for comparison purposes. This field must be numeric.
- **Secondary Database Name:** The name of the secondary database, the one you want to compare to the primary database.

The example ends by cleaning up memory as usual. It also closes the primary database and opens the comparison database. Opening and closing databases might not seem like a very big deal, but doing so can save time and reduce errors. Otherwise, you must rely on the user to open the correct database.

Working with Keys

When you work with a table, you must have some way to uniquely identify each record. The combination of fields that you choose for identification purposes is called a *key*.

In fact, because this particular kind of key is used to uniquely identify records in the current table, it's often called a *primary key*. An IDEA database is what is termed a *flat file*—that is, every table is standalone, so you only have to deal with keys for that one table. Consequently, throughout the book you'll hear the term key used to express the set of fields used to uniquely identify the records in the table that you're using for analysis.

Locating Duplicate Keys with DupKeyDetection

Sometimes, the database you work with is supposed to have a unique key, but someone adds a duplicate. For example, if you were to look at the `Sample-Customers` database, the CUSTNO field should be unique because each customer should have a different number. However, someone could add a second customer with the same CUSTNO value as another customer. When this problem occurs, you could find that your analysis doesn't work as expected. Consequently, one of the tests you perform for database integrity is checking for duplicate key values, as shown in Listing 8.3.

The code begins by opening the database. It then creates the `DupKeyDetection` task. The next step is to tell IDEA which fields to include in the output. In this case, the code only includes one field, CUSTNO. However, you can add as many fields as you wish.

After you define the output fields, you choose the fields that act as a key. In most cases, you choose fields that form a unique value within the database. For the `Sample-Customers` database, this means using the CUSTNO field.

When looking for duplicate keys you have two choices. You can either look for all of the records that aren't duplicated or you can look for the duplicates. The first method is useful when you want to output a clean database for analysis.

The second method is useful when you want to find all the duplicated keys and ask for an update that doesn't include them. When you set `OutputDuplicates` to `True`, you get just the records with duplicated keys (the duplicate being any record after the first record that contains the value in question).

Note

The trouble with using the Records without Duplicates option is that it produces a database containing the key values that only appear once. IDEA omits any value that occurs twice or more from the result. If you need to create a "clean" database with a complete set of "Key" values where each one only occurs once then use the `Summarization` task instead.

At this point, the code performs the task. The code then closes `Sample-Customers` and displays the Duplicates table. As expected, the database was created and you won't see any duplicates, which is actually a bit disappointing because you want to see some output. So, let's test a condition that will most definitely result in some output,

CREDIT_LIM. Change the fields to look like this:

```
' Add the key.
CheckKeys.AddKey "CUSTNO", "A"
CheckKeys.AddFieldToInc "COMPANY"
CheckKeys.AddFieldToInc "CREDIT_LIM"
```

After that, change the key value to look like this:

```
' Add the key.
CheckKeys.AddKey "CREDIT_LIM", "A"
```

LISTING 8.3 Locate Duplicate Keys

```
Sub Main
   ' Open the sample database.
   Dim db As Database
   Set db = OpenDB("Sample-Customers.imd")

   ' Create the task.
   Dim CheckKeys As Task
   Set CheckKeys = db.DupKeyDetection

   ' Tell IDEA to look at the CUSTNO field.
   CheckKeys.AddFieldToInc "CUSTNO"

   ' Add the key.
   CheckKeys.AddKey "CUSTNO", "A"

   ' Output only the duplicate values.
   CheckKeys.OutputDuplicates = True

   ' Perform the task.
   CheckKeys.PerformTask "Duplicates.imd", ""

   ' Clear memory.
   Set CheckKeys = Nothing
   Set db = Nothing

   ' Close the sample database and open the duplicates
   ' report.
   Client.CloseDatabase "Sample-Customers.imd"
   Client.OpenDatabase "Duplicates.imd"
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )
```

```
   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

When you run the example again, you see an overwrite prompt because the database already exists. Click **Yes**, then you see quite a bit of output because your key contains a number of duplicates. Figure 8.4 shows the results of the updated example. Now, if you look at the File Explorer, you see that there are 280 records in this new `Duplicates` database. (Make sure you click the Refresh List button on the File Explorer toolbar so you see the change.) Now, try changing `OutputDuplicates` to `False` . You'll see that you get a total of 34 records. When you add these two numbers together, you get 314, the number of records in the original database.

Duplicates.IMD

	CUSTNO	COMPANY	CREDIT_LIM
1	21105	Fine Jewellery Inc.	1000
2	21206	Wholesale Watches and Rings	1000
3	10003	Diseños de la Vendimia	2000
4	11207	Barbados Jewellery Company	2000
5	11301	Jewellery Now	2000
6	12203	Kara Jewels	2000
7	20008	Emitations	2000
8	20039	Hong Kong Fine Jewellery	2000
9	20414	Adrianna's Fine Jewels	2000
10	20756	Yuli's	2000
11	20764	Chinese Designers of Fine Jewellery	2000
12	20823	Gold Jewellery & Watches Inc.	2000
13	21092	Turquoise and Jewels	2000
14	21139	Lëtzebuergesch	2000
15	21274	Michaela's Fine Pendants	2000
16	21426	Lu-Pang's	2000
17	21646	Relojes Costa Rica	2000
18	30228	Joyería y cosas	2000
19	30704	Homeland Jewellery	2000
20	40310	Jewelery On Time	2000
21	40317	Fancy Cuts	2000
22	40401	The Look	2000
23	40605	Watches For All	2000
24	40617	Impressions	2000
25	40708	Fadi's	2000
26	40723	Aztlán Watches & Accessories	2000
27	40900	Rings By Joyce	2000

FIGURE 8.4 IDEA Can Find Any Duplicate Keys in a Database

Locating Exclusive Records Based on Key Using DupKeyExclusion

Sometimes you need to locate records that differ in just one respect. Records that are almost but not quite the same have an exclusive field, making them exclusive records. You use exclusive searches when you have a group of records that have similarities and you're most interested in how they differ. Often, such searches help you define a pattern based on differences, rather than similarities and a search of this kind can make the odd record stick out. Listing 8.4 shows how to perform an exclusive record search.

LISTING 8.4 Locate Exclusive Records

```
Sub Main
   ' Open the sample database.
   Dim db As Database
   Set db = OpenDB("Sample-Customers.imd")

   ' Create the task.
   Dim DupKeyEx As Task
   Set DupKeyEx = db.DupKeyExclusion

   ' Define the fields you want to see in the output.
   DupKeyEx.AddFieldToInc "CUSTNO"
   DupKeyEx.AddFieldToInc "COMPANY"
   DupKeyEx.AddFieldToInc "COUNTRY"
   DupKeyEx.AddFieldToInc "CREDIT_LIM"

   ' Define the key to use for the task.
   DupKeyEx.AddKey "CREDIT_LIM", "A"

   ' Define the difference field.
   DupKeyEx.DifferentField = "CUSTNO"

   ' Specify a criteria for selecting records.
   DupKeyEx.Criteria = "CREDIT_LIM = 2000"

   ' Perform the task.
   DupKeyEx.PerformTask "Exclusive.imd", ""

   ' Clear memory.
   Set DupKeyEx = Nothing
   Set db = Nothing

   ' Close the sample database and open the duplicates
   ' report.
   Client.CloseDatabase "Sample-Customers.imd"
   Client.OpenDatabase "Exclusive.imd"
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
```

```
   PathCheck = Dir( DBPath )

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

This example begins by opening the database and defining a task. After the code performs these basics, it defines a list of fields to display in the output database using the `AddFieldToInc()` method. If you want to include all of the fields, use the `IncludeAllFields()` method instead. You must define every field you want to appear in the output using either of these methods.

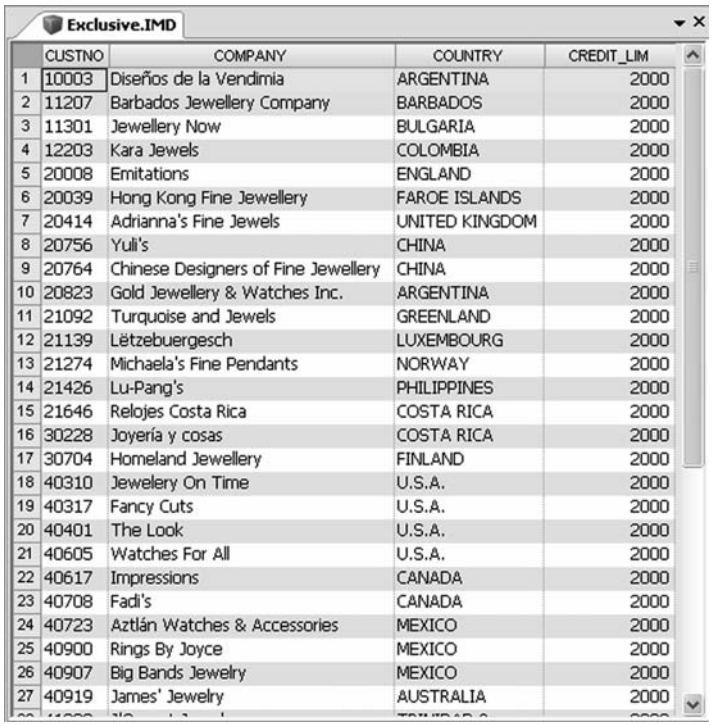
The next step is to add a key. In this case, the example uses CREDIT_LIM as the key, which means that IDEA accesses the database in CREDIT_LIM order before it performs the processing. The order you use is important because it determines what qualifies as a unique record. For example, try changing CREDIT_LIM to COUNTRY and you'll notice that the output record count changes from 39 to 25. That's because there are 14 records where there's just one country that has a record that matches the criteria, so that record is unique when sorted by COUNTRY. For example, there's only one record from Barbados, so it doesn't appear in the output when you set the key to COUNTRY, but it does when you set the key to CREDIT_LIM. You can always index the database as needed after you create the output, so worry more about how the key affects the output, than how it causes IDEA to present the data.

The next step is to determine which field to use to detect a difference. Remember, you're looking for just one difference. In this case, the code uses the `DifferentField()` method to set the value to CUSTNO, which is guaranteed to be unique for every record in the database.

You may not want every record in a database when performing some types of analysis. Most tasks include a `Criteria` property that lets you limit the output. In this case, the code sets the criteria to match only those records where CREDIT_LIM = 2000. Once the code sets all of the required arguments, it calls `PerformTask()` and places the output in `Exclusive.imd`. As usual, the code closes the source database and automatically opens the result database to save time and reduce errors. Figure 8.5 shows the output from this example.

Exporting a Database Using ExportDatabase

Being able to share data with others is an essential part of working with databases. Fortunately, IDEA makes it very easy to export data to other formats. The example shown in Listing 8.5 illustrates exporting data to another format, such as Microsoft Access or Microsoft Excel.



Exclusive.IMD

	CUSTNO	COMPANY	COUNTRY	CREDIT_LIM
1	10003	Diseños de la Vendimia	ARGENTINA	2000
2	11207	Barbados Jewellery Company	BARBADOS	2000
3	11301	Jewellery Now	BULGARIA	2000
4	12203	Kara Jewels	COLOMBIA	2000
5	20008	Emitations	ENGLAND	2000
6	20039	Hong Kong Fine Jewellery	FAROE ISLANDS	2000
7	20414	Adrianna's Fine Jewels	UNITED KINGDOM	2000
8	20756	Yuli's	CHINA	2000
9	20764	Chinese Designers of Fine Jewellery	CHINA	2000
10	20823	Gold Jewellery & Watches Inc.	ARGENTINA	2000
11	21092	Turquoise and Jewels	GREENLAND	2000
12	21139	Lëtzebuergesch	LUXEMBOURG	2000
13	21274	Michaela's Fine Pendants	NORWAY	2000
14	21426	Lu-Pang's	PHILIPPINES	2000
15	21646	Relojes Costa Rica	COSTA RICA	2000
16	30228	Joyería y cosas	COSTA RICA	2000
17	30704	Homeland Jewellery	FINLAND	2000
18	40310	Jewelery On Time	U.S.A.	2000
19	40317	Fancy Cuts	U.S.A.	2000
20	40401	The Look	U.S.A.	2000
21	40605	Watches For All	U.S.A.	2000
22	40617	Impressions	CANADA	2000
23	40708	Fadi's	CANADA	2000
24	40723	Aztlán Watches & Accessories	MEXICO	2000
25	40900	Rings By Joyce	MEXICO	2000
26	40907	Big Bands Jewelry	MEXICO	2000
27	40919	James' Jewelry	AUSTRALIA	2000

FIGURE 8.5 Look for Records That Vary by Just one Value to Find Data Patterns

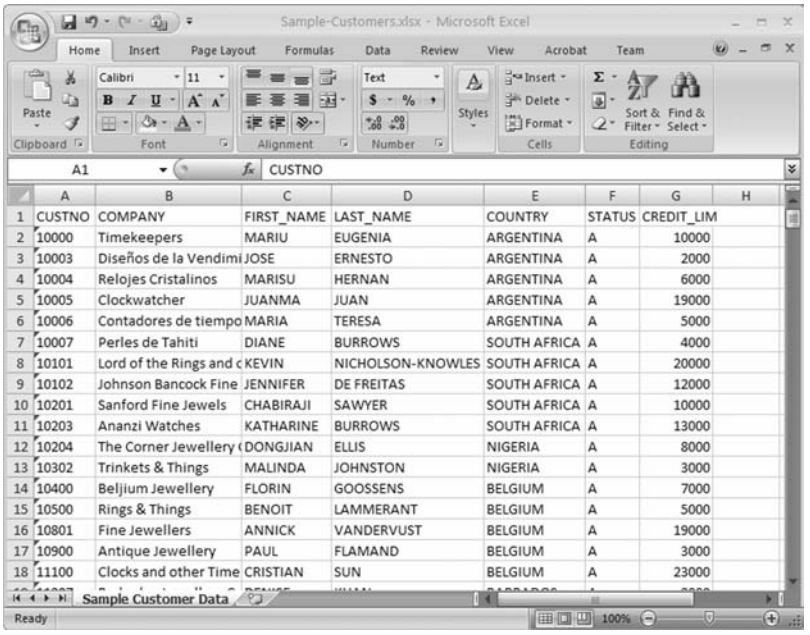
LISTING 8.5 Export a Database to a New Format

```
Sub Main
   ' Open the sample database.
   Dim db As Database
   Set db = OpenDB("Sample-Customers.imd")

   ' Export the database.
   ExportDB db, "Sample Customer Data", _
      "Sample-Customers.XLSX", "XLSX"

   ' Clear the memory.
   Set db = Nothing

   ' Close the database.
   Client.CloseDatabase "Sample-Customers.imd"

   ' Open the resulting Excel spreadsheet.
   Shell "cmd.exe /c " + Chr(34) + Client.WorkingDirectory + _
      "Sample-Customers.XLSX" + Chr(34)

   ' This is an alternative technique.
   ' Shell "C:\Program Files\Microsoft Office\OFFICE11\Excel.EXE" + _
```

```
   '   Chr(34) + Client.WorkingDirectory + _
   '   "Sample-Customers.XLSX" + Chr(34)
End Sub

Sub ExportDB(db As Database, TableName As String, _
   ExportPath As String, ExportType As String)

   ' Create the task.
   Dim ExportData As Task
   Set ExportData = db.ExportDatabase

   ' Output all of the fields.
   ExportData.IncludeAllFields

   ' Perform the task.
   ExportData.PerformTask ExportPath, TableName, _
      ExportType, 1, db.Count, ""

   ' Clear the memory.
   Set ExportData = Nothing
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

The code begins by opening the database. It then calls a subroutine named `ExportDB()`, which is discussed later in this section. After the conversion takes place, the code clears memory and closes the database used for export purposes.

At this point, you probably want to see the converted file. So, you have a choice of finding the file somewhere on your hard drive, double-clicking it in Windows Explorer, and watching Windows open it for you, or you can automate the task and let Excel open the file automatically. The `Shell()` function is the secret in this case. You can use the `Shell()` function to open a copy of the command processor `CMD.EXE` (don't worry if you've never heard about it—this is a cool, but hidden, program). The `/c` command line switch tells the command processor to execute a command and then close automatically after you close Excel. The command executed in this case is the full path to the file you



	A	B	C	D	E	F	G	H
1	CUSTNO	COMPANY	FIRST_NAME	LAST_NAME	COUNTRY	STATUS	CREDIT_LIM	
2	10000	Timekeepers	MARIU	EUGENIA	ARGENTINA	A	10000	
3	10003	Diseños de la Vendimi	JOSE	ERNESTO	ARGENTINA	A	2000	
4	10004	Relojes Cristalinos	MARISU	HERNAN	ARGENTINA	A	6000	
5	10005	Clockwatcher	JUANMA	JUAN	ARGENTINA	A	19000	
6	10006	Contadores de tiempo	MARIA	TERESA	ARGENTINA	A	5000	
7	10007	Perles de Tahiti	DIANE	BURROWS	SOUTH AFRICA	A	4000	
8	10101	Lord of the Rings and c	KEVIN	NICHOLSON-KNOWLES	SOUTH AFRICA	A	20000	
9	10102	Johnson Bancock Fine	JENNIFER	DE FREITAS	SOUTH AFRICA	A	12000	
10	10201	Sanford Fine Jewels	CHABIRAJI	SAWYER	SOUTH AFRICA	A	10000	
11	10203	Ananzi Watches	KATHARINE	BURROWS	SOUTH AFRICA	A	13000	
12	10204	The Corner Jewellery (	DONGJIAN	ELLIS	NIGERIA	A	8000	
13	10302	Trinkets & Things	MALINDA	JOHNSTON	NIGERIA	A	3000	
14	10400	Beljium Jewellery	FLORIN	GOOSSENS	BELGIUM	A	7000	
15	10500	Rings & Things	BENOIT	LAMMERANT	BELGIUM	A	5000	
16	10801	Fine Jewellers	ANNICK	VANDERVUST	BELGIUM	A	19000	
17	10900	Antique Jewellery	PAUL	FLAMAND	BELGIUM	A	3000	
18	11100	Clocks and other Time	CRISTIAN	SUN	BELGIUM	A	23000	

FIGURE 8.6 The Example Automatically Opens the Excel Spreadsheet for You

just created. Because of the way Windows is set up, you can use this indirect method of telling Windows which file to open. Windows will look up the correct application for opening the file in the Windows Registry and then open it for you as shown in Figure 8.6. Amazing! This technique works for just about any file on your machine, so whatever conversion format you use, you can usually open the resulting file automatically using the `Shell()` function and the full path to the file.

Notice the `Chr(34)` at both ends of the string that tells where the file is. It turns out that the command processor doesn't understand strings with spaces—you have to place them in double quotes, which is what `Chr(34)` does.

There's an alternative method for opening Excel, but really, it's error prone and you might want to avoid it unless you really do want people to use a specific version of Excel. You can tell the `Shell()` function the precise location of the copy of Excel you want to open and then provide the full path to the file as before.

Now it's time to look at the `ExportDB()` subroutine. This is one of those situations where you can create some pretty generic code that will reduce potential errors in your applications. The `ExportDB()` subroutine accepts the database you've opened for export, the name of the table you want to create in the exported file, the export path, and the export type as input. Using the correct export type is important. Table 8.1 shows the export types that IDEA supports.

The `ExportDB()` code begins by creating the `ExportDatabase` task. The subroutine makes some assumptions, as you probably will when you create a generic function, such as the need to include all fields in the primary (or source) database. Consequently,

TABLE 8.1 Database Export Types

Type of Export	Type
dBASE3	"DBF3"
dBASE4	"DBF4"
HTML Table	"HTM"
Microsoft Access 2000–2003	"MDB2000"
Microsoft Access 2007	"MDB2007"
Microsoft Access 97	"MDB97"
Microsoft Excel 2002 XML Spreadsheet	"XML1"
Microsoft Excel 2007	"XLSX"
Microsoft Excel 97-2002	"XLS8"
Microsoft Word	"DOC"
Tab Separated Variable	"TSV"
Text Delimited	"ASC"
Text Fixed Length	"FXD"
XML	"XML"

the code calls `ExportData.IncludeAllFields()` to add all of the fields in the primary database to the output.

The next step is to perform the task. You must supply the following arguments:

- **ExportPath:** The location of the exported file. If you supply just a file name, then IDEA will use the current `WorkingDirectory` location for the file.
- **TableName:** Many applications, such as Excel and Access, require a table name for the exported data. You don't have to supply a table name for some applications, such as Word. When working with an application like Word, simply provide an empty string (`""`) as input.
- **ExportType:** This argument defines the kind of output file that IDEA creates. You must supply one of the strings shown in Table 8.1.
- **Beginning Record:** The example makes an assumption that you want to output all of the records in the database. Consequently, the beginning record is 1. However, you can set this value to any record in the database. Export will begin at that record number in the current sort order.
- **Ending Record:** Again, the example assumes that you want to output all of the records, so it simply uses `db.Count` to obtain the number of the last record in the database. You can supply any number that is equal to or greater than the beginning record.
- **Criteria:** In some cases, you want to eliminate records based on some criteria. This argument lets you choose records based on any criteria expression to limit the output data.

After the example outputs the data, it cleans up memory by setting `ExportData` to `Nothing`. Notice that you don't set `db` to `Nothing` in this case because `db` comes

from the caller. The caller is responsible for cleaning up the memory used by `db`. In fact, you can cause some hard to locate errors in your application by freeing memory at the wrong time, so always exercise care in freeing memory within the subroutine or function that created the variable or object in the first place.

Working with Fields Using Field

There are many ways in which you can use fields to work with IDEA databases. The example in the "Committing the Database" section shows one such method. However, it's more likely that you'll want to create a field to use for auditing data. For example, you might want to calculate a total based on the content of several existing fields in the database or you might want to combine several fields to make them easier to use. A field can contain anything you need—you could even create a Boolean field to indicate whether you have audited a particular piece of data, as shown in Listing 8.6. An added field is normally called a *virtual* field because it's added to an existing database, but it doesn't have a physical presence in the database. *Native* fields are part of the database.

The example begins by opening the database and creating a `TableManagement` task. It's also necessary to create a `Table` object using `db.TableDef` .

At this point, the code creates the `Field` object, `AuditField`, using the `NewField()` method. The code then configures `NewField()` for use—giving it a name and description. As a minimum, you must also assign the field a type and an equation. Table 8.2 shows the acceptable types for fields. An editable field is one in which you can change the value, while a virtual field is one that doesn't actually appear as part of the database, but is something you add onto the database for your own use. IDEA automatically refreshes the displayed values while you work with the database as IDEA ensures the equation is recalculated and the displayed values are accurate.

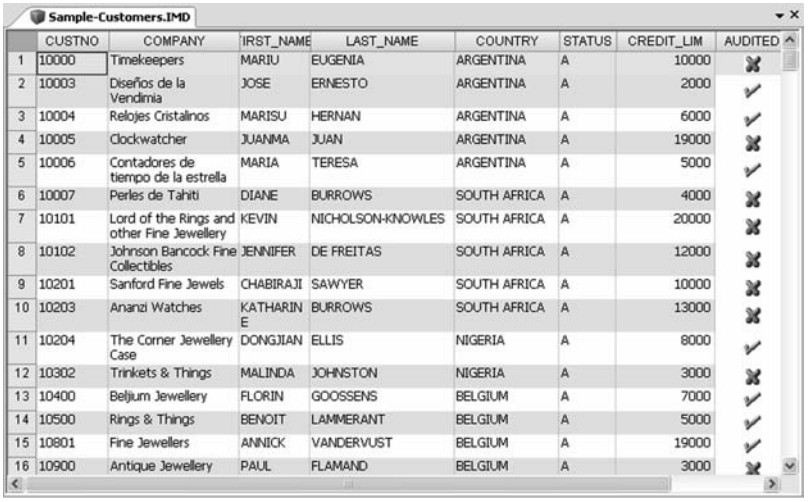
The equation can be anything that works with the field type you supply. This equation simply says to set all of the field entries to false since you don't know at the time of field creation whether any of the records have been audited.

TABLE 8.2 IDEA Database Field Types

Data Type	IDEAScript Constant
Boolean	"WI_BOOL"
Character	"WI_CHAR_FIELD"
Date	"WI_DATE_FIELD"
Editable Character	"WI_EDIT_CHAR"
Editable Date	"WI_EDIT_DATE"
Editable Numeric	"WI_EDIT_NUM"
Multistate (True, False, Unknown)	"WI_MULTISTATE"
Numeric	"WI_NUM_FIELD"
Virtual Character	"WI_VIRT_CHAR"
Virtual Date	"WI_VIRT_DATE"
Virtual Number	"WI_VIRT_NUM"

LISTING 8.6 Adding a Field to a Database

```
Sub Main
   ' Open the sample database.
   Dim db As Database
   Set db = OpenDB("Sample-Customers.imd")

   ' Create the task.
   Dim NewField As Task
   Set NewField = db.TableManagement

   ' Obtain a reference to the table.
   Dim ThisTable As Table
   Set ThisTable = db.TableDef

   ' Create a new field.
   Dim AuditField As Field
   Set AuditField = ThisTable.NewField

   ' Configure the new field.
   AuditField.Name = "Audited"
   AuditField.Description = "Have we audited this data?"
   AuditField.Type = WI_BOOL
   AuditField.Equation = 0

   ' Add the field.
   NewField.AppendField AuditField
   NewField.PerformTask

   ' Clean up memory.
   Set AuditField = Nothing
   Set ThisTable = Nothing
   Set NewField = Nothing
   Set db = Nothing
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```



Sample-Customers.IMD

	CUSTNO	COMPANY	IRST_NAME	LAST_NAME	COUNTRY	STATUS	CREDIT_LIM	AUDITED
1	10000	Timekeepers	MARIU	EUGENIA	ARGENTINA	A	10000	✗
2	10003	Diseños de la Vendimia	JOSE	ERNESTO	ARGENTINA	A	2000	✓
3	10004	Relojes Cristalinos	MARISU	HERNAN	ARGENTINA	A	6000	✓
4	10005	Clockwatcher	JUANMA	JUAN	ARGENTINA	A	19000	✗
5	10006	Contadores de tiempo de la estrella	MARIA	TERESA	ARGENTINA	A	5000	✓
6	10007	Perles de Tahiti	DIANE	BURROWS	SOUTH AFRICA	A	4000	✗
7	10101	Lord of the Rings and other Fine Jewellery	KEVIN	NICHOLSON-KNOWLES	SOUTH AFRICA	A	20000	✗
8	10102	Johnson Bancock Fine Collectibles	JENNIFER	DE FREITAS	SOUTH AFRICA	A	12000	✗
9	10201	Sanford Fine Jewels	CHABIRAJI	SAWYER	SOUTH AFRICA	A	10000	✗
10	10203	Ananzi Watches	KATHARIN E	BURROWS	SOUTH AFRICA	A	13000	✗
11	10204	The Corner Jewellery Case	DONGJIAN	ELLIS	NIGERIA	A	8000	✓
12	10302	Trinkets & Things	MALINDA	JOHNSTON	NIGERIA	A	3000	✗
13	10400	Beljium Jewellery	FLORIN	GOOSSENS	BELGIUM	A	7000	✓
14	10500	Rings & Things	BENOIT	LAMMERANT	BELGIUM	A	5000	✓
15	10801	Fine Jewellers	ANNICK	VANDERVUST	BELGIUM	A	19000	✓
16	10900	Antique Jewellery	PAUL	FLAMAND	BELGIUM	A	3000	✗

FIGURE 8.7 The New Field lets You Make Changes as Needed

The next step is to append the field to the database. You perform this task in two steps. First, call the `AppendField()` method to append the field. Second, call `PerformTask()` to make the change permanent. The code finishes by cleaning up the objects it has created.

When you run this example, you see a new field added to the `Sample-Customers` database as shown in Figure 8.7. When you click one of the entries, it changes from a red X to a green check mark. You can also modify this field programmatically to set it based on the results of analysis you perform.

To see how the change you just made affected the database structure, select Data > Field Manipulation. You'll see the Field Manipulation dialog box shown in Figure 8.8. As you can see, the new field appears at the bottom of the list using the correct data type and includes a description.

Note

IDEA may not let you delete a native field you create by mistake. If you see an error message when you try to delete a field, select View > Options. You'll see the System tab of the Options dialog box. Clear the Do Not Allow Deletion of Native Fields option and click OK. You can now delete the field. You can always delete a virtual field because it contains a calculated value and isn't actually part of the database.

Working with Records

As described in the "Setting Database Criteria" section in Chapter 7, you work either with an individual record or with a group of records. Individual records are normally the result

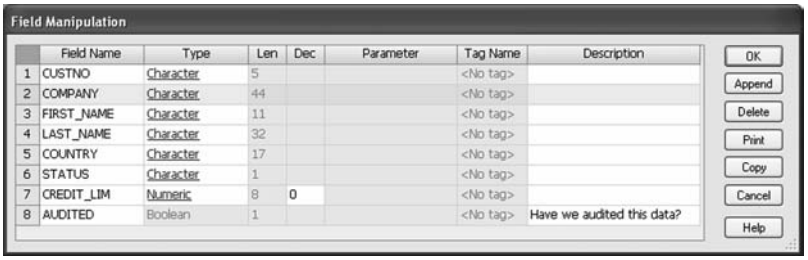


FIGURE 8.8 Check the Database Structure Change Created by the New Field

of a search of some kind or you know that a particular record contains the information you need. Record sets normally contain records that match particular criteria—the criteria give them something in common. However, a record set can contain any group of records at the outset of an analysis. The following sections discuss both the individual record and the record set.

Obtaining a Record Set

Most of your database experiences will begin with a record set because you have to search for a particular record and then interact with that record. The search is always performed on a record set. Listing 8.7 shows how to perform a near search using IDEAScript. In this case, you see three dialog boxes as output: one that shows matches for both criteria, one that shows records that matched just the country, and one that shows records that matched just the credit limit. Performing this kind of search helps you see what's available, even when the database doesn't have precisely the match you wanted.

LISTING 8.7 Performing Tasks with a Group of Records

```
Sub Main
   ' Create a handy variable for adding lines to the output.
   Dim vbCrLf As String
   vbCrLf = Chr(13)  + Chr(10)

   ' Open the sample database.
   Dim db As Database
   Set db = OpenDB("Sample-Customers.imd")

   ' Access the RecordSet.
   Dim RS As RecordSet
   Set RS = db.RecordSet

   ' Move to the first record.
   RS.ToFirst
   RS.Next

   ' Obtain some search criteria.
   Dim Country As String
```

(*continued*)

```
   Country = InputBox$("Type the name of the country you want to find.")
   Country = UCase(Country)

   Dim CredLim As String
   CredLim = InputBox$("Type the credit limit you want to find.")

   ' Define variables to hold the results.
   Dim MatchCountry As String
   Dim MatchCredLim As String
   Dim MatchBoth As String

' Locate matching records using a loop.
   Dim Count As Integer
   Dim CurrCountry As String
   Dim CurrCredLim As String
   For  Count = 1 To RS.Count - 1
      ' Get the current country and credit limit.
      CurrCountry = _
         RS.ActiveRecord.GetCharValue("COUNTRY")
      CurrCredLim = _
         CStr(RS.ActiveRecord.GetNumValue("CREDIT_LIM"))

      ' Check for a match and add it to the correct string.
      If  CurrCountry = Country And CurrCredLim = CredLim Then
         MatchBoth = MatchBoth  + _
RS.ActiveRecord.GetCharValue("CUSTNO") + _
            " " + CurrCountry + " " + CurrCredLim + vbCrLf
      ElseIf CurrCountry = Country Then
         MatchCountry = MatchCountry + _
            RS.ActiveRecord.GetCharValue("CUSTNO") + _
            " " + CurrCountry + " " + CurrCredLim + vbCrLf
      ElseIf CurrCredLim = CredLim Then
         MatchCredLim = MatchCredLim + _
            RS.ActiveRecord.GetCharValue("CUSTNO") + _
            " " + CurrCountry + " " + CurrCredLim + vbCrLf
      End If

      ' Go to the next record.
      RS.Next
   Next

   ' Display the result.
   MsgBox MatchBoth, MB_ICONINFORMATION, _
      "Records that Matched Both Criteria"
   MsgBox MatchCountry, MB_ICONINFORMATION, _
      "Records that Matched the Country"
   MsgBox MatchCredLim, MB_ICONINFORMATION, _
      "Records that Matched the Credit Limit"

   ' Clear memory.
   Set RS = Nothing
   Set db = Nothing
End Sub
```

```
Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

The example begins by opening the database and obtaining access to its record set. The code then moves the *record pointer* (the indicator that points to just one record within the record set so you can access it) to the first record.

At this point, the code displays two input boxes to request search criteria. The first input box requests a country name and the second requests a credit limit.

Remember that the example outputs three dialog boxes, so the next task is to create the three strings that accept this output. The variable names tell you that the first string contains country matches, the second credit limit matches, and the third those few records that match both criteria.

A `For...Next` loop helps the code examine each of the records in the database. After each loop is completed, the code executes the `RS.Next()` method to move to the next record in the record set. The code begins by obtaining the value of the COUNTRY and CREDIT_LIM fields of the current record. Notice that the code relies on a series of `If...ElseIf...Then` statements to check for matches. You can't use three `If...Then` statements because the `MatchCountry` and `MatchCredLim` strings would contain the precise matches.

Eventually, the code goes through each of the records and locates all of the records that contain the respective matches. It then displays the three dialog boxes showing the results, clears memory, and exits. Try this example out with Belgium as the country and a credit limit of 19000. Remember not to include the usual monetary symbol or other punctuation with the credit limit.

Interacting with an Individual Record

Listing 8.8 begins by opening the database, accessing the record set, and taking a single record from that record set. Always remember to place the record pointer at a specific record. Nothing terrible will happen if you don't, but the output will contain a lot of blanks and no information.

LISTING 8.8 Performing Tasks with a Single Record

```
Sub Main
   ' Create a handy variable for adding lines to the output.
   Dim vbCrLf As String
   vbCrLf = Chr(13)  + Chr(10)

   ' Open the sample database.
   Dim db As Database
   Set db = OpenDB("Sample-Customers.imd")

   ' Access the RecordSet.
   Dim RS As RecordSet
   Set RS = db.RecordSet

   ' Move to the first record.
   RS.ToFirst
   RS.Next

   ' Obtain the record.
   Dim Rec As Record
   Set Rec = RS.ActiveRecord

   ' Display information about the record.
   MsgBox "The record number is: " & Rec.RecordNumber & _
      vbCrLf & "The number of fields is: " & Rec.NumberOfFields & _
      vbCrLf & "The company name is: " & Rec.ValueAt(2) & _
      vbCrLf & "The contact name is: " & _
      Rec.ValueAt(3) & " " & Rec.ValueAt(4) & _
      vbCrLf & "The credit limit is: " & Rec.ValueAt(7)

   ' Clear memory.
   Set Rec = Nothing
   Set RS = Nothing
   Set db = Nothing

   ' Close the database.
   Client.CloseDatabase "Sample-Customers.imd"
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

FIGURE 8.9 Individual Records Can Tell You About the Content of Just That Record

In this case, the example shows some of the statistics you can obtain from a record. For example, you can obtain the record number (the current location of the record pointer) and the number of fields in the record. (The first record in the database is always record 0.) It's important to remember that the record number is only valid for the current database configuration. If you change the database in some way, such as creating an index, the record number will also change. The example also outputs some of the record data by using field numbers, rather than field names as used in previous examples. Figure 8.9 shows the output of this example.

Working with Tables

As you might have already guessed from previous discussions, tables are the gateway to the information contained in a database. Most of the examples you've seen so far modify an existing table because that's what you'll do most of the time. You use the `TableDef` object to work with existing tables because it provides the kind of access you need to perform tasks such as creating virtual fields.

In some situations, you need to create a table of your own. When you work with large data sets, creating a table can become time consuming using the `TableDef` object because it does everything immediately. The `TableManagement` object is the best choice when you need to create a table because it lets you set everything up and then tell IDEA to perform the task. As a result, the application saves time by working more efficiently. The following sections show how to work with both the `TableDef` and `TableManagement` objects.

Copying Databases Using TableDef

Sometimes you need to create a copy of a database for analysis purposes. For example, you might need to perform "what if" analysis on the data. The example in Listing 8.9 shows how to create a copy of the database you currently have open.

LISTING 8.9 Using the TableDef Object

```
Sub Main
   ' Create the database object.
   Dim db As Database
   Set db = Client.CurrentDatabase

   ' Create the input table object.
   Dim InTable As Table
   Set InTable = db.TableDef

   ' Locate the file name within the database name.
   Dim Pos As Integer
   Pos = 1
   Dim CurrName As String
   CurrName = db.Name
   Do While InStr(Pos, CurrName, "\")
      Pos = InStr(Pos, CurrName, "\")
      Pos = Pos + 1
   Loop

   ' Create a new database name.
   Dim NewName As String
   NewName = Left(CurrName, Pos - 1) + _
      "Copy-of-" + _
      Right(CurrName, Len(CurrName) - Pos + 1)
   MsgBox "Placing database copy in: " + NewName

   ' Create an output table.
   Dim OutTable As Table
   Set OutTable = Client.NewTableDef

   ' Copy the structure from the existing table.
   OutTable.CopyFrom db.TableDef

   ' Create the new database.
   Dim newDB As Database
   Set newDB = Client.NewDatabase (NewName, "", OutTable)

   ' Obtain or create the required records and recordsets.
   Dim InRS As RecordSet
   Set InRS = db.RecordSet
   Dim InRecord As Record
   Set InRecord = InRS.ActiveRecord
   Dim OutRS As RecordSet
   Set OutRS = newDB.RecordSet
   Dim OutRecord As Record
   Set OutRecord = OutRS.NewRecord

   ' Set the output table to allow writing.
   Set OutTable = newDB.TableDef
   OutTable.Protect = False

   ' Define objects and variables needed for copying.
   Dim RecCount As Integer
   Dim FldCount As Integer
   Dim CopyField As Field
```

```
   ' Begin a loop to copy the records.
   InRS.ToFirst
   For RecCount = 1 To db.Count
      ' Advance the Record Pointer
      InRS.Next

      ' Begin a loop to copy each field in a record.
      For FldCount = 1 To InTable.Count
         ' Obtain the current field.
         Set CopyField = InTable.GetFieldAt(FldCount)
           ' Copy only non-virtual fields.
         If CopyField.IsVirtual = False Then
            ' Determine the field type and use the appropriate
            ' method to copy it.
            If CopyField.IsNumeric Then
               OutRecord.SetNumValueAt _
                  FldCount, _
                  InRecord.GetNumValueAt (FldCount)
            ElseIf CopyField.IsCharacter Then
               OutRecord.SetCharValueAt _
                  FldCount, _
                  InRecord.GetCharValueAt (FldCount)
            Else
               OutRecord.SetDateValueAt _
                  FldCount, _
                  InRecord.GetDateValueAt (FldCount)
            End If
         End If
      Next
      ' Add the record to the output recordset.
      OutRS.AppendRecord OutRecord

      ' Clear the record for the next set of fields.
      OutRecord.ClearRecord
   Next

' Protect the output database.
   OutTable.Protect = True

   ' Save the database.
   NewDB.CommitDatabase

   ' Open the new database for viewing.
   Client.OpenDatabase NewName
      ' Clean up memory.
   Set OutRecord = Nothing
   Set OutRS = Nothing
   Set OutTable = Nothing
   Set NewDB = Nothing
   Set InRecord = Nothing
   Set InRS = Nothing
   Set InTable = Nothing
   Set db = Nothing
End Sub
```

This is a lot of code, but if you take it one piece at a time, you'll discover it really isn't hard to understand. Don't let this bigger example scare you! Someday you'll write code this long and longer without really thinking about it.

The code begins by creating a database object from the currently opened database. As you write more applications, you'll find that you often don't know what database the user has open, so you need a way to access that database without using the `OpenDB()` function used for earlier examples in the chapter. The `Client.CurrentDatabase` is the perfect way to perform this task. After the code gets the current database, it also creates an input table object.

You want to give the output database a name that the user will recognize. Otherwise, the user will create a copy and not know where to find it. The next bit of code shows how to obtain just the file name from the `Name` property. The code then adds `Copy-of-` to the file name and then adds the new file name to the path used by the original file. Now the original and the copy will appear in the same location and have similar names—making it very easy for the user to locate the file. The application even displays a message box showing the new file name.

The next step is to create an output table. Of course, this table doesn't have any structure, so the code uses the `CopyFrom()` method to copy the structure from the existing table to the new table.

Once you have a table structure, you can create a new database. The example uses the `NewDatabase()` method to create a new database by providing the new name the code created earlier, along with the new table definition.

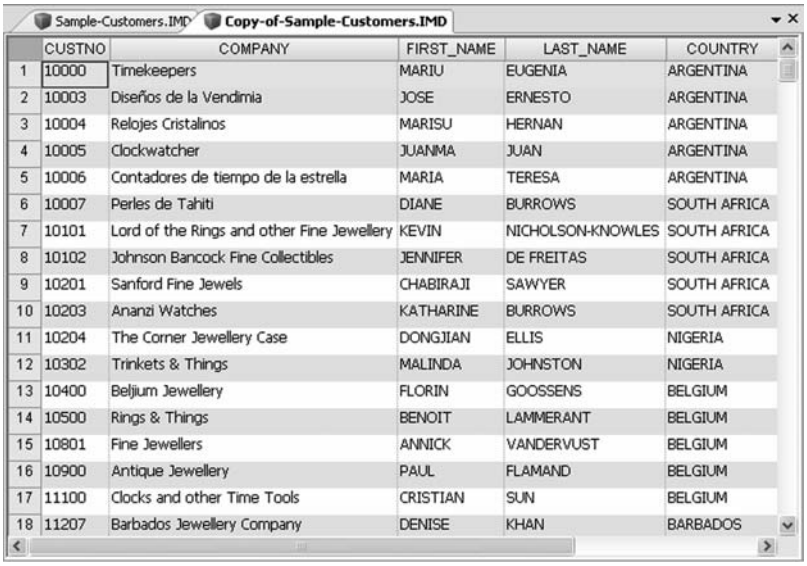
The copying process requires the code to perform a field-level copy from one database to the other. In order to perform such a copy, the code creates input and output record sets and records. The code also removes protection from the output table. This step is essential or the new table might be protected and the application will fail even though the code should work as expected.

It's time to copy the data from one table to the other. The actual copying process requires two steps. First, the code will access the current record. Second, the code will access each field in that record and copy it from the input database to the output database. Part of this looping process is to keep close track on both the current record number and the current field number, so the code creates the required variables.

At this point, the looping begins. There's a separate method for copying numbers (`SetNumValueAt()`/`GetNumValueAt()`), strings (`SetCharValueAt()`/`GetCharValueAt()`), and dates (`SetDateValueAt()`/`GetDateValueAt()`), so the loop uses an `If...ElseIf...EndIf` structure to perform the task based on the input type.

Field level copying produces a new record. In order to get the record into the output database, the code must use the `AppendRecord()` method. At this point, the record is dirty—it contains information that you don't want to appear in the next record. The code calls the `ClearRecord()` method to clean the record for the next field-level copy.

When the entire copying process is done, memory contains the new database. However, not all of that database appears on disk yet. Calling the `CommitDatabase()` method places the entire database on disk. However, before you save the database, you should always protect the table so that no changes can occur to the data without creating



Sample-Customers.IMD | Copy-of-Sample-Customers.IMD

	CUSTNO	COMPANY	FIRST_NAME	LAST_NAME	COUNTRY
1	10000	Timekeepers	MARIU	EUGENIA	ARGENTINA
2	10003	Diseños de la Vendimia	JOSE	ERNESTO	ARGENTINA
3	10004	Relojes Cristalinos	MARISU	HERNAN	ARGENTINA
4	10005	Clockwatcher	JUANMA	JUAN	ARGENTINA
5	10006	Contadores de tiempo de la estrella	MARIA	TERESA	ARGENTINA
6	10007	Perles de Tahiti	DIANE	BURROWS	SOUTH AFRICA
7	10101	Lord of the Rings and other Fine Jewellery	KEVIN	NICHOLSON-KNOWLES	SOUTH AFRICA
8	10102	Johnson Bancock Fine Collectibles	JENNIFER	DE FREITAS	SOUTH AFRICA
9	10201	Sanford Fine Jewels	CHABIRAJI	SAWYER	SOUTH AFRICA
10	10203	Ananzi Watches	KATHARINE	BURROWS	SOUTH AFRICA
11	10204	The Corner Jewellery Case	DONGJIAN	ELLIS	NIGERIA
12	10302	Trinkets & Things	MALINDA	JOHNSTON	NIGERIA
13	10400	Beljium Jewellery	FLORIN	GOOSSENS	BELGIUM
14	10500	Rings & Things	BENOIT	LAMMERANT	BELGIUM
15	10801	Fine Jewellers	ANNICK	VANDERVUST	BELGIUM
16	10900	Antique Jewellery	PAUL	FLAMAND	BELGIUM
17	11100	Clocks and other Time Tools	CRISTIAN	SUN	BELGIUM
18	11207	Barbados Jewellery Company	DENISE	KHAN	BARBADOS

FIGURE 8.10 Create Copies of Databases That You Can Then use for Analysis Purposes

an audit trail. The final steps are to open the new database and clean up memory. Figure 8.10 shows typical output from this example.

Creating New Tables Using TableManagement

You may eventually need to create your own database to house information that you don't want to appear in tables you're auditing. When this situation occurs, you use the `TableManagement` object to perform the task. The code normally performs three steps as shown in `Main()` in Listing 8.10:

1. Creates the database.
2. Fills the database with information.
3. Displays the database so you can interact with it.

`CreateDB()` begins by defining a new `Table` object and using the `Client.NewTableDef()` method to create it. `NewTable` is the center of creating the database definition.

You can add as many fields as desired for your database. These fields can be any type and you can call them anything you want. The example code shows how to add two fields. The first field contains up to 20 characters, while the second field contains numbers. In both cases, you must supply a field name and type. When working with character fields, you define the length of the field. On the other hand, when working with numeric fields, you define the number of decimal places the field must hold. After the code creates the fields, it adds them to `NewTable`.

LISTING 8.10 Creating a New Database

```
Sub Main
    ' Begin by creating the database.
   Dim db As Database
   Set db = CreateDB()

    ' Add data to the database.
   FillDB(db)

    ' Clear memory.
   Set db = Nothing

    ' Open the database.
   Client.OpenDatabase "SampleData.imd"
End Sub

Function CreateDB() As Database
    ' Create a table.
   Dim NewTable As Table
   Set NewTable = Client.NewTableDef

    ' Define a field for the table.
   Dim AddedField As Field
   Set AddedField = NewTable.NewField
   AddedField.Name = "Field1"
   AddedField.Type = WI_CHAR_FIELD
   AddedField.Length = 20

    ' Add the field to the table.
   NewTable.AppendField AddedField

    ' Perform the same steps for a second field.
   Set AddedField = NewTable.NewField
   AddedField.Name = "Field2"
   AddedField.Type = WI_NUM_FIELD
   AddedField.Decimals = 2
   NewTable.AppendField AddedField

    ' Create the database.
   Dim db As Database
   Set db = Client.NewDatabase("SampleData.imd", "", NewTable)

    ' Return the new database to the caller.
   Set CreateDB = db

    ' Clear memory
   Set db = Nothing
   Set AddedField = Nothing
   Set NewTable = Nothing
End Function
```

```
Sub FillDB(db As Database)
   ' Obtain the recordset.
   Dim RS As RecordSet
   Set RS = db.RecordSet

   ' Obtain a new record.
   Dim Rec As Record
   Set Rec = RS.NewRecord

   ' Change the table settings to allow writing.
   Dim TheTable As TableDef
   Set TheTable = db.TableDef
   TheTable.Protect = False

   ' Use the positional approach to add data.
   Rec.SetCharValueAt 1, "Character Value 1"
   Rec.SetNumValueAt 2, 11.1
   RS.AppendRecord Rec

   ' Clear the record of residual data.
   Rec.ClearRecord

   ' Use the field name method to add data.
   Rec.SetCharValue "Field1", "Character Value 2"
   Rec.SetCharValue "Field2", 22.2
   RS.AppendRecord Rec

   ' Protect the table before you commit it.
   TheTable.Protect = True

   ' Commit the database.
   db.CommitDatabase
   ' Clear memory.
   Set TheTable = Nothing
   Set Rec = Nothing
   Set RS = Nothing
End Sub
```

Now it's time to create the database using `Client.NewDatabase()`. The output of this method is a reference to the new database that you can use for other purposes. The `NewDatabase()` method requires that you provide a database path and the `Table` object as input. The example uses the default path and gives the database a name of `SampleData.imd`. `CreateDB()` ends by returning a reference to the new database and clearing memory.

`FillDB()` comes next. This subroutine fills the new database with information. In order to perform this task, the code creates a `RecordSet`, `RS`, and uses `RS` to create a new `Record`, `Rec`. `Rec` can add one record at a time, but you can clear it as necessary to add as many records to the database as desired.

Before the code can do anything else, however, it must unprotect the database fields so that the code can write data to them. To perform this task, the code creates a `TableDef` object, `TheTable`, and then changes its `Protect` property to `False`.

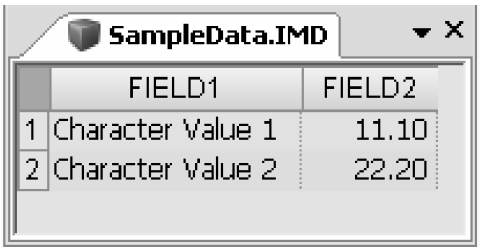


FIGURE 8.11 Create Databases as Required to Hold Data

The database is ready to receive new records. The code sets the value of each field and then uses the `AppendRecord()` method to add the record to `RS` . Always remember to clear `Record` after each addition using the `ClearRecord()` method.

When the code is finished adding records, it calls `CommitDatabase()` to send the changes to disk. It then clears memory and returns to the caller. `Main()` uses the `Client.OpenDatabase()` method to open the newly created database so you can see the additions shown in Figure 8.11.

Summary

This chapter has taken you another step into the world of using databases with IDEA. Databases are an essential part of working with IDEA because you need the data they contain to perform analysis. The examples in this chapter provide you with enough information to perform some basic verification tasks as well as some simple analysis. Most importantly, you've become familiar with keys, an important component of databases that helps you organize and search them, among other things.

In the "Adding One Database to Another Using AppendDatabase" section, you modified an example found in another chapter of this book to output another kind of table. The "Comparing Two Databases Using CompareDB" and "Locating Duplicate Keys with DupKeyDetection" sections also showed the result of modifications to the basic examples in the chapter. Experimentation of this sort is essential to discovering precisely how these IDEAScript tasks work. Before you move on to the next chapter, try modifications to some of the examples in the chapter to see what happens when you change something. Ask yourself whether the output is what you expected. If not, then you'll want to discover why the example you modified works differently than expected and try other modifications to get the result you anticipated. Don't worry about playing with the examples in this way—you can't break anything.

Chapter 9 gets into the specifics of more complicated database tasks. You also discover the IDEA database object model. Understanding this model is helpful because you need to know about the various objects that are available and the tasks you can perform using those objects. Chapter 9 builds on the knowledge you've gained in Chapters 7 and 8. Once you finish Chapter 9, you should be able to perform most common database-related tasks with IDEA. Chapters 13 and 17 expand your horizons even more, but before you get to these chapters, you must discover how to perform some other non-database-specific tasks.

CHAPTER 9

Considering the CaseWare IDEA Object Model

Up to this point, you've been looking at particular tasks that IDEAScript can perform. Each of these tasks relies on one or more objects to accomplish the work. The *task* defines what to do, while the *object* defines on what to do it. Your code defines how to perform the task using the object resources you provide.

There's a consistent relationship between these elements and you need to know how they work.

Objects don't simply exist by themselves. Consider an apple. The apple may be a single object, but it originally exists as part of a tree object, which is part of an orchard object, which is part of a farm object, which is part of . . . well, you get the idea. Objects are related to one another. In order to use a particular object effectively, you must know how that object is related to other objects. In short, you need to know about the *object model*, which is a representation of the relationship between various IDEAScript objects.

This chapter breaks the object model into pieces to make it easier to understand. Of course, before you can truly understand the object model, you must know more about objects, so that's the first topic discussed in this chapter. You discover that objects have certain features and that the way these features are implemented determines the functionality of the object.

Once you understand the basics of the object model, the chapter helps you understand the specifics of tasks and direct database manipulation. Tasks help you perform analysis and manipulate data in various ways. Direct database manipulation considers the underlying database structure and how it holds the data you manipulate. You need to know about both concepts in order to create useful applications.

Considering the IDEA Object Model

As previously stated, an object model is a representation of the relationship between objects in an application. Of course, it helps to understand what an object is before you begin talking about the object model, so the first section that follows discusses objects in general. The next section describes the client elements of the object model.

Understanding Objects

Many people find it hard to understand what an object is and why they should care about it. At one time, developers wrote applications to meet the requirements of the computer hardware. You literally wrote code to move information from one register in the processor to another. If you have no idea of what a register is, be glad, because this level of programming is extremely difficult and abstract. Newer languages are modeled on math concepts and procedures that are easier for humans to understand, but still don't have much bearing on the real world. Object Oriented Programming (OOP) is an effort to model development languages against the real world. In other words, programming languages have gone from catering to the needs of the hardware to catering to the needs of humans. So, objects, the focus of OOP, are important because they help you write applications based on your real-world knowledge.

The concept of a programming object is relatively easy to grasp as well. When you look around you, you see objects. You probably have a desk, and on the desk are a keyboard, monitor, mouse, and other objects. In short, you already know everything you need to know about objects based on your real world experiences. Objects aren't scary technology that only a programmer can love—everyone knows about them.

Objects are so common, however, that you probably haven't spent too much time thinking about them. After all, your desk is important because it holds other objects, not because it's an object. You don't go to your refrigerator looking for an object, you go there to find an apple. Consequently, even though you already know all about objects, you haven't really thought about them as objects before.

For many people, the difficulty in working with computer objects is that you can't truly touch them. No one can touch a database, it's abstract. Sure, the consequences of using a database are real, they have real world effects, but the database itself isn't really touchable. Even so, you can describe a database, imagine what it contains, see it in your mind, work with the content, and interact with it in other ways. It's a fact that you can do so many things with a database that it makes it an object that you can understand, even if you can't touch it.

In order to make objects something the computer understands as well, developers needed to come up with a common way to describe them. In short, no matter what application you rely on, no matter which language you use, no matter what computer you use, an object always consists of the same basic elements: classes, methods, properties, and events. These terms may sound foreign to you, but once you work with them for a while, you'll discover that they really do make a lot of sense. The following sections describe these four terms in detail.

Note

IDEAScript doesn't support events in the same sense as other OOP and Visual Basic-based languages. An event-driven programming language can wait to respond to an event, such as the user clicking a button, or suddenly divert to a subroutine when an

event occurs without having to call the event handler directly. IDEAScript behaves in a more procedural manner, you cannot create subroutines based on events such as "Click" or "Mouse Over" as you can with true Visual Basic. Instead, when handling a dialog box, for example, virtually the only thing that can invoke a step forward in the code is a click of a button even if you add a Dialog Function into the code. Don't worry too much about this issue for right now—you see how to interact with dialog boxes in Chapter 12. All you need to remember is that IDEAScript doesn't provide full event handling.

CLASSES A *class* sounds like a complex term, but it really isn't. The class is simply a blueprint for creating an object. It works almost precisely the same as a blueprint for a house. IDEAScript looks at the class, which is the plan for the object, and builds the object it describes for you. Consequently, when you type:

```
Dim MyField as Field
```

`Field` is the name of the class you want IDEAScript to build for you and `MyField` is the name of the resulting `Field` object. Every other object in IDEAScript works the same way. You use a class to describe what you want IDEAScript to build and IDEAScript builds it for you as an object. In fact, you've already seen a number of instances of object usage in previous examples.

Like blueprints, classes have certain features that are common. You might not find every feature in every class, but every class can have these features:

- Methods
- Properties
- Events

These three features describe what the class provides, much as the various lines describe what a blueprint provides. You'll eventually work with all three features using IDEAScript, but the most common features are methods and properties. The sections that follow describe all three features.

METHODS *Methods* describe actions you can perform with an object. Think of a door for a moment. A door has two actions: open and close. These terms describe the door methods. When you interact with a door, you can open it or close it. Likewise, you can perform actions with various IDEAScript objects. For example, when working with a `TableDef` class, you use the `NewField()` method to create a new field for the associated table object.

Some methods ask questions about the object. When interacting with a door, you might ask whether the door is opened or closed in order to know what to do next. Many IDEAScript objects also include methods that ask questions. For example, when working with a `Field` class, you can use the `IsTime()` method to ask whether the associated

object has data of a time type. Even in this case, you're still performing an action—you're asking a question about the object.

PROPERTIES *Properties* describe the object in some way. Using the previous example, a door can be brown or red. It might have raised panels or rely on a classic design. IDEAScript properties also describe object features. For example, the `Field` class has the `Name` property that contains the name of the associated object.

It's possible for properties to allow change or to deny change, but allow you to obtain their value. Developers use several terms for these activities, but the most common terms are `get` (for obtaining the property value) and `set` (for changing the property value). Looking again at the door, it's possible to change the color property by painting the door, so you can say that you can both get and set the door's color property. However, it would be very hard to change the door style—the use of raised panels as an example—so the door's style property is get-only (or read-only). When working with the `Field` class, you can both get and set the `Name` property, but you can only get the `TagName` property.

EVENTS *Events* provide a means for the object to interact with you. Think about a bell attached to the door. When someone opens the door, the bell rings—an event has occurred. You didn't initiate the action, someone else did and the door told you about that action. Events work the same way with IDEAScript objects. When a user clicks a button on a dialog box you create, the button creates an event that tells you about the button click. You can write code to react to the click event. Chapter 12 tells you more about how events work within dialog boxes.

Working with the Client Elements

The `Client` object is the top-level object in the IDEAScript object model. Under the `Client` object, you find a wealth of other objects that help you work with databases. However, everything begins with the `Client` object. IDEA creates the `Client` object for you when it starts and you use the same `Client` object for the entire session. In short, of all the objects described in this book, the `Client` object is the only constant.

The other objects you use can be divided in several different ways. The first way to view objects is whether they directly interact with the database or they let you perform specific tasks with the database. This section only discusses the database tasks, but you can read about the task-related objects in the "Working with the Task Object Model" section.

The second way to view objects is whether they work with existing databases or help you create new ones. The examples you worked with in Chapter 8 show both object types. Figure 9.1 shows an overview of objects that work with existing databases, while Figure 9.2 shows an overview of objects that work with new databases. Keep both of these figures in mind as you work through the sections that follow.

WORKING WITH THE DATABASE, TABLEDEF, RECORDSET, AND TABLEMANAGEMENT OBJECTS The `Database`, `TableDef`, and `RecordSet` objects let you work with existing databases. As you've seen in the examples through the chapters so far, you use one of the `Client`

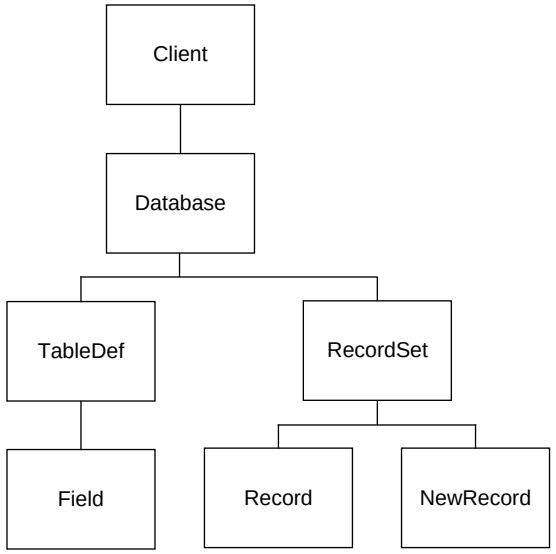


FIGURE 9.1 An Overview of the Objects That Deal with Existing Databases

object methods to obtain a reference to the `Database` object you want to use. It's then possible to use the `Database` object methods to obtain the `TableDef` and `RecordSet` objects, among others. In short, you follow the hierarchy of objects shown in Figure 9.1 to obtain successively finer access to the actual database.

It's interesting to note that Figure 9.1 shows `NewRecord`. You create a new record using an existing `RecordSet`. However, a new record is actually a new object and is actually part of the new database hierarchy. This strange combination of old and new occurs in the real world as well. When you want to create a widget, you first create

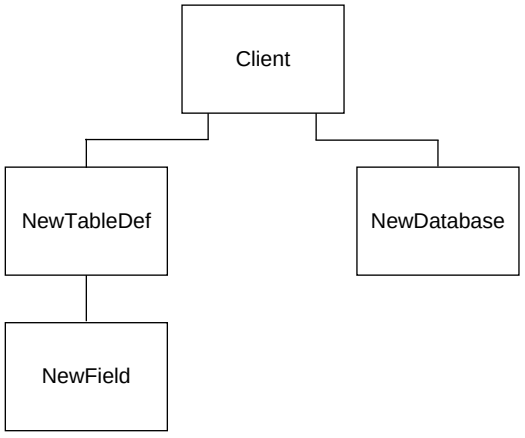


FIGURE 9.2 An Overview of the Objects That Deal with New Databases

a new machine to construct the widget. However, now the machine exists and it's no longer new. The widgets that the machine produces are indeed new, even though they come from an existing machine.

The `TableManagement` object is a kind of control panel. It lets you add, remove, and replace fields. In a way, this is a kind of maintenance action. You won't use the `TableManagement` object for normal database operations, but you need to keep it in mind when certain kinds of maintenance are required.

WORKING WITH THE NEWDATABASE, NEWTABLEDEF, AND NEWRECORD OBJECTS The `NewDatabase`, `NewTableDef`, and `NewRecord` objects help you create new databases using the hierarchy shown in Figure 9.2. Now it's important to understand what *new* means in this situation. The instant you create a `Database`, `TableDef`, or `RecordSet` object, it goes from being a new object to being an existing object. In other words, the time during which a particular object is new is quite short.

Look again at Listing 8.10. `Main()` contains calls to `CreateDB()`, during which time the various objects are new and you use the `NewDatabase` and `NewTableDef` objects to interact with them, and `FillDB()`, which works with the existing objects. Consequently, you must know when to switch from `NewDatabase`, `NewTableDef`, and `NewRecord` objects to the `Database`, `TableDef`, and `RecordSet` objects. The examples in this book will continue to demonstrate this particular distinction, but you must be aware that the distinction exists. Here's an overview of the three main objects used for working with new databases:

- **NewDatabase:** Creates a new database, which includes the database name, description, and the table definition. Even though it seems counterintuitive, you must always create the table definition before you create the database. In other words, you must have content to fill the database before you create the container to hold the content.
- **NewTableDef:** Creates a new table definition. A table definition consists of the fields and other structural elements used to store data, but it doesn't include the actual data. Consequently, you could define a field as having a particular name, holding characters, and being a certain length, but you don't say that the first record holds a certain value in that field.
- **NewRecord:** Whenever you want to add data to a database, you must create a new container to hold that data in the form of a `Record`. During the instant that your code creates the `Record`, the `Record` is new and you use the `NewRecord()` method to create the required object. However, once you create the object, it exists and you use the `Record` object to work with any data you want to provide. Since the `Database` that you use to hold the `Record` already exists, you actually create the `NewRecord` using the `RecordSet` object, as shown in Figure 9.1.

Working with the Task Object Model

Tasks are an essential part of working with IDEAScript because these objects define things you can do with a database. In fact, there are more task-related objects than just

about any other kind of object in IDEAScript. To understand where tasks fit into the object model, you need a detailed view of the entire IDEAScript object model as shown in Figure 9.3. This diagram is the one you should use for reference purposes as you work in other chapters of the book because it provides the most complete view of IDEAScript.

This diagram is useful because it shows where the various objects fit in and what you need to work with a particular object. For example, in order to work with a `TableDef` object, you need a `Database` object. Of course, the `Database` object comes from the `Client` object. Likewise, some tasks only require the `Client` object, while others require that you create a `Database` object. The following sections describe each of the task objects shown in Figure 9.3.

Note

This chapter describes the `Database` objects and tasks. You'll see the `Database` objects and tasks demonstrated throughout the book. Use this chapter as a reference when you need to discover more about a particular task or find the task related to work you need to perform. For example, you can see the `Aging` and `BenfordsLaw` tasks demonstrated in the "Employing Analysis" section in Chapter 10.

AdvancedFindTask

You use `AdvancedFindTask` to locate specific textual information within the database. In many respects, it works very much the same as the Find feature in a word processor or your e-mail program. You can use this task to locate information in one or more fields that are added to the task using the `AddFieldToInc()` method. It's also possible to look for data in multiple databases by using the `AddFile()` method.

Once you configure this task for use, the `PerformTask()` method accepts a number of inputs that control the report that IDEAScript generates for you. You supply the following information to perform the task:

- **Search String:** Contains the search string you want to use. The search string could consist of a single word or phrase. It's also possible to use advanced features in your search (see the "Advanced Features" entry for details).
- **Case Sensitivity:** Determines how IDEA treats letters during a search. When you set this argument to `1` for a case-sensitive search, IDEA treats "hello" differently than "Hello." A case-insensitive search will locate every instance of a word no matter what capitalization you use.
- **Whole Word Searches:** Determines whether IDEA searches only for whole words that match your search string or it checks for partial matches as well. Set this argument to `1` to match only whole words.
- **Advanced Features:** When performing the search using advanced features, you can add Boolean operators (`And`, `Or`, `Not`, and `Xor`), wildcard characters (`?` and `*`), and proximity operators (`/`, `@`, and `#`). It may seem as if using these advanced features

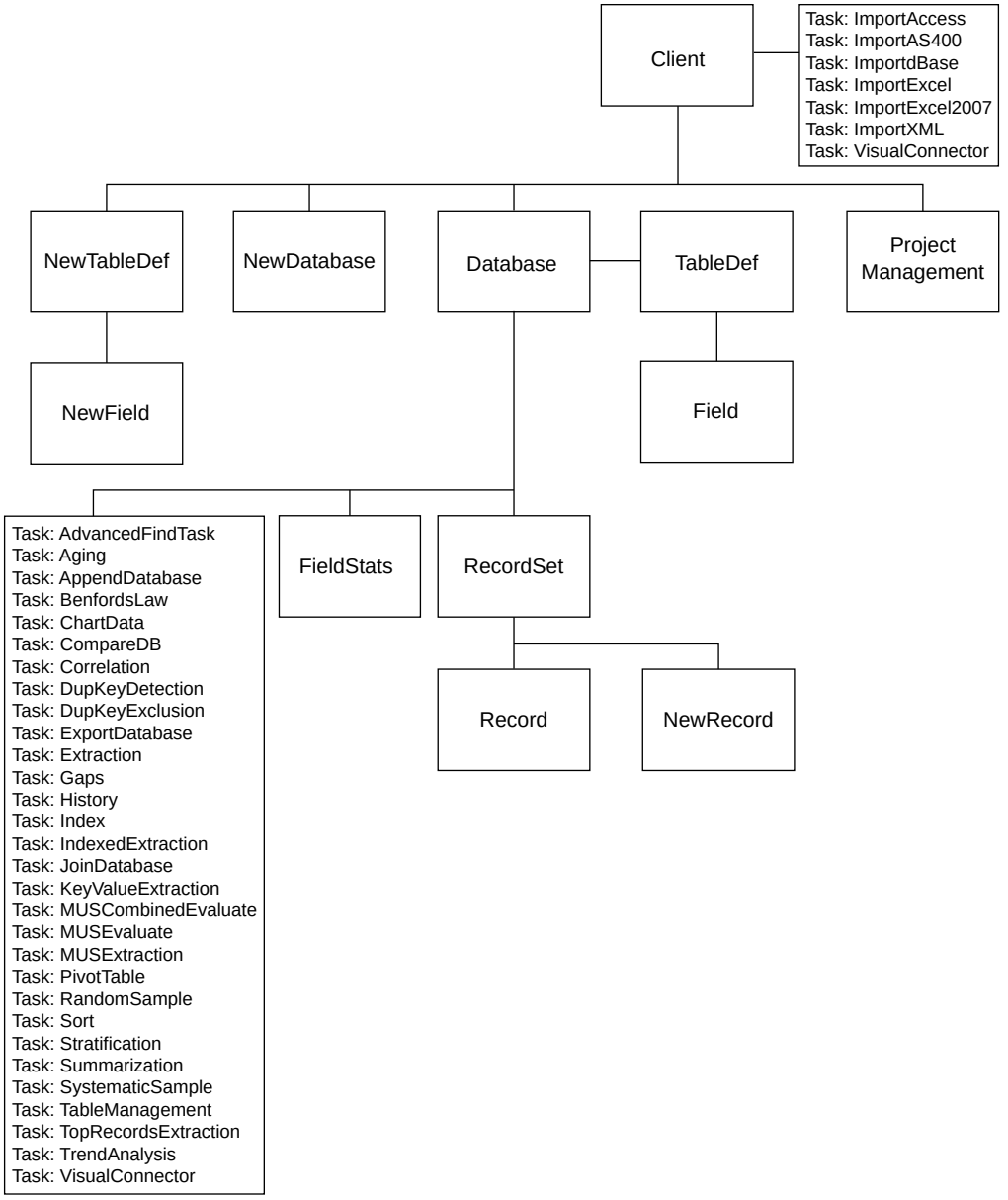


FIGURE 9.3 A Detailed View of the Entire IDEAScript Object Model

is always a good idea, but they can cause problems at times. For example, if you search for "Kline and Smith" with advanced features enabled, IDEA will treat "and" as a Boolean operator, not as part of the name. To turn advanced features off, set this argument to `0`.

If desired, you can send the output to an extraction database. Simply use the `RecordFilesPrefix()` method to define the prefix used to create the extraction database name. IDEA automatically generates the remainder of the name, which is guaranteed to be unique. Using a unique name makes it possible to perform multiple searches without worrying about a potential database name conflict. Chapter 13 tells you more about performing this task.

Aging

The `Aging` task helps you perform analysis on a database by splitting the database into six pieces based on age. For example, you could check an Accounts Receivable database for clients who are behind in their payments by so many months and then display the amount that they're behind. You decide on six continuous intervals to use for the analysis, such as 30, 60, 90, 120, 180, and 240 days (normally, the intervals are evenly sized, but you can use any intervals needed). Use the `IntervalTypeIndex` property to define whether aging occurs in days, months, or years.

In order to use this analysis, the database must have at least one date field and one numeric field. It's possible to create both detailed (using `CreateAgeDB()`) and summary (using `CreateSummaryDB()`) databases.

As with many other tasks, you can choose to include all of the source database fields using `IncludeAllFields()` or individual fields using either `AddFieldToInc()` or `AddFieldToIncAt()`. It's also possible to order the output database using `AddKey()`.

Before you can call `PerformTask()`, you must define the analysis criteria using `Info()`. This method accepts the aging date (the date from which the records are aged), the aging field (must be a date field), and the amount field (must be a numeric field). Chapter 10 tells you more about performing this task.

AppendDatabase

The `AppendDatabase` task helps you add two or more databases into a single database. You can use this task when you want to perform analysis on a single database structure, but from multiple sources (such as satellite offices of an organization).

In order to use this task, you begin by opening (and creating a reference for) one of the databases you want to appear in the output. You then use the `AddDatabase()` method to add as many additional databases as desired. When you finish adding databases, you call `PerformTask()` with the name of the output database. Add `Criteria` if you want to control the records that IDEA outputs to the database. Chapter 8 tells you more about performing this task.

BenfordsLaw

You use the `BenfordsLaw` task to detect patterns in the digits used for numeric fields in a database. This law states that the leading digits will fall into a predictable pattern in certain cases. When a database falls outside the pattern, you can use that information as a basis for looking for errors.

This particular task requires a good understanding of the principles behind Benford's Law, which are discussed in the "Using Benford's Law" section in Chapter 10. In order to perform this task, you must supply a number of input values including:

- **FieldToUse:** The name of the field you want to use for analysis purposes. This must be a numeric field.
- **ValueType:** Determines whether the analysis considers positive (`1`) or negative (`0`) numbers. Because of the way this analysis works, you can't analyze both positive and negative numbers as a single task.

There are also a number of optional values you can provide. For example, `CheckValues` will tell you the expected upper and lower boundaries for a particular digit when set to `True`. You can also determine whether IDEA actually creates a result by setting the value of `CreateResult`. Give the result a specific name by setting the `ResultName` property. As with most tasks, you can display the settings dialog box for the user by calling `DisplaySetupDialog()`.

After you set basic values, you call `AddAnalysis()` to define the kind of analysis that IDEA performs. The first argument defines the number of digits used for the analysis:

- **1:** Analyze only the first digit.
- **2:** Analyze the first and second digits.
- **4:** Analyze the first, second, and third digits.
- **8:** Analyze only the second digit.

The second argument contains the name of the file used to store the results. One or many analysis variations can be carried out simultaneously with this task. At this point, the code calls `PerformTask()` to complete the process. Chapter 10 tells you more about performing this task.

ChartData

You use the `ChartData` task to draw charts of your database. It's the same as selecting the **Data > Chart Data** command, except you don't have to perform the task manually. This task lets you define a number of optional chart characteristics, including the following:

- **ChartTitle:** Specifies the text to display as the chart title.
- **Criteria:** Defines the criteria used to identify the records you want processed during the task.
- **Legend:** Displays a legend when set to `True`.
- **LegendPosition:** Specifies the position of the legend. You can position the legend to the left (`1`), at the top (`2`), to the right (`3`), or at the bottom (`4`).
- **NumOfRecords:** Specifies the number of records to chart, starting with record number 1 using the current database order. Index or sort the database to obtain the records you want to use.

- **ResultName:** Specifies a unique result name.
- **Show3DChart:** Displays a 3-dimensional chart when set to `True`.
- **ShowGrids:** Displays the horizontal grid lines when set to `True`.
- **SnapShot:** Creates a snapshot of the database for the chart and then uses the snapshot to chart the data when set to `True`. A snapshot won't change, which means that your chart will remain static and not show database changes.
- **XFieldTitle:** Specifies the text to display on the X-axis.
- **YFieldTitle:** Specifies the text to display on the Y-axis.

Before you can use this task, you must define an X-axis field and a Y-axis field using the `XFieldName` property and the `AddYFieldName()` method. You can use `AddYFieldName()` multiple times to add more than one Y-axis field. In addition, you must specify the `NoOfSeries` (number of series), which is the number of lines or bars to use to display data. Finally, you must specify the `ChartType`. You can use any of these values for the `ChartType` property.

- Line - 0
- Bar - 1
- Curve - 2
- Scatter - 3
- Pie – 4
- Area – 5

After you define the optional and required values for the chart you want to create, you call `PerformTask()`. Chapter 18 tells you more about performing this task.

CompareDB

You use the `CompareDB` task to compare the content of a particular numeric field of two databases with the same matching key value. The comparison could work with the same database from different sources (say you want to analyze the data from satellite offices) or the same database at two different moments in time (to determine what has changed over time).

To perform this task, you use `AddMatchKey()` to define which fields to match. This method accepts the name of the key field from the primary database, the name of the key field from the secondary database, and the direction of sorting to use for the fields. When you call `PerformTask()`, you supply the path of the output database, the name of the comparison field (numeric type) in the primary database, the name of the comparison field (numeric type) in the secondary database, and the path of the secondary database. Chapter 8 tells you more about performing this task.

Correlation

You use the `Correlation` task to examine the mutual relationship between two fields held within a single database. For example, you might want to examine the relationship

between the stock price index and the current stock price for a particular company. To accomplish this task, you use the `AddFieldForCorrelation()` method to define the two fields containing this information in the database. You also need to define the field containing the company name using the `AuditUnitField()` method.

Now that you have the three pieces of required information, you might decide that you really don't want to examine every company in the database. In this case, you can use the `AddAuditUnit()` method to define which companies to examine.

At this point, you can set `CreateDB` to `True` to output a database. Use the `OutputDBName` property to assign the output database a name. The `CreateResult` and `ResultName` properties let you create a unique result name for the data. Finish the task as usual by calling `PerformTask()`. Chapter 18 tells you more about performing this task.

DupKeyDetection

You use the `DupKeyDetection` task to detect records that have the same key value in a database. In many cases, it's an error for the database to have multiple records with the same values—such as a customer database that has two clients with the same number, even though they're completely different clients.

To begin this task, you define which fields to include in the result using either `AddFieldToInc()` or `AddFieldToIncAt()`. If you decide to include all of the fields, simply use `IncludeAllFields()` to add them to the list. You must also use `AddKey()` to define the keys to look for and also sort the output. Optionally, you can determine which records to include using `Criteria`.

This task can output either the duplicate or non-duplicate records. Set `Output Duplicates` to `True` to output the duplicate values. Finally, call `PerformTask()` with the name of the output database. Chapter 8 tells you more about performing this task.

DupKeyExclusion

You use the `DupKeyExclusion` task to locate records that have the same values for one or more key fields and a different value for one key field. It's best to view this task in the context of `DupKeyDetection`. `DupKeyDetection` can be used to find all records where values in the key fields are duplicated across records. `DupKey-Exclusion` adds a further twist by insisting a further, critical field must hold a different value across the duplicated records. In short, it's a way to find the odd record in a group of similar database records. This task requires the same kinds of input as the `DupKeyDetection` task described in the previous section. Chapter 8 tells you more about performing this task.

ExportDatabase

You use the `ExportDatabase` task to output a database in a different format. For example, you might want to share your data with someone who only has Excel or you might want to perform additional auditing using Excel.

To begin this task, you define which fields to include in the result using either `AddFieldToInc()` or `AddFieldToIncAt()`. If you decide to include all of the fields, simply use `IncludeAllFields()` to add them to the list. You must also use `AddKey()` to define the keys to look for and also sort the output. Optionally, you can determine which records to include using `Criteria`.

When you call `PerformTask()`, you must provide a number of pieces of information as described in this list:

- **Output File:** Contains the name of the file that will receive the data. Make sure that the path contains the correct file extension for the external application you want to use.
- **Table Name:** Specifies the name of the table for applications, such as Microsoft Access, that require the information.
- **Type of Export:** Defines the kind of export to perform. Table 8.2 contains a complete list of the kinds of export that IDEA supports.
- **Start Record:** Defines the starting point for the export. If you want to start at the beginning record, set this value to `1`.
- **End Record:** Defines the ending point for the export. If you want to export every record, use the `Count` property of the `Database` object.
- **Criteria:** Specifies the records you want to appear in the output.

Chapter 8 tells you more about performing this task.

Extraction

You use the `Extraction` task to copy records that meet the criteria you specify from an existing database and place them in another database. This task makes it possible to work on a subset of the data and speed the analysis process.

To begin this task, you define which fields to include in the result using either `AddFieldToInc()` or `AddFieldToIncAt()`. If you decide to include all of the fields, simply use `IncludeAllFields()` to add them to the list. You must also use `AddKey()` to define the keys to look for and also sort the output. Optionally, you can determine which records to include using `Criteria`. It's possible to add a field that doesn't appear in the original database using `AddField()`. You must supply all of the usual information for adding a field including one of the field types defined in Table 8.3.

Once you decide on fields to include in the extraction, you use `AddExtraction()` to define the name of the output database and the equation to use for the extraction. Call `PerformTask()` with the starting and ending records you want to use for the extraction. If you want to export every record, use a starting value of `1` and an ending value of the `Count` property of the `Database` object. Chapter 14 tells you more about performing this task.

Gaps

You use the `Gaps` task to detect missing elements in a sequence of numbers, characters, or dates. This task allows you to include a number of optional values including:

- **Criteria:** Defines the criteria used to identify records you want to process during the task.
- **FromDate:** Specifies the starting date.
- **FromValue:** Specifies the starting key value.
- **Holidays:** Defines holiday dates. You provide the dates as a contiguous list, which can look pretty confusing if you run them all together: `task.Holidays = "05/2608/2512/2512/26"`. A better way is to add the dates as individual strings, such as `task.Holidays = "05/26" & "08/25" & "12/25" & "12/26".`
- **IgnoreHolidays:** Tells IDEA to ignore holidays you set with the `Holidays` property when set to `True`.
- **IgnoreWeekends:** Tells IDEA to ignore weekends when set to `True`.
- **Increment:** Specifies the numeric gap increment. This value defaults to `1`.
- **ToDate:** Specifies the ending date.
- **ToValue:** Specifies the ending key value.
- **UseCutoffs:** Analyzes records within a specified range when set to `True`. You must specify the range using a combination of the `FromDate`, `ToDate`, `FromValue`, and `ToValue` properties.

To start performing this task, you must specify the field you want to use in the `FieldToUse` property. If the field you've chosen is a character field, you must also provide a value for the `Mask` property that specifies the mask you want IDEA to use to search for gaps. For example, if your character field consists of four numbers, you'd provide a mask of `NNNN` where each `N` represents a number. You can use `C` to specify a character in place of a number.

Note

The `C` symbol requires further explanation. You can use the `C` symbol in the mask to mark part of the code as being a changeable prefix or suffix. It is not restricted to specifying characters; you can use numeric digits as well. For example, you might add a year prefix to an invoice number to specify a sequence of invoice numbers that IDEA resets every year.

07-00001
08-00001
09-00001

The mask to define the sequential elements of this code while keeping the "yearly" sequences apart would be is `CCCNNNNN`. And of course, there is always an option to ignore a character or digit using `X`—this is useful if the data is manually entered. Someone inputting data could enter the separator character in a number of ways, such as / or \.

After you select the field and provide a mask when necessary, you must choose an output of some type. Use one or both of `OutputDBName` and `ResultName` to specify the output. Finally, call `PerformTask` to tell IDEA to actually accomplish the task. Chapter 17 tells you more about performing this task.

History

You use the `History` task to add custom historical information to the database log. Using this feature can help you record events that occur within applications you create.

To perform this task, you begin by using `NewTask()`, including a task title, to create the task. Next, you use `AppendDatabaseInfo()` to add basic historical information: file name, description, and number of records for the database. At this point, you should date the entry using `DateStamp()`. Finally, you add details using the `AppendText()` method. Chapters 4 and 7 tell you more about performing this task.

ImportAccess

You use the `ImportAccess` task to import data from a Microsoft Access database into IDEA. Each Access table you import appears as a separate database within IDEA. Consequently, if the Access database has five tables in it, IDEA will add five databases to the File Explorer when you import all of the tables. As described in Chapter 7, IDEA relies on a flat-file database management system (DBMS) that contains only one table per database to improve performance and make the databases easier to manage. Unlike many of the tasks in this book, you use a special function, `GetImportTask()` with `Access` as the task argument to create this task.

To perform this task, your application begins by setting the `InputFileName` property to the path of the Access database, including the file extension. The code then adds single tables to the import list using `AddTable()` or adds all of the tables in the Access database using `AddAllTables()`. When adding a single table, you must provide the name of the table. Finally, the code must provide an output file name using either `OutputFileNameFromTableName()` or `OutputFileNamePrefix()`. When working with the `OutputFileNamePrefix()` method, you can ensure each database has a unique name by calling `UniqueFilenamePrefix()`. IDEA adds a number to the end of the prefix when the prefix is already in use to ensure every Access table receives a unique name.

In addition to the required inputs, you can also specify a few optional inputs. The following list describes each of these inputs:

- **CreateRecordNumberField:** Adds a record number field to the IDEA database so that each record from the Access database has a unique number.
- **DetermineMaximumCharacterLengths:** Defines the number of records to scan to determine the character lengths of the fields in the IDEA database. The predefined values are: `SCAN_ALL` (which scans all of the records) and `SCAN_NONE` (which doesn't scan any of the records). Using `SCAN_ALL` can greatly increase the import time, but ensures that all of the character fields are of sufficient length to avoid

truncating the input data. Using SCAN_NONE means that IDEA will use the default lengths for character fields, which almost certainly means that some data truncation will occur, but also ensures that the import proceeds swiftly.

After you configure the import conditions, calling PerformTask() imports the tables from the Access database and places each table in a separate IDEA database. Chapter 14 tells you more about performing this task.

ImportAS400

You use the ImportAS400 task to import data from an IBM AS/400 computer. Unlike many of the tasks in this book, you use a special function, GetImportTask() with "AS400" as the task argument to create this task.

To begin this task, you use InputDATFileName() to define the input data file name and InputFDFFileName() to define the input data file definition file name. You must also define an output file name using the OutputFileName property. Use the Client.UniqueFileName() method to generate a unique file name whenever necessary. At this point, you call PerformTask() to import the data. Chapter 14 tells you more about performing this task.

ImportExcel and ImportExcel2007

You use the ImportExcel task to import older versions of Excel files (those with an .xls extension) and the ImportExcel2007 task to import newer Excel files (those with either an .xlsx or .xlsm extension). Unlike many of the tasks in this book, you use a special function, GetImportTask() with "ImportExcel" or "Excel2007Import" as the task argument to create this task. All Excel files import using the same technique—only the actual task name differs.

Note

You might wonder why IDEA chose to use the special GetImportTask() function. Vendors continue to introduce new file formats and changing the Client object constantly isn't a good idea—having a consistent Client object is mandatory to ensure that your code doesn't break. Consequently, the best way to create a new import feature is to provide a flexible function to perform the task. The GetImportTask() function is flexible in that IDEA can simply introduce new import keywords as needed to handle new, updated, or improved external file formats.

To perform this task, your code begins by defining the FileToImport property. The code then defines SheetToImport. You must import each worksheet separately and each sheet will appear as a separate IDEA database. At this point, the code defines an

output file prefix `OutputFilePath()`. Ensure that each file name is unique by calling `UniqueFilePrefix()`, which adds a numeric value to a prefix when the prefix has already been used for a file name. This task supports the following optional inputs:

- **EmptyNumericFieldAsZero:** Determines whether IDEAScript treats blank numeric fields as 0.
- **FirstRowIsFieldName:** Determines whether IDEAScript uses the first row of the worksheet as field names for the database. You may have to rework the worksheet if it uses some other row for the field names (such as the second row).
- **OutputFilePath:** Outputs the full file name of a database based on the worksheet name.

After you define all of the required inputs, the code calls `PerformTask` to import the Excel worksheet as an IDEA database. Chapter 14 tells you more about performing this task.

ImportXML

You use the `ImportXML` task to import XML files into an IDEA database. IDEA doesn't necessarily let you import just any XML file. The file must be in a specific format, provide an `.xsd` (XML Schema Definition) file, or appear in `RDF` (Resource Description Framework) format. Unlike many of the tasks in this book, you use a special function, `GetImportTask()` with `"ImportXML"` as the task argument to create this task.

To perform this task, you begin by setting the `ImportFileName` property to the path of the file you want to import and the `OutputFileName` property to the path of the resulting database. If you want to ensure that the output file name is unique, call the `UniqueBaseFilename()` method and IDEA will generate a unique name for the file. You then call `PerformTask()` to perform the import. Use the `FullOutputFileName()` method to obtain the full path to the resulting database so you can open it for viewing. This task does support a number of optional inputs that you can set before you perform the task:

- **PutDecimalSeparator:** Sets the decimal separator used within the XML file. The default setting uses the regional settings configured for the machine.
- **PutThousandSeparator:** Sets the thousands separator used within the XML file. The default setting uses the regional settings configured for the machine.
- **XRDFFileName:** Provides the location of an eXtensible Resource Description Framework (XRDF) file that describes the structure of the data in the XML file. You can read more about XRDF at `http://gamma.wi-inf.uni-essen.de/rdf/xrdf/`.

Working with XML files can prove difficult in some cases because XML files have so many potential formats. It pays to check the format of your XML file using a test file to ensure it works with IDEA before you attempt a large import. Chapter 14 tells you more about performing this task.

Index

You use the `Index` task to create a sorted view of a database without creating an output database. The index is actually a view of the database that uses the settings you provide for ordering the information. You can select indexes you create in the Properties window for the selected database.

To perform an index, you begin by using `AddKey()` to create the keys used to order the information. `AddKey()` requires two inputs: the name of the field you want to use for the key and the direction you want the field sorted (`A` for ascending or `D` for descending). After you add all of the keys needed to define the index, you call `Index()` to perform the actual indexing task. `Index()` requires a single input that forces the creation of the index when set to `True`. Otherwise, `Index()` only creates the new index if it doesn't already exist. IDEA never updates indexes—it marks them out of date, so forcing the creation of an index means you're actually marking the old index as outdated and creating a new one.

It's possible to remove existing indexes using the `DeleteIndex()` method. Chapter 7 tells you more about performing this task.

IndexedExtraction

You use the `IndexedExtraction` task to create a new database that contains records that meet the criteria you specify. In addition, IDEA sorts the new database in the order you specify. This task provides the following optional inputs:

- **AndFieldValueIs2:** Specifies the second comparison operation to perform.
- **Criteria:** Defines which records to include in the output database.

To perform this task, the code begins by defining the fields to include in the output database. If you want to include all the fields, call `IncludeAllFields()`. Otherwise, call `AddFieldToInc()` or `AddFieldToIncAt()` to include specific fields. The code must also set the `FieldToUse` property to define which field to use for the comparison and the `FieldValueIs2()` method to define the comparison operation. `FieldValueIs2()` requires three inputs: the comparison operator, the value to use, and the data type of the field. Finally, the code must define the `OutputFileName`, which contains the name of the resulting database. Call `Client.UniqueFileName()` to obtain a unique file name for the output database.

At this point, the code can call `PerformTask()` to create the indexed extraction. You can call `DisplaySetupDialog()` if you want the user to see the extraction arguments that you're using to perform the extraction. Chapters 2 and 4 tell you more about performing this task.

JoinDatabase

You use the `JoinDatabase` task to join two databases together based on a common field, or fields (called a key). Many DBMSs use multiple tables to store information—such

as a customer table and a second table that contains a list of the orders each customer has made. The `JoinDatabase` task could join these two tables together into a single table so that you can see not only the list of customers, but also all of the orders the customers have made. The common field, in this case, would likely be a customer number field.

To perform this task, you must consider the two databases you want to join. Using the example of a list of customers and associated customer orders, the customer table would be the secondary database, while the customer orders table would be the primary database. It's important to keep the two tables separate or you'll encounter problems later. Making the customers table the primary would result in only one record for each customer (and therefore only one order would appear in the output).

As normal, your code begins by opening the primary database and then using the `FileToJoin` property to specify the name of the secondary database. The code then defines the fields to include in the output database from the primary database using `AddPFieldToInc()` or `AddPFieldToIncAt()` for individual fields. You can also call `IncludeAllPFields()` to include all of the fields found in the primary database. The code must also define the fields to use from the secondary database, using `AddSFieldToInc()` or `AddSFieldToIncAt()` for individual fields, or `IncludeAllSFields()` to include all of the fields.

The join relies on matching records in the primary database to those in the secondary database. You include this information using the `AddMatchKey()` method which accepts the name of the field to use in the primary database, the name of the field to use in the secondary database, and the sort order for these fields (`A` for ascending or `D` for descending) as arguments. Use the `AddMatchKey()` method multiple times if the relationship between the two databases involves more than one table. You can optionally use the `Criteria` property to limit the records in the output database based on some specific criteria, such as a range of customer numbers.

Once you've defined all of the required inputs, the code calls `PerformTask()`. In this case, `PerformTask()` accepts three input arguments: the path of the output database, an unused description (set to `" "`), and the kind of join. IDEA supports the joins shown in Table 9.1. Chapter 17 tells you more about performing this task.

KeyValueExtraction

You use the `KeyValueExtraction` task to create multiple output databases that contain unique values based on the field you define as a key. The main advantage of using a `KeyValueExtraction` is that it efficiently splits the records in a database into many subsets. To begin this task, you define which fields to include in the result using either `AddFieldToInc()` or `AddFieldToIncAt()`. If you decide to include all of the fields, simply use `IncludeAllFields()` to add them to the list. You must also use `AddKey()` to define the keys to look for and also sort the output. Optionally, you can determine which records to include using `Criteria`.

Once you decide on fields to include in the extraction, you define a key to use for the extraction and decide how to use that key. The key is used to create an array of values that you want to see in the output databases. The `ValuesToExtract` property receives this array as input. If you want each of these key values to appear in a single database,

TABLE 9.1 Types of Join that IDEA Supports

ID Value	Constant	Description
0	WI_JOIN_MATCH_ONLY	Records that match in both databases only. A customer that has made no orders won't appear in the output; neither will orders that have no known customer associated with them.
1	WI_JOIN_ALL_IN_PRIM	All records in the primary database appear in the output with their associated secondary database records. Orphaned secondary database records won't appear in the output. In short, every customer appears in the output, even if the customer hasn't made an order.
2	WI_JOIN_ALL_REC	All records in both databases appear in the output, even if the secondary record is orphaned.
3	WI_JOIN_NOC_SEC_MATCH	Records with no secondary match appear in the output. You'd use this kind of join to find customers who haven't made an order.
4	WI_JOIN_NOC_PRI_MATCH	Records with no primary match appear in the output. You'd use this kind of join to find orphaned records in the secondary database—such as orders with no customer associated with them (clearly errors in the secondary database).

you set the `CreateMultipleDatabases` to `False`. Set the `DBPrefix` property to ensure each of the output databases begins with the same name to make finding them in the File Explorer easier.

After you define all of the inputs you require, call `PerformTask()` to create the output database. Chapter 14 tells you more about performing this task.

MUSCombinedEvaluate

You use the `MUSCombinedEvaluate` task to evaluate multiple Monetary Unit Samples (MUS). `MUSCombinedEvaluate` creates an evaluation result using two or more MUS databases and, optionally, an MUS high values database. This task begins when you extract the samples you want to analyze from a primary database using the `MUSExtraction` task. Of course, you can use any appropriate task listed in this chapter to create the samples database. After you create the samples database, you use it to perform the `MUSCombinedEvaluate` task.

To perform this task, you open the database you want evaluated using `Client.OpenDatabase`, which requires the name of the database as input. You then open one or more sample databases using `AddSampleToEvaluation()`, which requires the following information:

- **Filename:** Specifies the name of the sample database.
- **BookField:** Defines the name of the book value field.

- **AuditField:** Defines the name of the audit amount field.
- **PopulationValue:** Specifies the value of the sampled population.
- **SampleSize:** Indicates the number of sample items.
- **BPP:** Specifies the Base Precision Pricing.
- **HighValuesFile:** Specifies the name of the high values database.
- **SampleInterval:** Identifies the sampling interval used to select the sample.
- **HighValuesBookField:** Defines the name of the book value field in the high values database.
- **HighValuesAuditField:** Defines the name of the audit amount field in the high values database.

The next step is to set the `ConfidenceLevel` for the samples you've selected for the task. The code also needs to define `ResultName`, which contains the path to the output database. You can use the `UniqueResultName()` method of the `Database` object to obtain the output file name. Finally, the code calls `PerformTask()`. Chapter 17 tells you more about performing this task.

MUSEvaluate

You use the `MUSEvaluate` task to evaluate a single MUS sample. `MUSEvaluate` creates an `Evaluation` result using a single MUS sample database and, optionally, an MUS high values database. This task requires that you provide the following information:

- **AuditAmountField:** Defines the name of the audit amount field.
- **BasicPrecisionPricing:** Specifies the Basic Precision Pricing level.
- **BookField:** Defines the name of the book value amount field.
- **HighValueAuditAmountField:** Defines the name of the high value audit amount field.
- **HighValueBookField:** Defines the name of the high value book value amount field.
- **HighValueFileName:** Specifies the name of the high values database.
- **HighValueHandling:** Determines the method for handling high value items. You have a choice of `WI_HighValueHandling_AGGREGATE`, which includes high value items in records to be sampled, or `WI_HighValueHandling_FILE`, which extracts high value items to a separate database.
- **HighValueReferenceField:** Defines the name of the high value reference field.
- **Method:** Determines the method for evaluating the sample. You have a choice of `WI_MUS_HIGH_ERROR_RATE`, which uses the method appropriate for high-error rate samples, or `WI_MUS_LOW_ERROR_RATE`, which uses the method appropriate for low-error rate samples.
- **PopulationValue:** Specifies the value of the sampled population.
- **PrecisionLimits:** Specifies the type of precision limits used.
- **ReferenceField:** Defines the name of the reference field.

Once you provide this information, you set the `ConfidenceLevel` for the sample you've selected for the task. The code also needs to define `ResultName`, which

contains the path to the output database. You can use the `UniqueResultName()` method of the `Database` object to obtain the output file name. Finally, the code calls `PerformTask()`. Chapter 17 tells you more about performing this task.

MUSExtraction

You use the `MUSExtraction` task to extract MUS samples. To perform this task, you open the database you want evaluated using `Client.OpenDatabase`, which requires the name of the database as input. The code then defines the fields to include in the output database. If you want to include all the fields, call `IncludeAllFields()`. Otherwise, call `AddFieldToInc()` or `AddFieldToIncAt()` to include specific fields. At this point, you must provide a number of input values for the task:

- **ChangeHighValueAmount:** (Optional) Specifies a high value amount other than the sample interval.
- **FieldToSample:** Defines the field to sample.
- **HighValueFilename:** Specifies the name of the high values database.
- **HighValueHandling:** Specifies the method for handling high value items. You have a choice of `WI_HighValueHandling_AGGREGATE`, which includes high value items in records to be sampled, or `WI_HighValueHandling_FILE`, which extracts high value items to a separate database.
- **RandomValue:** Determines the numeric seed value.
- **RangeOfValues:** Sets the range of values to sample. You have a choice of: `WI_RangeOfValues_POSITIVE`, which is positive values only, `WI_RangeOfValues_NEGATIVE`, which is negative values only, or `WI_RangeOfValues_ABSOLUTE`, which is all records based on the absolute value of the field to sample.
- **SampleInterval:** Specifies the numeric sample interval.
- **TaskType:** Specifies the monetary unit selection method type. You have a choice of `WI_TaskType_FIXED`, which is a fixed interval, or `WI_TaskType_CELL`, which is a cell selection interval.

Once you provide this information, you set `MUSExtractionFilename`, which contains the path to the output database. You can use the `UniqueResultName()` method of the `Database` object to obtain the output file name. Finally, the code calls `PerformTask()`. Chapter 17 tells you more about performing this task.

PivotTable

You use the `PivotTable` task to display specified data using a grid format. Pivot tables are helpful because they can help you summarize information or gain a new view of the data without becoming overwhelmed with detail. To begin this task, you define these inputs:

- **AddColumnField:** Specifies a field to put in the pivot table column.
- **AddDataField:** Specifies a field to put in the data area. The `DataField()` method requires that you provide the name of the field, the display name for the field, and the operation you want to perform. The valid operations include: sum (`1`), average (`2`), count (`3`), min (`4`), and max (`5`).

- **AddPageField:** (Optional) Specifies a field to put in the pivot table page to create a separate table (page) for each value of the field.
- **AddRowField:** Specifies a field to put into the pivot table row.

Once you provide all of the required inputs, you define `ResultName` for the pivot table and call `PerformTask()`. IDEA displays the resulting pivot table on screen. Chapter 17 tells you more about performing this task.

RandomSample

You use the `RandomSample` task to randomly sample records from a database. IDEA places the selected record in a new database. Remember that a random sample on a computer isn't going to be completely random—it will follow a pattern that varies according to the algorithm used and the seed value chosen. However, the results are random enough that no one except the computer will notice.

To perform this task, the code begins by defining the fields to include in the output database. If you want to include all the fields, call `IncludeAllFields()`. Otherwise, call `AddFieldToInc()` or `AddFieldToIncAt()` to include specific fields. At this point, the code calls `PerformTask()`, which requires a number of arguments in this case:

- **DatabaseName:** Specifies the output database name.
- **Desc:** No longer used. Set this parameter to `" "`.
- **NumRecs:** Defines the number of records to choose randomly.
- **StartRec:** Determines the starting record for the sample selection.
- **EndRec:** Determines the ending record for the sample selection.
- **Seed:** Specifies the number used to create the random output. To ensure maximum randomness, you want to set this value to a value that changes constantly, such as `Second(Now)`. However, if you need to extend a sample, using the same seed value will ensure that the second sample includes those records already selected plus additional unique records.
- **DupRecs:** Determines whether IDEA performs the sampling with replacement. Setting this value to `True` means that IDEA can select any record more than one time and place it in the output database. Set this property to `False` when you want the sampling performed without replacement, such that when a record is selected, it's taken out and can't be selected again.

Chapter 17 tells you more about performing this task.

Sort

You use the `Sort` task to actually sort the content of a database and save the result to a new database. Sorting usually requires time to perform when you initially perform the task, but it can make your application run faster in the long run as long as you choose your sorting key carefully. Use sorting when you intend to perform a number of tasks on a database using the same key.

To perform this task, use `AddKey()` to add sorting keys to the task. `AddKey()` requires that you provide both the field name and the direction you want to sort that field. After you select all of the keys you want to use, call `PerformTask()` with the name of the output database to obtain the sorted results. Chapter 7 tells you more about performing this task.

Stratification

You use the `Stratification` task to place the data in layers or strata by a date, numeric, or character field. `Stratification` lets you look at the distribution of related data with greater ease. What you see in the output is the number of records that fall between the limits you set and the total of a numeric field for each layer. In addition, you can click a link to see specific records within a layer.

To perform this task, the code begins by defining the fields to include in the output database. If you want to include all the fields, call `IncludeAllFields()`. Otherwise, call `AddFieldToInc()` or `AddFieldToIncAt()` to include specific fields.

After you decide on which fields to display, you must define `FieldToStratify`. In addition, in order to see an output, you must select one or more fields to total using `AddFieldToTotal()`. As a minimum, you must also define several limits for the stratification field using `AddUpperLimit()` (use `AddUpperCharLimit()` when working with a character field). You can also provide these optional inputs:

- **Criteria:** Defines the criteria used to select records for processing.
- **CutOffs:** Specifies cutoffs to filter group records to be stratified.
- **GroupBy:** Defines a key field used to group the stratification result. This will create a stratification result for each unique value in the selected field.
- **OutputDBName:** Specifies the name of the optional database created when running the `Stratification` task. This will be a copy of the original database but with an additional field identifying which layer each record falls into.
- **IncludeInterval:** Includes upper and lower limit columns in the stratification database when set to `True`.
- **LowerCharLimit:** Defines the lowest value that you want to use to start defining your character intervals.
- **LowerLimit:** Defines the lowest value that you use to start defining your numeric intervals. The default is `0`.

At this point, you give the output database a name by setting the `OutputDBName` property. You can also save the result under a specific name by setting the `ResultName` property. Calling `PerformTask()` creates the output. Chapter 17 tells you more about performing this task.

StratifyRndSample

You use the `StratifyRndSample` task to stratify the database and then select a random sample from the result. To begin this task, you perform a stratification of the database you want to use.

The `StratifyRndSample` code begins by defining the fields to include in the output database. If you want to include all the fields, call `IncludeAllFields()`. Otherwise, call `AddFieldToInc()` or `AddFieldToIncAt()` to include specific fields.

At this point, you use `StratifyOnBand()` to define the interval and the number of samples. The interval number should be a stratum number that resulted from the previously performed stratification. The number of samples shouldn't exceed the number of records for each interval. The last input is `ResultName`, which contains the name of the result as it appears in the Properties window. When you call `PerformTask()`, you must provide the following arguments:

- **DatabaseName:** Defines the name of the output database.
- **Desc:** Not used. Set this parameter to `" "`.
- **StratumField:** Specifies the name of the stratified field.
- **Seed:** Specifies the number used to create the random output. To ensure maximum randomness, you want to set this value to a value that changes constantly, such as `Second(Now)`.

Chapter 17 tells you more about performing this task.

Summarization

You use the `Summarization` task to group records based on the fields you specify. It's then possible to create a result with certain statistics, including sum, minimum, average, or maximum value for a specified field.

To perform this task, the code begins by defining the fields to include in the optional output database. If you want to include all the fields, call `IncludeAllFields()`. Otherwise, call `AddFieldToInc()` or `AddFieldToIncAt()` to include specific fields.

Note

Because a summarization collates records together, usually condensing many records down to one, it is normal not to include any fields in the database output. IDEA automatically includes the fields selected for grouping and for statistics. When you select additional fields, IDEA carries the value from only one record carried to the output. This could be the first in a group or the last depending on user preference.

The next step is to define the summarization field(s) using `AddFieldToSummarize()`. Every unique combination of values for the specified keys form a grouping. In addition, you can optionally tell IDEA which field to total using `AddFieldToTotal()`. This task also lets you specify a number of optional inputs including:

- **CreatePercentField:** Creates a percent field when set to `True`.
- **Criteria:** Defines the criteria used to select records for processing.

- **StatisticsToInclude:** Specifies the statistics to add to the output. Available statistics are count (`SM_COUNT`), sum (`SM_SUM`), maximum (`SM_MAX`), minimum (`SM_MIN`), variance (`SM_VARIANCE`), average (`SM_AVERAGE`), and standard deviation (`SM_STD_DEV`).
- **UseFieldFromFirstOccurence:** Uses the value from "included fields" from the first occurrence within a group when set to `True` and from the last occurrence when set to `False`.

After you define the required input arguments, you must provide an output for the result. IDEAScript accepts either (or both) a database name using `OutputDBName()` and a result name using `ResultName`. At this point, you can call `PerformTask` to produce the output. Chapter 17 tells you more about performing this task.

SystematicSample

You use the `SystematicSample` task to obtain a sample of a database that's distributed evenly over a population range. To perform this task, the code begins by defining the fields to include in the output database. If you want to include all the fields, call `IncludeAllFields()`. Otherwise, call `AddFieldToInc()` or `AddFieldToIncAt()` to include specific fields.

Once you define the fields you want in the output, you call `PerformTask()`. In this case, `PerformTask()` requires the following arguments:

- **DatabaseName:** Defines the name of the output database.
- **Desc:** Not used. Set this parameter to `" "`.
- **NumRecs:** Defines the number of records to sample.
- **StartRec:** Specifies the starting (beginning) record number for the sample.
- **EndRec:** Specifies the last (ending) record number for the sample.
- **Interval:** Specifies the interval between records.

Chapter 17 tells you more about performing this task.

TableManagement

The `TableManagement` task isn't actually a single task. It's a means of managing your table—sort of like a control panel. You can use `TableManagement` to perform any of these three management tasks:

- **AppendField:** Appends a field to the database.
- **RemoveField:** Removes a field from the database by name.
- **ReplaceField:** Replaces a field in the database by name.

As usual, you call `PerformTask()` to make the actual changes to the database. Chapter 8 tells you more about performing this task.

TopRecordsExtraction

You use the `TopRecordsExtraction` task to extract the specified number of records within a unique key (or key combination) to another database. In many cases, you use this kind of extraction to view a sample of the database for testing or other purposes.

To perform this task, the code begins by defining the fields to include in the output database. If you want to include all the fields, call `IncludeAllFields()`. Otherwise, call `AddFieldToInc()` or `AddFieldToIncAt()` to include specific fields. You must also use `AddKey()` to define the keys to look for and also sort the output. Optionally, you can determine which records to include using `Criteria`.

Once you create the basic inputs, you use the `NumberOfRecordsToExtract` property to define the number of records you want to see in the output. You must also define `OutputFileName`, which is the location of the extracted data. The final step is to call `PerformTask()`. Chapter 14 tells you more about performing this task.

TrendAnalysis

You use the `TrendAnalysis` task to reveal patterns in past data and to predict future values based on the past data. For example, you might predict the future value of a stock based on its past performance.

To perform this task, the code begins by defining the `TrendField`, which must contain numeric values. Additionally, this particular task provides a host of optional inputs you can provide to modify the output as described in the following list:

- **AddAuditUnit:** Adds an audit field value to include in the analysis task. If this method is not called, the analysis will be performed for all audit field values.
- **AuditUnitField:** Groups the records in the database by the selected field for the analysis task. If this method is not called, no grouping of the records will occur.
- **CalendarValue:** Sets the calendar interval type for the time scale. You may specify any of these values as interval types: calendar day interval (`0`), calendar month interval (`1`), four-week month interval (`2`), quarterly interval (`3`), or calendar year interval (`4`).
- **ClockValue:** Sets the clock interval type for the time scale. You may specify any of these values as clock intervals: hour (`0`), minute (`1`), or second (`2`).
- **CreateAnyDB:** Determines whether IDEA creates any output databases (standard output, forecast, or MAPE). Set to `False` if you don't want to create any output databases.
- **CreateDB:** Creates an output database when set to `True`.
- **CreateForecastDB:** Creates a forecast database when set to `True`.
- **CreateMAPEDB:** Creates a MAPE (Mean Absolute Percentage Error) database when set to `True`.
- **GenerateForecasts:** Enables or disables the forecasting option.
- **RefField:** Creates a reference graph for the analysis.

- **TimeScale:** Sets the type of time scale used in the analysis. You may specify any of these values for the time scale: index (`0`), calendar (`1`), or clock (`2`).
- **TimeScaleStartAndIncrement:** Sets the starting position and the increment value of the time scale used in the analysis.

After you define the required input arguments, you must provide an output for the result. IDEAScript accepts either (or both) a database name using `OutputDBName()` and a result name using `ResultName`. If desired, you can use `UniqueFileName()` to create a unique file name for the database. At this point, you can call `PerformTask` to produce the output. Chapter 18 tells you more about performing this task.

VisualConnector

You use the `VisualConnector` task to create a single database from two or more other databases. To perform this task, the code begins by defining the databases you want to use by calling `AddDatabase()` with the database path. One of these databases must be the primary database and you pass its object to IDEAScript using the `MasterDatabase` property.

After you define the databases, you define the fields to include in the output database. If you want to include all the fields, call `IncludeAllFields()`. Otherwise, call `AddFieldToInclude()` to include specific fields. The `AddFieldToInclude()` method has an extra argument that defines which database field to draw from as described here:

- **id0:** The first database selected (the primary or master database)
- **id1:** The second database selected
- **id2:** The third database selected
- **id*X*:** Where *X* is a number that defines the database selected

You can also define these optional inputs:

- **AddRelation:** Creates a relationship between two databases. Call this method for each relationship you want to create. This input requires that you provide the source database identifier, source database field, target database identifier, and target database field.
- **AppendDatabaseNames:** Appends the name of the database from which the field originated to the end of each field name in the output database. Set to `True` to append database names to field names. Otherwise, set to `False`.
- **IncludeAllPrimaryRecords:** Choose between All Records in Primary or Matches Only. Set to `True` to include all records in the primary database. Set to `False` for matches only.

Once you finish defining the inputs, you specify the output database name using `OutputDatabaseName`. Call `PerformTask()` to complete the process. Chapter 17 tells you more about performing this task.

Summary

This chapter has helped you understand the IDEAScript object model, which is an essential skill in working with IDEA databases. Of course, no one is expecting you to memorize this material. As you write more applications, you become familiar with the material, but you should also expect to revisit this chapter as needed to learn more about the object model. Even after many years of working with a product, most developers need to go back to their references when working on something new and this is a reference chapter.

One of the techniques that many good developers use to expand their knowledge and ability is to make notes and learn something new each day. To be effective, such learning has to be targeted. As you go through this chapter again, look for new objects that you think you might need for upcoming projects. Take one object per day and write a short application using it. Try the object out—play with it. See how changing various aspects of the object modifies the output of your application. Play time is an essential component of learning any macro language.

Chapter 10 takes a step back from intense database management tasks and looks at the math you use to work with databases. Of course, you can use math in a lot of ways that don't involve working with databases. Chapter 10 helps you understand some of the more complex math features provided with IDEAScript, such as the use of trigonometric functions. It also builds on what you've learned previously by helping you understand the techniques for manipulating database data.

CHAPTER 10

Performing Mathematical Tasks

A lot of the analysis performed on databases requires some type of math. Even theoretically simple tasks can require some use of math. In some cases, you might not even think about the math connection because it hides in the background. The math might occur because you've called on one of the IDEAScript functions, used a task object, or relied on a calculated field. In short, you need to know something about the math functionality that IDEAScript provides as part of working with databases.

Many applications never do anything more than simple math. The basic math functions of add, subtract, multiply, and divide can perform powerful and interesting tasks when combined. Equations that would drive the average human crazy are quite simple for the computer when properly defined. These equations often consist of nothing more than simple math performed repetitively to produce a specific result. Most monetary calculations are nothing more than simple math applied in a specific manner.

Sometimes you do need to perform complex math. For example, when working with area, you might need to use trigonometric functions. A special function helps you obtain the square root of a number, which is part of many math calculations. IDEAScript provides a number of advanced math functions that make it easier to perform tasks that would ordinarily require some very fancy coding. Imagine trying to create a successive approximation routine for finding the square root of a number! It's better to let IDEAScript perform these complex tasks for you.

In some cases, you need to perform certain kinds of mathematical analysis. For example, statistical analysis sometimes requires use of random sampling, something that IDEAScript helps you perform with little effort. The final section of this chapter discusses math used as an analysis tool. You discover how to use functionality, such as randomization, to perform certain tasks with IDEAScript that would be hard to perform manually.

Performing Basic Math

As the name implies, basic math is the math you learned in grade school. It includes addition, subtraction, multiplication, and division. Using these four kinds of math can help you perform all kinds of tasks on a computer because a computer doesn't have the problems with numbers that the average human does. You can tell the computer to perform some tasks repetitively in order to achieve a given result. For example, determining

n! (n-factorial) can be time consuming and error prone, but the computer performs the task with ease and complete accuracy.

Of course, before you begin writing complex math equations in your applications, you have to understand a little more about basic math in a computer. Unlike you, computers make a differentiation between integers (those without a decimal portion) and real numbers (those with a decimal portion). Table 5.1 describes all of the numeric types that IDEAScript supports and provides you with some information about their ranges. However, it's time to look at the numbers in another way.

Table 5.4 shows you the operators that IDEAScript supports and tells you about their use. This table should give you an idea that basic math on a computer can have slightly different rules than the rules that humans rely upon. Of course, some things are the same. For example, 5 + 2 still produces 7. However, try out this simple program.

```
Sub Main
   Dim FirstNum As Integer
   FirstNum = 5

   Dim SecondNum As Integer
   SecondNum = 2

   Dim Result As Integer
   Result = FirstNum / SecondNum

   MsgBox "5 divided by 2 equals " & Result
End Sub
```

The output of this application is 3, which isn't correct from a human perspective. What you really should have seen is 2.5 as the result. At the very least, you expected to see 2 as the output because that's the integer portion of the result. As it turns out, the result is precisely correct. The computer sees that Result can only hold an integer, so it rounds the output up to the next number because the decimal portion of the result is equal to or greater than 0.5. The following code demonstrates the correct method of obtaining a result you can accept as correct.

```
Sub Main
   Dim FirstNum As Integer
   FirstNum = 5

   Dim SecondNum As Integer
   SecondNum = 2

   Dim Result As Integer
   Result = Int(FirstNum / SecondNum)

   Dim Remainder As Integer
   Remainder = FirstNum Mod SecondNum

   MsgBox "5 divided by 2 equals " & Result _
      & " with a remainder of " & Remainder
End Sub
```

In this case, the computer spits out an answer that's consistent with what you'd expect from a child in grade school, "5 divided by 2 equals 2 with a remainder of 1." The answer is correct this time, but you had to use an unconventional method to obtain it. The `Int()` function tells IDEA to provide just the integer portion of the answer without any rounding. Of course, there's a remainder. To obtain the remainder, you use the `Mod` operator as shown in the example.

Often, basic math requires use of some tricky programming. You know you need to perform a task a certain number of times, but you don't know what values to use or what the end value will be. For example, computing a factorial (`n!`) requires recursion, a tricky kind of programming where a function calls itself until it achieves a specific value. Recursion is such a useful technique that everyone who writes code—even someone who writes macros—should know about it. Listing 10.1 shows how to use recursion to compute `n!`.

LISTING 10.1 Computing the Value of n!

```
Sub Main
   ' Obtain the value of n that the user wants.
   Dim n As Integer
   n = InputBox("Type the value of n that you want to compute: ")

   ' Obtain the n! result.
   Dim Result As Integer
   Result = 1
   Factorial n, Result

   ' Display the result.
   MsgBox "The value of n: " & n & _
      " produces a factorial, n!, of: " & Result
End Sub

Sub Factorial(ByVal n As Integer, ByRef Result As Integer)
   ' When using recursion, you must provide a means
   ' of exiting the recursive loop. In this case, the loop
   ' exits when the recursive loop reaches 1.
   If n = 1 Then
      Exit Sub
   Else
      ' Multiply Result by n during each loop.
      Result = Result * n

      ' Subtract 1 to advance to the next loop.
      n = n - 1

      ' Call the Sub recursively.
      Factorial n, Result
   End If
End Sub
```

The `Main()` subroutine displays an input box where the user inputs a value of `n` to calculate `n!`. `Main()` then calls `Factorial()` and displays the result on screen.

`Factorial()` uses several new techniques in addition to recursion to obtain the desired result. First, notice that this is a subroutine and not a function. You could use either approach, but using a subroutine is often easier because you pass the calculated value with each loop. It may interest you to know that you could also calculate `n!` using a simple loop—this example is simply a means to show how recursion works.

Second, notice that `n` is passed by value, `ByVal`, and `Result` is passed by reference, `ByRef`. (The default is to pass arguments by reference, but it helps document your code to use `ByVal` and `ByRef` when there could be a question from anyone viewing it.) What's that all about? It turns out that when you pass something by value, the subroutine or function can't modify the passed value directly. In other words, when `Factorial()` is finished doing its work, the original value of `n` in `Main()` remains unchanged. However, using `ByRef` means that the subroutine or function can modify the passed value directly. In this case, we do want `Result` to hold the value of `n!` after the recursive loop is finished, so you must pass `Result` by reference.

Third, unlike many other subroutines, `Factorial()` provides a specific escape mechanism. If you don't provide this mechanism, the loop will continue until the machine runs out of memory or you manually stop the application. In this case, every call to `Factorial()` reduces the value of `n` by `1` until `n` reaches `1`, at which point, the loop exits.

Calculating the value of `n!` comes next. The value is calculated using the number you provide as a starting point and multiplying by each previous value. For example, if you provide `4` as an input, then `n!` is `4 * 3 * 2 * 1`. After the loop calculates the current value of `n!`, it calls `Factorial` with the new values. This loop continues until `n` reaches `1`.

Warning

Recursive functions are an elegant way to calculate some values. However, a recursive function can cause an overflow without too much trouble. For example, if you use a value larger than 7 for the example in this section, the example will overflow. You could extend this example by using Long, rather than Integer arguments, but the example will still overflow when the number you calculate exceeds the storage capacity of the variable. Always be sure to test your application for potential overflow problems. After testing, set up error checking code to prevent the user from entering values that are too high.

Using Advanced Math

Advanced math refers to the kinds of tasks that normally require something more than the four common operators (+, −, /, and *). For example, IDEA provides access to a full set of trigonometric functions. You may not use advanced math functionality every day,

TABLE 10.1 Advanced Math Functions in IDEAScript

Function	Description	Example
Abs	Returns the absolute value of the input value.	Abs(−1.1) = 1.1
Atn	Returns the arctangent of the specified angle in radians. The input value of 0.7853975 radians equates to 45 degrees.	Atn(0.7853975) = 0.6658
Cos	Returns the cosine of the specified angle in radians.	Cos(0.7853975) = 0.7071
Exp	Returns the natural log, e, raised to the power of the input number (e ^ <number>). The approximate value of e is 2.718281828.	Exp(5) = 148.4132
Log	Returns the natural log of a number.	Log(148.4132) = 5
Randomize	Sets the seed value for generating random numbers using the `Rnd()` function and making the `Rnd()` function output less predictable.	This function doesn't generate output.
Rnd	Returns a random number in the range from 0 to 1. The number is actually pseudorandom because a truly random number isn't possible using computers without special hardware. If you truly need random numbers, you can obtain them at sites such as `http://www.random.org/`.	Rnd() = 0 < <Output> < 1
Sin	Returns the sine of the specified angle in radians.	Sin(0.7853975) = 0.7071
Sqr	Returns the square root of the input number.	Sqr(4) = 2
Tan	Returns the tangent of the specified angle in radians.	Tan(0.7853975) = 1.0

but it's good to know that it exists when you do need it. The following sections consider advanced math functionality in IDEA.

Considering the Advanced Math Functions

IDEAScript provides access to a number of advanced functions. You use these functions just as you do any other function in IDEAScript. Table 10.1 provides a list of these functions and shows short examples of their usage.

You may have noticed that the trigonometric functions require you to input the value in radians. Most people are used to working in degrees. Fortunately, you can use the following function to convert values from degrees to radians for use in your IDEAScript equations.

```
Function ConvertDegrees2Radians(Degrees As Double) As Double
   ' Create a constant for pi.
   Const pi = 3.14159265358979323846264338327795

   ' Perform the conversion.
   ConvertDegrees2Radians = Degrees * (pi / 180)
End Function
```

Using Advanced Math in an Application

Precisely how you use the functions shown in Table 10.1 depends on the application you want to create. For example, you might need to determine the power factor for an industrial plant as part of an energy audit. Here's a simple application that shows the calculation of the standard trigonometric values for 45 degrees.

```
Sub Main
   ' Create a handy variable for adding lines to the output.
   Dim vbCrLf As String
   vbCrLf = Chr(13)  + Chr(10)

   ' Select a random number of degrees.
   Randomize(Second(Now))
   Dim Degrees As Double
   Degrees = Rnd() * 360

   ' Obtain the value of the random degree value as radians.
   Dim Rads As Double
   Rads = ConvertDegrees2Radians(Degrees)

   ' Display the result.
   MsgBox "Using " & Degrees & " degrees as input:" & vbCrLf & _
      "Cosine: " & Cos(Rads) & vbCrLf & _
      "Sine: " & Sin(Rads) & vbCrLf & _
      "Arctangent: " & Atn(Rads) & vbCrLf & _
      "Tangent: " & Tan(Rads)
End Sub
```

This example begins by setting the random number generator seed to the current number of seconds. Using the `Second()` function to create the seconds value like this is a common programmer trick to make a pseudorandom number generator act in a more random manner. However, the numbers still aren't truly random, but they're sufficiently random for all but the most critical uses. The code then places a random number of degrees in `Degrees`—from `0` to `360`.

The next step is to convert the degrees into radians using the `ConvertDegrees2Radians()` function found in the previous section of the chapter. At this point, the code can use the various trigonometric functions to output information about the random degree value. Figure 10.1 shows a typical result; however, your output will vary according to the random number.

Employing Analysis

IDEAScript provides a couple of tasks that definitely fall into the mathematical analysis category. These tasks help you work with databases to look for data patterns and to see the effect of time on the data. The following sections describe two more of the tasks you can perform using IDEAScript.

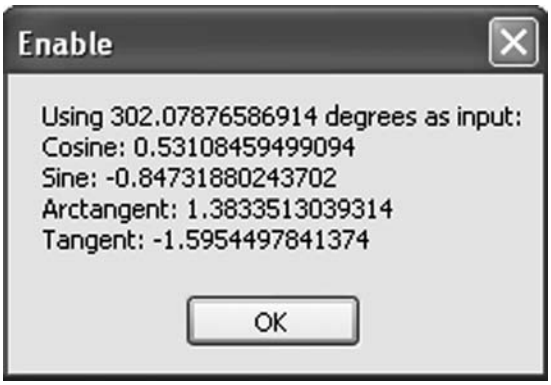


FIGURE 10.1 Computing the Trigonometric Values for a Random Number of Degrees

Performing Aging

Aging helps you review data from a historical perspective. The `Aging` task helps you break a database into six time frames that you can use to locate data by time. For example, you might find that some payments aren't being made within the required time interval. Listing 10.2 shows the `Aging` task in action.

The example begins by opening the `Sample-Bank Transactions` database. The database you choose for aging must have at least one date field and one numeric field. Consequently, the `Sample-Customers` database won't work because it only includes the numeric field.

Next, the code sets the task and then begins configuring the task. In this case, the code begins by defining the end date of the analysis, the field used for aging purposes (must be a date field), and the field used for value information (must be a numeric field). The code also selects an aging interval, which is days for this example, and the six aging intervals. The transactions in the `Sample-Bank Transactions` database take place roughly over the course of a year, so the intervals are divided approximately equally into that time frame.

The `Aging` task supports a number of outputs. The example shows both the summary and the detailed output. The summary output is simply an overview of which entries fit within a particular time frame as shown in Figure 10.2. The detailed output shown in Figure 10.3 includes additional information such as the actual number of aged days for a particular record (AGED_DAYS field) and the interval into which the record fits (AGED_INT field).

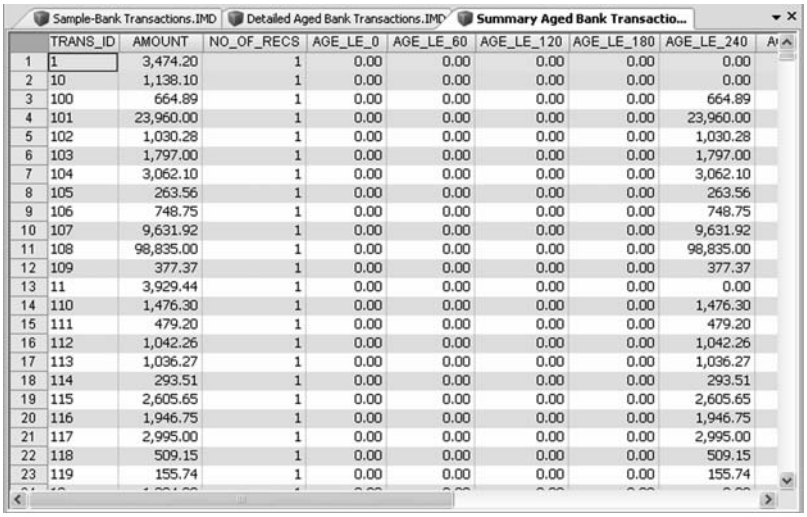
Tip

The AGED_INT field is particularly helpful when looking for a pattern. For example, you might find that most transactions take place in the 300 day interval.

LISTING 10.2 Performing Aging on a Database

```
Sub Main
   ' Use the current database as a starting point.
   Dim db As Database
   Set db = Client.OpenDatabase("Sample-Bank Transactions.imd")

   ' Create the task.
   Dim AgeIt As Task
   Set AgeIt = db.Aging

   ' Set the end data, aging date, and total values.
   AgeIt.Info "2008/11/07", "DATE", "AMOUNT"

   ' Define the interval to use: days = 0, months = 1, or years = 2.
   AgeIt.IntervalTypeIndex = 0

   ' Define the six aging intervals used for the analysis.
   AgeIt.Intervals 60, 120, 180, 240, 300, 360

   ' Set the detailed aging database name.
   AgeIt.CreateAgeDB "Detailed Aged Bank Transactions.imd", ""

   ' Set the summary aging database name.
   AgeIt.CreateSummaryDB "Summary Aged Bank Transactions.imd", ""

   ' Include all of the fields in the database output.
   AgeIt.IncludeAllFields

   ' Add a key for sorting the output.
   AgeIt.AddKey "TRANS_ID", "A"

   ' Perform the task.
   AgeIt.PerformTask

   ' Open the result databases.
   Client.OpenDatabase "Detailed Aged Bank Transactions.imd"
   Client.OpenDatabase "Summary Aged Bank Transactions.imd"

   ' Clear memory
   Set db = Nothing
   Set AgeIt = Nothing
End Sub
```

The next step is to tell the `Aging` task which fields to include. You can use `IncludeAllFields()` to include all of the fields or add individual fields with the `AddFieldToInc()` or `AddFieldToIncAt()` methods. The example also sorts the data by the TRANS_ID field using `AddKey()`. Finally, the example performs the task and opens the databases for viewing.



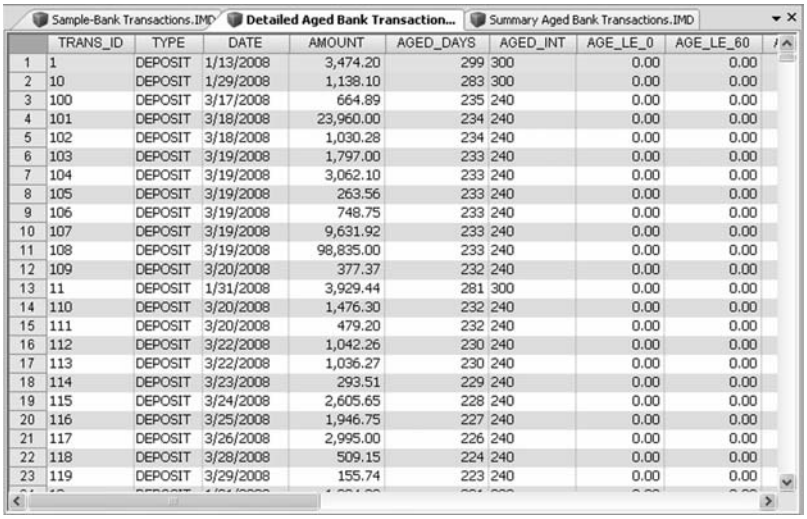
Sample-Bank Transactions.IMD | Detailed Aged Bank Transactions.IMD | **Summary Aged Bank Transactio...**

	TRANS_ID	AMOUNT	NO_OF_RECS	AGE_LE_0	AGE_LE_60	AGE_LE_120	AGE_LE_180	AGE_LE_240
1	1	3,474.20	1	0.00	0.00	0.00	0.00	0.00
2	10	1,138.10	1	0.00	0.00	0.00	0.00	0.00
3	100	664.89	1	0.00	0.00	0.00	0.00	664.89
4	101	23,960.00	1	0.00	0.00	0.00	0.00	23,960.00
5	102	1,030.28	1	0.00	0.00	0.00	0.00	1,030.28
6	103	1,797.00	1	0.00	0.00	0.00	0.00	1,797.00
7	104	3,062.10	1	0.00	0.00	0.00	0.00	3,062.10
8	105	263.56	1	0.00	0.00	0.00	0.00	263.56
9	106	748.75	1	0.00	0.00	0.00	0.00	748.75
10	107	9,631.92	1	0.00	0.00	0.00	0.00	9,631.92
11	108	98,835.00	1	0.00	0.00	0.00	0.00	98,835.00
12	109	377.37	1	0.00	0.00	0.00	0.00	377.37
13	11	3,929.44	1	0.00	0.00	0.00	0.00	0.00
14	110	1,476.30	1	0.00	0.00	0.00	0.00	1,476.30
15	111	479.20	1	0.00	0.00	0.00	0.00	479.20
16	112	1,042.26	1	0.00	0.00	0.00	0.00	1,042.26
17	113	1,036.27	1	0.00	0.00	0.00	0.00	1,036.27
18	114	293.51	1	0.00	0.00	0.00	0.00	293.51
19	115	2,605.65	1	0.00	0.00	0.00	0.00	2,605.65
20	116	1,946.75	1	0.00	0.00	0.00	0.00	1,946.75
21	117	2,995.00	1	0.00	0.00	0.00	0.00	2,995.00
22	118	509.15	1	0.00	0.00	0.00	0.00	509.15
23	119	155.74	1	0.00	0.00	0.00	0.00	155.74

FIGURE 10.2 The Summary Output Shows a Basic Overview of the Aging Data

Using BenfordsLaw

`BenfordsLaw` basically says that data will follow a predictable pattern in many cases. It relies on a logarithmic scale where the number 1 will appear as the first digit almost one third of the time, while 9 only appears as the first digit one time in twenty. When the digit sequence or digit dataset falls outside predictable patterns, the database could contain



Sample-Bank Transactions.IMD | **Detailed Aged Bank Transaction...** | Summary Aged Bank Transactions.IMD

	TRANS_ID	TYPE	DATE	AMOUNT	AGED_DAYS	AGED_INT	AGE_LE_0	AGE_LE_60
1	1	DEPOSIT	1/13/2008	3,474.20	299	300	0.00	0.00
2	10	DEPOSIT	1/29/2008	1,138.10	283	300	0.00	0.00
3	100	DEPOSIT	3/17/2008	664.89	235	240	0.00	0.00
4	101	DEPOSIT	3/18/2008	23,960.00	234	240	0.00	0.00
5	102	DEPOSIT	3/18/2008	1,030.28	234	240	0.00	0.00
6	103	DEPOSIT	3/19/2008	1,797.00	233	240	0.00	0.00
7	104	DEPOSIT	3/19/2008	3,062.10	233	240	0.00	0.00
8	105	DEPOSIT	3/19/2008	263.56	233	240	0.00	0.00
9	106	DEPOSIT	3/19/2008	748.75	233	240	0.00	0.00
10	107	DEPOSIT	3/19/2008	9,631.92	233	240	0.00	0.00
11	108	DEPOSIT	3/19/2008	98,835.00	233	240	0.00	0.00
12	109	DEPOSIT	3/20/2008	377.37	232	240	0.00	0.00
13	11	DEPOSIT	1/31/2008	3,929.44	281	300	0.00	0.00
14	110	DEPOSIT	3/20/2008	1,476.30	232	240	0.00	0.00
15	111	DEPOSIT	3/20/2008	479.20	232	240	0.00	0.00
16	112	DEPOSIT	3/22/2008	1,042.26	230	240	0.00	0.00
17	113	DEPOSIT	3/22/2008	1,036.27	230	240	0.00	0.00
18	114	DEPOSIT	3/23/2008	293.51	229	240	0.00	0.00
19	115	DEPOSIT	3/24/2008	2,605.65	228	240	0.00	0.00
20	116	DEPOSIT	3/25/2008	1,946.75	227	240	0.00	0.00
21	117	DEPOSIT	3/26/2008	2,995.00	226	240	0.00	0.00
22	118	DEPOSIT	3/28/2008	509.15	224	240	0.00	0.00
23	119	DEPOSIT	3/29/2008	155.74	223	240	0.00	0.00

FIGURE 10.3 The Detailed Output Shows Additional Information

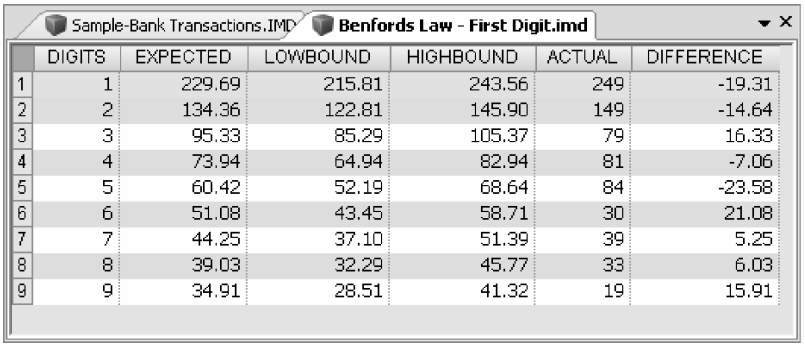
LISTING 10.3 Using BenfordsLaw to Analyze Data

```
Sub Main
   ' Use the current database as a starting point.
   Dim db As Database
   Set db = Client.OpenDatabase("Sample-Bank Transactions.imd")

   ' Set the task.
   Dim BL As Task
   Set BL = db.BenfordsLaw

   ' Choose a field to use for the analysis.
   BL.FieldToUse = "AMOUNT"

   ' Decide whether you want to look for positive or negative values
   ' (0 for positive values and 1 for negative values).
   BL.ValueType = 0

   ' Request one or more analyses to perform. Use 1 for first digit,
   ' 2 for first two digits, 4 for first three digits, or 8 for
   ' the second digit.
   BL.AddAnalysis 1, "Benfords Law - First Digit.imd"

   ' Perform the task.
   BL.PerformTask

   ' Open the result.
   Client.OpenDatabase "Benfords Law - First Digit.imd"

   ' Clear memory.
   Set BL = Nothing
   Set db = Nothing
End Sub
```

errant data. Consequently, when you perform this analysis, you work with a numeric field of some type. (Theoretically, you could use `BenfordsLaw` to analyze something like the number of city names in the United States beginning with the letter A, but that kind of analysis falls outside of the tasks you normally perform with IDEA.) You choose the depth of analyses. There are four variations of analyses:

- Testing the first digit
- Testing the first two digits
- Testing the first three digits
- Testing the second digit

Listing 10.3 begins by opening the database and setting the task. The database you use has to have a numeric field. In this case, `Sample-Bank Transactions` has the AMOUNT field, which is numeric.


Sample-Bank Transactions.IMD | Benfords Law - First Digit.imd

	DIGITS	EXPECTED	LOWBOUND	HIGHBOUND	ACTUAL	DIFFERENCE
1	1	229.69	215.81	243.56	249	-19.31
2	2	134.36	122.81	145.90	149	-14.64
3	3	95.33	85.29	105.37	79	16.33
4	4	73.94	64.94	82.94	81	-7.06
5	5	60.42	52.19	68.64	84	-23.58
6	6	51.08	43.45	58.71	30	21.08
7	7	44.25	37.10	51.39	39	5.25
8	8	39.03	32.29	45.77	33	6.03
9	9	34.91	28.51	41.32	19	15.91

FIGURE 10.4 BenfordsLaw Focuses on the Distribution of Data in a Dataset

At this point, you must decide what kind of analysis to perform. The first choice is whether to use positive or negative numbers—you can't use both when working with `BenfordsLaw`. The second choice is which digits to use. Traditionally, `BenfordsLaw` focuses on the first digit, but you can gain some interesting insights by using the other digit combinations.

Now the code performs the task and opens the resulting database shown in Figure 10.4. It's at this point that you see something interesting. The number of records in the analysis doesn't add up to the number of records in the database. It turns out that the AMOUNT field has both positive and negative numbers—only the positive numbers appear in the analysis. In addition, `BenfordsLaw` only applies to numbers 10 or larger because single digit numbers don't fall within the law's domain. To prove this to yourself, try the same analysis on the CREDIT_LIM field of the `Sample-Customers` database.

Summary

This chapter has helped you understand the math feature of IDEAScript. A single chapter can't really demonstrate everything there is to know about math, much less math on a computer. The purpose of this chapter is to expose you to what's possible. Other chapters in the book will add to the knowledge you gained in this chapter. However, these principles are essential because math truly is different from the computer's perspective and you need to understand that perspective.

Now that you know what's possible, you need to consider your use of math for the real-world tasks you perform. Only you can answer precisely how you employ math on a daily basis. Often, the difference between a good application and a great application is the math functionality that the great application provides. Now that you have some idea of what to do, try converting some of your math into something the computer can understand. Write small test programs like the ones found in this chapter to test your understanding of how computer math works.

Chapter 11 looks at another interesting issue that relates to databases, but doesn't necessarily work directly with them—arrays. It might interest you to know that some

kinds of math equations actually rely on lookup tables found in arrays or use arrays to perform the actual math task. This chapter is interesting because it's the first time you'll explore something somewhat complex from a code-writing perspective. Don't worry though: you'll find that this chapter eases you into working with arrays and that you'll be using them in your own applications soon.

CHAPTER 11

Interacting with Arrays

You look in a filing cabinet drawer and see multiple file folders inside. Some of them contain a lot of information, some of them a little—a few contain nothing at all. The file folders are individual containers used to hold something—it doesn't matter what that something is, just that the storage capability exists. The need for filing cabinet drawers with all of those file folders is real—can you imagine the chaos that would result if they didn't exist? Civilization as we know it might cease to exist!

Programs also have a need for organized data storage—a filing cabinet drawer of sorts—and inside that data storage are little containers—a sort of file folder. These cabinet drawers come in two forms: arrays and collections. IDEAScript only supports arrays, so that's what this chapter discusses. However, you may encounter collections when interacting with other applications through IDEAScript, so it's important to remember the term.

Note

Arrays are the simpler of the two storage techniques and you normally use them for general storage. For example, you can use an array to store a group of related numbers. A collection normally provides additional functionality, but it's usually attached to a specific object and can prove more difficult to use. Work with collections when the object provides the functionality for you.

Arrays provide simple storage of similar elements. You can move arrays around as needed, fill them with data, view, change, copy, and do anything else that you normally do with the data in your application. Arrays provide packaging that make your applications efficient and easier to understand. The following sections describe arrays in more detail.

Understanding How Arrays Work

Many people view arrays as simply a series of numbered boxes. In fact, you might hear of arrays discussed as a kind of apartment mailbox with individual boxes for each

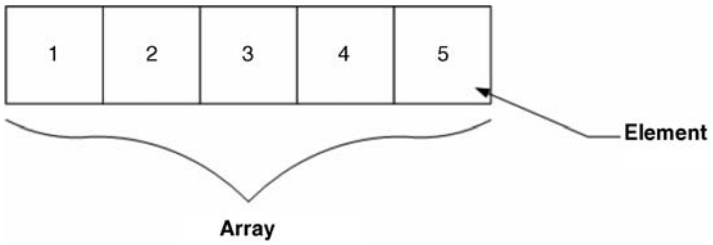


FIGURE 11.1 Arrays Provide an Organized Method of Storing Like Items

apartment. No matter how you think about arrays, they conceptually look like Figure 11.1. The whole storage area is the array, while the individual storage box is an element.

Elements hold the data. Of course, the data isn't very useful if you can't access it in some way. When you look at a filing cabinet drawer, each file folder has some kind of identification so that you know what it contains. Likewise, when you look at one of those apartment mailboxes, each individual box has a number or letter identifying the apartment to which it belongs. In the same way, array elements have a numbering scheme that helps you to access the individual elements. The elements are normally numbered from 0 to whatever length you make the array. To access the content of a particular array, you supply its number to IDEAScript.

Note

Most computer languages use arrays that start with element 0, just like IDEAScript. However, some products, such as Microsoft Office, use a 1-based array system. In this case, the array elements begin with 1 and end at the length you select—effectively giving you one less element to work with. See Chapter 16 for details on working with Excel.

Unlike the physical entities that an array mimics, you can resize arrays in some cases. You can remove information from just one element and replace it with something else if you like. In every respect, an array and its elements are simply a way to organize your data without resorting to using a database or an external file.

Creating and Using Arrays

The basic array can store any data type, including user-defined types. You can also use an array to store objects. No matter what you store in an array, you follow the same techniques for creating and using the array. Listing 11.1 shows how to create a basic array and perform simple tasks with it.

LISTING 11.1 Creating and Using Basic Arrays

```
Sub Main
   ' Define an array of three elements.
   Dim Letters(3) As String

   ' Show the lower bound of the array.
   MsgBox "The lower bound is: " & LBound(Letters)

   ' Fill the array with letters.
   Dim Count As Integer
   For Count = 0 To UBound(Letters)
      Letters(Count) = Chr$(65 + Count)
   Next

   ' Diplay the letters on screen.
   Dim Letter As String
   For Each Letter In Letters
      MsgBox Letter
   Next

   ' Modify an array element.
   Letters(2) = "Hello"
   MsgBox Letters(2)

   ' Clear the array.
   Erase Letters

   ' Show that the array is empty.
   MsgBox Letters(3)
End Sub
```

The code begins by creating the array. To perform this task, you follow the name of the array with the number of elements you want to create in parentheses. However, IDEAScript uses 0-based arrays by default. Consequently, even though the code appears to create three elements, it actually creates four when you consider element 0. Some programmers use element 0 to hold an essential piece of information about the array, such as the array size.

To demonstrate that the array really does begin with element 0, the code displays a message box next using the `LBound()` function. This function always tells you the starting point of the array. As shown in Figure 11.2, this array really does start at `0`.

The next step is to fill the array with letters. The capital letters begin at character 65 in the ASCII character set. So using the `Chr$()` function fills the array with the letters A through D. If you prefer lowercase letters, then you'd start with character 97. At this point, you have an array filled with 4 letters of the alphabet. The code demonstrates this fact by displaying each of the letters in the array in turn using a loop.

You may not actually want the letter C in `Letters(2)`, however, so the code shows you how to change it. When you want to access a particular array element,

FIGURE 11.2 IDEAScript Defaults to Using 0-Based Arrays

you supply the array name, followed by the number of the element you want to use. In this case, the code changes the content of `Letters(2)` to `Hello` , as shown in Figure 11.3.

At some point, you may want to clear all of the content from an array. For example, you might want to use the array for another task and not want to take the chance that the array will contain old content. Simply use the `Erase()` function to perform this task. All of the array elements will be empty after you use this function. The example demonstrates this fact by showing the content of `Letters(3)`, which is now blank.

You'll eventually want to work with other applications. Those applications might use 1-based arrays. Trying to get your code to work with those other applications can become nightmarish as you try to remember where the array is supposed to start. Save yourself some grief and pain; use 1-based arrays when working with those other applications. In order to perform this task, you simply use the `Option Base 1` statement as shown in Listing 11.2. This statement must appear outside any subroutine or function in the application so it is usually inserted near the beginning of the code.

This example also demonstrates another interesting IDEAScript feature. Notice that `Numbers` doesn't include the number of elements it contains as part of the `Dim` statement. That's right. `Numbers` is wandering about with a memory problem because it doesn't know what size it should be! We'll take care of that problem by asking the user what size to make the array using an input box.

FIGURE 11.3 You Can Change the Content of Any Array to Any Permissible Value for That Element

LISTING 11.2 Creating and Using 1-Based Arrays

```
' Create a 1-based array.
Option Base 1

Sub Main

   ' Create a blank array.
   Dim Numbers() As Integer

   ' Determine the number of array elements.
   Dim Size As Integer
   Size = InputBox("Type the number of array elements you want.")

   ' Resize the array to fit.
   ReDim Numbers(Size)

   ' Show that this is a 1-based array.
   MsgBox "The lower bound is: " & LBound(Numbers) & _
      " and the upper bound is: " & UBound(Numbers)

   ' Fill the array with data.
   Dim Count As Integer
   For Count = 1 To UBound(Numbers)
      Numbers(Count) = Count
   Next

   ' Show the content.
   Dim Result As String
   For Count = 1 To UBound(Numbers)
      ' Convert the number to a string.
      Result = Result & Numbers(Count) & " "
   Next
   MsgBox Result
End Sub
```

Once the application knows what size to make the array, it uses the `ReDim` statement to give the array a specific size. You can only use the `ReDim` statement with arrays that don't have an upper bound when you initially `Dim` them. Otherwise, you get an error message stating that the array is already dimensioned.

Of course, you don't actually know that the array has the correct dimensions unless you check. The example uses the `LBound()` and `UBound()` functions to check both the lower and upper array bounds and then displays that information on screen. Figure 11.4 shows an example of the output you might see.

In this case, the user has typed 5 as the desired number of array elements. The code fills the array with consecutive numbers next, and then displays the result on screen. Using the previous input as a starting point, Figure 11.5 shows the output from this example.

FIGURE 11.4 This Message Box Verifies the Lower and Upper Bounds of the Array

FIGURE 11.5 The Result Contains Consecutive Numbers

Copying Data between Arrays

There are times where you need to copy the content of one array to another array. For example, you might want to perform some "what if" analysis and not want to damage your original data. Using a copy of the array lets you perform this analysis without potentially damaging your data. Listing 11.3 shows one technique for copying one array to another.

In this case, the code begins by creating a source array. This array contains six elements in total—the five elements requested in the code and element 0. The code fills the source array next. Notice the use of the `LBound()` and `UBound()` functions to ensure that the `For...Next` loop begins and ends at the right count. It's a good practice to use the `LBound()` and `UBound()` functions whenever you have any doubt of the starting and ending element numbers.

The code creates the target array next. Notice that the code can't simply assign the target array a size because you might not know the size. Instead, the target array relies on the `ReDim` statement to set its size to the size of the source array.

At this point, the code performs the copy from one array to the other. Notice that you simply set the individual elements of the target array to equal those of the source array. Now you have a perfect copy. The code outputs a message box showing that the target array content is indeed the same as the source array as shown in Figure 11.6.

LISTING 11.3 Copying Arrays

```
Sub Main
   ' Create the source array.
   Dim Source(5) As Integer

   ' Fill the source array.
   Dim Count As Integer
   For Count = LBound(Source) To UBound(Source)
      Source(Count) = Count
   Next

   ' Create the target array.
   Dim Target() As Integer

   ' Modify the target array size to match the source.
   ReDim Target(UBound(Source))

   ' Copy the Source array to the Target array.
   For Count = LBound(Source) To UBound(Source)
      Target(Count) = Source(Count)
   Next

   ' Show the content.
   Dim Result As String
   For Count = LBound(Target) To UBound(Target)
      ' Convert the number to a string.
      Result = Result & Target(Count) & " "
   Next
   MsgBox Result
End Sub
```

FIGURE 11.6 Copying Arrays Lets You Modify Data in the Copy without Affecting the Original

Summary

This chapter has helped you understand arrays. In general, you use arrays whenever you need to create a generic storage container of like items. Remember that an array can contain any item you desire.

Some people find arrays are a little difficult to understand. It really does pay to create a few arrays and try working with them in an application. The better you understand arrays, the easier it is to create certain kinds of applications. Try starting with very simple, very small arrays. If necessary, draw a picture of the array so that you can visualize how the array will work in your application. A little graph paper and a few lines can work wonders. Make before and after pictures of the array so that you can picture the effect of the code you write.

Previous chapters have used `MsgBox()` and `InputBox()` to provide a user interface. The applications you created were simple enough that a complex interface really wasn't necessary. However, databases can become complex and analysis can add to that complexity. In some cases, a simple `MsgBox()` or `InputBox()` won't do the job—you need something more. Chapter 12 shows how to create custom dialog boxes that can provide the flexibility needed to remove some of the complexity from your application. When you can communicate better with the user, your application becomes more effective and less difficult to write.

CHAPTER 12

Creating Interactive Dialog Boxes

Dialog boxes are an important part of most applications. You've seen dialog boxes used for many of the examples so far in the book. A dialog box can serve to inform (`MsgBox`) or obtain simple input (`InputBox`). However, sometimes you need to obtain more information than a simple input box can provide. Interactive dialog boxes are a cornerstone of many applications with any complexity, and they aren't even hard to create: All you do is drag and drop the items you want on the blank dialog box—really, it's that simple!

Of course, dragging and dropping controls onto a dialog box won't do much. First, you want to make sure that the controls appear in a logical manner and that they're easy to use. All of us have had to deal with dialog boxes that were less than appealing. Sometimes, a dialog box is constructed so poorly that it's simply frustrating to use. This chapter spends some time reviewing techniques you can use to keep users of your dialog boxes happy.

IDEAScript provides you with access to all of the standard controls (a visual element on a dialog box)—view them as a toolbox of controls. As with any toolbox, you need to know what the toolbox contains in order to use the tools inside efficiently. This chapter also spends time telling you about the controls that IDEAScript provides and helps you understand when each control works best, so you don't try to use a text box where a list box would work better (akin to pounding in a screw with a hammer—it works, but you won't enjoy it).

After you create the visual portion of a dialog box, you need to write some code to make the controls do something. This chapter focuses on three straightforward tasks that you'll use most often:

- Display the dialog box so the user can see it.
- Do something interesting with the controls on the dialog box.
- Pass information back to the *caller* (the subroutine or function that made the call) when necessary.

Finally, this chapter discusses a few interesting topics. For example, you'll discover how to add graphics to your dialog boxes. No, you won't discover how to create a multimedia extravaganza, but you'll learn how to add a bit of pizzazz to an otherwise

ordinary application. In addition, users will actually benefit from the visual cues you provide.

Creating Great Dialog Boxes

To successfully create dialog boxes, three kinds of talent are required.

First, you must have experience in the task that you're trying to present. You already have that experience, you've been working with IDEA for a while now and have a lot of knowledge of your trade—knowledge that most developers would be hard pressed to learn. Therefore, you're already ahead of many developers when it comes to creating dialog boxes for your particular application—imagine that!

Second, dialog boxes require that you know how to configure the visual elements in a pleasing way that works with the user. This section of the chapter teaches you when to use dialog boxes and how to lay them out so that others understand what you're trying to communicate. Dialog boxes are a kind of art—you communicate your ideas to others using visual elements (with a few textual cues added).

Third, dialog boxes require careful coding and testing. Many developers make things too complicated. Don't follow their example; make things as simple as you can. The later sections of this chapter tell you about the coding requirements. For now, the following section provides you with the initial requirements for working with dialog boxes in the most efficient manner possible.

Best Practices for Using Dialog Boxes

The best dialog box is the one that you don't have to create. Many of the examples in this book have already shown you how to use `MsgBox` and `InputBox`. In addition, you have access to `DisplaySetupDialog()` for many tasks. These three, pre-made, dialog box types answer a considerable number of needs.

You should also avoid using dialog boxes when you can obtain the information you need from some other source, such as a database or directly from IDEA. Asking the user for information doesn't always produce the results you envisioned. There are some problems with dialog boxes that you need to consider when you design your application:

- Dialog boxes make the application more complex because you're asking for user interaction and the user might not always understand what input you require.
- Displaying a dialog box stops your application. If the user isn't available to interact with the dialog box, then the application waits until the user is available—perhaps a very long time for a task that the user expected to run for a while. Even if the user is available and reacts to the dialog box immediately, the presentation, interpretation, and processing of a dialog box takes time that reduces the overall efficiency of your application.
- Whenever you display information on screen, there's a potential for someone other than the intended recipient to be reading that information. Given the privacy and security laws in effect for data management, using dialog boxes when they aren't really necessary could cause problems for your organization.

- Custom dialog boxes are unfamiliar to users. In many cases, this means that the user will require additional training to use the dialog box, which increases the overall cost of using your application. To keep costs low, use as few, simple, dialog boxes as you can.

Of course, there are times when you need to create a custom dialog box. For example, Chapter 13 emphasizes the use of custom dialog boxes for search forms. IDEA doesn't provide access to a premade form that fully answers the needs of a good search form, in this case, because your data is unique and a search form should consider the data to which it provides access. Theoretically, you could cobble several pre-made dialog boxes together to answer this need, but the result would be horrid to use and error prone as well. A dialog box needs to communicate well with the user. Consequently, don't be afraid to use a custom dialog box when you have a specific need that isn't addressed by IDEA (often because the IDEA developers can't anticipate your requirements).

You also need to create a custom solution when the setup dialog box displayed with `DisplaySetupDialog()` provides too much information. It's important to keep things simple and sometimes the setup dialog box presents too many settings for your user. For example, the Aging setup dialog box shown in Figure 12.1 is relatively complex. If you create a custom dialog box that presents just the required information, such as the database name and intervals, you can reduce the complexity of working with the `Aging` task for the user.

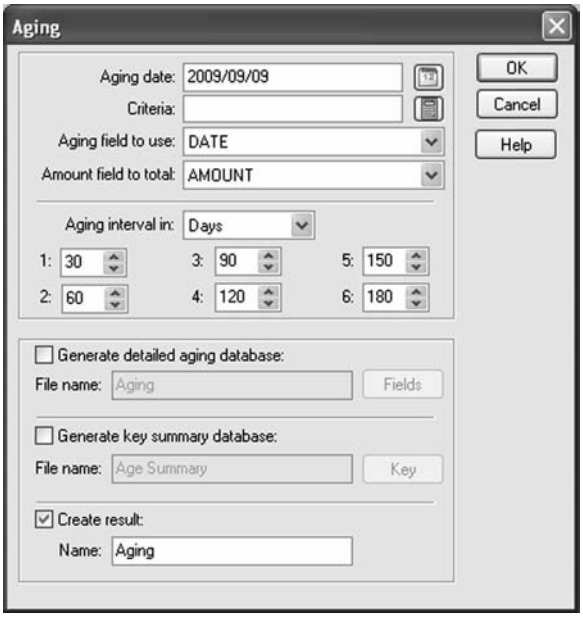


FIGURE 12.1 The Aging Setup Dialog Box Demonstrates One Time where A Simplified Custom Dialog Box Is Helpful

When you do create a dialog box for your application, remember to test it. Ask anyone who will use the dialog box to work with it in the performance of actual tasks. You may find that the dialog box that seems perfectly usable and understandable to you, doesn't appeal to the user at all. Don't worry about it—developers face the same problem all over the world. Simply take time to work through the dialog box problems with the user and fix them to ensure the dialog box works as expected.

Designing a Good Dialog Box Layout

Dialog boxes normally follow certain rules. For most countries, the controls in dialog boxes flow from left to right, top to bottom. A form where you started in the middle and then worked your way out would be original, but it would also be confusing. Users have adopted certain ways of viewing dialog boxes over time, so you need to follow the directions that users expect. Of course, you might live in a country where users are used to reading from right to left or from bottom to top. The best way to design your dialog box is to ensure the controls flow in the same direction that you'd read a book or newspaper.

Unfortunately, as with many things in life, there are exceptions to the positioning rules that you need to consider. Many dialog boxes place buttons on the right side or the bottom of the dialog box, with the right side being most common. Where you place buttons depends on the complexity of the dialog box. For most of the dialog boxes you'll create with IDEAScript, the buttons will appear on the right side of the dialog box, from top to bottom, in the order the user is more likely to require them (OK is usually at the top, Cancel is at the bottom, and any other buttons appear in the middle).

Align the controls on a dialog box whenever possible. As odd as it may sound, sloppy dialog boxes are distracting. A user will concentrate better when the controls on your dialog box are aligned and the dialog box itself looks neat. Well-ordered dialog boxes also demonstrate a certain pride in the application you've created—they show that you care about your work.

Make sure you include good prompts—a short piece of text that tells users about the purpose for the control. Many people have a problem with creating prompts that are too short or not including prompts at all. For example, imagine you provide four controls to hold someone's name: title, first name, middle initial, and last name. If you provide a prompt that simply says "Name," people using your application won't know that the first space is actually the title. A prompt that says, "Name (Title, First, Middle, and Last)" is longer, but better. However, you can provide prompts that are too long, too. If you feel as if you're writing a novel, then the prompt is too long.

Using the Basic Controls

IDEAScript provides you with access to a number of standard controls. You won't find anything fancy in the list, but you'll find everything needed to create a useful custom dialog box. The controls automatically appear in the Dialog Tools window shown in

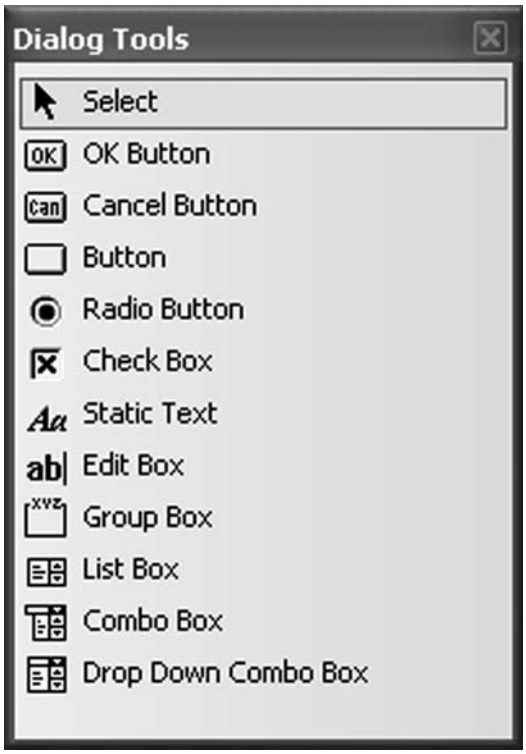


FIGURE 12.2 IDEAScript Provides You with a Basic Set of Useful Controls

Figure 12.2 whenever you create a new dialog box by right-clicking Dialogs in the Project window and selecting New Dialog from the context menu.

In general, IDEAScript provides access to just the basic properties for each control. You also trap events by looking for specific outputs from calls you make using the dialog box. Don't worry too much about events for now—you'll explore them in more detail in the "Creating a Basic Dialog Box" section. The following sections describe each of the controls that IDEAScript provides.

Considering the Select Tool

In many cases, you won't add new controls to your dialog box—you simply want to modify existing controls. When this happens, you use the Select tool—the one that looks like a mouse pointer at the top of Figure 12.2. When you have the Select tool highlighted, clicking on a control selects it. You see the properties for the control in the Properties window shown in Figure 12.3. In this case, you're looking at the properties for an `OKButton` control, but the same principles apply to any control you select.

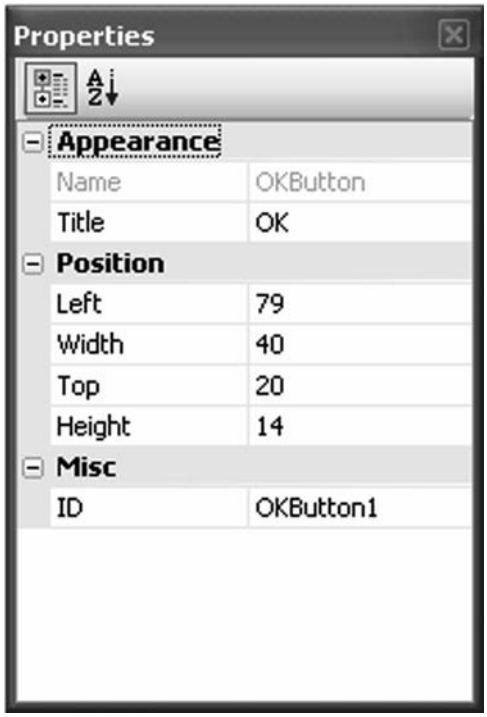


FIGURE 12.3 Use the Select Tool to Pick Controls in Your Dialog Box

Tip

You can sort the contents of the Properties window by category (as shown in Figure 12.3) or in alphabetical order. Categories group like properties together, making them easier to find when you need to change the values of a number of associated properties, such as the size or position of a control. Use alphabetical order when you remember the name of a particular property and want to find it quickly in the list. To change the sort order of the properties, simply click one of the two buttons that appear at the top of the list.

The Select tool works the same as the mouse cursor in any application. If you want to choose just one control, simply click on it. On the other hand, when you want to choose more than one control for tasks such as aligning controls, Ctrl+Click or Shift+Click the controls you want to select. IDEA adds each control that you Ctrl+Click or Shift+Click to the selected group of controls. If you Ctrl+Click or Shift+Click the control again, IDEA deselects it.

Tip

It's possible to select multiple controls using what is termed the rubberband or lasso technique. Place the mouse cursor in a position outside of the group of controls you want to select. Click the left mouse button and without releasing the button, drag the mouse cursor in the direction of the controls you want to select. You'll see a box appear. Any control within that box is selected. When you have all of the controls that you want to select within the box, release the left mouse button. You'll see that the controls are selected.

Adding Clicks Using the Generic Button

A general `Button` control (or `PushButton`) lets the user signal some event to you. Of course, the most common uses for a `Button` is to signal that it's OK to proceed with processing based on the inputs provided to a dialog box or to tell the application to cancel the action. IDEAScript also includes the `OKButton` and `CancelButton`, which appear in the sections that follow.

Configuring a `Button` is straightforward because this control only supports a few properties. Table 12.1 shows the properties that a `Button` supports.

Whenever the user clicks a `Button`, the `Button` returns a value that matches the position of the `Button` in the `Button` hierarchy. The first `Button` added to the dialog

TABLE 12.1 Button, OKButton, and CancelButton Properties

Property	Description
Name	Shows the name of the control. You can't change this value.
Title	Modifies the text displayed on the button. Whenever you use a generic `Button`, you should change this property to match the purpose of the `Button`. You can also change the `OKButton` and `CancelButton` Title to meet specific needs, but normally you won't change them.
Left	Determines the position of the control from the left side of the client area of the dialog box in pixels. The client area is the area in which you can place controls and doesn't include the title bar or the window border.
Width	Determines the width of the control in pixels. The default `Button` width is 40 pixels, which will accommodate most short to moderately sized prompts.
Top	Determines the position of the control from the top of the client area of the dialog box in pixels.
Height	Determines the height of the control in pixels. The default height is 14 pixels.
ID	Contains the name of the control as you access it in your code. Many developers use special prefixes to make it easier to work with their controls and know which control is of a particular type. For example, a `Button` control commonly uses `btn` as a prefix, such as `btnOK` for an OK button. IDEAScript doesn't care what you call your `Button`, as long as you give it an `ID`.

box will return a value of 1, the second a value of 2, and so on. The value you receive from the Button identifies the specific Button that the user clicked.

Tip

You can make your dialog boxes easier to use by including speed keys (also called accelerator keys or short-cut keys). A speed key is simply the underlined letter that you see on many menus and buttons for applications you already have. To select the button based on its speed key, you press Alt+<Speed Key Letter>. For example, if you underline the O on an OK button, the user presses Alt+O to click that button without using the mouse. To underline a particular character in a Button or other control Title property, simply precede that letter with an ampersand (&). Use two ampersands (&&) to add an actual & in the Title.

Adding Clicks Using the OKButton

The OKButton is a special version of the generic Button control (see the "Adding Clicks Using the Generic Button" section). The OKButton defaults to a Title of OK. It also returns a value of −1, rather than a value of 1 or above, when clicked in a dialog box. The OKButton is the default Enter button. If you have the cursor in a TextBox or other data control and you press Enter, it's the same as clicking OK. You can only have one OKButton in your dialog box.

Adding Clicks Using the CancelButton

The CancelButton is a special version of the generic Button control (see the "Adding Clicks Using the Generic Button" section). The CancelButton defaults to a Title of Cancel. It also returns a value of 0, rather than a value of 1 or above, when clicked in a dialog box. Pressing Esc automatically selects the CancelButton. You can only have one CancelButton in your dialog box.

Providing Options Using the RadioButton

The RadioButton gets its name from radios where you press a button to change stations. You typically find these radios in your car, but you might see them in other places as well. In an application, a RadioButton lets a user choose just one option out a group of options. Most developers use RadioButton controls when space isn't a concern and they want to be sure the user sees all of the available options. A ListBox or ComboBox can also work for option selections and they consume less space in most cases, but neither of these controls is as accessible as the RadioButton. A RadioButton has the properties shown in Table 12.2.

TABLE 12.2 RadioButton Properties

Property	Description
Name	Shows the name of the control. You can't change this value.
Title	Modifies the text displayed to the right of the control. This text should reflect the option the user chooses when selecting the `RadioButton`.
Left	Determines the position of the control from the left side of the client area of the dialog box in pixels.
Width	Determines the width of the control in pixels. The default width is 40 pixels, which will accommodate most short to moderately sized prompts.
Top	Determines the position of the control from the top of the client area of the dialog box in pixels.
Height	Determines the height of the control in pixels. The default height is 14 pixels.
ID	Contains the name of the control as you access it in your code. The standard prefix for a `RadioButton` is `opt`, but you can choose anything you wish.
GroupID	Contains the name of the group to which a `RadioButton` belongs. A user may only select one `RadioButton` in a group. Consequently, if your application requires the user to select multiple options, you must provide multiple `RadioButton` groups. You can have as many groups on a dialog box as desired.

Although a `RadioButton` will work just fine if you place it directly on the dialog box, it's common practice to place `RadioButton` controls in the same group in a `GroupBox` (see the "Grouping Related Items Using the GroupBox" section for details). Using a `GroupBox` to hold the `RadioButton` controls provides a visual grouping so users don't get confused.

It turns out that the `GroupID` is important for another reason. When a user clicks OK on a dialog box, you can read the `RadioButton` selection through the `GroupID`. The `RadioButton` values begin with `0` and increase by `1` each time you add a new `RadioButton`. Because this is a bit confusing, let's look at a quick example.

Let's say you have a dialog box that contains three `RadioButton` controls, an `OKButton`, and a `CancelButton`, as shown in Figure 12.4. The `RadioButton` controls are all part of `grpColors`. When the user selects `Red`, the group will provide a return value of `0`. Likewise, when the user selects `Blue`, the group will provide a return value of `1`. Even though each of the `RadioButton` controls has a different `ID`, all of them have the same `GroupID`.

Now that you've seen the dialog box, let's look at some code. Listing 12.1 shows how to work with the `RadioButton` controls in this instance.

The code begins by creating an object that represents the dialog box shown in Figure 12.4. Whatever name you give the dialog box, you use to create that dialog box in your code (see the "Creating a Basic Dialog Box" section for additional information). The code uses the `Dialog()` function to display the dialog box on screen. When the user clicks OK or Cancel, IDEAScript places the selection in `Result`. Because the `CancelButton` always returns `0`, if `Result = 0`, the user clicked Cancel and the application exits. Otherwise, the user must have clicked OK.

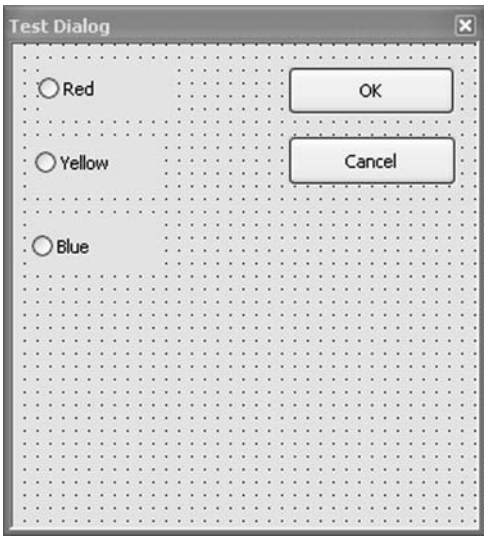


FIGURE 12.4 A Test Dialog Box to Demonstrate Radio Buttons

LISTING 12.1 Using RadioButton Controls

```
Sub Main
    ' Create the dialog
   Dim MyDialog As dlgTest

   ' Display the dialog on screen.
   Dim Result As Long
   Result = Dialog(MyDialog)

   ' Make sure the user hasn't clicked Cancel.
   If Result = 0 Then
      Exit Sub
   End If

   ' Get the selection.
   Dim Choice As Integer
   Choice = MyDialog.grpColors

   ' Display the selection
   Select Case Choice
      Case 0
         MsgBox "You chose Red"
      Case 1
         MsgBox "You chose Yellow"
      Case 2
         MsgBox "You chose Blue"
   End Select
End Sub
```

The object you create from `dlgTest` also gives you access to the controls. In this case, the code places the value of the group, `grpColors`, in `Choice`. The code then relies on a `Select Case` statement to determine which selection the user had made. In short, you rely on the group, not the individual `RadioButton`, to determine which option the user has selected.

The group is also handy for another reason. Let's say you want to select something other than the default `RadioButton` for the user before IDEAScript displays the dialog box on screen. In this case, you set the group equal to a value as shown here:

```
' Choose the Blue button.
MyDialog.grpColors = 2
```

Providing Options Using the CheckBox

`CheckBox` acts as an on/off switch. You use it to select an option regardless of any other option you selected. In short, you use this control to create a checklist of items that the user may or may not want. `CheckBox` is a good option when you want to display a list of items and ensure the user sees every option. Like `RadioButton`, `CheckBox` uses additional space on screen. If space is at a premium, you should use `ListBox` or `ComboBox` instead. Table 12.1 shows the properties associated with `CheckBox` (they're the same as `Button`).

`CheckBox` controls are either checked (`True`) or not (`False`). You read a `CheckBox` through the dialog box object as shown here (assuming you have created a dialog box object named `MyDialog`):

```
If MyDialog.cbBird Then
   Choices = "You love birds."
Else
   Choices = "You don't love birds."
End If
```

The default `CheckBox` state is clear (or `False`). However, you can set the `CheckBox` state before IDEA displays it by setting `CheckBox` to either `True` or `False`, as shown here:

```
MyDialog.cbBird = True
```

Displaying Prompts Using StaticText

The `StaticText` control doesn't do very much on its own. You can't type information into it or click it. However, this control is very important because it provides the prompts that users need to understand your dialog box. When you work with this control, you simply type the text that you want to appear on screen in the `Title` property. The properties for `StaticText` match those in Table 12.1 (the same as `Button`).

One special use for `StaticText` is with the `EditBox` control. An `EditBox` control lacks a `Title` property. The `StaticText` control normally appears immediately above

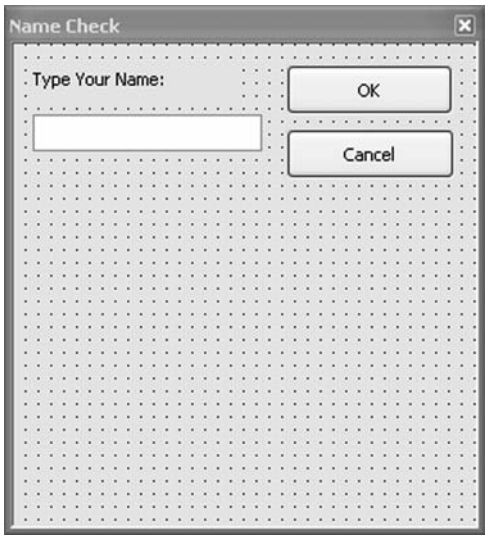


FIGURE 12.5 Combine the StaticText and EditBox Controls to Provide Documentation and Speed Key Support

or to the left of the `EditBox` control. If you want to add a speed key for the `EditBox` control, you do it using the `StaticText` control. Simply add an ampersand immediately before the letter you want underlined as you normally would. Figure 12.5 shows an example of a combination of `StaticText` and `EditBox` controls.

Getting User Input Using the EditBox

The `EditBox` (or `TextBox`) control lets the user type text into the dialog box that your application can interpret. You use an `EditBox` control in situations where you can't easily predict the user's input in advance. For example, you'd use an `EditBox` control to obtain the user's password. The `EditBox` control serves an essential purpose in that it's the only true freeform data entry control that IDEA provides. In general, you want to avoid using `EditBox` controls whenever possible for the following reasons:

- Users may not understand what you want them to type, so they can type something completely inaccurate. If you catch the error and tell the user to type the value again, the user might repeat the mistake. The cycle can continue until the user is completely frustrated.
- Typos are a fact of life—everyone makes them. An `EditBox` control offers the opportunity for users to type what they think is the correct information, but actually ends up being a misspelling. When the information is verified, the user can end up getting frustrated as a minimum.
- Using `EditBox` controls opens your application to potential security problems. You can avoid some of these problems, but probably not all of them. It's important to

verify every piece of information a user provides through an `EditBox` control. A simple check, such as checking the length of the input, can help you locate and dispose of security issues presented by `EditBox` controls. Chapter 20 helps you more with security issues.

- Due to all of the other problems in this list, `EditBox` controls require extra verification and validation, which means you have to write more code to use an `EditBox` control correctly. More code means that the probability of introducing an error into your code is higher and it also means that your application won't run as fast.

Tip

As your coding skills improve, you can usually reduce your use of the `EditBox` control. For example, the system already knows the name of your user, so you can often ask it for a username. The system knows a lot of details about the user, the network, the applications on the local drive, and all sorts of other topics. This book doesn't cover many advanced programming topics in detail—it's meant to introduce you to IDEAScript. However, there are techniques that you can learn for working with the Windows Application Programming Interface (API) that can move your code from extremely useful to highly innovative.

Don't get the idea that using the `EditBox` control is evil or anything like that. All that this section implies is that using the `EditBox` control presents certain risks that you should consider. There are situations where you simply have no other choice. Table 12.3 shows the properties for the `EditBox` control.

TABLE 12.3 EditBox Properties

Property	Description
Name	Shows the name of the control. You can't change this value.
Title	Even though this property appears in the list, you can't change it.
Left	Determines the position of the control from the left side of the client area of the dialog box in pixels.
Width	Determines the width of the control in pixels. The default width is 40 pixels, which will accommodate most short to moderately sized prompts.
Top	Determines the position of the control from the top of the client area of the dialog box in pixels.
Height	Determines the height of the control in pixels. The default height is 14 pixels.
ID	Contains the name of the control as you access it in your code. The standard prefix for an `EditBox` control is `txt`, but you can choose anything you wish.
Style	Determines the style of the control. For example, you can configure the control to display a special character in place of the actual characters the user types for a password. It's also possible to create a multiline text box that the user can use for typing longer text items.

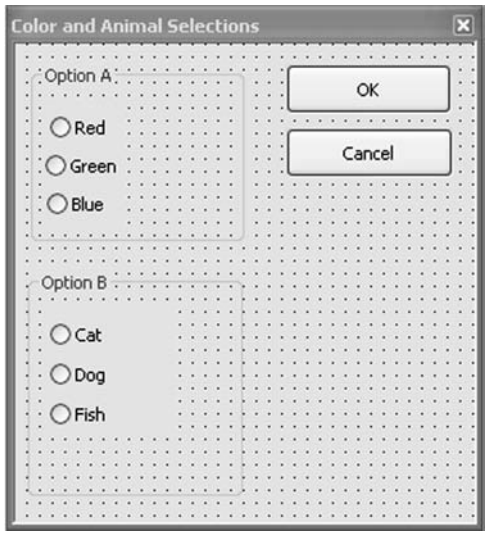


FIGURE 12.6 Use the GroupBox Control to Arrange Like Items Together

Grouping Related Items Using the GroupBox

The `GroupBox` control provides a visual grouping of controls. You use it to make it apparent which controls belong to each other on screen. Figure 12.6 shows an example of a `GroupBox` control in action. `GroupBox` controls have the same properties as listed for the `Button` control in Table 12.1.

Even though a `GroupBox` control consumes screen real estate, it serves an important purpose. Imagine trying to figure out which options go together in Figure 12.6 without the benefit of a `GroupBox` control. Other than providing the arrangement and classification (with the `Title` property) feature, the `GroupBox` control doesn't provide any runtime functionality you need to consider.

Providing Options Using the ListBox

A `ListBox` control provides a list of items from which a user can choose. In many respects, a `ListBox` control works just like a group of `RadioButton` controls. The user sees a list of items, from which the user can select only one item. Figure 12.7 shows an example of a `ListBox` control populated with colors.

As you can see, the `ListBox` control is more space efficient than a group of `RadioButton` controls would be. You can size the `ListBox` control to use as much space as you can, but you aren't required to allow space for every item. If the list of items exceeds the display area, the `ListBox` control displays a scrollbar on the right side so the user can scroll through the list of selections. The `ListBox` control uses the properties shown in Table 12.4.

FIGURE 12.7 The ListBox Control Works Like a Group of RadioButton Controls in Many Respects

As previously mentioned, the `ListBox` control works very much like a group of `RadioButton` controls. However, instead of obtaining the selection from the group ID, you simply use the `ListBox` control ID to access the user selection like this:

```
Choice = MyDialog.lstColors
```

The `AttachedList` property requires a little explanation as well. You provide the name of a variable that contains the list of items to display. For example, if the name of

TABLE 12.4 ListBox, ComboBox, and DropDownComboBox Properties

Property	Description
Name	Shows the name of the control. You can't change this value.
Title	Even though this property appears in the list, you can't change it.
Left	Determines the position of the control from the left side of the client area of the dialog box in pixels.
Width	Determines the width of the control in pixels. The default width is 40 pixels, which will accommodate most short to moderately sized prompts.
Top	Determines the position of the control from the top of the client area of the dialog box in pixels.
Height	Determines the height of the control in pixels. The default height is 14 pixels. You'll normally need to increase the height to allow for more than one item to display and also to make the scrollbars usable when the list of items exceeds the display space. A `ListBox` control that's 38 pixels high can accommodate 6 items.
AttachedList	Contains the name of an array variable that holds the values used to populate the control. Each item in the array appears on a separate line.
ID	Contains the name of the control as you access it in your code. The standard prefix for a `ListBox` control is `lst`, but you can choose anything you wish.

the array is MyColors, then you type MyColors() in the AttachedList property. Typing MyColors() dimensions the array for you. Consequently, you must ReDim the array to the size required to hold your list as shown here:

```
' Define the list used to populate the listbox.
ReDim MyColors(7)
MyColors(0) = "Black"
MyColors(1) = "Blue"
MyColors(2) = "Green"
MyColors(3) = "Teal"
MyColors(4) = "Brick"
MyColors(5) = "Magenta"
MyColors(6) = "Brown"
MyColors(7) = "Light Gray"
```

Tip

Many developers find it best to define the array as normal, using a Dim statement, and assigning the array a data type. When you create the list box and assign the array to it, IDEA automatically changes the Dim statement to a ReDim statement. If this method isn't used, then no data type is assigned to the array.

Providing Options Using the ComboBox

The ComboBox control is a combination of the ListBox and EditBox controls. You can type in a value or you can select one from a list. As you type text into the ComboBox control, the ComboBox control automation shows you choices that reflect what you have typed. It's also possible to simply scroll down the list and choose one of the entries by highlighting it. ComboBox, like EditBox, also lets you type freeform text—you don't have to select any of the available options. Consequently, the ComboBox control shares many of the same security concerns as the EditBox control, but it's considerably easier to use. No matter what you do, the selection you make appears in the edit area of the ComboBox control.

The ComboBox control is a great option for long lists. No one wants to scroll through a list of states. Using a ComboBox control lets the user type the first couple of letters in the state name and then select the desired option from the list. Because a ComboBox control also provides a list of acceptable options, it's easier to check the user input for invalid information. In short, the ComboBox control can be a safer sort of EditBox control that's also considerably easy to use.

Table 12.4 shows the properties for the ComboBox control. Like ListBox, you must provide an array containing the options you want the user to see. However, unlike ListBox, the output from ComboBox is a string, not a number. Consequently, when you type Selections = MyDialog.cbColors you receive a string as output. This

FIGURE 12.8 The ComboBox Control Is a Combination of the ListBox and EditBox Controls

string contains whatever appears in the `EditBox` portion of the `ComboBox` control. If the user doesn't type or select anything and the `EditBox` portion is blank, then you receive an empty string. Figure 12.8 shows a typical example of a `ComboBox` control (the top control). Below it is a `DropDownComboBox` control, which is discussed in the next section of this chapter.

Providing Options Using the DropDownComboBox

The `DropDownComboBox` control is the most space efficient selection for presenting a list of items. When the user isn't working with this control, it occupies the same space as the `EditBox` control. However, when the user wants to make a selection, the `DropDownComboBox` control displays a list in about the same way as the `ComboBox` control as shown in Figure 12.9.

Like the standard `ListBox` control, the user must select one of the options in the list and can't type any non-list values. However, the user can type the first letter of an option and the `DropDownComboBox` control will highlight an option with that letter. Type the letter again to see a second option with the same letter. In fact, you can quickly go through every item in the list with a specific letter or number simply by typing the first letter or number as required. The `DropDownComboBox` control has the properties described in Table 12.4.

The `DropDownComboBox` control defaults to the first item in the list, even if the user doesn't select anything. Consequently, you always receive a value. In addition, the output from a `DropDownComboBox` control is a number, just like the `ListBox` control,

FIGURE 12.9 The DropDownComboBox Control Is Space Efficient and Easy to Use

so you use a `Switch...Case` statement with the array you created to discover which option the user selected.

Obtaining the Visual Appearance You Want

Maintaining a neat dialog box appearance doesn't need to be difficult or time consuming. IDEA provides a number of commands for aligning and positioning your controls on the dialog box. All of these controls appear on the Standard toolbar shown in Figure 12.10. IDEA automatically displays this version of the Standard toolbar when you open a dialog box. The following sections describe how to use the various alignment and positioning buttons.

FIGURE 12.10 The Standard Toolbar Contains Helpful Positioning Controls

Note

Remember that you can select individual controls by clicking them using the Select tool. However, many of the visual appearance tools require that you select multiple controls. In this case, you select the first control in the group, and then Ctrl+Click or Shift+Click the other controls you want to work with. The last control you click is the one that determines what action IDEA will take. For example, if you select a group of controls, and then click Make Both Sizes the Same, all of the controls you selected will be changed to appear the same size as the last control you selected in the group. The same holds true for alignment and other tasks—the last control is the one that IDEA uses to perform the task.

Aligning Controls

The first four buttons in Figure 12.10 are the usual New, Open, Save, and Print buttons. After that, you see six buttons that help you align controls. Aligning your controls improves the visual presentation of your application. Users can be distracted by controls that are misaligned and difficult to follow. More importantly, the way you align controls makes it easier for users to see the flow you want to provide in your application. Here are the seven alignment commands and how you use them.

- **Align Lefts:** Aligns the left side of all of the controls you select. You commonly use a left alignment for the entry controls (labels and fields) of data entry forms.
- **Align Centers:** Aligns the centers of all the controls you select. When added to horizontal centering on the dialog box, the controls appear completely centered. You commonly use center alignment for titles and graphical elements.
- **Align Rights:** Aligns the right side of all the controls you select. You commonly use right alignment for buttons that appear on the right side of the dialog box, such as the OK and Cancel buttons.
- **Align Tops:** Aligns the tops of all of the controls you select. You commonly use top alignment for labels and their associated fields. Top alignment also works well to align the topmost data entry field with the OK button.
- **Align Middles:** Aligns the middles of all the controls you select. You commonly use middle alignment for graphical elements that are also centered on the dialog box.
- **Align Bottoms:** Aligns the bottoms of all of the controls you select.
- **Align to Grid:** Aligns one or more selected controls with the grid (the dots that appear on the design surface). Aligning controls with the grid makes it easier to position them accurately because the grid lines are a specific distance apart.

Making Controls the Same Size

The sizing commands appear after the alignment commands shown in Figure 12.10. Making controls, such as buttons, the same size gives your dialog box a uniform appearance,

especially when these controls are aligned (as buttons often are on the right side or along the bottom of the dialog box). Using the sizing commands saves you time having to manually modify each control in a group to the right size.

It isn't always necessary to make controls the same size. For example, the fields on a data entry form may vary in size to give the user a visual cue as to the amount of data that you expect them to provide or the maximum amount of data that they can provide. The following list describes each of these controls.

- **Make Same Width:** Changes the width of all of the controls you select to match the width of the last control you select.
- **Make Same Height:** Changes the height of all of the controls you select to match the height of the last control you select.
- **Make Both Sizes Same:** Changes the width and height of all of the controls you select to match the width and height of the last control you select.

Defining the Horizontal and Vertical Spacing

Accurate spacing between controls can be one of the hardest things to achieve when creating a dialog box. Using these commands makes it possible to provide the same spacing between controls, even if the controls aren't the same size, and increase or decrease spacing as needed. The following list describes each of the spacing controls, which appear after the sizing commands on the Standard toolbar shown in Figure 12.10:

- **Make Same Horizontal Spacing:** Changes the space between controls you select (not the starting or ending point) so that each control has the same horizontal space between it. You commonly use this command to space buttons placed along the bottom of a dialog box or to space multiple data entry fields on the same line.
- **Increase Horizontal Spacing:** Changes the space between controls and the ending point to increase the horizontal space between controls by eight pixels. This command doesn't change the starting point of the controls (the leftmost control in the group).
- **Decrease Horizontal Spacing:** Changes the space between controls and the ending point to decrease the horizontal space between controls by eight pixels. This command doesn't change the starting point of the controls (the leftmost control in the group).
- **Remove Horizontal Spacing:** Removes all of the horizontal space between controls without causing the controls to overlap. This command changes the space between controls and the ending point, but doesn't change the starting point of the controls.
- **Make Same Vertical Spacing:** Changes the space between controls you select (not the starting or ending point) so that each control has the same vertical space between it. You commonly use this command to space buttons placed along the right side of a dialog box or to space multiple data entry fields on different lines.
- **Increase Vertical Spacing:** Changes the space between controls and the ending point to increase the vertical space between controls by eight pixels. This command doesn't change the starting point of the controls (the topmost control in the group).

- **Decrease Vertical Spacing:** Changes the space between controls and the ending point to decrease the space between controls by eight pixels. This command doesn't change the starting point of the controls (the topmost control in the group).
- **Remove Vertical Spacing:** Removes all of the vertical space between controls without causing the controls to overlap. This command changes the space between controls and the ending point, but doesn't change the starting point of the controls.

Centering Controls on a Dialog Box

The centering commands appear after the spacing commands on the Standard toolbar shown in Figure 12.10. Centering controls on the dialog box tends to give them focus. Users tend to pay more attention to the controls in the middle. Consequently, many developers center titles and graphic elements so the user will be sure to see them. In many cases, a developer will also select all of the controls in a dialog box and center them after completing the dialog box design. Using this approach gives the dialog box a finished look so that the controls are spaced equally from the sides of the dialog box. The following list describes the centering commands:

- **Center in Form Horizontally:** Centers the selected controls horizontally in the dialog box. If you select a single control, that control will appear in the horizontal center. Selecting a group of controls will center the entire group in the horizontal center without affecting the relationship between controls.
- **Center in Form Vertically:** Centers the selected controls vertically in the dialog box. If you select a single control, that control will appear in the vertical center. Selecting a group of controls will center the entire group in the vertical center without affecting the relationship between controls.

Using the Grid for Alignment

The grid is exceptionally useful for alignment tasks. If you want to align all controls you add to your dialog box to the grid, select Format > Snap to Grid. IDEA places a check mark next to the Snap to The Grid option, which shows that all controls will be aligned to the grid. If you later decide that you don't want to align controls to the grid, select Format > Snap to The Grid again.

Interacting with Dialog Boxes Using Code

Now that you know all of the basics of creating a custom dialog box, it's time to look at a few examples. The following sections provide some basics of working with custom dialog boxes. Of course, you'll see more examples as the book progresses—these examples are only here to get you started doing some fantastic things with your applications using custom dialog boxes.

Creating a Basic Dialog Box

It's always a good idea to start with a basic dialog box example. Designing and coding your first dialog box can be a little frustrating, but this example gets you started with minimum fuss. The following steps get you started.

1. In the **Project** window, right-click **Dialogs** and select **New Dialog** from the context menu. IDEA creates a new dialog box for you. You see the new dialog box, along with the **Properties** and **Dialog Tools** windows. Let's change some of the dialog box properties.
2. In the **Title** field of the **Properties** window, type **Name Check**. IDEA changes the title of the dialog box. Always give your dialog boxes a name that matches the dialog box purpose. This example will ask you your name and then display a dialog box containing the result. It's not an exciting example, but asking people their name is one of the things that developers do quite often. In the **Name** field of the **Properties** window, type **dlgName**. It's important to give your dialog box a name that's descriptive and easy to remember because you need this information to interact with the dialog box in code.
3. In the **Dialog Tools** window, click **OK Button**. Now, click anywhere in the dialog box. IDEA adds a new `OKButton` control to your dialog box. The `OKButton` probably isn't in the right spot, but you can fix that easily by dragging the button to the upper right corner of the dialog box. Leave a little space, of course. If you want to position the control precisely, change the **Left** field in the **Properties** window to **100** and the **Top** field to **5**.
4. In the **Dialog Tools** window, click and drag **Cancel Button** to the dialog box. You'll notice that the mouse cursor changes to have a little plus sign next to the pointer. Drop the `CancelButton` control directly below the `OKButton` control. Some people like the point-and-click method used with the `OKButton` control and others like the drag-and-drop method used with the `CancelButton` control. Either method will add controls to your dialog box—use the method that works best for you.
5. **Ctrl+Click** the `OKButton` control. You'll see that IDEA has selected both the `CancelButton` and the `OKButton` controls. Click the **Align Lefts** button. The `CancelButton` control now appears directly below the `OKButton` control. Wasn't that easy?
6. **Ctrl+Click** the `OKButton` control. Notice that IDEA has deselected the `OKButton` control, but keeps the `CancelButton` control selected. In the **Top** field of the **Properties** window, type **25**. The `CancelButton` control moves an appropriate distance below the `OKButton` control.
7. Add a `StaticText` control to the dialog box using one of the techniques you used for the other controls. In the **Properties** window, type **&Type Your Name:** in the **Title** field, **5** in the **Left** field, **45** in the **Width** field, **5** in the **Top** field, and **9** in the **Height** field. Making the `StaticText` control shorter lets you place an associated control closer to it without potentially hiding part of that other control.
8. Add an `EditBox` control to the dialog box. In the **Properties** window, type **80** in the **Width** field and **txtName** in the **ID** field. Making the `EditBox` wider will make it easier for the user to type their name.

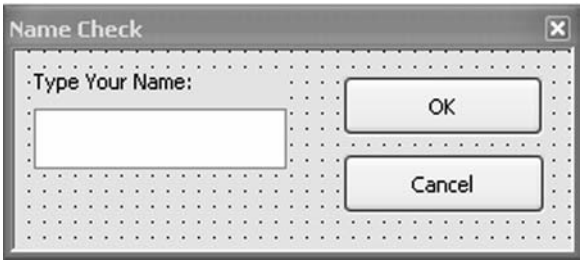


FIGURE 12.11 The Basic Dialog Box Used for This Example

9. **Ctrl+Click** the `StaticText` control. IDEA selects both the `EditBox` and the `StaticText` controls. On the Standard toolbar, click the **Align Lefts** button and then the **Remove Vertical Spacing** button. The `EditBox` control now appears directly below its associated `StaticText` control.
10. In the designer window, click the dialog box to select it. In the **Height** field of the **Properties** window, type **60**. IDEA resizes the dialog box so that the dialog box is now the right size to hold the controls you added. Normally, you'll want to resize your dialog box as needed to ensure it doesn't look too big or small. Figure 12.11 shows how your dialog box should look at this point. See, it's actually easy and fun to create a dialog box!

If you run the example at this point, nothing will happen. Sure, the dialog box is there, but you haven't told IDEA to do anything with it—not even display it. Even a basic example has five steps that you normally consider in the code:

1. Create the dialog box object.
2. Perform any required control configuration.
3. Display the dialog box and wait for the user to do something.
4. Check the dialog box result.
5. Act on the dialog box result.

It's important to remember that this is a very basic dialog box. Many of the dialog boxes you create will perform more work, but look at Listing 12.2 for now to see how you'd handle these basic steps.

The code begins by creating the dialog box object, `MyDialog`. Notice that you use the dialog box's `Name` property to create the object. That's why you want to use a name that's easy to remember and descriptive.

The next step is to configure controls. The `EditBox` control is blank right now. One way to help users know what you expect them to type is to provide a sample value or a descriptive prompt in the `EditBox` control. In this case, the code adds `Your Name Here` in the `EditBox` control. Don't worry, when the user types their name, they'll replace the `Your Name Here` value with their actual name (at least, you hope they will).

LISTING 12.2 A Simple Dialog Box Example

```
Sub Main
   ' Create the dialog
   Dim MyDialog As dlgName

   ' Type in some default text.
   MyDialog.txtName = "Your Name Here"

   ' Display the dialog on screen.
   Dim Result As Long
   Result = Dialog(MyDialog)

   ' Make sure the user hasn't clicked Cancel.
   If Result = 0 Then
      Exit Sub
   End If

   ' Obtain the selection and display it.
   MsgBox "Your name is: " & MyDialog.txtName
End Sub
```

Once it configures the controls, the code can display the dialog box using the `Dialog()` function. At this point, the code waits for the user to type something in the `EditBox` control and then click either the OK button or the Cancel button. The `Dialog()` function places the value of the button the user clicks in `Result`.

Now your code is on hold, waiting for the user to do something. While you're waiting, press Alt+T. You'll see that IDEA selects the `EditBox` control for you. After the user clicks a button, the code checks `Result`. Remember that the `CancelButton` control produces an output of `0`, so if the user clicks Cancel, the application exits using `Exit Sub`.

If the user does click the OK button, the code displays the value they typed in the `EditBox` control. Notice that in writing to or reading from the `EditBox` control, you must go through the dialog box object, `MyDialog` as shown in the code.

Try this example again. This time, after you enter a value in the `EditBox` control, try pressing Enter. Notice that the application displays the message box with the value you typed because pressing Enter in the Type Your Name field is the same as clicking the OK button. Run the example a third time and this time press Esc. The application exits without displaying the message box. Pressing Esc is the same as clicking the Cancel button. Congratulations! Your first dialog box is a success!

Working with the Dialog Box Function Using Beep

You can create a wealth of interesting and useful dialog boxes using the techniques described in the previous section, but these dialog boxes have a limitation. The moment a user clicks a button, any button, the dialog box closes. Sometimes you want to provide a button that doesn't close the dialog box, but does something else instead.

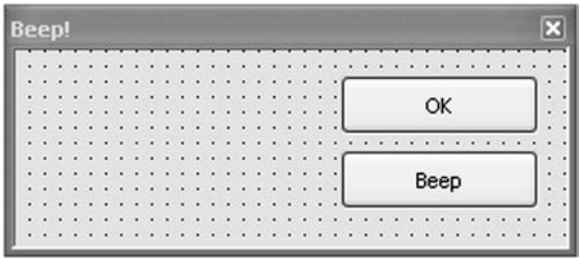


FIGURE 12.12 A Dialog Box with a Beep Button

Tip

If you plan to create a dialog box with controls that are similar to another dialog box, highlight the controls you want to use and then select Edit > Copy. This act copies the controls to the Clipboard. Now, select the new dialog box and select Edit > Paste. The controls you copied will appear in the new dialog box so you don't have to recreate them.

The example in this section does something simple. When the user clicks the `Beep` button, the application beeps. Consequently, this is also your first multimedia application, even though the multimedia is undeniably simple. To begin this example, you create a dialog box like the one shown in Figure 12.12. Table 12.5 lists the changes you need to make for the various controls.

At this point, you have a nice new dialog box designed. Listing 12.3 shows the code you'll need to add to make it functional.

The `Main()` subroutine does pretty much as you expect it to do. It creates the dialog box and then displays it on screen. However, `Main()` seems to be missing

TABLE 12.5 Beep Button Dialog Box Control Settings

Control	Property	Value
Dialog Box	Name	dlgTest
	Title	Beep!
	Height	70
	Function	HandleDlgTest
OKButton1	Left	100
	Top	5
PushButton	Title	&Beep
	Left	100
	Top	20
	ID	btnBeep

LISTING 12.3 Using Dialog Box Functions

```
Sub Main
   ' Create the dialog
   Dim MyDialog As dlgTest

   ' Display the dialog on screen.
   Dim Result As Long
   Result = Dialog(MyDialog)

   ' Display the result.
   MsgBox Result
End Sub

Function HandleDlgTest (ControlID As String, Action As Integer, _
                        SuppValue As Integer)
   ' Determine which button was clicked.
   If ControlID = "btnBeep" Then
      ' Don't close the dialog box if the user clicks Beep.
      HandleDlgTest = 1

      ' Sound a beep for the user.
      Beep
   Else
      ' If the user clicks OK (the only other button),
      ' close the dialog box.
      HandleDlgTest = 0
   End If
End Function
```

a few steps in this case. That's because of the entry in the dialog box `Function` property of `HandleDlgTest`. This property tells IDEA to look for a function named `HandleDlgTest()` whenever the user clicks a button.

Of course, the next part of the code shows the `HandleDlgTest()` function. This function accepts three input values:

- **ControlID:** Whenever the code calls `HandleDlgTest()`, it passes the value it finds in the `ID` property for the control. You can use the `ControlID` to determine what to do with the other information that IDEA provides.
- **Action:** The `Action` argument tells you what IDEA is doing at the time it calls your function. In some cases, you really don't care what action IDEA is doing (as in our example), but in other cases, the action is very important. Table 12.6 describes the various action values and what they mean.
- **SuppValue:** A few of the controls provide supplementary information during certain actions. You won't actually use this argument very often, but Table 12.7 tells you about the various supplementary values that your application could receive.

TABLE 12.6 IDEA Dialog Box Actions

Action	Description
1	The dialog box hasn't become visible yet. This action lets you perform control configuration and other pre-display tasks.
2	The user has clicked a button. The button the user clicked is supplied as the `ControlID` value.
3	The user has changed the value of a `TextBox` or `ComboBox` control. IDEA sends this action after the control loses focus, but before the user works with the next control. IDEA doesn't send this action unless the content of the `TextBox` or `ComboBox` control changes. Consequently, you can use this action to perform tasks such as validating the input users provide to ensure that it meets the criteria you set.
4	The user has pressed Tab to move to another control. `SuppValue` contains the numeric value of the control that's losing focus, while the `ControlID` contains the name of the control that will receive focus. You can't display a message box or any other dialog box while processing this action.

The `HandleDlgTest()` function begins by checking the `ControlID` value. You'll normally use a `Select...Case` structure to check through a list of controls, but using a simple `If...Then...Else` statement works fine this time.

When the user clicks Beep, IDEA passes the `ControlID` value of `btnBeep` to `HandleDlgTest()`. The code then sets the return value to `1`, which tells IDEA to keep the dialog box open. It also calls the `Beep` function and you hear a beep come from the computer. On the other hand, when the user clicks OK, you want to close the dialog box, so the code sets the return value to `0`.

TABLE 12.7 Supplementary Information Provided by IDEA

Control	Supplementary Value Passed
ListBox, DropListBox, ComboBox	Provides the number of the item that the user selected in the control, where the first item is `0`.
CheckBox	Contains `1` when the check box is checked and `0` when it's cleared.
OptionButton	Provides the number of the option button the user selected, where the first option button is `0`.
TextBox	Contains the number of characters in the `TextBox` control.
ComboBox	Contains the number of characters in the `ComboBox` control when the `Action` is `3`.
CommandButton	Provides the value of the button the user clicked. You obtain precisely the same information from the `ControlID` argument.

At this point, the application could process a return value in `Main()`. In this case, the only possible `Result` value is -1, which means the user clicked OK. In fact, that's the result the message box shows after the dialog box closes.

Selecting a Control Using DlgFocus

You hear a lot about focus in Windows applications. The *focus* simply means that the dialog box or control has the user's attention. When a user is working with a particular dialog box, that dialog box has the focus. The control that the user is currently interacting with has the focus. Sometimes you want to *set the focus* (change the control that the user is working with). For example, when an application starts, you want to start the user on the right path by giving the first control they should work with the focus. That's the purpose of the example shown in Listing 12.4. It gives the `txtFirstName` control the focus. This example uses the dialog box shown in Figure 12.13, which is simply a variation on the dialog boxes you created earlier.

LISTING 12.4 Setting the Control Focus

```
Sub Main
   ' Create the dialog
   Dim MyDialog As dlgTest

   ' Display the dialog on screen.
   Dim Result As Long
   Result = Dialog(MyDialog)

   ' Make sure the user hasn't clicked Cancel.
   If Result = 0 Then
      Exit Sub
   End If

   ' Obtain the selection and display it.
   MsgBox "Your name is: " & MyDialog.txtFirstName & _
      " " & MyDialog.txtMI & " " & _
      MyDialog.txtLastName
End Sub

Function HandleDlgTest(ControlID As String, Action As Integer, _
   SuppValue As Integer)

   ' Set the initial control focus.
   If Action = 1 Then
      DlgFocus "txtFirstName"
   End If

   ' The only two button options are OK and Cancel, so pass a 0.
   HandleDglTest = 0
End Function
```

FIGURE 12.13 A Dialog Box Containing Three EditBox Controls

The example begins by creating the dialog box object and displaying the dialog box. It's during the dialog box display process that IDEA calls `HandleDlgTest()` with an `Action` value of `1`. In this case, that means that the code calls `DlgFocus` with the name of the control that should receive focus, `txtFirstName`. Because there are only two buttons to consider, in this case, `HandleDlgTest()` always exits with a value of `0`. Now, the important thing to consider is that the focus is changed before the dialog box is displayed so the user doesn't even realize you changed the focus.

The rest of the example works as you might expect. When the user closes the dialog box by clicking OK, the example displays their name on screen. Otherwise, when the user clicks Cancel, the application exits without displaying anything.

Changing Control Values Using DlgValue, DlgVisible, and DlgEnable

Sometimes you need to manipulate the dialog box while it's running. For example, a user might select a check box that makes additional options available, as in the example shown in this section. In this case, when a user checks Provide Full Name, the dialog box displays the Middle Initial and Last Name fields as shown in Figure 12.14. Otherwise, the user sees only the First Name field. Because you probably want full information in this case, this example also shows how to check Provide Full Name before the dialog box appears (encouraging the user to provide full information).

In order to perform this task, the code must detect the state of Full Name and then change the enabled and visible states of the affected controls. Because the dialog box is running at the time, you must perform this task within `HandleDlgText`, rather than in `Main()` (which looks the same in this example as it does in Listing 12.4). Listing 12.5 shows how to perform this task.

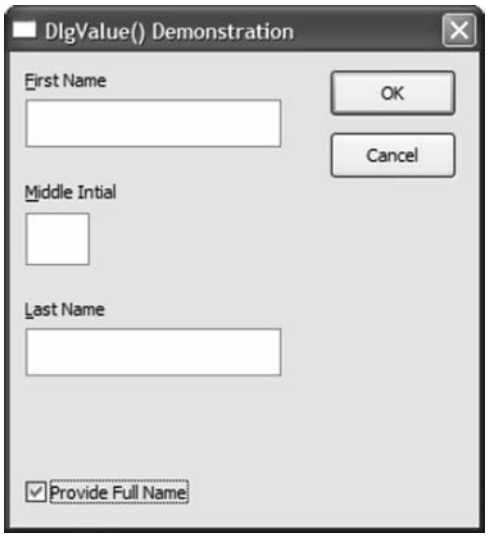


FIGURE 12.14 Changing the Appearance of a Dialog Box Based on Settings

The example begins as usual by creating and displaying the dialog box. In this case, before IDEA displays the dialog box, it calls `HandleDlgTest()` with `Action` equal to `1`. The code calls `DlgValue` with two arguments: the name of the control (`cbFullName`) and the value to set the control (`1`, or checked).

`Action 4` occurs every time one of the controls loses focus. It's important to remember that `ControlID` always contains the name of the control that's gaining focus and `SuppValue` contains the numeric identifier of the control losing focus. We want to check `cbFullName` as it loses focus in order to determine its checked status. To do this, you get the numeric identifier for `cbFullName` using the `DlgControlId()` function. This function accepts a control name as input and outputs its numeric value. Always use this approach, rather than trying to guess the numeric value of the control.

When `cbFullName` is losing focus the code checks its value using `DlgValue`. When `cdFullName` is checked, the code makes the Middle Initial and Last Name fields visible and enables them for use (you could simply disable the controls if desired). The code makes controls visible using the `DlgVisible()` function and it changes the control enabled status using `DlgEnable()`. Both functions accept the name of a control and its status as input (`1` for enabled or visible, `0` for hidden).

Adding Pictures to Your Dialog Boxes

Pictures can provide a bit of extra pizzazz for an application. You can use them for a host of purposes, including graphics for reports or simply the occasional bit of decoration. Fortunately, IDEA provides a limited amount of graphics capability in the form of the `Picture` control. This example uses a simple setup to demonstrate the `Picture` control

LISTING 12.5 Enabling and Disabling Controls

```
Function HandleDlgTest(ControlID As String, Action As Integer, _
   SuppValue As Integer)

   ' Encourage the user to enter the full name.
   If Action = 1 Then
      DlgValue cbFullName, 1
   End If

   ' Check for a change in full name status.
   If Action = 4 Then
      If SuppValue = DlgControlId("cbFullName") Then If DlgValue("cbFullName")
= 0 Then[CE Query: Move the following to flush left? NO. VM]

            ' Make everything invisible.
            DlgEnable "txtMI", 0
            DlgVisible "txtMI", 0
            DlgEnable "txtLastName", 0
            DlgVisible "txtLastName", 0
            DlgVisible "lblMI", 0
            DlgVisible "lblLastName", 0
         Else

            ' Make everything visible.
            DlgEnable "txtMI", 1
            DlgVisible "txtMI", 1
            DlgEnable "txtLastName", 1
            DlgVisible "txtLastName", 1
            DlgVisible "lblMI", 1
            DlgVisible "lblLastName", 1
         End If
      End If
   End If
End Function
```

as shown in Figure 12.15. In this case, you need `StaticText`, `EditBox`, `Picture`, `Button`, and `CancelButton` controls to create the required dialog box.

There's an emphasis on simplicity in this case. Working with most `Picture` controls is a grueling process because the developer normally gives you way too much functionality—most of which is never used. The IDEAScript version is simple enough that you really don't write much code to use it. The `Picture` control works in tandem with the `DlgSetPicture()` function as shown in Listing 12.6 to display graphics.

The example begins as usual by creating the dialog box and displaying the components. In many respects, the code you write is as simple as the Beep example earlier in the chapter. The only major difference is that you use `DlgSetPicture()` to display the requested image on screen.

FIGURE 12.15 You Can Use the Picture Control to Display Graphics in Your Application

LISTING 12.6 Adding Pictures to Your Application

```
Function HandleDlgTest(ControlID As String, Action As Integer, _
   SuppValue As Integer)

   ' Determine which button was clicked.
   If ControlID = "btnOpen" Then
      ' Don't close the dialog box if the user clicks Open.
      HandleDlgTest = 1

      ' Display the selected picture.
      DlgSetPicture "Picture1", DlgText("txtFilename")
   Else
      ' If the user clicks Cancel (the only other button),
      ' close the dialog box.
      HandleDlgTest = 0
   End If
End Function
```

The `DlgSetPicture()` function accepts two input values. The first is the name of the control you want to use, which is `Picture1` in this case. The second is the path of the picture you want to display.

Notice that this example uses `DlgText()` to obtain the path information from `txtFilename`. You can use this particular function whenever you need to know the value contained on an onscreen control. The amazing thing about this example is that it really is quite simple—you can add graphics without all the frustration normally associated with working with graphics on computers.

Summary

This chapter has demonstrated the basics of working with dialog boxes in IDEAScript. At this point, you have the tools required to create just about any dialog box you might ever need. However, it's important to realize that you can quickly become mired in a world of detail when it comes to dialog boxes. Authors have written entire books on just dialog box design—some of them quite thick—and not covered the topic entirely. The most important concept you can take away from this chapter is to keep things simple. Let others worry about making the dialog boxes fancy—you keep them simple to use and you'll have happy application users.

Now that you've seen a few examples of how to work with dialog boxes, it's time to try a few experiments of your own. For example, try working with each of the controls and discovering how it works best for your particular needs. Do things like trying out various control sizes. Check for ease of reading versus efficient use of screen real estate (larger characters are easier to read, but you can present fewer controls on a dialog box when you use large characters). Most importantly, go back to some of the previous examples in the book that relied on message boxes and input boxes. Try using custom dialog boxes for some of these examples to see if using a custom dialog box would make the example easier to use and understand.

Chapter 13 is going to put some of what you discovered in this chapter to use. In this chapter, you begin searching for information in a database. It's a very common task, so it's one of the more important topics in the book. However, many people focus on clever search techniques and forget the most important element of all—the search form. A search form is simply another kind of dialog box. You take the information you learned in this chapter to discover techniques for creating great search forms in Chapter 13.

CHAPTER 13

Locating Information in Databases

Finding what you want can be a problem. You've probably had the experience of remembering seeing some piece of data (consider it the needle), but not knowing where it appears in the haystack of data. Depending on the size of the database, you could potentially spend days looking for just that piece of data and it's not assured that you'll even find it. Looking for data can be frustrating, even using a computer. Everyone needs to find data and they usually need to find it quickly, so it shouldn't surprise you that one of the tasks you'll want to automate regularly is locating information.

There's no answer that satisfies every search need, but this chapter explores some of the ways in which you can use automation to make looking for data considerably easier. In some cases, it's simply a matter of knowing where to look and in others, it's a matter of knowing how to look. For example, choosing the right field to search can make a big difference. Rather than look for some obscure number, you might choose to look for easily remembered character data instead. Using wildcards (special symbols that can represent any character or phrase) can also make searches significantly easier when you use them effectively.

Performing Searches Efficiently

No matter where you search for data, you have to consider the best way to find what you want. Searching for data with IDEA isn't quite the same as searching with other products, such as a word processor, because of the amount of data typically involved. In addition, it's quite possible that your search will include multiple hits and you might even need to create an extraction database in order to hold all of the results. Database searches are also unique in that you must search across multiple fields and records. In short, you need a special kind of search mechanism when working with databases.

The manual IDEA Search task may look somewhat complex to you, as shown in Figure 13.1. This dialog box is necessary to provide you with a full range of search options. As you can see from Figure 13.1, you can search across multiple fields, in multiple databases, and even output the results to an extraction database. Some of the complexity is hidden. You can use special terms and characters to produce well-defined results from the search (see the "Using the Built-in Search Features" section for details). In short, if it can be found, the Search dialog box helps you find it.

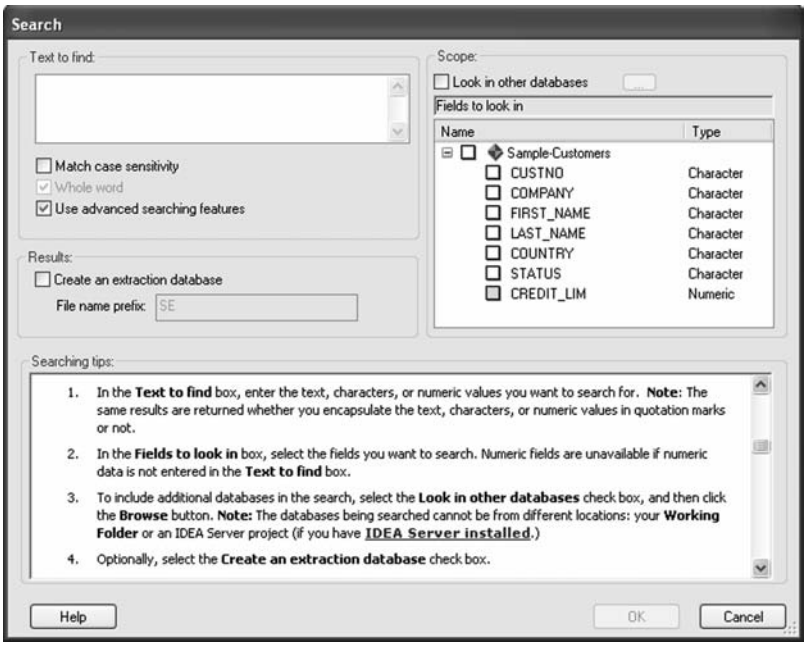


FIGURE 13.1 The Standard Search Dialog Box May Look a Little Complex, but Provides Essential Functionality

Unfortunately, the Search dialog box might prove to be a little too complex for the typical user in your organization. Using automation can help you simplify things and make the user feel a lot more comfortable. In some cases, you know things about your organization that you can include as part of an application—things that a user might not consider, but you know they should. You could also make it easier to perform specialized searches, such as proximity searches, that will improve the results less skilled users obtain.

The best way to become more efficient at searching is practice. Read about the various techniques in the "Using the Built-in Search Features" section, and then practice them on your own databases. It won't take long to figure out methods for obtaining the results you want using the right search term. Remember, the fewer outputs you have to search, the faster you'll find the data you want. Sometimes you simply have to work through a number of "what if" scenarios before you begin to understand how searches work and the best way to use them in your organization.

Write down the search scenarios you find successful. You can use this information when designing your application. Everyone finds it helpful when the application can make useful suggestions as to how to locate information more quickly. Even you will forget some of the things you learn and having them part of your application will save you time too!

Using the Built-in Search Features

IDEA provides a significant number of built-in search features. In fact, you can often find a particular record or group of records using more than one approach simply because there are so many ways to conduct a search. One of the things to consider as you work through the basics of performing a search is to devise methods that work best for you. Not everyone thinks about data in the same way. A search that makes perfect sense to one person may leave someone else baffled.

The following sections help you understand the built-in search features. After you complete these sections, spend some time searching for things in the sample databases. You'll likely find that a particular search technique works best for you. When you discover this technique, you can begin putting it down into code, then customizing it as needed to perform searches using fewer criteria and an easily understood search dialog box of your own design.

Creating a Basic Search

A basic search begins with the data fields. The search addresses the content directly. If you want to find a particular name, you search for the name directly, rather than search for a range of particular values. Likewise, you search for numeric values and dates directly, rather than rely on ranges of values or equations to express the value you need.

Basic searches are straightforward and easy to understand. You normally obtain the results of a basic search quickly. The basic search is the basis for other search types that add to the elements of a basic search. However, basic searches are also inflexible and might not find the data you want. Consequently, you often need to perform a more advanced search to locate the information you want without spending days doing it.

Note

IDEA lets you perform searches across databases. However, the databases must be in the same location, such as your working folder or an IDEA Server project. The examples in this section focus on a single database for the sake of clarity. It's important to know how to perform the task using a single database before you move on to working with multiple databases.

The following sections describe the basic searches based on field data type. Each of the sections provides a small demonstration on one of the sample databases. You'll also find a short code sample showing how you'd perform the same search using IDEAScript. These elements combine to help you better understand how the searches work so you can use them with greater efficiency later.

SEARCHING CHARACTER FIELDS From the perspective of looking for something, searching a character field is simple. Type the information you want to find in the Text to find field

of the Search dialog box, and then check one or more character fields in the Fields to look in list.

When working with a character field, you can also choose to look only for whole words and you can also make the search case sensitive. For these simple searches, you may want to clear the Use advanced searching features option. Using an extraction database to hold the results of the search will make it easier to code the search as a macro.

Whole word searches are helpful because you won't find words that contain the word you want to find. A whole word search often reduces the number of false leads you have to dig through to find the one hit that you really wanted. For example, instead of finding all of the cities in Oregon, you'll find Ore City, Texas. However, whole word searches can also reduce the chance of finding something you need. For example, you may need to find every variation of a particular word by searching for a word subset. Adding too much search precision can be as bad as not having enough.

Case-sensitive searches are helpful in finding words with the correct capitalization. For example, you might want to find Ore City, Texas, not the unrefined metal. However, a case-sensitive search assumes that no one has entered words using the wrong case. Someone might have entered Ore City, Texas as "ore city, texas". If you use a case-sensitive search, you won't find the entry that uses the incorrect case.

Note

It doesn't matter whether you place search values in quotes or not when working in the Search window. However, when you create code to perform the search, the quotes are important. IDEAScript will register an error if you attempt to create a search macro that doesn't include the quotes.

Now that you have the basics of a character field search in mind, it's time to view the code used to perform one. Listing 13.1 shows a typical example of creating a simple character field search.

The example begins by obtaining the path to the databases used for the example. It then deletes the old copy of the search output database, if one exists. The next step is to open the search database, which in this case is `Sample-Customers.imd`. If there aren't any errors opening the database, the code proceeds to building the task.

All of the searches you create rely on the `db.Search` task. This example adds a character field to the list, which is COMPANY for the example. You can add as many fields as necessary to perform the search. The example also provides a `RecordFilesPrefix` property value of `Search`, which tells IDEA to create an extraction database, `Search-Sample-Customers.imd`.

Once the task is configured, the code calls `DoSearch.PerformTask()`. The import argument, at this point, is the first one, which contains the search word. Notice that the word appears in double quotes. The remaining three arguments are set to `0` for this example.

Here are the uses for each argument:

- **Search String:** Contains the string that you want to find.
- **Case Sensitive:** Performs a case-sensitive search when you set this argument to `1`. In this case, the search would find companies that have Jewel in their name, but not companies that have JEWEL in their name.
- **Whole Word:** Looks only for whole words when set to `1`. In this case, the search would find companies with the word Jewel in their name, but not companies that have Jewelry or Jewels in their name.
- **Advanced Features:** Performs a search that requires advanced features. Use a value of `1` to search with advanced features and a `0` to search without them.

LISTING 13.1 Performing a Simple Character Field Search

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Delete the old file.
   DeleteOld Path & "Search-Sample-Customers.imd"

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path & "Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Set the task type.
   Dim DoSearch As Task
   Set DoSearch = db.Search

   ' Configure the task.
   DoSearch.AddFieldToInc "COMPANY"
   DoSearch.RecordFilesPrefix = "Search"

   ' Perform the task.
   DoSearch.PerformTask "Jewel", 0, 0, 0

   ' Open the result.
   OpenDB(Path & "Search-Sample-Customers.imd")

   ' Clear memory.
   Set DoSearch = Nothing
   Set db = Nothing
End Sub
```

(*continued*)

LISTING 13.1 (*Continued*)

```
Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub

Function OpenDB(DBPath As String) As Database
   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

The code ends by opening the result database, as shown in Figure 13.2. It then clears up memory before it exits.

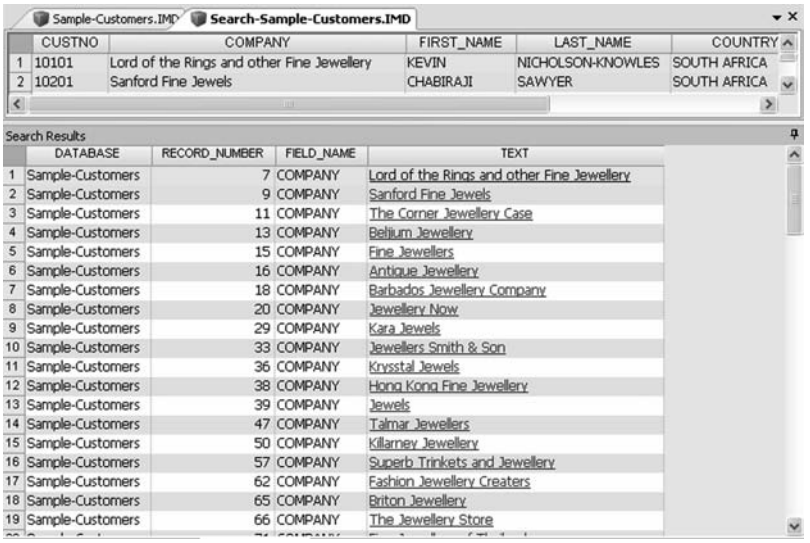


FIGURE 13.2 The Search Results Show What Happens When You Look for the Word Jewel in the COMPANY Field.

SEARCHING NUMERIC FIELDS Look at the Search dialog box shown in Figure 13.1 again. Notice that the only numeric field in the list, CREDIT_LIM, is not enabled. Don't reach the wrong conclusion; you really can perform numeric searches. The CREDIT_LIM entry in the Fields to look in field becomes enabled when you enter a number in the Text to find field. Of course, your user might not know this little fact and decides that numeric searches aren't allowed. Your custom code can help solve this problem by making numeric searches easy to perform.

Numeric searches can't contain spaces. You must enter the number without spaces, such as 12.3. A number can include the decimal point, but you don't provide thousands separators, so 1,000 would appear in a search as 1000. It's also possible to search for negative values—just add a minus sign in front of the number like this, −4567.8. Listing 13.2 shows you how to perform a numeric search.

LISTING 13.2 Performing a Simple Numeric Field Search

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Delete the old file.
   DeleteOld Path & "Search-Sample-Customers.imd"

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path & "Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Set the task type.
   Dim DoSearch As Task
   Set DoSearch = db.Search

   ' Configure the task.
   DoSearch.AddFieldToInc "CREDIT_LIM"
   DoSearch.RecordFilesPrefix = "Search"
      ' Perform the task.
   DoSearch.PerformTask "1000", 0, 0, 0

   ' Open the result.
   OpenDB(Path & "Search-Sample-Customers.imd")

   ' Clear memory.
   Set DoSearch = Nothing
   Set db = Nothing
End Sub
```

(*continued*)

LISTING 13.2 *(Continued)*

```
Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub

Function OpenDB(DBPath As String) As Database
   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

The majority of this example is the same as the simple character search. However, there are some important differences. The first major difference is that the search looks in the CREDIT_LIM field instead of the COMPANY field.

The second major difference is that the `DoSearch.PerformTask()` arguments are different. For the example, the numeric value you want to find is `1000`. Notice that the `1000` contains no thousands separator. The search can't be case sensitive or rely on a whole word, so these two arguments are set to `0`. A numeric search also can't use advanced features such as Boolean operators, so the advanced features argument should normally be set to `0`. Figure 13.3 shows the output from this example.

SEARCHING DATE FIELDS Date fields represent an important kind of search because time is a focus for most people. Unlike numeric fields, data fields are always enabled as a search target in IDEA. However, in order to search a date field, you must enter the information in YYYYMMDD format (four-digit year, two-digit month, and two-digit day). For example, if you want to find 23 May 2010, you must provide the search date as 20100523. The search would output an empty result database if you provided a value of 2010523 because you didn't enter a two-digit month.

This example uses the `Sample-Bank Transactions` database. It searches for records of transactions that occur on 12 January 2008. Listing 13.3 shows you how to perform this task.

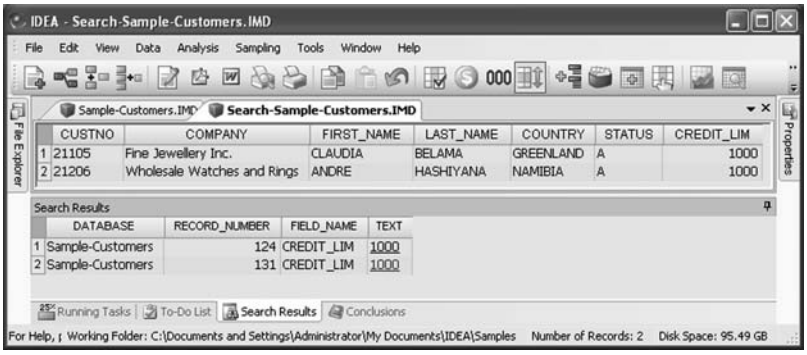


FIGURE 13.3 The Search Results Show What Happens When You Look for the Number 1000 in the CREDIT_LIMIT Field

LISTING 13.3 Performing a Simple Date Field Search

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Delete the old file.
   DeleteOld Path & "Search-Sample-Bank Transactions.imd"

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path & "Sample-Bank Transactions.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Set the task type.
   Dim DoSearch As Task
   Set DoSearch = db.Search

   ' Configure the task.
   DoSearch.AddFieldToInc "DATE"
   DoSearch.RecordFilesPrefix = "Search"

   ' Perform the task.
   DoSearch.PerformTask "20080112", 0, 0, 0

   ' Open the result.
   OpenDB(Path & "Search-Sample-Bank Transactions.imd")

   ' Clear memory.
   Set DoSearch = Nothing
   Set db = Nothing
End Sub
```

(*continued*)

LISTING 13.3 (*Continued*)

```
Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub

Function OpenDB(DBPath As String) As Database
   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

This example works very much like the character search shown in Listing 13.1. There are some differences, of course, such as the target database, the search field, and the search value that appears as part of `DoSearch.PerformTask()`. The element that makes this a date search (for this example) is the format of the search string. (Theoretically, the same string could find data in a character or numeric field, assuming that the field has the required information.) As you can see, the string is in the date format consisting of the four-digit year, two-digit month, and two-digit day. Figure 13.4 shows the output from this example.

Using Boolean Operators

A *Boolean operator* describes a logical relationship between two items. Boolean operators let you look for records using multiple criteria. For example, you could look for records that contain both Jewel and Fine in the COMPANY field. Only when a record contains both strings within the specified field will IDEA add it to the result. In this case, you'd use the `AND` Boolean operator. Of course, you might not be looking for just one complex string—you might want any record that contains one of a number of strings, in which case, you'd use the `OR` Boolean operator instead of the `AND` Boolean operator. IDEAScript supports the following Boolean operators.

- **And:** The records must meet both of the criteria you supply to appear as part of the result.

- **Or:** The records can meet either of the criteria you supply to appear as part of the result. For example, if you provide `Jewel` or `Fine` as the criteria, then any record that contains either `Jewel` or `Fine` will appear as the output.
- **Not:** The records must not meet the criteria you supply to appear as part of the result. For example, if you provide `Not Jewel` as the criteria, then the result will contain all of the records that don't contain the word Jewel in them.
- **Xor:** The records can meet either the criteria you supply, but not both criteria. For example, if you provide `Jewel XOR Fine` as the criteria, then the result will contain all of the records that have Jewel or Fine in the requested field, but not both strings. As a specific example, if you perform an `XOR` search on the COMPANY field of the `Sample-Customers` database using `Jewellery XOR Fine` as the criteria, then you'd see `Beljium Jewellery` and `Johnson Bancock Fine Collectibles` in the output, but not `Lord of the Rings and Other Fine Jewellery`.

Note

In order to perform a Boolean search, you must use the Advanced Features option shown in Figure 13.1. Checking this option automatically checks the Whole Word option as well. Consequently, you must provide precise strings for your search. Creating a search for `Jewel` won't find results that contain `Jewellery`.

It's also possible to combine Boolean operators to create complex searches. For example, you can combine `Jewellery` and `Fine not Thailand` as criteria. In this case, records that have both `Jewellery` and `Fine` in them will appear in the results as long as the result doesn't also include `Thailand`. A Boolean search can become quite complex and provide you with specific results. Listing 13.4 shows how to use a Boolean search to find specific results.

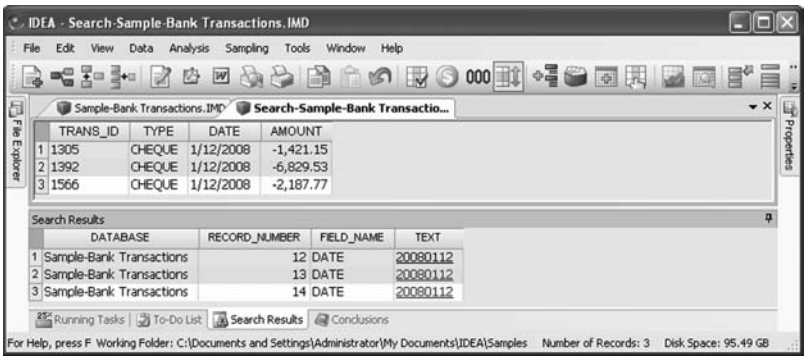


FIGURE 13.4 The Search Results Show What Happens When You Look for 12 January 2008 in the DATE Field.

LISTING 13.4 Using Boolean Operators in a Search

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Delete the old file.
   DeleteOld Path & "Search-Sample-Customers.imd"

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path & "Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Set the task type.
   Dim DoSearch As Task
   Set DoSearch = db.Search

   ' Configure the task.
   DoSearch.AddFieldToInc "COMPANY"
   DoSearch.RecordFilesPrefix = "Search"

   ' Perform the task.
   DoSearch.PerformTask "Jewellery AND Fine NOT Thailand", 0, 0, 1

   ' Open the result.
   OpenDB(Path & "Search-Sample-Customers.imd")

   ' Clear memory.
   Set DoSearch = Nothing
   Set db = Nothing
End Sub

Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub
```

```
Function OpenDB(DBPath As String) As Database
   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

The code for this example works much like the code shown in Listing 13.1. However, it does have some interesting differences. For example, notice that the code uses criteria of `Jewellery AND Fine NOT Thailand` against the COMPANY field. The code uses uppercase for the Boolean terms to make them easier to see, but IDEAScript will accept lowercase (or any other case for that matter). Using uppercase is a convention that makes the Boolean operators easier to see and serves no other purpose.

This is an advanced search, so the advanced features argument is set to `1` instead of `0`. You don't have to set the whole word argument to `1` because the search always uses whole words. Figure 13.5 shows the results of this example. Compare this result to the result shown in Figure 13.2. As you can see, creating a complex Boolean expression has greatly reduced the number of records in the result.

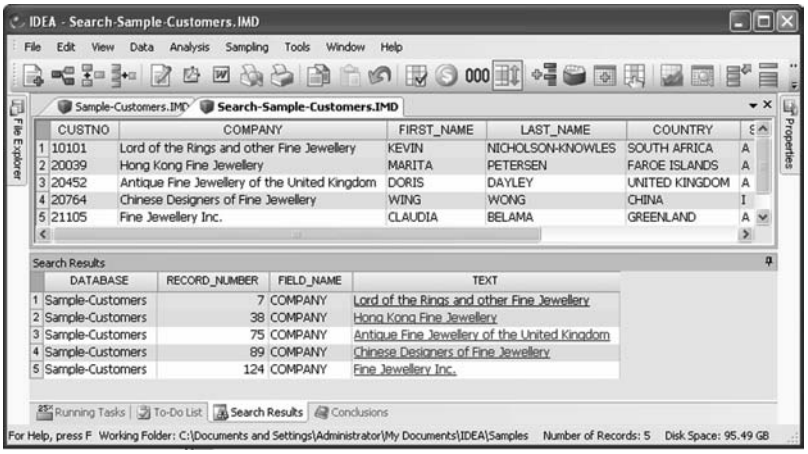


FIGURE 13.5 This Search Combines Multiple Simple Searches Using Boolean Operators

Note

You can't perform Boolean, wildcard, or proximity searches on numeric fields.

Using Wildcards

Wildcard searches are a kind of pattern matching, where you describe a pattern of characters to find and IDEA locates them for you. IDEA supports two wildcard characters:

- *** (asterisk):** The asterisk matches any number of characters.
- **? (question mark):** The question mark matches just one character.

The position of the wildcard character makes a difference in how IDEA views it during the search. For example, `*day` would match any day of the week: "Sunday," "Monday," "Tuesday," "Wednesday," "Thursday," "Friday," or "Saturday." Likewise, `sun*` would match "sunshine" or "sunny". You can even place the wildcard character in the middle. Specifying `t*y` would match words such as "today" and "toy". You can even double the wildcard characters. Using `*@*.com` would match any e-mail address with a .com domain, such as "george@mycompany.com." However, it wouldn't match "sam@anothercompany.net."

Sometimes you want to be a little more precise than matching any number of characters. In this case, you use the question mark in place of the asterisk. For example, `t?y` would match "toy" and "try," but it wouldn't match "today" because "today" has more than one character between the "t" and the "y." If you want to specifically match "today," you could specify `t???y`. In this case, the search wouldn't match "toy" or "try" because they don't have enough characters between the "t" and the "y." Naturally, the question mark can appear anywhere in the search term, just like the asterisk.

You can use these characters separately or in combination. In many cases, you can obtain more precise search results using the question mark and asterisk in combination. For example, `???@*.com` would match any three-letter name in the .com domain. It would match "ann@mycompany.com" and "sam@anothercompany.com," but wouldn't match "nelly@mycompany.com" because "nelly" contains too many letters.

This example shows how you can create a unique search using wildcards. If you perform a case-sensitive search for the word `Watches` in the COMPANY field of the `Sample-Customers` database, you get 54 results. However, let's say you want to use a wildcard to look for companies that begin with the letter `C`, have six characters in their name, and also have the word `Watches` in the name. You'd use `C????? Watches` as your search. In this case, you only get four return values:

- Custom Name Watches
- Happy Corner Watches
- Irish and Celtic Watches
- Custom Watches

In all four cases, the `C` word has six characters in its name and appears before the word `Watches`. Notice that the `C` word doesn't necessarily appear at the beginning of the company name. In addition, the `C` word and `Watches` do not need to appear together—they only need to appear in the order you specified. Listing 13.5 shows the code used to create this example.

LISTING 13.5 Using Wildcards in a Search

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Delete the old file.
   DeleteOld Path & "Search-Sample-Customers.imd"

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path & "Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Set the task type.
   Dim DoSearch As Task
   Set DoSearch = db.Search

   ' Configure the task.
   DoSearch.AddFieldToInc "COMPANY"
   DoSearch.RecordFilesPrefix = "Search"

   ' Perform the task.
   DoSearch.PerformTask "C????? Watches", 1, 0, 1

   ' Open the result.
   OpenDB(Path & "Search-Sample-Customers.imd")

   ' Clear memory.
   Set DoSearch = Nothing
   Set db = Nothing
End Sub

Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )
```

(*continued*)

LISTING 13.5 (*Continued*)

```
   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub

Function OpenDB(DBPath As String) As Database
   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

This example works much like the example shown in Listing 13.1. However, the search term provided as input to `DoSearch.PerformTask()` is different for this example, `C????? Watches`. In addition, notice that both the case-sensitive and the advanced search arguments are set to `1`. Figure 13.6 shows the output from this example.

Performing Proximity Searches

In some cases, a standard search will obtain results you don't want because the words don't appear in the required proximity. For example, you might want to see `Custom`

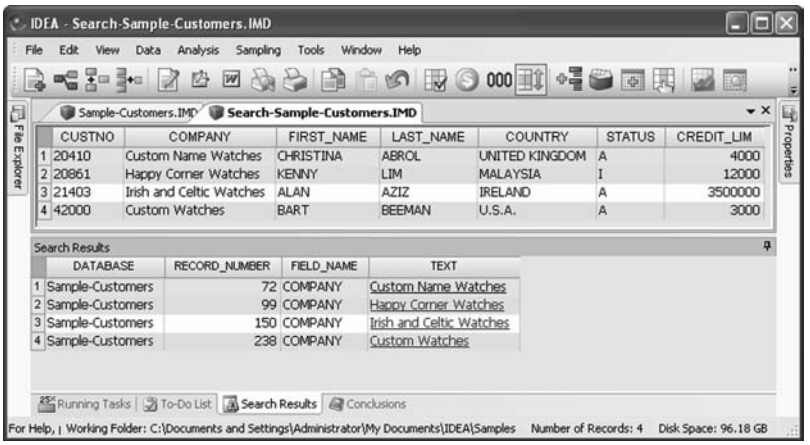


FIGURE 13.6 This Search Combines Multiple Simple Searches Using Wildcards

`Watches` in the search results, but not `Custom Name Watches` because the search terms are too far apart. In this case, you can perform a proximity search to obtain results closer to the ones you want.

A proximity search can search within a specific field (called a word proximity search) or between records. For example, you might have a database that contains groupings of records, and you can use a proximity search to look for records within the group that match specific criteria.

Word proximity searches can also be ordered or unordered. When using an ordered word proximity search, the order of words in your search term matters. An unordered proximity search simply finds the words in your search term in any order.

Each proximity search type uses a special symbol. You surround the search term(s) within quote marks, then place the symbol outside the search term and follow it by a number that specifies the search range. Here are the symbols used for each of the proximity search types.

- **/:** Specifies an ordered word proximity search.
- **@:** Specifies an unordered word proximity search.
- **#:** Specifies a record proximity search.

The range of a proximity search begins with the first word found. As a result, when working with an ordered word proximity search, the first word in your search term begins the search range. When working with an unordered word proximity search or a record proximity search, the first word found in the search term begins the range.

Now that you have some basics, it's time to look at a few specific examples. Let's use the COMPANY field of the `Sample-Customers` database as a starting point and the terms `Design` and `Jewel`. Proximity searches require precise terms unless you supply a wildcard, so the actual terms will be `Design*` and `Jewel*` to match all variations of Design and Jewel. Starting with an ordered word proximity search with a range of 3, `"Design* Jewel*"/3`, you get two records to match. Let's try the search term the other way around. `"Jewel* Design*"/3` also outputs two records, but two different records. To see all four records at once, try an unordered search of `"Jewel* Design*"@3`. Here are the four records:

- Hans Andersen Designer Jewellery (First ordered search)
- Personal Designs Jewelry (First ordered search)
- Girl's Best Friend Jewellery Design (Second ordered search)
- Finnish Jewellery Designers (Second ordered search)

You can see how the three different searches obtained the results they did. Let's see what happens when you increase the range to 4. When using an unordered word search, `"Jewel* Design*"@4`, you get five records as output with the addition of `Chinese Designers of Fine Jewellery`. This means that an ordered word search of `"Design* Jewel*"/4` will increase to three records.

Just to see what happens, try a record proximity search of `"Design* Jewel*"#2` to see where records that have `Design* or Jewel*` are next to each other. In this

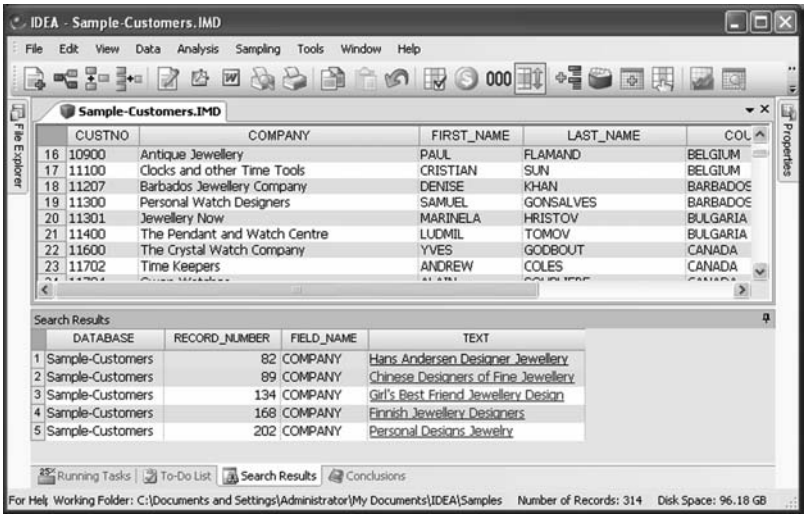


FIGURE 13.7 This Search Combines Multiple Simple Searches Using Proximity Operators

case, you get 9 records as output, as shown in Figure 13.7. When a record range value is 1, it means that both terms appear in the same record. So, in this case, because `Barbados Jewellery Company` is next to `Personal Watch Designers`, `Barbados Jewellery Company` appears in the results. Listing 13.6 shows how you'd create a proximity search.

LISTING 13.6 Using Proximity Operators in a Search

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Delete the old file.
   DeleteOld Path & "Search-Sample-Customers.imd"

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path & "Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Set the task type.
   Dim DoSearch As Task
   Set DoSearch = db.Search
```

```
   ' Configure the task.
   DoSearch.AddFieldToInc "COMPANY"
   DoSearch.RecordFilesPrefix = "Search"

   ' Perform the task.
   DoSearch.PerformTask """Jewel* Design*""@4", 0, 0, 1

   ' Open the result.
   OpenDB(Path & "Search-Sample-Customers.imd")

   ' Clear memory.
   Set DoSearch = Nothing
   Set db = Nothing
End Sub

Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub

Function OpenDB(DBPath As String) As Database
   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

This example works very much like the one shown in Listing 13.1, except for the search term supplied with `DoSearch.PerformTask()`, `"""Jewel* Design*""@4"`. This search term may look a little complicated until you take it apart. Begin with the search term itself:

```
"Jewel* Design*"
```

To this search term you add another set of double quotes to show that it's a string within a string:

```
""Jewel* Design*""
```

You then add in the `@4` to show the kind of proximity search to perform:

```
""Jewel* Design*""@4
```

The final step is to place the entire search term within a string so that IDEAScript can interpret it:

```
"""Jewel* Design*""@4"
```

Use this pattern for any proximity search you perform. Figure 13.7 shows the output from this example.

Creating a Custom Search

The "Using the Built-in Search Features" section shows how to use all of the individual search features that IDEA provides as IDEAScript applications. You can mix and match search techniques as needed to locate specific information based on user input. A single search could combine wildcard and proximity searches, as shown in Listing 13.6. You might decide to perform a numeric search first, and then a character string or date search next to locate specific records. Perhaps you want to find all of the companies that spent over $10,000 in the time frame between Jan 1, 2010 and Jan 31, 2010.

Your application could use a wizard-like approach to perform the task. A search could start with a wildcard or simple character string search. The output could appear in an extraction database that the user could review. Your application would simply display a small dialog box that the users could move out of the way until they are ready for the next step.

The next step of your wizard could ask whether the kind of data the user wants appears in the output or whether the application needs to start searching again. If the user is pleased with the preliminary result, you might ask questions to refine the search and output a second extraction database. Perhaps the second phase would use a proximity search to locate records that have a particular relationship to each other or fields that contain a certain kind of word structure.

At each stage your application could offer to move forward to refine the results or move back to the previous results. Using this application programming technique means that the user is going to focus on the results, not on how to perform the search. All logic required to perform the actual search resides within your application.

Summary

This chapter has shown how to create searches using IDEAScript. As you now know, the secret to finding what you need is to know what you want, define what you want, where to find it, and how to search for it in the most efficient manner. Automation can make searches considerably faster by improving search technique and performing some tasks automatically.

For many people, performing searches efficiently doesn't come easy because they don't have a method that works well for them. The only way to develop a good technique is to practice. Define a list of things you want to find, then search for them using the approaches described in this chapter. Find the one or two methods that work best for you. It's important to remember that different kinds of searches require different techniques. If nothing else, design a cheat sheet that lists the techniques you've tried and work best for you.

Chapter 14 moves on to another common database task, importing data from other sources. Most of your analysis will rely on obtaining data from outside sources—from common applications used by other people. In order to use this data, you need to know how to get it into IDEA. Of course, once you complete an analysis, you might want to send it to someone else, which could mean exporting it to yet another format. Chapter 14 is all about moving data around as needed to communicate well with other people.

CHAPTER 14

Importing and Exporting Data

You won't generate new data in IDEA for the most part. In most cases, you obtain data to analyze from an external source of some kind—probably not IDEA. Because you'll perform this task relatively often, creating applications to import data is one of the biggest time savers. Using an application to perform the task will also make things considerably easier. You won't have to remember every nuance of importing data you need—all you need to do is figure things out once, then let the application do it for you. In short, creating applications to import data is a double win for anyone using IDEA. Fortunately, IDEA makes it easy to import data in just about any format.

At some point, you'll probably have to export the data to send to someone else. After all, once you analyze the data, you'll want to share the results with someone—that's where exporting the data to another format comes into play. In addition, you might need to export data when you want to share it with other people using a common format. You'll sometimes need to export data to create reports. For example, you might want to create a complex graph or chart of the data. IDEA provides a significant number of methods for exporting your data to another data format.

This chapter discusses both importing and exporting data. This is perhaps one of the more important chapters in the book because it's a task that just about everyone will perform at some point when using IDEA.

Considering the Import and Export Features

IDEA provides a wealth of import and export features. In fact, IDEA supports a number of the most common formats such as Microsoft Access, Microsoft Excel, dBASE, and Portable Document Format (PDF). IDEA tries to make importing and exporting data as easy as possible. However, sometimes the details can prove a little overwhelming to even seasoned users, so your application will likely streamline the process for most users. You'll likely use your application to ensure the user imports and exports data using the techniques preferred by your organization. In doing so, you also make things easier for the user because the user will need to make fewer decisions. The following sections provide an overview of the import and export tasks that IDEAScript provides.

Even though each data import requires unique information, they follow a predictable pattern:

1. Create the required task.
2. Specify an output file name.
3. Specify input file names as needed.
4. Perform the import task.
5. Perform any required cleanup.
6. Open the imported database.

The examples in this section use the files located in the\IDEA\Samples directory found in your My Documents (or equivalent) directory. Make sure you have the samples available when trying out one of the import or export examples in the following sections.

Tip

IDEA doesn't support absolutely every application file format on the market—no one could do that. However, IDEA does support the most common formats. If you use an application that IDEA doesn't support directly, consider exporting the data from the application in a common format, and then importing the common format file into IDEA. In most cases, this double conversion technique works fine and you won't lose any data doing it.

ImportAccess

This task helps you import Microsoft Access tables into IDEA. See the "Managing Access Data" section for additional details on importing and exporting Access data.

ImportAS400

The AS/400 is a minicomputer originally developed by IBM. If you have data to import from this platform, you normally need the data (.dat) file and a Format Description File (.fdf). You provide the name of these two files as input to the task as shown in Listing 14.1.

ImportdBASE

dBASE is a relatively old (by computer standards) file format originally created by C. Wayne Ratliff and later licensed by Ashton-Tate. A dBASE file can include any file that conforms to the dBASE standard, even those created using products such as Nantucket Clipper. In fact, dBASE files come from a very wide variety of sources such as Microsoft FoxPro, so you might have a dBASE file and not even know it! Normally, a dBASE file has a .dbf extension.

LISTING 14.1 Importing an AS/400 File

```
Sub Main
   ' Create the task.
   Dim ImportAS400 As Task
   Set ImportAS400 = Client.GetImportTask("AS400")

   ' Set the output name of the database.
   ImportAS400.OutputFileName = "Sample-AS400.imd"

   ' Obtain the input path.
   Dim InputPath As String
   InputPath = Client.WorkingDirectory

   ' Define the input data filename and definition file.
   ImportAS400.InputDATFilename = InputPath + "AS400.dat"
   ImportAS400.InputFDFFilename = InputPath + "AS400def.fdf"

   ' Perform the task.
   MsgBox "Importing: " + InputPath + "AS400.dat"
   ImportAS400.PerformTask

   ' Clear memory.
   Set ImportAS400 = Nothing

   ' Open the database for viewing.
   Client.OpenDatabase("Sample-AS400.imd")
End Sub
```

IDEA lets you create a link to a dBASE file, which saves disk space, or import the file for use. Normally you want to import the file to improve application performance. Using a link slows things down considerably and isn't recommended for the most part considering disk space is relatively cheap today. Listing 14.2 shows the code you can use to perform a basic dBASE import.

Note

If the Link option is used, the source dBASE file must remain in the location it was imported from and remain accessible. If you move the database or make it inaccessible in some way, you'll break the link to it.

A dBASE import represents one of the exceptions to the rule. In this case, you can use either the `ImportDbaseFile()` method or the `GetImportTask()` method to perform the import. This is one of the few situations where IDEA provides a special method to perform the import task.

LISTING 14.2 Importing a dBASE File

```
Sub Main
    ' Configure the import task.
   Client.AddRecordNumberFieldForImport = False

    ' Obtain the input path.
   Dim InputPath As String
   InputPath = Client.WorkingDirectory

    ' Perform the import task.
   Client.ImportDbaseFile InputPath + "DBASE.DBF", _
      "Sample-dBASE.imd", False, False

    ' Open the new database.
   Client.OpenDatabase("Sample-dBASE.imd")
End Sub
```

Note

You might wonder why there are two methods for accessing a dBASE file. Originally, IDEA only supplied a fixed number of imports, which is where the `ImportDbaseFile()` method comes from. A few years back, IDEA added the ability to let the user download custom imports. However, to do this meant IDEA could no longer support specific APIs for each import, such as `ImportMP3Info()`, as this import may not have been considered when the product went out the door. To fix this problem, IDEA now has a generic API (`GetImportTask`) to wrap existing imports (dBASE for example). Yet for backward compatibility reasons, IDEA still provides support for the original API, `ImportDbaseFile()`.

Notice that the use of a special method does reduce the size and complexity of the code you write, but not by much. The `ImportDbaseFile()` method requires four inputs:

- **PathName:** Provides the file name of the database to import. The file name should have the `.dbf` extension.
- **DatabaseName:** Contains the file name of the output database with an `.imd` extension.
- **ComputeStats:** Computes file statistics during import when set to `True`.
- **LinkToFile:** Creates a link to the file, rather than importing it, when set to `True`.

This example also shows how to use the `AddRecordNumberFieldForImport` property. Setting this property to `True` adds a field to the imported database that contains the record number. In this case, the code sets the property to `False`, so you only see the original data.

ImportExcel

This task helps you import Microsoft Excel worksheets into IDEA. You use this task for worksheets with an `.xls` or `.xlsx` extension. See the "Managing Excel Data" section for additional details on importing and exporting Excel data.

ImportXML

eXtensible Markup Language (XML) appears just about everywhere today because it provides a convenient method for transferring information over the Internet without any loss of data context (the meaning behind the data). In addition, XML works across machine types and doesn't rely on a particular product or operating system. XML does have its own quirks and you can encounter some odd problems when working with it, but most people agree that XML works very well for the tasks it's designed to perform. If you want to know more about how XML works, you can find a useful tutorial at `www.w3schools.com/xml/default.asp`.

If you've tried to import XML using other products and have been disappointed at the amount of work you must perform to accomplish the task, be prepared for a pleasant surprise. IDEA doesn't require that you know anything about the content of the XML file—it locates the data for you automatically and imports it based on the structure the data uses. All you need to know is the file name as shown in Listing 14.3.

Note

While you can perform an import without knowing the content of the XML file, importing XML files successfully does require a bit more input. If you don't select which tagged elements are required, there is a risk of ending up with a database containing lots of surplus fields and data. The example XML file imports a field called NONAMESPACESCHEMALOCATION, which holds the name of the XML schema file repeated on every record.

PublishToMicrosoftWord

There are actually two methods you can use to export data to Microsoft Word. The first method is to publish the data. If you were to perform this task manually, you would:

Select **File** > **Print** > **Publish to Microsoft Word** and follow the instructions IDEA provides.

The second method is to actually export the data. In this case, you would:

1. Select **File** > **Export Database**, select **Microsoft Word** in the **Export as** field of the **Export Database** dialog box and perform any other required configurations.
2. Click **OK** to complete the task.

LISTING 14.3 Importing an XML File

```
Sub Main
    ' Create the task.
   Dim ImportXML As Task
   Set ImportXML = Client.GetImportTask ("ImportXML")

    ' Define the output filename.
   ImportXML.OutputFileName = "Sample-OrderDetails.imd"

    ' Obtain the input path.
   Dim InputPath As String
   InputPath = Client.WorkingDirectory

    ' Define the input filename.
   ImportXML.InputFileName = InputPath + "OrderDetails.xml"

    ' Perform the task.
   MsgBox "Importing: " + InputPath + "OrderDetails.xml"
   ImportXML.PerformTask

    ' Clear memory.
   Set ImportXML = Nothing

    ' Open the file.
   Client.OpenDatabase("Sample-OrderDetails.imd")
End Sub
```

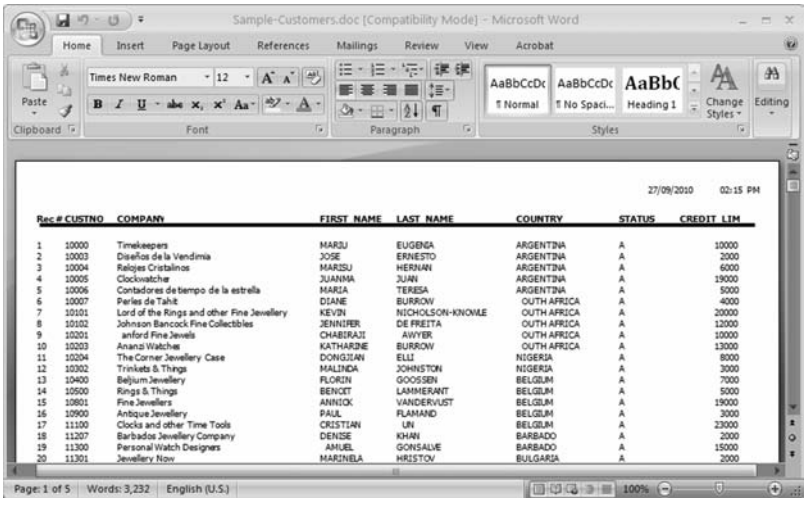


FIGURE 14.1 Publishing Produces a Report

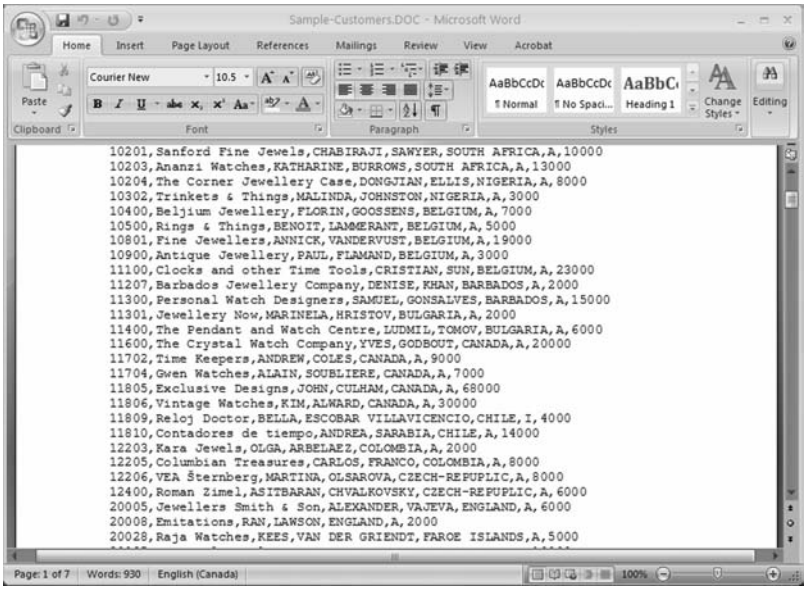


FIGURE 14.2 Exporting Produces Data You Can Analyze and Use in Other Ways.

The two output techniques produce significantly different results. Figure 14.1 shows the result of publishing the `Sample-Customers` database to Word, while Figure 14.2 shows the result of exporting the same database. You use the first technique for creating a report and the second technique when you want to use the data directly. Because the first technique does produce a report, you might see the message shown in Figure 14.3 when you use it. The user will have to choose an option based on output requirements.

You'll find a number of other differences when working with the two output techniques. Publishing produces a formatted result, while exporting relies on plain text that you'll need to format in Word. Exporting also provides some additional flexibility. For example, you can choose a sort order and selection criteria. In addition, you can

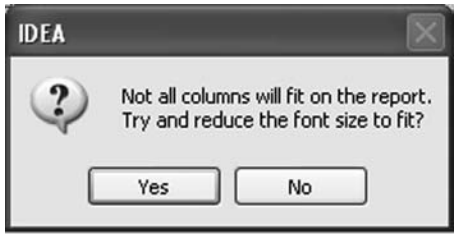


FIGURE 14.3 When Publishing the Data, the User May Have to Choose an Output Strategy.

choose a range of records to export. Listing 14.4 shows both publishing and exporting techniques.

The application begins by getting the working directory, which defaults to the location of the `IDEA\Samples` directory on your machine. `Main()` then calls `PrintToMSWord()` to show the printed form of the output and `ExportDatabaseDOC()` to show the exported form of the output. In both cases, `Main()` passes the `Path` so the routines know where to place the data.

Let's begin by looking at `PrintToMSWord()`. This subroutine begins by opening the database using `OpenDB()`. You might remember `OpenDB()` from Chapters 7 and 8.

LISTING 14.4 Exporting Data to Microsoft Word

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Publish the database first, and then export it.
   Call PrintToMSWord(Path)
   Call ExportDatabaseDOC(Path)
End Sub


' File: Print - Microsoft Word
Sub PrintToMSWord(Path As String)
   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path + "Sample-Customers.imd")

   ' Check for errors.
  If db Is Nothing Then
    Exit Sub
  End If

  ' Set the publishing filename.
  db.FilenameForPublishing = Path + "Sample-Customers1.doc"

  ' Remove the old copy of the file.
  DeleteOld Path + "Sample-Customers1.doc"

  ' Publish the database.
  MsgBox "Publishing: " & Path & "Sample-Customers1.doc"
  db.PublishToMicrosoftWord

  ' Close the database.
  db.Close

  ' Clear memory
  Set db = Nothing
End Sub
```

```
' File - Export Database: DOC
Sub ExportDatabaseDOC(Path As String)
  ' Open the database.
  Dim db As Database
  Set db = OpenDB(Path + "Sample-Customers.imd")

' Check for errors.
  If db Is Nothing Then
    Exit Sub
  End If

  ' Create the task.
  Dim task As ExportDatabase
  Set task = db.ExportDatabase

  ' Configure the task.
  task.IncludeAllFields
  task.AddKey "CREDIT_LIM", "A"

  ' Remove the old copy of the file.
  DeleteOld Path + "Sample-Customers2.doc"

  ' Perform the task.
  MsgBox "Exporting: " + Path + "Sample-Customers2.doc"
  task.PerformTask Path + "Sample-Customers2.doc", _
    "Database", "DOC", 1, db.Count, ""

  ' Close the database.
  db.Close

  ' Clear memory
  Set db = Nothing
  Set task = Nothing
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
   ' exist and we need to exit the routine.
   If Len(PathCheck) <  1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)
```

(*continued*)

LISTING 14.4 (*Continued*)

```
   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function

Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub
```

This example uses a more advanced form that provides error trapping as shown in the following code:

```
Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
   ' exist and we need to exit the routine.
   If Len(PathCheck) <  1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

The error trapping begins with the `Dir()` function that you'll learn more about in Chapter 15. This function looks at the path you provide and obtains the name of the first file that matches. In this case, the code provides an exact path for a single file, the

database you want to open. If the file doesn't appear in the location you specify, then `Dir()` returns an empty string. Otherwise, it returns the name of that database file in `PathCheck`.

The next step is to check for an error. If `PathCheck` has a length of zero, which means that `Dir()` didn't find the file you specified, `OpenDB()` responds by displaying an error message the user can understand, then exits the function. Otherwise, the code performs precisely the same task it did in Chapters 7 and 8. The code creates a `Database` object using `Client.OpenDatabase()`, sets `OpenDB()` equal to this object, cleans up memory, and exits.

When `OpenDB()` returns to `PrintToMSWord()`, it can provide two output values. Either it returns `Nothing` or it returns a `Database` object. The next check in `PrintToMSWord()` is to check for `Nothing`. If `db` is `Nothing`, then the subroutine simply exits and doesn't perform any more work.

Now that the code knows that the database you want to use really does exist, it sets the `FilenameForPublishing` property to the proper output value. In this case, the code uses `Sample-Customers1.doc` and places the output in the same directory as the database. Of course, the directory could already contain a `Sample-Customers1.doc` file. The next step is to remove this old file using `DeleteOld()`, as shown in the following code:

```
Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir(Filepath)

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
      MsgBox "Deleting old copy of " + PathCheck
      Kill Filepath
   End If
End Sub
```

`DeleteOld()` begins by using the same technique used to verify the presence of the database. It sets `PathCheck` to the name of the file using `Dir()` when the file exists in the target directory. If the file exists, the code displays a message box telling the user that it's deleting the old copy of the file. It then uses the `Kill()` function to actually delete the file (see Chapter 15 for more details about the `Kill()` function).

Warning

You may encounter a problem when working with some functions. Deleting a file is one of those situations. Every time an application object gains access to the database in some way, it increases the reference count for that database. The *reference count* basically says that so many objects are referencing the database. When an application

(*continued*)

(*Continued*)
object releases its access to the database, then the reference count decreases. It isn't possible to delete the database until its reference count is 0. In short, you must release every reference to the database before you can delete it. The same holds true for any other IDEA object. It isn't possible to delete the object until the object's reference count is 0.

Tip

It's important to look at the technique used in this chapter to determine whether a file exists. In one case, using the `Dir()` function ensures that the file exists before attempting to open it. In the second case, using the `Dir()` function checks for the file so that the application can remove it before creating another file of the same name. Checking for the file before you attempt to work with it means avoiding an error, rather than experiencing the error, then handling it later. When you're proactive about checking for potential errors before they happen, you maintain control of the application and its environment, making it less likely that your application will experience a nasty unrecoverable error. Error handling is best used for situations you can't anticipate. Use proactive checking techniques, such as those found in this chapter, to avoid errors before they occur whenever possible.

At this point, the code returns once again to `PrintToMSWord()` where the code displays a message box saying that the application is going to publish the data in the database. The actual publishing process is simple—you use the `PublishTo MicrosoftWord()` function to do it. The `PrintToMSWord()` code ends by closing the database and clearing memory.

`ExportDatabaseDOC()` uses many of the same techniques as `PrintTo MSWord()` does. It begins by using `OpenDB()` to open the database and performs the same check on return from the call. However, exporting relies on the `ExportDatabase` task, rather than a direct `Database` object call. As with any task, the code tells IDEA which fields to include and adds a key for sorting the data.

The code calls `DeleteOld()` to remove the old file next and displays the familiar exporting message. The code relies on `PerformTask()` to export the data. In this case, you must include the name of the file you want to use, the name of the table in the exported document, the kind of export to perform, the starting record, the ending record (obtained using the `Database` object's `Count` property), and the export criteria (if any). The code ends by calling `db.Close()` and cleaning up memory.

PublishToPDF

This task helps you export IDEA data to PDF format. See the "Managing PDF Data" section for additional details on importing and exporting PDF data.

Performing Data Extractions

Extracting data, mining precisely what you want from a database, is an important part of any analysis. IDEA supports a number of types of extraction. The following sections describe three extraction types:

- **Direct:** You use Direct Extraction to obtain a specific subset of the data in the database.
- **Key Value:** You use Key Value Extraction to create one or more databases based on the key you specify. This extraction method lets you determine whether each key value creates a separate output database or all of the key values appear in a single database. The extraction is always based on the key and associated values you choose.
- **Top Records:** You use Top Records Extraction to identify the highest or lowest value records within a data set or within key groups. You can define how many records within each group are extracted.

Using Extraction

The `Extraction` task helps you to directly extract information from the target database and place the result into a secondary database. For example, you might want to extract all of the customers from Argentina from the `Sample-Customers` database. The output database will contain just the 18 records related to Argentina customers. Listing 14.5 shows how to perform this task.

LISTING 14.5 Extracting Data

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path + "Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Delete the old file.
   DeleteOld Path + "Sample-Customers-Extract.imd"

   ' Create the task.
   Dim task As Extraction
   Set task = db.Extraction

   ' Configure the task.
```

(*continued*)

LISTING 14.5 (*Continued*)

```
   task.IncludeAllFields
   task.AddKey "CREDIT_LIM", "A"
   task.AddExtraction Path + "Sample-Customers-Extract.imd", _
      "", " COUNTRY  = ""ARGENTINA"""

   ' Perform the task.
   task.PerformTask 1, db.Count

   ' Clear memory.
   Set task = Nothing
   Set db = Nothing

   ' Open the new database.
   Client.OpenDatabase (Path + "Sample-Customers-Extract.imd")
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
   ' exist and we need to exit the routine.
   If Len(PathCheck) <  1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function

Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub
```

The code begins by opening the database using the `OpenDB()` function described in the "PublishToMicrosoftWord" section. It then uses `DeleteOld()` to remove any old copies of the database from the target directory, which is the default working directory in this case.

The next step is to create the `Extraction` task. The code then configures the task by including all of the fields, adding a key to sort the output, and the extraction criteria. The `AddExtraction()` method requires three inputs:

- **DatabaseName:** Specifies the output database name.
- **Desc:** Not used. Set this parameter to `""`.
- **Equation:** The equation to use to extract records from the source database.

At this point, the code calls `PerformTask()` to create the output database. In this case, `PerformTask()` accepts the starting and ending record numbers to extract from the source database. The example ends by clearing memory and opening the newly created database.

Using KeyValueExtraction

When performing the `KeyValueExtraction` task, IDEA relies on an array containing the values you want to extract based on the key you select. For example, if you want to extract the records associated with ARGENTINA, AUSTRALIA, and AUSTRIA, based on the COUNTRY field, you can use this task to produce the required output. Listing 14.6 shows how to perform this task.

The example begins by opening the database using the `OpenDB()` function described in the "PublishToMicrosoftWord" section. The next step is to create the task.

Before the code can configure the task, it must create a two-dimensional array containing a list of key values to use in the extraction. (There is a good reason to use a two-dimensional array. If a multi-field key is used, the first element is used to represent the primary key [the first field selected] and the second element is used for the secondary fields—identified by elements 0 to 6 if the key is defined from the maximum eight fields.) The code configures each array element as shown in the example. Notice that the first array element number changes, not the second. The key values are case sensitive, so "Argentina" isn't the same as "ARGENTINA". The `CreateMultipleDatabases` property controls whether IDEA creates one database for all key values or one database for each key value in `myArray`.

At this point, the code begins configuring the task. It begins by defining the database prefix value using the `DBPrefix` property. IDEA combines this prefix with the key values when you use multiple databases or uses the prefix alone when you use a single database for output. For example, if you choose multiple databases and set the prefix value to `KeyVal`, the first output database might be `KeyVal = ARGENTINA.imd` where ARGENTINA is the first key value in the array.

The next step is to set the fields that should appear in the output and define the key used to match the key values, the criteria used to select records, and the values to extract from the database based on the key value. You must always set `AddKey` before you can set `ValuesToExtract`. Otherwise, your application will end with an error.

LISTING 14.6 Performing a Key Value Extraction

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path + "Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Create the task.
   Dim task As KeyValueExtraction
   Set task = db.KeyValueExtraction

   ' Create an array to hold the key values and fill it
   ' with the key values used for the extraction.
   Dim myArray(2,0)
   myArray(0,0) = "ARGENTINA"
   myArray(1,0) = "AUSTRALIA"
   myArray(2,0) = "AUSTRIA"

   ' If you create one database for each key value, each output
   ' database will have the prefix you provide here.
   task.DBPrefix = "KeyVal"
   task.CreateMultipleDatabases = False

' Configure the task.
   task.IncludeAllFields
   task.AddKey "COUNTRY", "A"
   task.Criteria = " CREDIT_LIM>=10000"
   task.ValuesToExtract myArray

   ' Perform the task.
   task.PerformTask

   ' Obtain the name of the output database(s) and
   ' open them.
   If task.CreateMultipleDatabases Then
      ' Create a variable to hold the current count.
      Dim Count As Integer

      ' Create a variable to hold the individual database names.
      Dim Filename As String

      ' Define a loop to open the databases.
      For Count = 0 To UBound(myArray, 0)
```

```
            ' Define the database name.
            Filename = task.DBPrefix + "=" + myArray(Count, 0) + ".imd"
            Client.OpenDatabase(Filename)
         Next
      Else
         dbName = task.DBName
         Client.OpenDatabase(dbName)
      End If

      ' Clear memory.
      Set task = Nothing
      Set db = Nothing
End Sub

Function OpenDB(DBPath As String) As Database
      ' Verify that the database exists.
      Dim PathCheck As String
      PathCheck = Dir( DBPath )

      ' If PathCheck has a 0 length, the database doesn't
      ' exist and we need to exit the routine.
      If Len(PathCheck) <  1 Then
         MsgBox "Couldn't file the file: " & DBPath
         Exit Function
      End If

      ' Define a database object.
      Dim db As Database

      ' Open the database using the default client folder.
      Set db = Client.OpenDatabase(DBPath)

      '  Return the database object.
      OpenDB = db

      ' Clear the memory used by db.
      Set db = Nothing
End Function
```

Warning

Notice that this example doesn't use `DeleteOld()` to remove the old database if it exists. The problem is that the output directory could contain any number of databases with the requested prefix. You could accidentally remove a database that the user wanted to keep. In this case, it's better to let the user see the replacement message and respond to it, than handle things automatically as the other examples in the chapter have done.

Now the application calls `PerformTask()` to create the required output. Of course, the example could create multiple outputs depending on the setting of `CreateMultipleDatabases`. The example looks at `CreateMultipleDatabases` next and takes one of two actions based on its value.

When `CreateMultipleDatabases` is `True`, the code must open multiple files. To do this, it relies on a `For...Next` loop to create the output file names and then uses `Client.OpenDatabase()` to open them. An output database name includes the database prefix, an equals sign (=), and the key value, followed by the `.imd` extension.

When `CreateMultipleDatabases` is `False`, the code obtains the name of the single output database from `task.DBName`. It then uses this value with `Client.OpenDatabase()` to open the resulting database. Finally, the example clears memory.

Using TopRecordsExtraction

In some cases, you need the top or bottom records that meet particular criteria. For example, you may need the top records for the CREDIT_LIM field. You can add criteria that further limit the output from the extraction, such as those records from Argentina or companies whose name begins with "A." However, the main objective is to find the top (or bottom) records for the CREDIT_LIM field in this case. Of course, you must also define the number of records you want to review as part of this task. Listing 14.7 shows a simple example of the `TopRecordsExtraction` task.

LISTING 14.7 Performing a Top Records Extraction

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path + "Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Delete the old file.
   DeleteOld Path + "TopRecordsExtraction.imd"

   ' Define the task.
   Dim task As TopRecordsExtraction
   Set task = db.TopRecordsExtraction

   ' Configure the task.
```

```
   task.IncludeAllFields
   task.AddKey "CREDIT_LIM", "D"
   task.OutputFileName = "TopRecordsExtraction.imd"
   task.NumberOfRecordsToExtract = 100
   task.PerformTask

   ' Clear memory.
   Set task = Nothing
   Set db = Nothing

   ' Open the output database.
   Client.OpenDatabase ("TopRecordsExtraction.imd")
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
   ' exist and we need to exit the routine.
   If Len(PathCheck) <  1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function


Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub
```

The example begins by opening the database using `OpenDB()`. It then deletes the old database (when present) using `DeleteOld()`.

At this point, the code creates the task and begins configuring it. In this case, the example includes all of the fields. It uses `AddKey()` to add the key to use as the basis for extracting the top records. The direction of sorting determines whether the example outputs the top records or the bottom records. Use a descending (`D`) sort for the top records and an ascending (A) sort for the bottom records (even though this might sound counterintuitive at first, try changing the example to an ascending sort to see the results for yourself). The `OutputFileName` property contains the name of the output database and the `NumberOfRecordsToExtract` property contains the number of records to extract into that database. Calling `PerformTask()` creates the output.

The example ends by clearing memory. It then displays the resulting database on screen.

Managing PDF Data

PDF is one of the most common user data formats after pure text. It contrasts with XML, which is a format often used by developers to transfer information. Just about everyone knows what a PDF file is and has worked with a PDF file at some point. In addition, many applications now support PDF output (although more support pure text output). Consequently, the PDF file represents a great choice for transferring data in an easily readable form that includes full formatting (unlike text, which doesn't include any formatting whatsoever).

IDEA can both import and export PDF data, which means that you can use PDF files for exchange purposes with anyone who doesn't own a copy of IDEA. You can also use PDF files to create reports or to import data from reports. The following sections discuss both PDF import and export.

Importing PDF Data

You can import PDF data of nearly any complexity. However, this is one situation where IDEA can't simply look at the PDF file and determine the difference between text and tabular information. In addition, a PDF file may actually contain more than one table, making it necessary for you to define which table to import. For this reason, you must create a template before you can design an application to import PDF data. Follow these steps to start creating a template:

1. From the IDEA main menu, select **File** > **Import Assistant** > **Import to IDEA**. You will see the **Import Assistant** dialog box.
2. In the **Select the format** list, select **Print Report and Adobe PDF**.
3. Click the ellipses beside the **File name** field. You will see the **Select File** dialog box.

4. In the **Files of type** field, select **Adobe Acrobat (*.pdf)**. Locate the file you want to import and click **Open**. IDEA places the name of the file in the **File name** field of the **Import Assistant** dialog box.
5. Click **Next**. IDEA automatically opens the **Report Reader** for you. You use the **Report Reader** to define the template for your PDF file. The Report Reader Help file provides full details on how to perform this task.

Once you create the template, you can save it to disk and use it to create an import application. Listing 14.8 shows how to perform this task.

LISTING 14.8 Importing PDF Data

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Delete the old file.
   DeleteOld Path + "PDF-Import.imd"

   ' Perform the import.
   Client.ImportPrintReport _
      Path + "Sample-Customers.jpm", _
      Path + "Sample-Customers.pdf", _
      "PDF-Import.imd", False

   ' Open the resulting database.
   OpenDB(Path + "PDF-Import.imd")
End Sub

Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' Define a database object.
   Dim db As Database
```

(*continued*)

LISTING 14.8 (*Continued*)

```
   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

The code begins by deleting the old database, if it exists. It then uses the `Client.ImportPrintReport()` method to import the data. The `ImportPrintReport()` method requires four arguments as input:

- Name of the template file
- Name of the PDF file
- Name of the IDEA database
- `True` if you want to compute statistics for the new database or `False` if you don't

During the import process, you see a Report Reader dialog box with a progress bar showing the progress in importing the data. After the import process is complete, the code uses `OpenDB()` to open the resulting database.

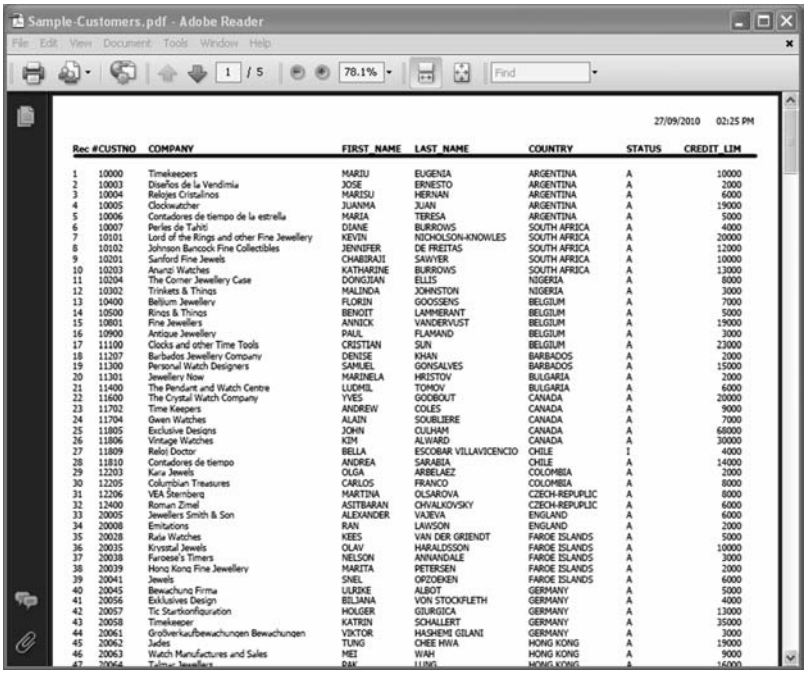
Exporting PDF Data

Exporting a database as a PDF file is more a process of printing than actual exporting. You're creating a report of sorts that someone will read directly. Unlike importing a PDF file, you don't require any special template files to export a PDF file. Listing 14.9 shows how to perform this task.

LISTING 14.9 Exporting PDF Data

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path + "Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If
```

```
   ' Delete the old file.
   DeleteOld Path + "Sample-Customers.pdf"

   ' Output the file.
   db.FilenameForPublishing = Path + "Sample-Customers.pdf"
   db.PublishToPDF

   ' Open the PDF
   Shell "cmd.exe /c " + Chr(34) + Path + _
      "Sample-Customers.pdf" + Chr(34)
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
   ' exist and we need to exit the routine.
   If Len(PathCheck) <  1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function


Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub
```



Rec #	CUSTNO	COMPANY	FIRST_NAME	LAST_NAME	COUNTRY	STATUS	CREDIT_LIM
1	10000	Timekeepers	MARIU	EUGENIA	ARGENTINA	A	10000
2	10003	Diseños de la Vendimia	JOSE	ERNESTO	ARGENTINA	A	2000
3	10004	Relojes Cristalinos	MARISU	HERNAN	ARGENTINA	A	6000
4	10005	Clockwatcher	JUANMA	JUAN	ARGENTINA	A	19000
5	10006	Contadores de tiempo de la estrella	MARIA	TERESA	ARGENTINA	A	5000
6	10007	Perles de Tahiti	DIANE	BURROWS	SOUTH AFRICA	A	4000
7	10101	Lord of the Rings and other Fine Jewellery	KEVIN	NICHOLSON-KNOWLES	SOUTH AFRICA	A	20000
8	10102	Johnson Buncock Fine Collectibles	JENNIFER	DE FREITAS	SOUTH AFRICA	A	12000
9	10201	Sanford Fine Jewels	CHABIRAJI	SAWYER	SOUTH AFRICA	A	10000
10	10203	Anunzi Watches	KATHARINE	BURROWS	SOUTH AFRICA	A	13000
11	10204	The Corner Jewellery Case	DONGJIAN	ELLIS	NIGERIA	A	8000
12	10302	Trinkets & Things	MALINDA	JOHNSTON	NIGERIA	A	3000
13	10400	Bellium Jewellery	FLORIN	GOOSSENS	BELGIUM	A	7000
14	10500	Rings & Things	BENOIT	LAMMERANT	BELGIUM	A	5000
15	10801	Fine Jewellers	ANNICK	VANDERVUST	BELGIUM	A	19000
16	10900	Antique Jewellery	PAUL	FLAMAND	BELGIUM	A	3000
17	11100	Clocks and other Time Tools	CRISTIAN	SUN	BELGIUM	A	23000
18	11207	Barbados Jewellery Company	DENISE	KHAN	BARBADOS	A	2000
19	11300	Personal Watch Designers	SAMUEL	GONSALVES	BARBADOS	A	15000
20	11301	Jewellery Now	MARINELA	HRISTOV	BULGARIA	A	2000
21	11400	The Pendant and Watch Centre	LUDMIL	TONOV	BULGARIA	A	6000
22	11600	The Crystal Watch Company	YVES	GOOBOUT	CANADA	A	20000
23	11702	Time Keepers	ANDREW	COLES	CANADA	A	9000
24	11704	Gwen Watches	ALAIN	SOUBLIERE	CANADA	A	7000
25	11805	Exclusive Designs	JOHN	CULHAM	CANADA	A	68000
26	11806	Vintage Watches	KIM	ALWARD	CANADA	A	30000
27	11809	Reloj Doctor	BELLA	ESCOBAR VILLAVICENCIO	CHILE	I	4000
28	11810	Contadores de tiempo	ANDREA	SARABIA	CHILE	A	14000
29	12203	Kara Jewels	OLGA	ARBELAEZ	COLOMBIA	A	2000
30	12205	Columbian Treasures	CARLOS	FRANCO	COLOMBIA	A	8000
31	12206	VEA Šternberg	MARTINA	OLSAROVA	CZECH-REPUPLIC	A	8000
32	12400	Roman Zimel	ASITBARAN	CHVALKOVSKY	CZECH-REPUPLIC	A	6000
33	20005	Jewellers Smith & Son	ALEXANDER	VAJEVA	ENGLAND	A	6000
34	20008	Emitations	RAN	LAWSON	ENGLAND	A	2000
35	20028	Rala Watches	KEES	VAN DER GRIENDT	FAROE ISLANDS	A	5000
36	20035	Krvsstal Jewels	OLAV	HARALDSSON	FAROE ISLANDS	A	10000
37	20038	Faroese's Timers	NELSON	ANNANDALE	FAROE ISLANDS	A	3000
38	20039	Hong Kong Fine Jewellery	MARITA	PETERSEN	FAROE ISLANDS	A	2000
39	20041	Jewels	SNEL	OPZOEKEN	FAROE ISLANDS	A	6000
40	20045	Bewachung Firma	ULRIKE	ALBOT	GERMANY	A	5000
41	20056	Exklusives Design	BILJANA	VON STOCKFLETH	GERMANY	A	4000
42	20057	Tic Startkonfiguration	HOLGER	GJURGICA	GERMANY	A	13000
43	20058	Timekeeper	KATRIN	SCHALLERT	GERMANY	A	35000
44	20061	Großverkaufbewachungen Bewachungen	VIKTOR	HASHEMI GILANI	GERMANY	A	3000
45	20062	Jades	TUNG	CHEE HWA	HONG KONG	A	19000
46	20063	Watch Manufactures and Sales	MEI	WAH	HONG KONG	A	9000
47	20064	Talitac Jewellers	RAY	LUNG	HONG KONG	A	16000

FIGURE 14.4 Printing the Data As a PDF File Produces a Report Like This.

The code begins by opening the database using `OpenDB()`. As normal, the code checks for potential errors before proceeding. The code also deletes the existing PDF file, if any. At this point, the code sets the `FilenameForPublishing` property to the location of the PDF file you want to create. It then calls `PublishToPDF()` to perform the actual task.

At this point, the code uses the same technique shown in Chapter 8 to open the resulting file. The code creates a copy of the command processor, passes the name of the file to it, and then the command processor opens that file using the features found in Windows. The user sees the PDF file open with the report loaded, as shown in Figure 14.4. The command processor window closes when the user closes Adobe Acrobat (or another application used to view PDF files).

Managing Text Data

Text files are the most common of all file types. Every computer ever made can use text files in some manner. Consequently, you'll very likely use text files at some point as a means of importing and exporting data. Using text files isn't always straightforward because text files come in several formats. The following sections tell you more about text files and how to work with them.

Considering the Text Data Types

Text files don't have any formatting—they're simply characters that appear in a file. A text file won't preserve any special information about your database and when you import a text file, you must supply an interpretation of the text for IDEA. However, text files have to have some means of indicating individual records and fields within those records. Consequently, developers have created some common methods for including this information in the text file—the text file format. IDEA supports the three kinds of formatting described in the following sections (IDEA uses the associated file extensions for export, but doesn't care what the file extension is for import as long as the file provides the specified text formatting):

- Tab Separated Value (`.tsv` extension)
- Delimited (`.asc` extension)
- Fixed Length (`.fxd` extension)

Note

Text files normally have a `.txt` file extension. IDEA uses the special `.tsv`, `.asc`, and `.fxd` file extensions to make it easier to know which kind of format a text file uses to store data. You can change the file extension of any files you receive from someone else to match the IDEA extensions to make it easier to import the files. However, you may also need to change the IDEA extension to `.txt` for users of other computer systems.

TAB SEPARATED VALUE (TSV) A TSV file uses the tab control character to separate fields and a carriage return/line feed character combination to separate records as shown in Figure 14.5. This file format is quite flexible because it only uses two control characters for formatting—characters that are unlikely to interfere with database data in most cases. Even if the database contains special characters, you'll likely find that this text output is unaffected (the only time you'll encounter problems is when the database contains tabs and the carriage return/line feed combination).

This type of formatting does have a few problems. For example, it's used less frequently than the delimited file format and not all applications will support it. As you can see from the figure, the tabs do provide distinct spacing between fields as long as there's a space between the end of the data and the next tab stop. Depending on how the viewer you use works with tabs, you might not easily see fields in the file. This is also the only text output that doesn't include field names in the first row as an option, which means that someone importing this file into another application may not use the same field names as you do. Using different field names can add a confusion factor when you need to interact with the other party.

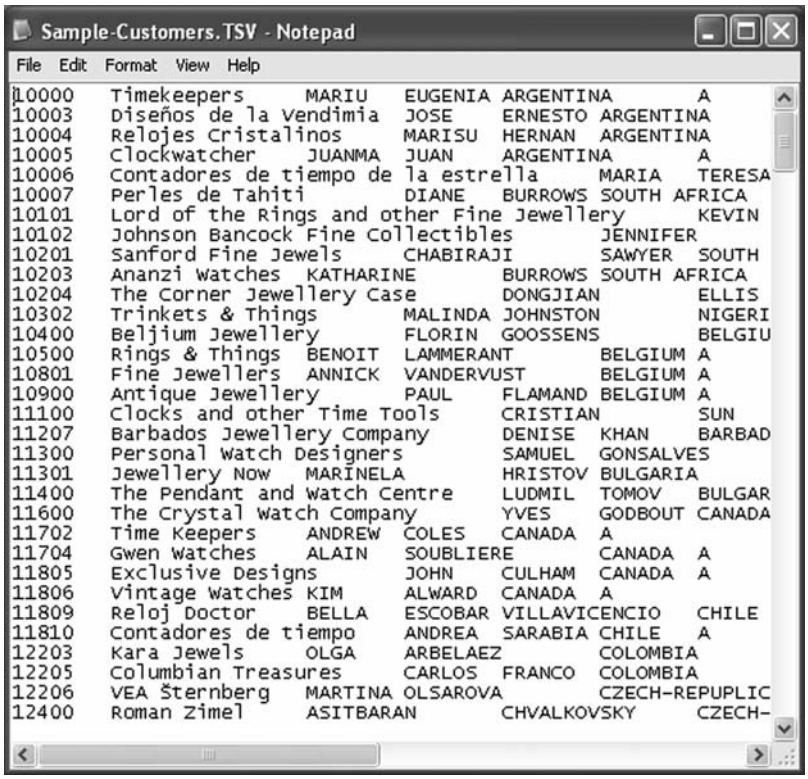


FIGURE 14.5 TSV Files Look Like This When Viewed in Notepad

DELIMITED Delimited files have seen common use since the days when mainframes ruled the day and PCs weren't even an idea yet. The term *delimited* simply means that each field is encased (delimited) in a special character and that fields are separated by another special character. The most common delimiter is the double quote (`"`). The field separator is the comma (`,`). IDEA does let you set the field separator character when importing data to avoid potential problems with use of commas in your data.

Each record in a delimited file is separated by a special control character. Most modern applications use a carriage return/line feed combination (as used for some of the earlier examples in this book to format text for message boxes). However, you might also find that some computers use only a line feed (such as UNIX-based systems) or a carriage return. The point is that each record is separated from the others by a special control character.

IDEA uses the `.asc` (for ASCII) extension for delimited files. Another common extension is `.csv` (for comma separated value). In fact, many database applications use the `.csv` extension when exporting data to a text format, so you'll probably encounter this file extension at some time. No matter what extension the file uses, a delimited file normally looks like the one shown in Figure 14.6.

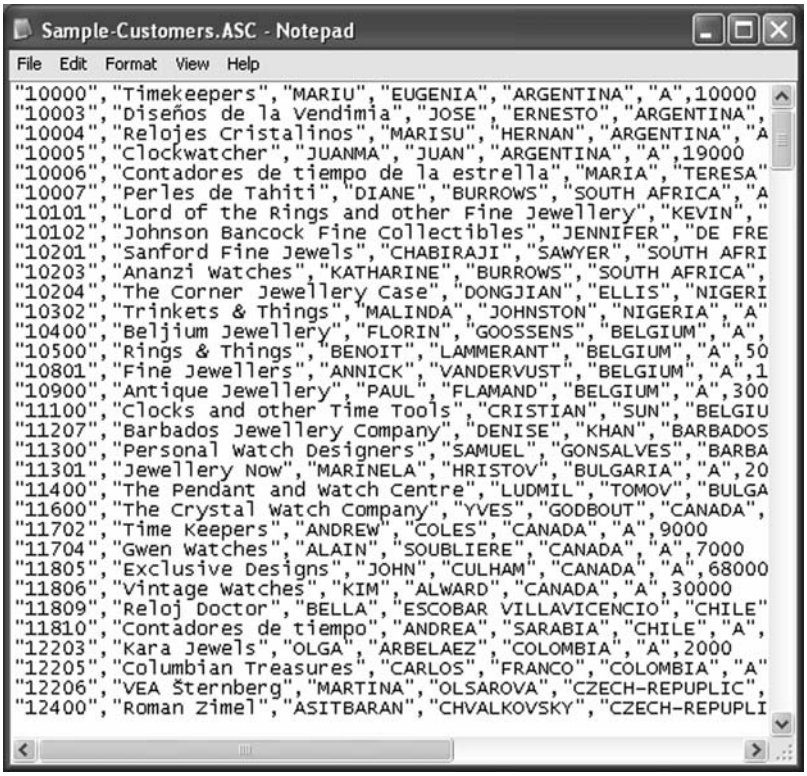


FIGURE 14.6 Delimited Files Look Like This When Viewed in Notepad

Notice that the first row of this file contains the names of the fields. IDEA gives you the option of including or not including the field names. In most cases, it helps to include the field names to make it easier to import the file later. When someone exports a file for you, make sure you ask that they include the field names in the first row of the file.

FIXED LENGTH The fixed length text file uses a fixed length for every field and the carriage return/line feed combination between records. (There isn't any guarantee that a source file will include the carriage return/line feed combination, but normally it will.) Like delimited files, fixed length files have been around for many years and many applications support them. Of all the text output types, fixed length is the easiest to view directly, as shown in Figure 14.7. Because the fields are padded with spaces, you can easily see where each field begins and ends in most cases. This format is also easy to see when you need to import data into IDEA.

Like every other format, you do have to consider the negatives when working with a fixed length file. In this case, the most significant negative is the amount of space this file requires. The example file is 45 KB in size, while the other two formats consume a

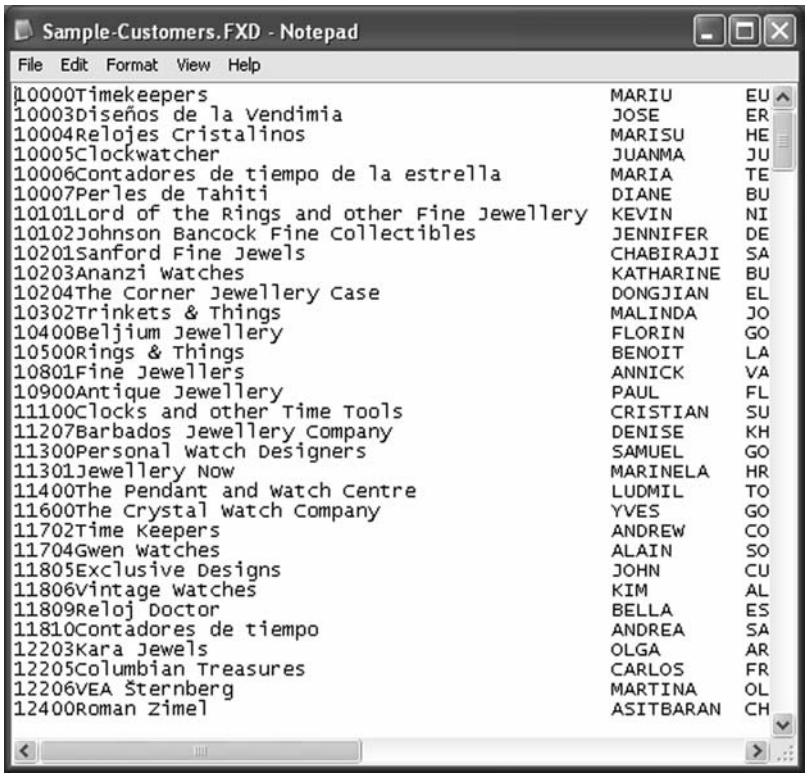


FIGURE 14.7 Fixed Length Files Look Like This When Viewed in Notepad

mere 22 KB each. If you're exporting or importing a large database, the size differential can make a big difference.

In some cases, you may also experience some problems figuring out where one data element ends and another begins. Look at the figure again. Notice that CUSTNO runs right into COMPANY. If you aren't careful, you could end up making a mistake during import that would leave the field names incorrect (as a minimum).

Creating a Definition File

Before you can create an application to import text data, you must create a Record Definition File (RDF). IDEA makes it exceptionally easy to create this file. All you do is follow the steps you'd normally take to manually import the file of your choice. The last step of the process asks you to supply a name for the RDF. You use this file in your application (make sure you supply it to any application users as well). Use these steps to begin the process of creating the RDF:

1. From the IDEA main menu, select **File** > **Import Assistant** > **Import to IDEA**. You'll see the **Import Assistant** dialog box.
2. In the **Select the format** list, select **Text**.
3. Click the ellipses beside the **File name** field. Use the **Select File** dialog box to locate the file you want to import and then click **Open**.
4. Click **Next**. At this point, you see selections for the type of text file. Normally, IDEA chooses the correct type for you. Simply follow the specific text definition steps from this point on to create the definition (they differ by file type—see the IDEA help file for additional information).

At the last step of the Import Assistant wizard, you see the **Save record definition as** field. This is where you enter the name of the RDF you want to create. Make sure you keep this name in mind when you write your code.

Importing Text Data

There are essentially two techniques you use to import text data. The first works with any file that's delimited in some way, such as `.tsv` and `.asc` files. The second works with any file that uses a fixed length format, such as `.fxd` files. Listing 14.10 shows how to perform both types of import.

LISTING 14.10 Importing Text Data

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Perform the two different kinds of import.
   Call ImportTSVAndASC(Path, "ASC-Sample-Customers.imd")
   Call ImportFixedLength(Path, "FXD-Sample-Customers.imd")
End Sub

' File - Import Assistant: Delimited Text
Sub ImportTSVAndASC(Path As String, Filename As String)
   ' Remove the old database.
   DeleteOld Path + Filename

   ' Import the data.
   Client.ImportDelimFile "Sample-Customers.ASC", _
      Path + Filename, False, "", _
      "ASC-Sample-Customers.RDF", True

   ' Open the resulting database.
   OpenDB Path + Filename
End Sub
```

(*continued*)

LISTING 14.10 (*Continued*)

```
' File - Import Assistant: Fixed Length Text
Sub ImportFixedLength(Path As String, Filename As String)
   ' Remove the old database.
   DeleteOld Path + Filename
      ' Import the data.
   Client.ImportDatabase Path + "Sample-Customers.FXD", _
      Path + Filename, False, False, "", _
      Path + "FXD-Sample-Customers.RDF"

   ' Open the resulting database.
   OpenDB Path + Filename
End Sub

Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )


   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

The code in `Main()` begins by obtaining the path information. It then passes the path information, along with a database name, to the two subroutines responsible for importing the data, `ImportTSVAndASC()` and `ImportFixedLength()`. In this case, `ImportTSVAndASC()` imports an `.asc` file, but the same technique works fine for a `.tvs` file.

The `ImportTSVAndASC()` subroutine begins by deleting the old file when it exists. The subroutine then calls `Client.ImportDelimFile()`, which requires these arguments:

- **ImportFilename:** Contains the name of the delimited text file you want to import.
- **OutputDatabaseName:** Contains the name of the IDEA database that receives the imported data. The IDEA database has an `.imd` file extension.
- **ComputeStats:** Determines whether IDEA calculates field statistics for the database. Set this argument to `True` to create field statistics during import. Adding this option can make the import process extremely slow, so use it with care.
- **RDFFilename:** Provides the name of the RDF to use during the import process. The RDF describes the format and structure of the textual data. An RDF always has an `.rdf` extension.
- **First line contains the field names:** Determines whether IDEA uses the first record as a means to determine the field names. Set this argument to `True` if the first line contains the database field names.

When the import is completed, the code opens the resulting database using `OpenDB()`.

The `ImportFixedLength()` subroutine also starts by deleting the old file. However, this subroutine calls `Client.ImportDatabase()` to import the fixed length file. The `ImportDatabase()` method requires the following arguments:

- **ImportFilename:** Contains the name of the fixed length text file you want to import. The file name should have a file extension that indicates it's an ASCII (American Standard Code for Information Interchange) or EBCDIC (Extended Binary Coded Decimal Interchange Code) file, such as `.txt` or `.dat`.
- **OutputDatabaseName:** Contains the name of the IDEA database that receives the imported data. The IDEA database has an `.imd` file extension.
- **LinkToFile:** Determines whether IDEA creates a link to the external file or imports it as a new database. Set this to `True` if you want to link and save disk space. Set to `False` if you want to import instead and save time. Normally you want to import the file to improve application performance. Using a link slows things down considerably and isn't recommended for the most part considering disk space is relatively inexpensive.

Tip

If the Link option is used, the source file must remain in the location it was imported from and remain accessible. If you move the database or make it inaccessible in some way, you'll break the link to it.

- **ComputeStats:** Determines whether IDEA calculates field statistics for the database. Set this argument to `True` to create field statistics during import. Adding this option can make the import process extremely slow, so use it with care.
- **RDFFilename:** Provides the name of the RDF to use during the import process. The RDF describes the format and structure of the textual data. An RDF always has an `.rdf` extension.

As before, the subroutine ends by opening the resulting database file using `OpenDB()`.

Exporting Text Data

Exporting a database to text format will let you exchange information with just about every computer platform in existence. In fact, you'd be hard pressed to find an exception to the rule. When you export a database to text format, always choose a text file format that works well with the recipient's software. In many cases, the best choice is to use a delimited file because most applications support this format. The basic export strategy relies on the `ExportDatabase` task. However, each of the standard text formats differs in some small points, such as setting the separators when working with the delimited format. Listing 14.11 shows how to perform all three text outputs and focuses on the minor differences between them.

LISTING 14.11 Exporting Text Data

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Perform the three conversions provided with IDEA.
   Call ExportDatabaseTSV(Path, "Sample-Customers.imd")
   Call ExportDatabaseASC(Path, "Sample-Customers.imd")
   Call ExportDatabaseFXD(Path, "Sample-Customers.imd")
End Sub

' File - Export Database: TSV
Sub ExportDatabaseTSV(Path As String, Filename As String)

' Delete the existing output file.
   DeleteOld Path + "Sample-Customers.TSV"

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path + Filename)

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If
```

```
   ' Create the task.
   Dim task As ExportDatabase
   Set task = db.ExportDatabase

   ' Configure the task.
   task.IncludeAllFields

   ' Perform the task.
   task.PerformTask Path + "Sample-Customers.TSV", _
      "Database", "TSV", 1, db.Count, ""

   ' Clear memory.
   Set db = Nothing
   Set task = Nothing

   ' Open the file.
   Shell "Notepad " + Chr(34) + Path + "Sample-Customers.TSV" + Chr(34)
End Sub

' File - Export Database: ASC
Function ExportDatabaseASC(Path As String, Filename As String)

 ' Delete the existing output file.
   DeleteOld Path + "Sample-Customers.ASC"

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path + Filename)

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Create the task.
   Dim task As ExportDatabase
   Set task = db.ExportDatabase

   ' Configure the task.
   task.IncludeFieldNames = True
   task.IncludeAllFields
   task.Separators ",", "."

   ' Perform the task.
   task.PerformTask Path + "Sample-Customers.ASC", _
      "Database", "ASC", 1, db.Count, ""

   ' Clear memory.
   Set db = Nothing
   Set task = Nothing
```

(*continued*)

LISTING 14.11 (*Continued*)

```
   ' Open the file.
   Shell "Notepad " + Chr(34) + Path + "Sample-Customers.ASC" + Chr(34)
End Function

' File - Export Database: FXD
Function ExportDatabaseFXD(Path As String, Filename As String)
   ' Delete the existing output file.
   DeleteOld Path + "Sample-Customers.FXD"

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path + Filename)

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Create the task.
   Dim task As ExportDatabase
   Set task = db.ExportDatabase

   ' Configure the task.
   task.IncludeFieldNames = True
   task.IncludeAllFields

   ' Perform the task.
   task.PerformTask Path + "Sample-Customers.FXD", _
      "Database", "FXD", 1, db.Count, ""

   ' Clear memory.
   Set db = Nothing
   Set task = Nothing

   ' Open the file.
   Shell "Notepad " + Chr(34) + Path + "Sample-Customers.FXD" + Chr(34)
End Function

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
   ' exist and we need to exit the routine.
   If Len(PathCheck) <  1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If
```

```
   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function


Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub
```

The `ExportDatabaseTSV()` function shows the simplest of the three export methods. This function follows these steps, which are found in the other two functions, `ExportDatabaseASC()` and `ExportDatabaseFXD()`, as well.

1. Deletes the old file when it exists.
2. Opens the source database and checks for errors.
3. Creates the task.
4. Configures the task (which means telling IDEA which fields to include as a minimum). This is the step where the three functions differ.
5. Performs the task. In this case, you must provide six arguments to `PerformTask()`: the path and file name of the output file, the name of the table as it appears in the output, the kind of conversion you want to perform (`.tsv`, `.asc`, or `.fxd`), the starting record number, the ending record number, and the equation you want to use to express the filtering criteria.
6. Clears memory.
7. Opens the resulting file. Because Notepad resides in the Windows directory, you can call it directly. All that you need to provide is the path to the file you want to open as shown in the example code.

The `ExportDatabaseASC()` function has two additional task configuration items. First, you can set `IncludeFieldNames` to determine whether IDEA outputs the field

names in the first row. Setting this value to `True` outputs the field names. The default is not to output the field names. Second, you must set the `Separators` property, which takes two arguments. The first argument sets the field delimiter, while the second sets the character used for decimals.

The `ExportDatabaseFXD()` function sets just one additional task configuration item. Like the `ExportDatabaseASC()` function, you can set `IncludeFieldNames` to `True` if you want to output the field names as the first row.

Managing Excel Data

Excel is a popular spreadsheet application that offers a number of ways to analyze data. Microsoft provides a wealth of built-in equations you can use to look at data in certain ways. You can also graph and chart the data with ease. For these reasons, and others, you might find that you export data to Excel for your own use, as well as to let others see the data you've worked with. The following sections describe how to import and export Excel data.

Importing Excel Data

Excel data isn't nearly as hard to import as other kinds of data, such as that found in text, because Excel uses a grid layout for data that translates extremely well into a database. One of the considerations for importing Excel data is that the field names you want to use should appear in the first row of the worksheet. In addition, you must remove total, subtotal, and blank rows. Listing 14.12 shows how to perform this task.

LISTING 14.12 Importing Excel Data

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Delete the old file.
   DeleteOld Path + "Excel-Sample-Customers.imd"

   ' Locate the input file.
   Dim dbName As String
   dbName = Client.LocateInputFile (Path + "Sample-Customers.xlsx")

   ' Define the task.
   Dim task As ImportExcel
   Set task = Client.GetImportTask("ImportExcel")

   ' Configure the task.
   task.FileToImport = dbName
```

```
   task.SheetToImport = "Sample-Customers"
   task.OutputFilePrefix = "Excel"
   task.FirstRowIsFieldName = "True"
   task.EmptyNumericFieldAsZero = "False"

   ' Perform the task.
   task.PerformTask

   ' Clear memory.
   Set task = Nothing

   ' Open the new database.
   ' dbName = task.OutputFilePath("Sample-Customers")
   OpenDB(Path + "Excel-Sample-Customers.imd")
End Sub

Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

The code begins by obtaining the working directory and deleting the old file as normal. The code then obtains the complete path information for the input file using `Client.LocateInputFile()`.

The next step is to create and configure the task. As with other examples in the chapter, the code relies on `Client.GetImportTask()` to create the task. The

configuration process includes setting the input file name, the name of the worksheet to import, and the prefix to use for the output databases. `FirstRowIsFieldName` lets you tell IDEA that the first row of the worksheet contains field names when set to `True`. The `EmptyNumericFieldAsZero` property determines whether IDEA interprets blank numeric fields in the worksheet as zeroes. (Setting this property to `True` creates a column made up of cells that contain either numbers or blanks, but no other data formats. An undocumented consequence of using this feature is IDEA imports columns that contain only dates or blanks as date type.)

At this point, the code calls `PerformTask()` to perform the import process. Finally, the code clears memory and opens the resulting database using `OpenDB()`.

Exporting Excel Data

There's only one technique for importing Excel data, no matter what form the Excel data takes. However, exporting Excel data isn't quite as straightforward. IDEA supports three variations of Excel export. Fortunately, the techniques for all three levels are about the same. The only difference is the value you use for the kind of export. When working with Excel, you have these three export options:

- **XML1:** Creates an Excel 2002 XML spreadsheet. (This option may produce an output file that Excel will have a problem reading.)
- **XLSX:** Creates an Excel 2007 spreadsheet.
- **XLS8:** Creates an Excel 97 to Excel 2003 spreadsheet.

Because there's only this one little difference, once you know how to export to Excel using one technique, you know them all. Listing 14.13 shows how to perform an Excel 2007 spreadsheet export.

LISTING 14.13 Exporting Excel Data

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path + "Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If
      ' Delete the old file.
   DeleteOld Path + "Sample-Customers.XLSX"
```

```
   ' Define the task.
   Dim task As ExportDatabase
   Set task = db.ExportDatabase

   ' Configure the task.
   task.IncludeAllFields

   ' Perform the task.
   task.PerformTask Path + "Sample-Customers.XLSX", _
      "Sample-Customers", "XLSX", 1, db.Count, ""

   ' Clear memory.
   Set db = Nothing
   Set task = Nothing

   ' Open the spreadsheet
   Shell "cmd.exe /c " + Chr(34) + Path + _
      "Sample-Customers.XLSX" + Chr(34)
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
   ' exist and we need to exit the routine.
   If Len(PathCheck) <  1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function


Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )
```

(*continued*)

LISTING 14.13 (*Continued*)

```
   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub
```

The code begins by obtaining the working directory and opening the source database using `OpenDB()`. After the source database is opened, the code checks for errors as normal. At this point, the code deletes the old worksheet, if any.

The next step is to create the task, which relies on the `ExportDatabase` object. The code tells IDEA which fields to include next (all of them in this case), and then calls `PerformTask()`. In this case, `PerformTask` requires these inputs:

- Output file name
- Table name in the output file
- Type of export
- Starting record number
- Ending record number
- Equation used to define the filtering criteria

When IDEA completes the export task, the code clears memory. The final task is to open the exported file using the `Shell()` function.

Managing Access Data

Like Excel, you can import and export Access data. Microsoft Access is a small database product—sort of a smaller version of SQL Server. Many people use it as a desktop database because it provides everything needed to manage information using a relational database setup. The following sections describe how to import and export Access databases.

Importing Access Data

Because Access is another database product, importing data from it is relatively straightforward. One of the major differences with IDEA databases though is that a single Access database can contain multiple tables as well as other data items. (IDEA is only capable of importing the tables from an Access database so any queries that you need to import should be converted to tables.) Consequently, the import process must consider which table you want to import within the Access database, as shown in Listing 14.14.

LISTING 14.14 Importing Access Data

```
Sub Main
   ' Define constants used to determine the amount of scanning
   ' performed for record length.
   Const SCAN_ALL = -1
   Const SCAN_NONE = 0

   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Delete the old file.
   DeleteOld Path + "Access-SampleCustomers.imd"

   ' Create the task.
   Dim task As ImportAccess
   Set task = Client.GetImportTask("Access")

   ' Configure the task.
   task.InputFileName = Path + "Sample-Customers.ACCDB"
   task.OutputFileNamePrefix = "Access"
   task.CreateRecordNumberField = False
   task.DetermineMaximumCharacterFieldLengths = 10000
   task.AddTable("SampleCustomers")

   ' Perform the task.
   task.PerformTask

   ' Clear memory.
   Set task = Nothing

   ' Obtain the resulting filename and open the database.
   OpenDB(Path + "Access-SampleCustomers.imd")
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

(*continued*)

LISTING 14.14 (*Continued*)

```
Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub
```

Just because Microsoft touts Access as a desktop database doesn't mean you can't store a lot of information using it. Consequently, the Access tables can be quite large. However, IDEA still needs to know what size to make the fields of the target database. Therefore, one of the things that you commonly see defined for an Access database import is constants that tell whether to scan all of the records in the table or none of the records in the table. You can also supply a specific number of records to scan, as described later in this section.

The code begins by obtaining the working directory and deleting the old import database. You have to exercise some caution during the import process because it's possible that you'll have to delete multiple databases when you import multiple tables from Access.

The next step is to create and configure the task. As with an Excel import, an Access import relies on `Client.GetImportTask()` to create the task. The code must define an input file name and an output file name prefix. In addition, the code determines whether the output database has a record number field using the `CreateRecordNumberField` property.

It's extremely important that you define the `DetermineMaximumCharacterFieldLengths` property. Scanning all of the records in a large database can cause delays in importing the data. Of course, you need to scan enough records to ensure that the fields in the target IDEA database are large enough. The example uses 10,000 records, which is usually more than enough to get a good sample.

The code finishes configuring the task by determining which Access table to process. You can use `AddTable()` repeatedly to add multiple tables from a single Access database. The target databases will use a combination of the database name prefix and the source table name. For example, if the `OutputFileNamePrefix` property is `Access` and the Access table name is `SampleCustomers`, the resulting IDEA database name is `Access-SampleCustomers.imd`.

At this point, the code calls `PerformTask()` to perform the actual import. The code clears memory and finishes by opening the resulting database.

Exporting Access Data

Exporting data to Access works very much like the Excel export described in the "Exporting Excel Data" section. In fact, as shown in Listing 14.15, the main difference is the export option you choose. Access uses the following three keywords for export purposes:

- **MDB2000:** Creates an Access 2000/2003 database.
- **MDB2007:** Creates an Access 2007 database.
- **MDB97:** Creates an Access 97 database.

LISTING 14.15 Exporting Access Data

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path + "Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Delete the old file.
   DeleteOld Path + "Sample-Customers.ACCDB"

   ' Define the task.
   Dim task As ExportDatabase
   Set task = db.ExportDatabase

   ' Configure the task.
   task.IncludeAllFields

   ' Perform the task.
   task.PerformTask  Path + "Sample-Customers.ACCDB", _
      "SampleCustomers", "MDB2007", 1, db.Count, ""

   ' Clear memory.
   Set db = Nothing
   Set task = Nothing

   ' Open the database
   Shell "cmd.exe /c " + Chr(34) + Path + _
      "Sample-Customers.ACCDB" + Chr(34)
End Sub
```

(*continued*)

LISTING 14.15 *(Continued)*

```
Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
   ' exist and we need to exit the routine.
   If Len(PathCheck) <  1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function


Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub
```

Note

IDEA also supports Open Database Connectivity (ODBC) data sources. ODBC is an extremely flexible option because most database products support it directly or by using drivers you download from third-party sites. Using ODBC is more complex than other options because you first locate an appropriate driver. Just because you

have a driver for MySQL doesn't mean the driver will also work for Sybase. After you download the driver, you must configure it using the ODBC Data Source Administrator. Every driver has a different configuration routine. ODBC is also notoriously slower than other import solutions. Consequently, you obtain significant flexibility and reliability (because ODBC has been around so long, the drivers tend to work exceptionally well), for slower speeds and considerable complexity. Because ODBC doesn't provide a well-defined import route, it isn't covered in this book.

Summary

This chapter has helped you understand the requirements for importing and exporting data. In addition, you've discovered a few new techniques for extracting data so that you export just the data you need or analyze just the data you want to view. IDEA provides a formidable array of options for importing and exporting data. If you're looking for the best format for transmitting data, PDF files appear on just about every machine—so it's very much a winning solution.

Now that you understand how to import and export data, you should give it a try. Begin by making a list of the file formats used most often by your organization. Create a plan for dealing with each of the formats (make sure the plan is written so that others can use it too). Now, take the time to start writing some test applications. It's important to write applications that test your ability to move data from one format to another without error. Once you build applications that can perform the task successfully, you can begin working with actual data and perform imports and exports for real.

Chapter 15 considers use of another kind of data transfer, working with files. You can use external files for all kinds of purposes other than working with data. For example, you can use text files to store settings for your application. Text files are also one of the most common ways to exchange information between different platforms (with XML becoming far more common because it retains the data formatting information).

CHAPTER 15

Working with Files

You have files all over your hard drive. Some files are quite specific. For example, when you open an IDEA database, you're opening a file. Likewise, when you open Microsoft Excel to see some data you exported from IDEA, you're opening a file. These specific kinds of files don't require any special processing because IDEA already has functionality built into it to handle them. However, you might encounter files that are unique in some way—they might not even contain data that you analyze, but rather contain information of a different sort—perhaps configuration information for your application. IDEAScript can work with these kinds of files too! All you need to do is provide a little code to tell IDEAScript what to do with the file. This chapter is all about working with these special kinds of files.

Considering the File Format

Your hard drive contains many different kinds of files. Some of the files are executables, which are completely unreadable by humans. A few of the files contain pure text data, which is easily read by humans using a simple editor such as Notepad. Many of the files are a mix of readable and unreadable information. For example, you can open an XML file in Notepad and probably figure out what the XML means, but it isn't as easy to understand as pure text. Files, such as those created by Microsoft Word and Microsoft Excel, contain readable information, but the unreadable information mixed with it makes it hard for most people to understand. In fact, without some special tricks, you can't normally read Word and Excel files directly at all.

The format of the file you interact with is important because it determines just how easy it will be to read the information using IDEAScript. Theoretically, with the right programming, you could read any file that exists on your hard drive, even executables (not that you'd understand the executable content). In short, even though you can read just about anything using IDEAScript, as a practical matter, the files that are easiest to read are those that contain the most human readable information. You need to consider this issue when you try to create an application to read files using IDEAScript.

In addition to the raw data format, you also need to consider how much information you have about the file. Some files have detailed specifications that explain how the file is constructed. The more information you have about the file, the easier it is to build an application to read it. Consequently, the easiest files to read are often the files you create and document yourself (assuming that you provide full file documentation).

Another factor in reading files is your ability to understand complex formats. Some files require you to understand individual bits within the file and to know how to perform bit manipulation. Even if you have good documentation, working at the bit level is cumbersome, error prone, and difficult. Consider your skill level before you begin working with complicated files that require special programming.

The bottom line is that you need to think about the kind of file you want to read before you begin writing code to read it. Depending on the complexity of the file, you might choose to find some other method of interacting with it. For example, the application that created the file might provide an alternative output format that's easier to read.

Using the File IO Features

IDEAScript has a wealth of functions you can use to interact with files. These functions let you do things like set the location on the hard drive, read and write files, and determine when you're at the end of a file. In fact, there's a function for every major file task as described in the sections that follow.

ChDir

The `ChDir()` function changes the directory that IDEAScript is currently using. Normally, your application starts out using your `My Documents` folder (or simply `Documents`, depending on the version of Windows you have installed). In order to work with other locations on the hard drive, you must change directories. The `ChDir` function can accept a number of arguments as shown here:

```
ChDir [[drive:/] | [/]]dir[/dir[/dir...]]
```

As a minimum, you must supply the directory you want to change to. However, you can add a drive specification if you want. If you do that, you must remember to add the slash (/) or backslash (\). You can also add just a slash or backslash to start at the root directory of the current drive. It's also possible to change to a directory at any level in the directory structure. Here are examples of the `ChDir()` function.

- `ChDir "Test"`
- `ChDir "\Test"`
- `ChDir "C:\Test"`
- `ChDir "C:\Test\IDEA"`
- `ChDir "Test\IDEA"`

Destination directories that begin with a drive specification, slash, or backslash are called *absolute* locations. The destination begins from an absolute location and will always end at a specific place on the hard drive. Destination directories that begin with a directory name are called *relative* locations. The destination is relative to the current location. For example, if you use `ChDir IDEA` the `ChDir()` function will move to the IDEA subdirectory of the current directory, which could be anywhere on the hard drive. Listing 15.1 shows some examples of the `ChDir()` function in action. Before you begin working with this example, create a `Test` directory off the *root* (the topmost directory, which is normally `C:\`) of your hard drive using Windows Explorer. The result is a directory named `C:\Test` in most cases. The `Test` directory lets you try things out without possibly damaging other data on your machine.

LISTING 15.1 Working with Directories

```
Sub Main

   ' Display the starting location.
   MsgBox "Starting at: " & CurDir

   ' Make sure we're on the C drive.
   ChDrive "C:"

   ' Make sure we're using the test directory.
   ChDir "\Test"

   ' Display the current directory.
   MsgBox "Current Directory: " & CurDir$

   ' Create a new directory.
   MkDir "IDEA"

   ' Change to the new directory.
   ChDir "IDEA"

   ' Display the new directory location.
   MsgBox "Current Directory: " & CurDir

   ' Change back to the root directory.
   ChDir ".."

   ' Display the new directory location.
   MsgBox "Current Directory: " & CurDir

   ' Remove the directory we created.
   RmDir "IDEA"

End Sub
```

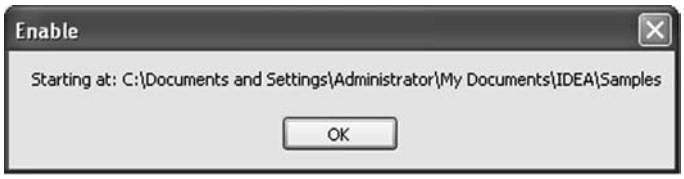


FIGURE 15.1 The First Message Box Shows the Starting Location for the Application

This example begins by showing the default directory—the one in which the application begins. You can read about the `CurDir()` and `CurDir$()` functions in the "CurDir and CurDir$" section. Figure 15.1 shows an example of the output. Of course, your message box will show the default directory for your machine.

The next step is to change the drive and directory. The `ChDrive()` function changes the drive. You can read about this function in the "ChDrive" section. The `ChDir()` function uses an absolute directory location of `\Test`, which is the `Test` subdirectory of the root directory of your hard drive. The next call to the `MsgBox()` function displays the new drive and directory location, as shown in Figure 15.2.

At this point, the code uses the `MkDir()` function to create the IDEA subdirectory below the `\Test` directory. Consequently, the new directory's absolute location is `C:\Test\IDEA`. You can read more about the `MkDir()` function in the "MkDir" section. The default directory is still `C:\Test`. To change directories to the new subdirectory, the code calls `ChDir()` using a relative directory location of IDEA. You can see the results in Figure 15.3.

It's easy to move up or down the directory list using relative locations if you know a little trick. When you call `ChDir ".."` the `..` (two dots) represent the parent directory. The default directory moves up one directory location, as shown in Figure 15.2.

The final step is to clean up the `C:\Test` directory using the `RmDir()` function. This function removes the IDEA directory that the application created earlier. You can read more about the `RmDir()` function in the "RmDir" section. You can see an additional example of the `ChDir()` function in Listing 15.2 in the "Close" section.

FIGURE 15.2 Changing the Drive and Directory Sets the Default Directory to a Known Location on the Hard Drive

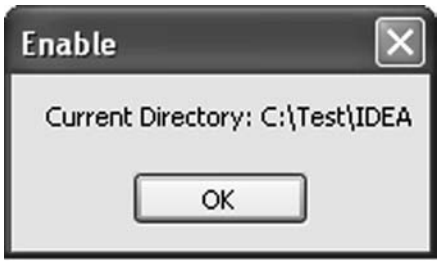


FIGURE 15.3 The New Drive Location Is the IDEA Subdirectory.

ChDrive

The `ChDrive()` function changes the drive that IDEAScript is currently using to process files. Normally, your application starts on the same drive as the `My Documents` (or simply `Documents`) folder. If you want to access files on other drives or you want to be sure that the application is using the correct drive, you use the `ChDrive()` function to change it. The `ChDrive()` function can accept the argument shown here:

```
ChDrive drive:[/]
```

LISTING 15.2 Working with Files

```
Sub Main
    ' Change directories to the test directory.
   ChDir "C:\Test"

   ' Define the file path.
   Dim FilePath As String
   FilePath = CurDir & "\TestFile.txt"

   ' Create some content.
   WriteToFile FilePath

   ' Read the content we created.
   ReadFromFile FilePath

   ' Copy the content to another file.
   CopyFile FilePath

   ' Perform a binary mode copy.
   CopyBinary FilePath

   ' Obtain the file statistics.
   GetFileStatistics FilePath
End Sub
```

As a minimum, you must supply the letter of the drive you want to use, followed by a colon, such as `C:`. You may optionally supply a slash or backslash. Here are some examples of the `ChDrive()` function in use.

- `ChDrive C:`
- `ChDrive D:\`
- `ChDrive E:\`

Close

The `Close()` function closes a file you open using a function such as `Open()`. To close a file, you need to supply `Close()` with the file number used to open it. A file number is a representation of the *handle* for that file—the means you use to grab the file and work with it. Think about a pot or a pan—in order to grab the pot or pan you need a handle. Likewise, the file number is the handle you use to perform tasks with files.

The examples used to demonstrate the various file manipulation techniques that IDEAScript supports are connected together so you can better see how they work. Listing 15.2 shows the `Main()` function used to connect the file-related examples together.

This part of the application begins by changing directories to the `C:\Test` directory. Notice that this form of the `ChDir()` function includes a drive letter. You don't actually need to use the `ChDrive()` function to change drives unless you want to do so for clarity or other reasons. The next step is to create the `FilePath` variable, which contains the complete path and file name for the test file, `TestFile.txt`. Instead of typing out the entire path, the example uses the `CurDir()` function to retrieve the required information.

In order to provide you with a relatively complete view of what IDEAScript can do, the example performs five different tasks with the example file:

1. Writes to the file, actually creating a new file and then placing content in it. You'll discover two essential techniques for writing to files.
2. Reads data from the file to show what the file contains. The example uses several different techniques to perform this task.
3. Creates a copy of the file. It's essential to know how to perform this task so that you can perform tasks on a copy of the file, rather than modify the original.
4. Performs a binary copy of the file. Files that you can read directly are called ASCII (American Standard Code for Information Interchange) or *text* files. These files typically don't include any formatting or other special information, just characters that you can read directly. *Binary* files contain characters that humans can't read directly. For example, a binary file might contain special codes that format the text. Executable files—those that contain code that the machine can understand and graphics files are also binary files. A binary copy lets you create a copy of both ASCII and binary files.
5. Obtains file statistics. Although this part of the example is short, it's often helpful to know specifics about a file, such as whether the file is executable, rather than human readable. As you perform work with more files, knowing as much as you can about the file becomes more important.

LISTING 15.3 Checking the Existence of a File

```
Function CheckFile(SearchSpec As String) As Boolean
   ' Get the specified directory.
   TestPath = Dir(SearchSpec)

   ' Check for the specified directory.
   If Len(TestPath) > 1 Then
      CheckFile = True
   Else
      CheckFile = False
   End If
End Function
```

Many of the examples need to know whether a file exists before they attempt to make changes to the file. For example, you can't open a file for reading unless you know that the file exists. Likewise, you can't create a new file with the same name as an existing file. Trying to copy an existing file to a file that has the same name as an existing file won't work either. For all these reasons, and many more, you need to know whether a file exists before you use it. Listing 15.3 shows a simple function that returns `True` or `False` based on whether a file exists.

The `CheckFile()` function begins by using the `Dir()` function to look for the file. Because this is a specific file path with a full path and file name, only one file can match the specification. When `Dir()` is successful, `TestPath` contains the path and file name as output. When the length of `TestPath` is greater than `1`, the test succeeds—the file exists and `CheckFile()` returns `True`. Otherwise, `CheckFile()` returns `False`.

It's time to look at the first part of the example code, `WriteToFile()` as shown in Listing 15.4. This part of the example creates a text file as output. The text file contains some simple text on completion.

The example begins by checking for the existence of the output file as shown in Figure 15.4. If you've already run the example once, then the file exists and the code needs to delete it before recreating it. The code asks whether you want to delete the file as shown in Figure 15.5. If you answer Yes, then the code uses `Kill()` to delete the file. Otherwise, it exits the subroutine using `Exit Sub`.

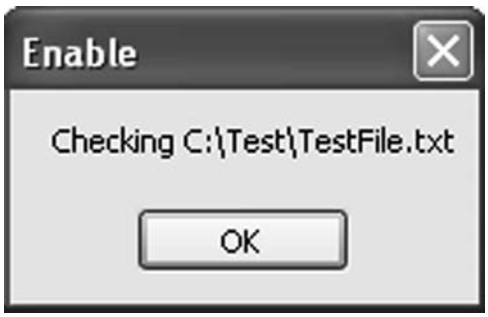


FIGURE 15.4 The Application Displays the Path of the File It Plans to Check.

LISTING 15.4 A Simple File Writing Example

```
Sub WriteToFile(FilePath As String)
    ' Verify that the file we want to create doesn't already exist.
   MsgBox "Checking " & FilePath
   If CheckFile(FilePath) = True Then

      ' Ask about the existing file.
      Response = MsgBox("Do you want to overwrite the file?", MB_YESNO)

      ' Act on the response.
      If Response = IDYES Then
         ' Remove the old file.
         Kill FilePath
      Else
         ' Exit without changing the file
         Exit Sub
      End If
   End If

   ' Obtain the next file number.
   FileNum = FreeFile

   ' Create a new file.
   Open FilePath For Output As FileNum

   ' Write data to the file.
   Write #FileNum, "Hello", "World"

   ' Use the Print function instead.
   Print #FileNum, "Some ", "More ", "Input!"

   ' Close the file.
   Close FileNum
End Sub
```

The next step is to open a file. The act of opening a file that doesn't exist creates one. Instead of opening the file, you get a new one. The code begins with the `FreeFile()` function, which returns the next unused file handle. Always use the `FreeFile()` function, rather than hard code handle values for your code to prevent potential accidents.

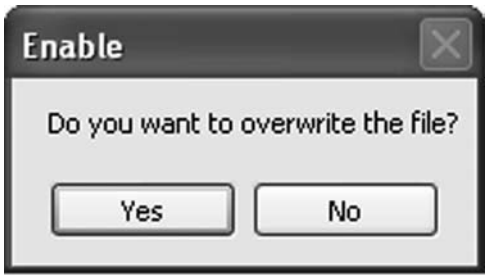


FIGURE 15.5 The Application Asks Whether You Want to Delete the Existing File

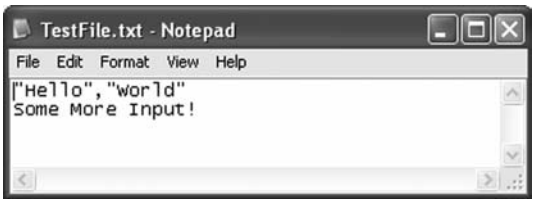


FIGURE 15.6 Writing to a File Produces a Different Result Than Printing to a File

After the code obtains a handle, it uses the `Open()` function to create the file. In most cases, you must specify three things to make the `Open()` function work:

1. The name of the file, which is contained in `FilePath` for this example.
2. How you want the file opened—essentially, what task you want to perform with the file. In this case, the code uses the file for output, so it uses the `For Output` keywords.
3. The handle you want to use to work with the file, which is contained in `FileNum` for this example.

At this point, the file is ready for use. The example provides two different output methods: `Write #()` and `Print #()`. These two functions output the data in different ways, as shown in Figure 15.6. Notice that the `Write #()` function places each parameter you provide in double quotes. Each of the parameters is separated by a comma. In addition, the `Write #()` function ends by adding a carriage return/line feed combination to the output. The `Print #()` function handles things differently. It outputs the text precisely as you supply it. If you provide three expressions, but don't include the required spaces between them, you won't see the spaces in the output.

The `WriteToFile()` function ends by closing the file using the `Close()` function. It's essential that you always close the file when you're done using it. Once you close the file, the handle used to access it becomes invalid. You can see examples of using the `Close()` function in a number of examples in this chapter—most of which show how to close a single file. Listing 15.9 in the "Get" section shows how to close two files at once using the `Close()` function.

Warning

Failure to close files after you modify them could result in data loss, among other problems. In addition, not closing files could create a *memory leak* in your application, where memory becomes inaccessible because Windows thinks that the memory is still in use. A third problem with not closing files is that the file handles become inaccessible, which means you must constantly allocate new file numbers. In summary, not closing files can cause all kinds of problems in your application. The best idea is to immediately write the `Close()` function every time you use a function that opens a file in some way.

CurDir and CurDir$

The `CurDir()` and `CurDir$()` functions return the current directory that IDEAScript is using. The current directory is also called the *path* and it includes both the drive letter and the absolute directory location. Neither of these functions require (or accept) input arguments. The `CurDir()` function returns a `Variant` object that contains the path, while the `CurDir$()` function returns a string that contains the path. Listing 15.1 in the "ChDir" section shows how to use the `CurDir()` and `CurDir$()` functions with other functions to work with a number of locations on the hard drive. You can see additional examples of the `CurDir()` and `CurDir$()` functions in Listing 15.2 in the "Close" section and Listing 15.7 in the "FileCopy" section.

Dir

The `Dir()` function returns zero or more file names and directories based on a path specification you provide. You can also search for files based on file attributes, such as whether a file is marked as Read-only. If you combine both a path specification and a list of attributes, you can search for file names or directories with specific characteristics. For example, you could search for just the files that have their archive attribute set in order to perform a backup of the files on a drive. The "GetAttr" section tells you more about using attributes.

A path specification can appear in a number of formats. You might decide to look for a specific file name. In this case, you'd provide a full path and full file name specification such as `C:\Test\TestFile.txt`. However, if you're interested in the content of an entire directory, you'd provide just the directory specification, such as `C:\Test\`.

The path specification can also include wildcard characters. The asterisk (`*`) includes any number of characters, while the question mark (`?`) represents just one character. A specification such as `C:\Test\T*.*` would look for any file that begins with "T" and has any extension associated with it. If you want to be more particular, you might provide a file specification of `C:\Test\T*.txt`, which would limit the search to text files. A file specification of `C:\Test\T??.*` would limit the search to file names that contain three letters, begin with the letter "T," and have any file extension. As you can see, path specifications can prove quite flexible in defining precisely what you want to see as output.

The first time you use the `Dir()` function, you get back a single file name or directory, assuming that `Dir()` finds one that matches the combination of path specification and attributes that you specified. After the first call, each successive call to `Dir()` without any additional input provides another directory or file name. Listing 15.5 shows an example of how you might use the `Dir()` function in a loop to read a list of files or directories.

In this example, the code begins by changing directories to a folder that contains files to check. The example then creates `DirList`, which is a string that contains the output. It uses `DirList` as the output for the first call to `Dir()`, which includes the file specification of `*.txt`. If this first call fails, there aren't any files in the directory that match the specification. The code displays an informative message and then exits.

LISTING 15.5 Displaying a List of Files

```
Sub Main
   ' Change directories to the test directory.
   ChDir "C:\Test"

   ' Obtain the first directory listing.
   Dim DirList As String
   DirList = Dir("*.txt")

   ' Verify that the application actually found some files.
   If Len(DirList) = 0 Then
      MsgBox "No files found!"
      Exit Sub
   End If

   ' Now that you have an initial entry, start processing any
   ' additional entries.
   Dim ThisEntry As String
   ThisEntry = ""
   Do
      ThisEntry = Dir
      DirList = DirList & Chr(13) & ThisEntry
   Loop While ThisEntry <>  ""

   ' Display all of the entries on screen.
   MsgBox "Files in the Directory:" & Chr(13) & DirList
End Sub
```

When `Dir()` does find a file or directory that matches the specification, `DirList` will have a length greater than `0`. The code then creates `ThisEntry`, a variable that contains a single directory or file name. The use of a `Do...Loop While` loop ensures that the code will at least check once for the next entry using `Dir()`. The code adds anything found in `ThisEntry` to `DirList` and separates the entry with a carriage return so it appears on another line. The code continues until there aren't any more entries to process and then displays the list as output. Figure 15.7 shows typical output from this example.

Of course, there are many ways to use the `Dir()` function. You can see an additional example of the `Dir()` function in Listing 15.3 found in the "Close" section.

EOF

Every file has an end. Of course, your code doesn't know about that until you tell it that it has reached the end. The `EOF()` function performs one task—it provides an indicator when there isn't any more data to read from a file.

IDEAScript provides a number of methods for reading data from a file. For example, you can simply place the entire content of a file into a variable using a single read. Listing 15.6 shows two different ways to read a file.

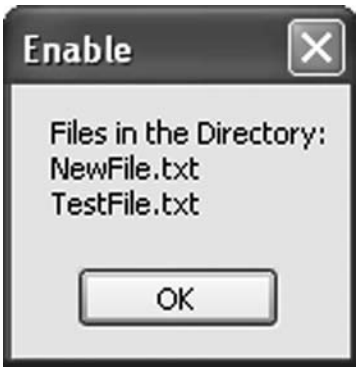


FIGURE 15.7 Dir( ) Works within a Loop to Display All of the Directories and File Names That Match a Specification

The example begins by obtaining a file handle number using `FreeFile()`. It then opens the file specified by `FilePath` for input using the `Open()` function. At this point, the code can start reading the file.

The first read example uses the `Input()` function. Because the file is already open, the code uses the `LOF()` function to obtain the file length as the first argument for the

LISTING 15.6 Reading a File

```
Sub ReadFromFile(FilePath As String)
   ' Reopen the file for reading.
   FileNum = FreeFile
   Open FilePath For Input As FileNum

   ' Read the entire file.
   FileContent = Input(LOF(FileNum), FileNum)

   ' Display the content on screen.
   MsgBox "The entire file contains: " & FileContent

   ' Look for a particular location in the file.
   Seek FileNum, 9

   ' Read to the end of the file one character at a time.
   FileContent = ""
   Do While Not EOF(FileNum)
      FileContent = FileContent & Input$(1, FileNum)
   Loop

   ' Display the content on screen.
   MsgBox "Starting from position 9: " & FileContent

   ' Close the file.
   Close FileNum
End Sub
```

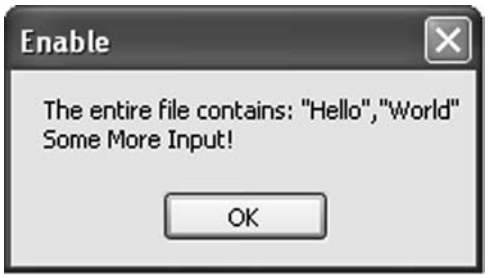


FIGURE 15.8 Use a Combination of Input() and LOF() to Display the Entire File

`Input()` function. You could also use the `FileLen()` function to obtain the file length if the file were closed. The second argument is the file handle. The code then displays `FileContent`, which contains the full content of the file, as shown in Figure 15.8.

You may decide that you don't want to read from the beginning of the file. The second read example shows how to perform this task. In this case, the code begins by using `Seek()` to place the file pointer at the ninth character in the file. The *file pointer* is an indicator that IDEAScript maintains that shows where the last read occurred. Every time your code reads from the file, IDEAScript moves the file pointer the required number of characters so that the next read resumes where you left off.

The next step is to read the remaining data from the file and place it in `File Content`. Instead of reading the entire file at once, this part of the example reads the file one character at a time. Using this technique lets you check each character as you read it and change it if necessary. The `Do While...Loop` structure relies on the `EOF()` function to determine when the entire file has been read. Notice that this loop uses the `Input$()` function with a read length of only `1` to read the input one character at a time. Figure 15.9 shows the output from this part of the example.

The `ReadFromFile()` subroutine ends with a call to `Close()`. Always remember to close files after you work with them.

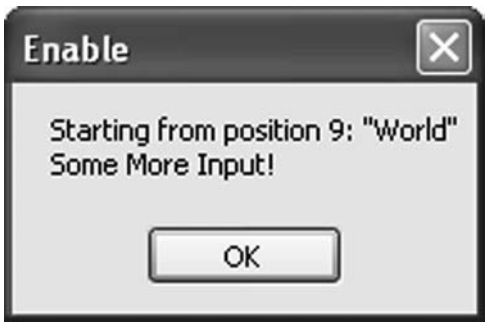


FIGURE 15.9 Use Seek() to Locate a Particular Starting Point in a File

FileCopy

The `FileCopy()` function lets you copy a file from one location on the hard drive to another location. The result is two files with precisely the same information. You could use the file copy to perform various tasks without affecting the original copy. The `FileCopy()` function could also provide a means for performing backups of your data files. Listing 15.7 shows how to copy files, rename them, and then read the content to verify the content of the copy.

`CopyFile()` begins by creating a copy of the file. You don't need to open the file to copy it (in fact, you don't want to open it). The original file specified by `FilePath` is copied to `TestFile2.txt`. Notice the use of `CurDir$()` to place the copy in the current directory.

At this point, the example tries to rename the file. However, before the code can rename the file, it must ensure that the file doesn't exist. The code calls `CheckFile()` (see Listing 15.3) to check for the existence of the file. If the file exists, the code calls on `Kill()` to remove it. The example then uses the `Name()` function to rename the file to `NewFile.txt`.

The code uses `Open()` to open the new file for input. The fact that `Open()` works shows that the `Name()` function worked as expected. The code calls `Line Input#()` to read just the first line of the new file and display it on screen using the `Print()`

LISTING 15.7 Copying Files

```
Sub CopyFile(FilePath As String)
   ' Copy the file to another location.
   FileCopy FilePath, CurDir$ & "\TestFile2.txt"

   ' Rename the file something else.
   Dim NewFilePath as String
   NewFilePath = CurDir$ & "\NewFile.txt"
   If CheckFile(NewFilePath) Then
      Kill NewFilePath
   End If
   Name CurDir$ & "\TestFile2.txt" As NewFilePath

   ' Open this new file for reading.
   FileNum = FreeFile
   Open NewFilePath For Input As FileNum

   ' Obtain a single line of text from the file.
   Line Input #FileNum, FileContent

   ' Display the text on screen.
   Print "Content from NewFile.txt: ", FileContent

   ' Close the file.
   Close FileNum
End Sub
```

FIGURE 15.10 The Print( ) Function Works Like a Simple Form of the MsgBox( ) Function

function. The `Print()` function works like a very simple form of the `MsgBox()` function, as shown in Figure 15.10.

The `CopyFile()` function ends by calling `Close()`. Always close any file you open when you're done using it.

FileLen

The `FileLen()` function obtains the size of a file on disk without opening it. You simply provide the path and name of the file you want to examine. The `FileLen()` function comes in handy for the same sorts of tasks as the `LOF()` function. However, you can also use it to determine the memory requirements for working with a file in advance. When you combine the `FileLen()` function with the `GetAttr()` function, you can determine whether your application should even process a particular file. Figure 15.8 shows how to obtain file statistics for a file (the constants used for this example appear in Listing 15.11 in the "GetAttr" section).

The `GetFileStatistics()` function begins by obtaining the length of the requested file. All you need for this part of the example is the `FileLen()` function.

The next step is to set the file attributes to a known value using the `SetAttr()` function (explained fully in the "SetAttr" section). File attributes determine whether a file is a system file (which you probably don't want to open), hidden, read-only, or otherwise unsuitable for your application. You can also use attributes to determine whether there's a

LISTING 15.8 Obtaining File Statistics

```
Sub GetFileStatistics(FilePath As String)
   ' Get the file length.
   FileLength = FileLen(FilePath)

   ' Set the file attributes.
   Dim Attributes As Integer
   Attributes = Archive Or ReadOnly
   SetAttr FilePath, Attributes

   ' Get the file attributes.
   Dim FileType As String
   Attributes = GetAttr(FilePath)
```

(*continued*)

LISTING 15.8 *(Continued)*

```
   ' Create a string that shows the attributes.
   If (Attributes And ReadOnly) = ReadOnly Then
      FileType = Chr(13) & Chr(9) & "Read-Only"
   End If
   If (Attributes And Hidden) = Hidden Then
      FileType = FileType & Chr(13)  & Chr(9) & "Hidden"
   End If
   If (Attributes And System) = System Then
      FileType = FileType & Chr(13)  & Chr(9) & "System"
   End If
   If (Attributes And Directory) = Directory Then
      FileType = FileType & Chr(13)  & Chr(9) & "Directory"
   End If
   If (Attributes And Archive) = Archive Then
      FileType = FileType & Chr(13)  & Chr(9) & "Archive"
   End If
   If (Attributes And Normal) = Normal Then
      FileType = FileType & Chr(13) & Chr(9) & "Normal"
   End If
   If (Attributes And Temporary) = Temporary Then
      FileType = FileType & Chr(13)  & Chr(9) & "Temporary"
   End If
   If (Attributes And Compressed) = Compressed Then
      FileType = FileType & Chr(13)  & Chr(9) & "Compressed"
   End If
   If (Attributes And Offline) = Offline Then
      FileType = FileType & Chr(13)  & Chr(9) & "Offline"
   End If
   If (Attributes And Indexed) = Indexed Then
      FileType = FileType & Chr(13)  & Chr(9) & "Indexed"
   End If
   If (Attributes And Encrypted) = Encrypted Then
      FileType = FileType & Chr(13)  & Chr(9) & "Encrypted"
   End If

   ' Display the statistics.
   MsgBox "File Length: " & CStr(FileLength) & Chr(13) & _
      "Attributes for " & CStr(Attributes) & ": " & FileType

   ' Reset the attributes.
   SetAttr FilePath, Normal
End Sub
```

need to create a backup of the file. Any time a file has the archive attribute set, it requires backup. In this case, the example sets the file to have its `Archive` and `ReadOnly` attributes set. Notice that you use a logical `Or` to add the two values together.

The code then uses the `GetAttr()` function to determine the attributes of the file. The "GetAttr" section tells more about the `GetAttr()` function. The attributes that you

receive may actually represent more than one attribute type. For example, you might get attributes for both hidden and read-only back from `GetAttr()`. The series of `If...Then` statements that follow determine which of the attributes is set. Notice that the code uses a logical `And` to perform the comparison and that the `And` portion of the expression is in parentheses to ensure the code performs that part of the task first. In this case, the attributes include both `Archive` and `ReadOnly`. Figure 15.11 shows the output from this example.

When the code is finished, it resets the file attributes to `Normal` using `SetAttr()` again. If you leave the file in a read-only state, a second run of the example will fail because Windows won't let the code delete the file. You can see another example of the `FileLen()` function in Listing 15.9 in the "Get" section.

FreeFile

The `FreeFile()` function provides the next available handle (essentially a number) to use with the `Open()` function. It's important to use the `FreeFile()` function instead of hard coding handle numbers into your code to ensure your code works properly.

FIGURE 15.11 File Statistics Can Help You Work with Files More Efficiently

Otherwise, you could use the wrong handle to access a particular file and end up damaging data or creating other problems in your application. You can see an example of the `FreeFile()` function in Listing 15.4 in the "Close" section. In addition, you can see this function used in Listing 15.6 in the "EOF" section and Listing 15.9 in the "Get" section.

Get

The `Get()` function lets you perform a binary read of a data file. (A binary read lets you look at all of the characters in a file so that you can see control characters such as carriage return and line feed. Normally, you won't use binary reads in IDEA unless you want to perform a backup or work with files that have binary data in them, such as graphics files.) You can choose any starting position within the file and read only the amount of data you want. The `Get()` function requires three inputs: the handle for the file you want to read, the starting position for the read, and a buffer to hold the data from the read. A *buffer* is an area of memory that you set aside to hold data—think of it as you would any other storage container. The size of the buffer determines how much data it will hold. Listing 15.9 shows how to work with the `Get()` function.

The example begins by obtaining the attributes for the file you want to copy. Theoretically, the example code can copy any type of file—it doesn't have any limitations. However, in practice, you probably want to limit the kinds of files that someone can copy. If the file has its `Archive` attribute set or is a `Normal` file, the example code will copy it. Otherwise, the code displays an error message and exits.

The next step is to obtain the next available file handle and then use it to open the file. Notice that this version of the `Open()` function opens the file in `Binary Access Read` mode, which is a requirement for using `Get()`. Remember that `Get()` requires a buffer. In order to allocate enough memory to hold the file content, you must discover the size of the file and then create a buffer of the required size. (IDEA has a maximum buffer size of 65,534 bytes, so make sure you don't exceed this limit.) The next step performs this task using `String()`, which takes the number of characters to allocate (obtained using `FileLen(FilePath)`) and the kind of character to use for fill (a space works fine). At this point, the code calls `Get()` to obtain the file content.

Tip

If you find that the `Get()` function isn't working or not working as you expect, verify that you have created a buffer of sufficient size to hold the required data. Using a blank string for input to the `Get()` function always returns a buffer that doesn't contain any data. If the buffer is too small, the data is truncated.

Now that you have content to copy, it's time to create the output file. The example begins by creating an output file name. In this case, the code simply creates a file name that has a `.bak` extension in place of the original extension. The code then obtains the next available file handle and opens the file in Binary Access Write mode. You must use Binary Access Write mode with the `Put()` function (described in the "Put" section).

LISTING 15.9 Creating a Binary Copy for Backup Purposes

```
Sub CopyBinary(FilePath As String)
   ' Verify that the file is a standard file.
   Attributes = GetAttr(FilePath)
   If Not ((Attributes = 32) Or (Attributes = 128)) Then
      MsgBox "Can't copy anything but standard files!"
      Exit Sub
   End If

   ' Open the input file in binary mode.
   InFileNum = FreeFile
   Open FilePath For Binary Access Read As InFileNum

   ' Allocate space for the data in the input file.
   Dim FileContent As String
   FileContent = String(FileLen(FilePath), " ")

   ' Use Get instead of Input or Input$ to read the file.
   Get InFileNum, 0, FileContent

   ' Create an output filename.
   OutFilePath = Left(FilePath, InStr(1, FilePath, ".")) & "bak"

   ' Open the output file in binary mode.
   OutFileNum = FreeFile
   Open OutFilePath For Binary Access Write As OutFileNum

   ' Output the data to the backup file.
   Put OutFileNum, 0, FileContent

   ' Close the files.
   Close InFileNum, OutFileNum

   ' Set the attributes of the output file.
   SetAttr OutFilePath, Normal

   ' Reopen the output file.
   OutFileNum = FreeFile
   Open OutFilePath For Input As OutFileNum

   ' Read the entire file.
   FileContent = Input(LOF(OutFileNum), OutFileNum)

   ' Display the content on screen.
   MsgBox "The backup file contains: " & FileContent
End Sub
```

The example then writes the data found in `FileContent` to the output file using the `Put()` function. At this point, the code closes both the input and output files.

Because the output file is a backup, you really don't want to have the archive attribute set. Consequently, the next step is to use `SetAttr()` to set the output file's attributes to

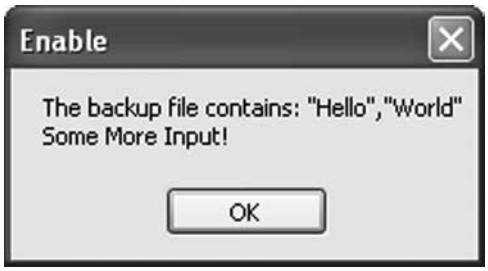


FIGURE 15.12 The Copy You Create Contains Precisely the Same Content As the Original

`Normal`. You might wonder why the code didn't perform this task earlier. Always close the file before you set the attribute, or IDEAScript might change the attribute as it closes the file to reflect the change in content.

It's important to verify that the file contains the content that you think it does. The example performs this task using a simple dialog box. The process begins by reopening the file for `Input`. The code then uses the `Input()` function to obtain the file content. Remember that you must tell `Input()` how many bytes to read, which is easily obtained using the `LOF()` function. Figure 15.12 shows that the backup file does indeed contain a full copy of the original file.

GetAttr

The `GetAttr()` function retrieves the attributes of a file so that you know more about it. Attributes are special settings that are stored with the file on disk. In some cases, the operating system automatically sets file attributes. For example, if you create a file in a folder that has file compression on, the operating system will automatically make the file compressed and set its compressed attribute. Here's a list of the attributes you can check with the `GetAttr()` function.

- **Read-only:** The file is set as read-only, which means you can't write to it, but you can read from it. The operating system will actually stop you from writing to the file. Normally, you don't want to work with read-only files.
- **Hidden:** The user can't see hidden files when using normal Windows Explorer settings. You can set Windows Explorer to show hidden files, but even then, they appear grayed out. In addition, hidden files are invisible at the command prompt. Using hidden files is one way to keep users from trying to modify application configuration files outside of the application.
- **System:** System files are those that are used by the operating system. It's always a bad idea to modify system files.
- **Directory:** The directory attribute tells you that an entry is a directory instead of a file. Directory entries are used as a kind of storage container for files and other directories.
- **Archive:** Any time you modify a file in some way, the operating system sets the archive bit to show that the file has been modified. When you write a backup pro-

gram, you can check for the archive bit and create a copy of those files that have the bit set. As you copy the files for backup purposes, reset the archive attribute to show that it has been backed up.

- **Normal:** The precise definition of Normal varies a bit by operating system. Under older versions of Windows, `Normal` simply meant that the file could be used for both read and write purposes. Under Windows XP and newer operating systems, `Normal` means that the file isn't compressed or encrypted, and that it's indexed. A Normal file doesn't have the archive bit set.
- **Temporary:** Sometimes the operating system or an application needs more storage space than memory can easily provide. In this case, the operating system or application creates a temporary file where it stores information. When the file is no longer required, it gets erased.
- **Compressed:** In days gone by, hard drive space was expensive and processing cycles were plentiful. Users would compress files so that they consumed less space on disk. A compressed file is written to the disk compressed and automatically decompressed during a read, so the process is invisible. Today, hard drive space is cheap, so many users don't compress their files. However, some users still compress files because doing so can improve disk performance. It requires less time to read a compressed file from the hard drive than it does a file that isn't compressed, especially when the file is easily compressed to a small size (such as graphics).
- **Offline:** An offline file is one that normally exists somewhere permanently, such as a network drive, but that you copy to your local drive so you can access it when you're disconnected from the network. Windows automatically performs a synchronization process when you log back into the network and immediately before you log out of the network to ensure your local copy of the file remains in sync with the network copy.
- **Indexed:** Using the indexing service can make locating files significantly faster. However, indexing every file on the system, even those that you don't normally search would slow the system down excessively, because the operating system has to search the file and create an index to it. The Indexed attribute shows that the file is part of the index. You only want to index files that you actually plan to search at some point (assuming that you even have the Indexing Service enabled).
- **Encrypted:** The operating system can automatically encrypt files so that other people can't read them. When you want to read the file, the operating system automatically decrypts it for you.

The operating system assigns a bit to each of these attributes and then places a number that represents those bits on the hard drive with the file. There are 32 bits available for each file, so a file can have up to 32 attributes associated with it. Here are the bits that are associated with each attribute:

- 0 – ReadOnly
- 1 – Hidden
- 2 – System
- 4 – Directory

- 5 – Archive
- 7 – Normal
- 8 – Temporary
- 11 – Compressed
- 12 – Offline
- 13 – Indexed
- 14 – Encrypted

`GetAttr()` adds these bits together to create a number that represents the attributes. For example, if a file has both `ReadOnly` and `Hidden` attributes set (bits `0` and `1`) that's 11 binary or 3 in decimal. This whole issue of bits can become quite confusing and you really don't need to worry about it unless you really want to. Using the Windows Calculator can help you understand things a lot better. Let's look at an example. Listing 15.10 shows a quick method for viewing attributes. You can use the `TestFile.txt` file created for the other examples in this chapter to follow along.

The code for this example begins by changing directories to the `C:\Test` directory that contains the files created for the other examples in this chapter. The code creates a `FilePath` variable that points to the `C:\Test\TestFile.txt` file. It then displays the number returned by `GetAttr()` for the `C:\Test\TestFile.txt` file.

Let's try setting a value or two to see how these bits work. First, right-click the `TestFile.txt` file in Windows Explorer and select Properties from the context menu. You'll see the Properties dialog box shown in Figure 15.13. Look at the bottom of this dialog box and you'll see Read-only and Hidden check boxes. These check boxes let you apply the read-only and hidden attributes to the file.

Click Advanced... and you see the Advanced Attributes dialog box shown in Figure 15.14. This is where you see the archive, indexed, compressed, and encrypted attributes. Checking the appropriate box will apply that attribute to the file after you click OK twice to close the two dialog boxes.

As a first experiment, try clearing all of the check boxes except the fast searching entry on the Advanced Attributes dialog box and checking both Read-only and Hidden on the Properties dialog box. Click OK to close the Properties dialog box. Now, run the example application and you see the output of 3 as shown in Figure 15.15.

LISTING 15.10 A Quick Method for Viewing Attributes

```
Sub Main
   ' Change directories to the test directory.
   ChDir "C:\Test"

   ' Define the file path.
   Dim FilePath As String
   FilePath = CurDir & "\TestFile.txt"

   ' Display the test file attributes.
   MsgBox GetAttr(FilePath)
End Sub
```

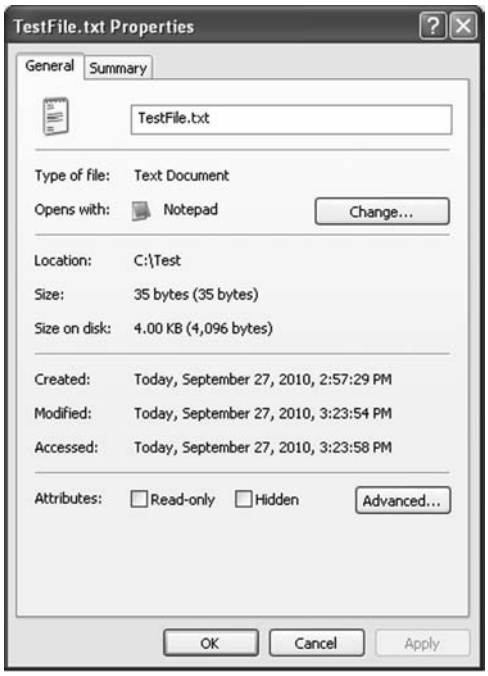


FIGURE 15.13 The Read-only and Hidden Attributes Appear on the Properties Dialog Box

Now, let's see which bits are selected. Open the Windows Calculator. Select View > Scientific to place the Calculator in scientific mode. Type 3. Now, select the Bin option and you see the results in Figure 15.16. Bits 0 and 1 are both set to 1, which means that both Read-only and Hidden are set. See how it works?

FIGURE 15.14 The Archive, Indexed, Compressed, and Encrypted Attributes Appear on the Advanced Attributes Dialog Box

FIGURE 15.15 The First Example Shows Read-only and Hidden

Let's try another example. Keep the Read-only and Hidden check boxes selected on the Properties dialog box. In addition, select the Compressed check box on the Advanced Attributes dialog box. Now, run the application again. You get a value of `2051`. If you enter this value in the Windows Calculator with Dec selected and then switch to Bin mode, you see the output shown in Figure 15.17. Notice that bits `11`, `1`, and `0` are now selected because they're set to `1`.

Try other examples until you understand a bit better how things work. Check and uncheck different boxes to see what the result is when you run the example application. The only odd attribute is the Indexing attribute. In order to turn this attribute on, you must clear the check box. Of course, trying to remember all of these values yourself would be terribly hard. That's why the example code includes a list of constants that

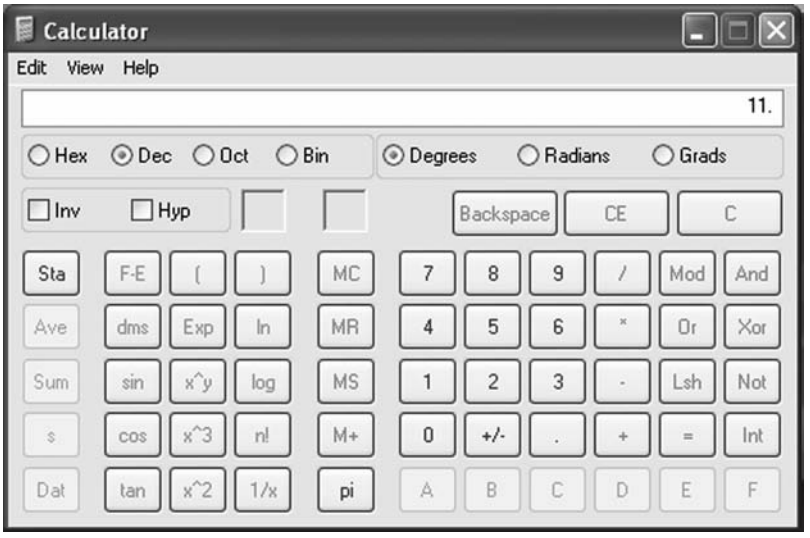


FIGURE 15.16 Windows Calculator Tells You About the Bits

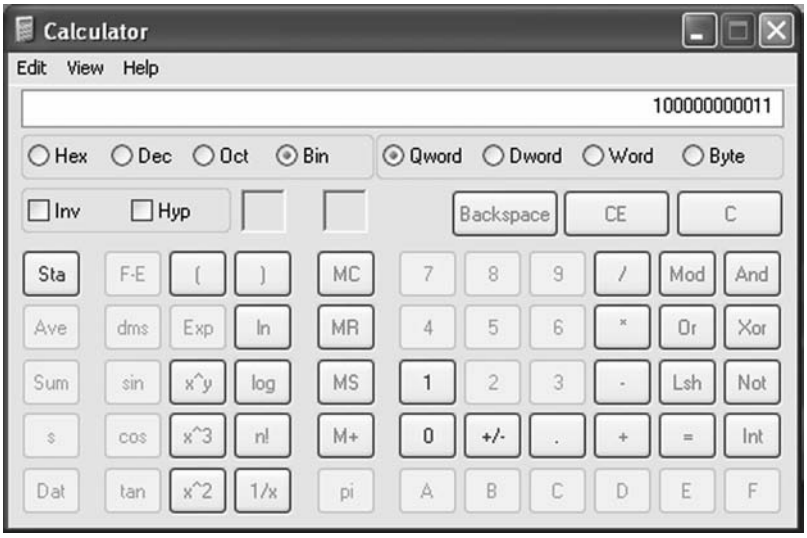


FIGURE 15.17 The Second Example Adds the Compressed Bit

are easier to remember as shown in Listing 15.11. These constants use the hexadecimal equivalents for each of the binary values.

Getting and setting attributes is actually a whole lot easier than understanding precisely how they work. You can see an example of the `GetAttr()` function in Listing 15.8 in the "FileLen" section. Another example of the `GetAttr()` function appears in Listing 15.9 in the "Get" section.

Input and Input$

The `Input()` and `Input$()` functions read a specified number of characters from a file starting from the current file pointer position. To use the `Input()` and `Input$()` functions, you supply the number of characters to read as the first argument and the

LISTING 15.11 Constants Used for Attributes

```
' Constants used for file types.
Const ReadOnly = &H00000001
Const Hidden = &H00000002
Const System = &H00000004
Const Directory = &H00000010
Const Archive = &H00000020
Const Normal = &H00000080
Const Temporary = &H00000100
Const Compressed = &H00000800
Const Offline = &H00001000
Const Indexed = &H00002000
Const Encrypted = &H00004000
```

file handle of the file you want to read as the second. The `Input()` function returns a variant containing the data found in the file. The `Input$()` function returns a string containing the data. You can see an example of the `Input()` and `Input$()` functions in Listing 15.6 in the "EOF" section and in Listing 15.9 in the "Get" section.

Kill

The `Kill()` function deletes a file from the hard drive. It's important to understand that the file doesn't go into the Recycle Bin or another location where you can easily recover it—the file is actually deleted. (You could use a file recovery utility to potentially get it back, but you shouldn't count on such measures.) To use the `Kill()` function, you simply provide the name and path of the file you want to delete. You can see an example of the `Kill()` function in Listing 15.4 in the "Close" section. A second example appears in Listing 15.7 in the "FileCopy" section.

Line Input

The `Line Input #()` function reads an entire line from a file open for sequential access. In this case, the number of characters you receive is determined by the length of the line. A line is terminated by either a line feed or carriage return character (or both). To use the `Line Input #()` function, you supply the file handle as the first argument and a string variable used to hold the line of text as the second argument. You can see an example of the `Line Input #()` function in Listing 15.7 in the "FileCopy" section.

LOF

The Length of File, `LOF()`, function returns the number of bytes found in an open file. To use this function, you provide the file handle of the file you want to measure. It's also possible to obtain the length of a closed file using the `FileLen()` function described in the "FileLen" section. You can see an example of the `LOF()` function in Listing 15.6 in the "EOF" section and in Listing 15.9 in the "Get" section.

MkDir

The `MkDir()` function creates new directories on the hard drive. Directories are a kind of storage container for files. You normally use directories to organize the files to make them easier to interact with and to find. When you get done using a directory, you can use the `RmDir()` function described in the "RmDir" section to remove it. The `MkDir()` function accepts the argument shown here:

```
MkDir [path]newDir
```

As a minimum, you must supply the name of the directory you want to create. If you supply just the directory name, `MkDir()` creates the new directory as a subdirectory of the current default directory. In other words, supplying just a directory name creates a new relative directory. When you supply the optional path, `MkDir()` creates the new

directory in the precise location that you specify. The path can be a relative or absolute directory. Here are some examples of the `MkDir()` function in use.

- `MkDir "MyDir"`
- `MkDir "MyDir\Test"`
- `MkDir "\MyDir"`
- `MkDir "D:\MyDir"`

Now that you have a better idea of how the `MkDir()` function works, it's time to see it in action. Listing 15.1 in the "ChDir" section shows how to use the `MkDir()` function with other functions to work with a number of locations on the hard drive.

Warning

You can't create a directory that already exists. If the directory already exists, IDEAScript will display a "Cannot create a file when that file already exists" error message.

Name

The `Name()` function lets you rename a file. To use this function, you simply provide the original file name as the first argument and the new file name as the second argument. When you rename a file, the new file name can't exist in the directory. Otherwise, IDEAScript displays an error message telling you about the renaming problem. You can see an example of the `Name()` function in Listing 15.7 in the "FileCopy" section.

Open

The `Open()` function lets you open a file for some use. You might want to read the file or write to it. In some cases, you open the file for binary access so that you can manipulate the data it contains, including non-ASCII characters used for formatting purposes. In all cases, you use the `Open()` function by specifying the name of the file, the mode you want to use to open it, and the file handle to use to access the file. Although the mode information is theoretically optional, you should specify it so that it's clear how you plan to work with the file. The syntax for the `Open()` function is as follows:

```
Open fileName$ [For mode] As [#] fileNumber
```

The `Open()` function supports four different modes:

- Append
- Binary (Access Read or Access Write)
- Input
- Output

The choice of mode is important. The `Append` mode automatically places the file pointer at the end of the file. `Append` mode makes it possible for you to add information to an existing file. Anytime you use `Print()` or `Write #()`, the data automatically appears at the end of the file. The `Append` mode provides sequential file access. If the file doesn't exist, IDEAScript automatically creates a new one for you when you use this mode.

When you select `Binary` mode, you can work with random areas of the file. Random access has advantages when you want to peek at the file data and poke at other areas to discover information about the file. `Binary` mode also provides low-level access to all of the information in a file. You can specify `Binary` mode by itself, in which case, you can both read and write the file. If you want to be a little more precise with the file, you can specify `Access Read` or `Access Write` as part of opening the file. If the file doesn't exist and you specify either `Binary` or `Binary Access Write`, IDEAScript will create it for you. On the other hand, if you specify `Binary Access Read`, the file must exist or IDEAScript will display an error message.

The `Input` mode provides sequential read (or input) access to the file starting from the beginning of the file. Every time your code performs a read of the file, IDEAScript moves the file pointer to the next position. When the code finishes reading the file, `EOF()` becomes true (see the "EOF" section for details). The file you want to read must exist or IDEAScript will display an error message.

The `Output` mode provides sequential write (or output) access to the file starting from the beginning of the file. If you write to a file that already has data in it, using the `Write #()` or `Print()` functions will delete the existing content. If you want to add to the existing content, you must use the `Append` mode. If the file doesn't exist, IDEAScript automatically creates a new one for you when you use this mode.

The `Open()` function is used a number of times in this chapter. You can see an `Output` mode example of the `Open()` function in Listing 15.4 in the "Close" section. Listing 15.6 in the "EOF" section and Listing 15.7 in the "FileCopy" section both show examples of `Input` mode. There's an example of how to use `Open` in the `Binary Access Read` and `Binary Access Write` modes in Listing 15.9 in the "Get" section.

Warning

You can't create a file that already exists. If the file already exists, IDEAScript will display a "Cannot create a file when that file already exists." error message.

Print

The `Print()` function doesn't actually print content, it displays content in a dialog box. All you do is provide a comma separated list of expressions you want to see displayed in the dialog box. In short, this is a simplified form of the `MsgBox()` function found in many examples in this book. You can see an example of the `Print()` function in Listing 15.7 in the "FileCopy" section.

Print

The `Print #()` function outputs data to a file that you have open for writing. To use the `Print #()` function, you supply a handle and then a list of expressions that you want to output. The expressions you output are printed literally as you provide them, without any formatting. You can see an example of the `Print #()` function in Listing 15.4 in the "Close" section.

Put

The `Put()` function lets you perform a binary write of a data file. You can choose any starting position within the file and write only the amount of data you want. The `Put()` function requires three inputs: the handle for the file you want to write, the starting position for the write, and a buffer to hold the data to write. You can see an example of the `Put()` function in Listing 15.9 in the "Get" section.

RmDir

The `RmDir()` function removes directories that you no longer need (sometimes created using the `MkDir()` function explained in the "MkDir" section). It's important to remember that directories serve as storage containers for other directories and for files. Just as you wouldn't throw out a box that still contains useable items, you can't remove a directory until it's empty. The `RmDir()` function can accept the argument shown here.

```
RmDir path
```

In order to remove a directory, you must supply the absolute or relative directory path to it. The path argument can contain just the name of the subdirectory you want to remove, the location from the root directory of the current drive, or a location on another drive. Here are some examples of the `RmDir()` function in use.

- `RmDir "MyDir\Test"`
- `RmDir "MyDir"`
- `RmDir "\MyDir"`
- `RmDir "D:\MyDir"`

Now that you have a better idea of how the `RmDir()` function works, it's time to see it in action. Listing 15.1 in the "RmDir" section shows how to use the `RmDir()` function with other functions to work with a number of locations on the hard drive.

Warning

You can't remove a directory that contains other directories or files. If you attempt to remove a directory that contains other directories or files, you'll receive a "Path/File access error." message. It's also an error to try to remove a directory that doesn't exist. In this case, you receive a "Path not found" error message.

Seek

Sometimes you don't want to start reading data from a file starting at the beginning. In this case, you use the `Seek()` function to place the file pointer at the position that you do want to start reading from. To use the `Seek()` function, you provide the file handle for the file as the first argument and the position you want to start reading from as the second argument. You can see an example of the `Seek()` function in Listing 15.6 in the "EOF" section.

SetAttr

As described in the "GetAttr" section, attributes are essentially descriptions of file or directory characteristics. An attribute helps everyone understand a particular issue about a file or directory, such as whether it's read-only. Be sure you go through the "GetAttr" section as part of learning to use `SetAttr()`. The `SetAttr()` function sets attributes on a file or directory, rather than getting them. In some cases, you set an attribute after performing a task, such as resetting (clearing) the archive attribute after you perform a backup. You can see an example of the `SetAttr()` function in Listing 15.8 in the "FileLen" section. Another example of the `SetAttr()` function appears in Listing 15.9 in the "Get" section.

Write

The `Write #()` function outputs data to a file you have opened for sequential access using the `Output` or `Append` modes. To use the `Write #()` function, you supply a file handle and a list of parameters you want to write. Each parameter appears in double quotes and the parameters are separated by commas. The `Write #()` function also adds a carriage return/line feed combination at the end of each write to form records between `Write #()` calls. You can see an example of the `Write #()` function in Listing 15.4 in the "Close" section.

Using External Variables

External variables provide a means of storing application settings to disk. Except for the task of adding and deleting external variables as you need them, IDEA does all of the required work for you. When you load a macro that has external variables, IDEA automatically loads the associated variables as well. Likewise, IDEA automatically saves any changes you make to the external variables. All of the external variables appear in an `.evars` file that has the same name as your macro. For example, if your macro is named `ExternalVars.iss`, then the external variables appear in `ExternalVars.evars`. Never modify the `.evars` file because doing so can cause your application to break.

Using external variables can make your application work better because it remembers settings between sessions. You'll find that external variables are also quite flexible.

In fact, you can use any of these data types:

- Character (String)
- Integer
- Long
- Boolean
- Double

Use the following steps to create a new external variable:

1. Select **Insert** > **External Variables**. You see the **External Variables** dialog box shown in Figure 15.18.
2. Click **Insert**. The **External Variables** dialog box changes to edit mode. You can type the name of the variable in the **Variable** field.
3. Type the name of the variable, such as **MyVar**, and press **Tab**. The next field is highlighted. Notice that IDEA automatically adds an **e_** to the variable name, such as **e_MyVar**.
4. Select a variable type from the drop-down list. The example uses **String**, but you can select any of the other types in the **Type** field.
5. Type a comment in the **Description** field. Adding a comment will make it easier for you to remember why you created the external variable later.
6. Type a value for the variable in the **Initial Value** field. Don't add double quotes for String values. The example uses an initial value of **Hello**. It's a good idea to give the variable an initial value so that it always has a known value when you start the application.
7. Repeat steps 2 through 6 to add any additional variables required by your application.
8. Click **OK**. At this point, you have an external variable to use.

Let's try a little code with the external variable you just created. Listing 15.12 shows the simple code used for this example.

When you run this example, you see two dialog boxes. The first dialog box always says `Hello` and the second always says `Goodbye`. No matter how many times you run the example, the sequence is the same unless you change the value of the external variable.

FIGURE 15.18 The External Variables Dialog Box Holds All of Your External Variables

LISTING 15.12 Using an External Variable

```
Sub Main
   ' Display the intial variable value.
   MsgBox e_MyVar

   ' Change the variable value.
   e_MyVar = "Goodbye"

   ' View the new value.
   MsgBox e_MyVar
End Sub
```

You can change any aspect of an existing external variable, including its name. Simply select Insert > External Variables and click on the item you want to change, such as the external variable's initial value. Type the new value and click OK to complete the action.

If you find that you've created a variable you don't need, highlight the entry in the External Variables dialog box and click Delete. You'll see that IDEA removes the variable from the list. When you click **OK**, the variable is gone.

Summary

This chapter has demonstrated techniques you can use to work with files on your system. Of course, you can read any file on any drive you can access—even remote files on mapped drives. It's possible to read any file to which you have access. However, the ability to understand the content of the file depends on the file type and whether the content is documented in any way. Writing a file requires different rights from reading it and you also need to know enough about the file content to avoid damaging the data within the file.

Now that you've seen the things you can do with files, try several experiments. The easiest way to work with files is to read a simple text file, so you should try that kind of example first. After you successfully read a text file, try creating a new file and writing data to it. When you understand this technique, try reading and writing files that contain a level of formatting. For example, try creating an application that uses a file to maintain its settings. It won't take long and you'll start seeing possibilities for all kinds of uses for files.

Chapter 16 looks at a new area of working with IDEAScript, accessing Excel directly. Excel can help you perform some types of analyses that you might have a hard time performing in IDEA, such as creating charts and graphs to see data visually (sometimes a picture really is worth a thousand words). Chapter 16 helps you understand how to access Excel and then use it to perform a number of interesting tasks.

CHAPTER 16

Working with Other Applications

You may have a need to work with other applications when using IDEA. In some cases, it's simply a matter of convenience. Yes, you can easily perform the task in IDEA, but perhaps the other application has some special feature to offer or your colleagues use the other application, making it easier to transport information to them. In other cases, the application may provide special functionality you won't find in IDEA, such as the ability to generate letters. This chapter looks at two such applications, Microsoft Word and Microsoft Excel.

It's possible to use Microsoft Word to create letters and other printed material. You could create some of this material in IDEA, but using Word offers an additional level of refinement—your output will look a little more professional. In addition, Word can help you create complex reports. In short, Word is a good choice when you have intricate printed output to produce.

Microsoft Excel provides a place for you to perform additional analysis, create graphs and charts, and exchange data with other people. Of course, you can perform all of these tasks manually, but if you perform them often enough, it makes sense to create applications to perform much of the work for you automatically. The purpose of this chapter is to demonstrate techniques you can use to interact with Excel from IDEA and IDEA from Excel. By linking the two applications together, you can do amazing things.

Considering IDEAScript and Visual Basic for Applications (VBA) Differences

You may not realize it, but you already know a considerable amount of information about working with Word and Excel through VBA. The IDEAScript language and the VBA language have an incredible number of similarities. For example, they both use the same structures (such as `If...Then`) and the same kind of looping (such as `For...Next`). Creating variables is about the same in both languages as well. In fact, you could write a simple program in VBA right this moment without reading another word in this chapter. Isn't that amazing?

IDEAScript and VBA do have some differences. VBA provides a number of constants that IDEAScript doesn't provide, such as `vbCrLf`. You may also find that functions with

the same name in both languages don't work precisely the same. Likewise, you'll find that some IDEAScript functionality isn't available in VBA. Fortunately, VBA provides you with help similar to that found in IDEAScript, so you can see differences as you type code in VBA. In addition, Word, Excel, and other Microsoft products that rely on VBA provide Help files just like those found in IDEA. In short, it won't take you very long at all to discover just how to use the language elements in VBA.

The big difference between IDEAScript and VBA is that VBA must accommodate the application you use it with, just as IDEAScript accommodates IDEA. Word includes a number of special objects such as Document, Selection, and Range. When you work with Excel, you must consider the use of Workbook, Worksheet, Chart, and Window objects, among other things. These objects are an important part of working with Word and Excel because they define what you can do with Word and Excel.

As an example, if you want to change the value in a Worksheet Cell object, then you need to access the Workbook first, then the Worksheet, and finally the Cell objects. You drill down into objects just as you do when working with IDEA. It isn't a surprising difference, but one that you must learn about in order to use Excel with IDEA fully. You can find a complete VBA programming reference at `http://msdn.microsoft.com/library/bb190882.aspx`.

The VBA code editor interface is also similar to the one used with IDEAScript, so you'll feel comfortable the first time you start it. However, as with everything else, there are differences between the two products. For the most part, you'll find that VBA is a little more feature rich and consequently, more complicated to use. You can choose to ignore features that you don't really need when using VBA with IDEA. You can find a very basic reference for the VBA editor at `http://msdn.microsoft.com/library/aa201750.aspx`. Even though this article is about Visio 2002, the VBA editor works the same for most Microsoft products, so the information you find here also works fine for Word and Excel. If you find that the Microsoft documentation is less than helpful (sometimes it's written in terms only a programmer could love), *VBA for Dummies* by John Paul Mueller provides a good place to start learning about the VBA language and how to use the VBA editor.

Understanding the Word and Excel Object Models

Later sections in this chapter will demonstrate that there's a lot you can do in Word or Excel by relying on the IDEAScript skills you've built. However, if you really want to work with either application, you need to know something about their object models—the part of the scripting language that works directly with application objects, such as documents and worksheets. There are entire books written on this topic, so a single section of a chapter can't possibly hope to cover the topic in any detail. Fortunately, Microsoft provides more than a little documentation online for both Word and Excel.

It's important to begin with an overview of the object models. The overview tells you some basics about how the object model works and how you can interact with it. You can find the Word overview at `http://msdn.microsoft.com/library/kw65a0we.aspx`. The overview doesn't provide you with many details about the various objects that Word supports. In fact, it may surprise you to learn that Word

supports hundreds of objects. Microsoft requires an entire six pages worth of documentation just to display them graphically. You can find a graphical presentation of all of the Word objects at `http://msdn.microsoft.com/library/aa165321.aspx`.

As with Word, you'll begin with an overview when working with the Excel object model. You can find an Excel object model overview at `http://msdn.microsoft.com/library/wss56bz7.aspx`. As with Word, Excel supports more than a few objects. Excel is a little less complex; it only requires three pages worth of documentation starting at `http://msdn.microsoft.com/library/aa141044.aspx`.

Running Word from IDEA

It's quite easy to run Word from IDEA. For that matter, even though this example is Word-specific, you can use the same technique for any application that provides the required interfaces. For example, you could easily use this technique with any Microsoft Office application. The only prerequisite is that you be able to work with the application as an object and that you have a reference to the object interfaces supplied by the application (see the "Understanding the Word and Excel Object Models" section for additional details).

This example interacts with Word in a way that lets you create custom content. The content could be anything. For example, you could use it to fill out a form that you've already designed in Word. To keep this example simple, it demonstrates the following tasks.

- Creates Word as an object.
- Creates a document within Word.
- Adds custom text to the document.
- Adds content from an IDEA database to the document.
- Saves the document to disk.
- Closes the document.
- Closes Word.

Now that you have an idea of what the example does, let's take a look at some code. Listing 16.1 shows the code needed to perform all these tasks. (You'll need to create a folder named `C:\Test` to use the example. Windows XP users can use the `C:\` folder if desired.)

LISTING 16.1 Accessing Word from IDEA

```
Sub Main
   ' Create the vbCrLf constant.
   vbCrLf = Chr(13) & Chr(10)

   ' Create the Word object.
   Dim WordDoc As Object
   Set WordDoc = CreateObject("Word.Application")
```

(*continued*)

LISTING 16.1 (*Continued*)

```
   ' Create a new document in Word.
   Dim ThisDoc As Object
   Set ThisDoc = WordDoc.Documents.Add()

   ' Set the text in the first paragraph.
   Dim ThisPara As Object
   Set ThisPara = ThisDoc.ActiveWindow.Document.Range()
   ThisPara.Text = "Here are the customers in Argentina:" & vbCrLf

   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path & "Sample-Customers.imd")

   ' Get the recordset.
   Dim RS As RecordSet
   Set RS = db.RecordSet

   ' Start at the first record.
   RS.ToFirst

   ' Loop through the recordset.
   For i = 1 To RS.Count

      ' Get the active record.
      Set Rec = RS.ActiveRecord

      ' Move to the next record.
      RS.Next

      ' Verify this customer is from Argentina.
      If Rec.GetCharValue("COUNTRY") = "ARGENTINA" Then

         ' If the customer is from Argentina, add it to Word.
         ThisPara.InsertAfter Rec.GetCharValue("COMPANY") & vbCrLf
      End If

   Next

   ' Save and close the document.
   ThisDoc.SaveAs "C:\Test\Test.doc"
   WordDoc.ActiveDocument.Close

   ' Exit Word
   WordDoc.Quit

   ' Clear memory.
   Set ThisPara = Nothing
   Set ThisDoc = Nothing
   Set WordDoc = Nothing
End Sub
```

LISTING 16.1 (*Continued*)

```
Function OpenDB(DBPath As String) As Database
    ' Define a database object.
    Dim db As Database

    ' Open the database using the default client folder.
    Set db = Client.OpenDatabase(DBPath)

    ' Return the database object.
    OpenDB = db

    ' Clear the memory used by db.
    Set db = Nothing
End Function
```

The example begins by creating a Word application object named `WordDoc`. To perform this task, you use the `CreateObject()` function and set the object variable, `WordDoc`, equal to the output. The `CreateObject()` function accepts the object name of the application as input. The string, `Word.Application`, tells `CreateObject()` what application to create. This string appears in the registry and is associated with Word. If you're interested, you can see other strings of this sort by starting the Registry Editor (`RegEdit`) and viewing the entries in `HKEY_CLASSES_ROOT\` as shown in Figure 16.1. Typically, the existence of a `CLSID` subkey tells you that

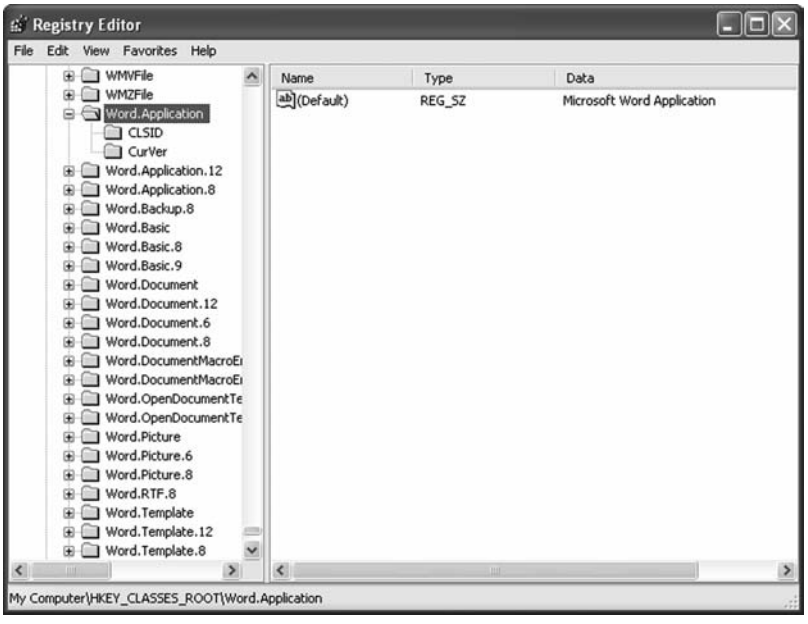


FIGURE 16.1 The Registry Editor Provides a List of Application Strings for Applications on Your Computer

the entry is usable as a `CreateObject()` string. You can find a list of the properties, methods, events, and child objects associated with `Word.Application` at `http://msdn.microsoft.com/library/aa221371.aspx`.

The `CreateObject()` call typically requires a few seconds to complete. That's because this call actually starts a copy of Word that exists out of sight in the background. If you open Task Manager, you won't see Word listed as one of the running applications. However, you'll see `WINWORD.EXE` listed on the Processes tab of the Task Manager because Word is executing in the background as a result of the `CreateObject()` call.

Now that the code has access to Word, it can add a document to the Word environment by calling `WordDoc.Documents.Add()`. This call tells Word to add a new document to the list of documents that it has in its documents list. You can interact with this document through the `ThisDoc` variable.

A document isn't much good without content. The first task is to gain access to a paragraph object, `ThisPara`, by making a call to `ThisDoc.ActiveWindow .Document.Range()`. This call tells Word to place the currently selected range, which is the first paragraph in the new document, in `ThisPara`. The `ThisPara` object contains a number of properties, including `Text`, that you can use to type custom text into Word. If you stopped the code right now and were able to peer into Word, you'd see the custom text, "Here are the customers in Argentina:" in the document.

Typing some custom text is great, but you really want to be able to send something from IDEA to Word. The code continues by opening the `Sample-Customers` database. It creates a `RecordSet` object, `RS`, and places the record pointer on the first record. At this point, the code begins processing the records. If the record, `Rec`, obtained from `RS.ActiveRecord`, contains the word ARGENTINA in the COUNTRY

FIGURE 16.2 Placing IDEA Data in Word Is Relatively Easy

field, the code sends the information from the COMPANY field to Word using the `ThisPara.InsertAfter()` method. Using this approach adds one record after another to the list so that Word eventually ends up with a list of companies that are in Argentina, as shown in Figure 16.2.

The document content is complete, but it isn't saved to disk yet. The `ThisDoc.SaveAs()` method saves the content to disk, after which, the code calls `WordDoc.ActiveDocument.Close()` to close the document. Always close the document before you exit Word. Otherwise, strange things can occur, such as data loss. Word might display one or more dialog boxes during the save process, depending on how you have Word configured. For example, you might see the document Properties dialog box. After the document is saved, the code calls `WordDoc.Quit()` to exit Word. As normal, the code ends by clearing memory.

Running IDEA from Excel

Just as you can run other applications from IDEA, you can run IDEA from other applications. As long as the host application supports some form of the `CreateObject()` function, you can start a copy of IDEA and perform tasks with it. Not every IDEA feature is exposed as part of the object you create with `CreateObject()`, but you'll find that you can perform most tasks quite easily.

The example in this section performs several tasks from Excel using IDEA. In order to use this example, create a new VBA macro in Excel by selecting Tools > Macro > Visual Basic Editor. After you open the Visual Basic Editor, right-click VBAProject in the Project window and select Insert > Module from the context menu.

When the example is finished, you'll see a copy of Excel loaded with data extracted from IDEA. The code shown in Listing 16.2 lets you extract data from any IDEA database and place it in Excel to perform whatever tasks you want.

The code in Listing 16.2 shows an overview of the process. It performs the following tasks:

- Creates an IDEA application object.
- Obtains the working directory from IDEA.
- Extracts data from the `Sample-Advanced Statistical Methods` database and places it in an extraction database.
- Exports the extraction database to `.xls` format.
- Closes the IDEA application.
- Clears memory.
- Opens the exported database in Excel.

Now that you have the overview of the process, let's begin looking at the code in detail. Listing 16.3 shows the code used to extract the database in IDEA through Excel.

The thing to notice about the code in Listing 16.3 is that it roughly corresponds to how you'd write the code in IDEA. However, there are differences you need to consider. First, because Excel knows nothing about IDEA objects, everything is declared as an `Object` type, rather than as a specific type such as `Database`. Excel automatically uses the correct type for you once it talks with IDEA about the request.

LISTING 16.2 Accessing IDEA from Excel

```
Sub AccessIDEA()
   ' Create an instance of the IDEA application.
   Dim Client As Object
   Set Client = CreateObject("IDEA.IDEAClient")

   ' Get the current directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Tell IDEA to extract data from the Sample-Advanced Statistical
   ' Methods database.
   Dim Output As String
   Output = DirectExtraction(Client, Path)

   ' Export the extracted information to .xls format.
   Dim ExportName As String
   ExportName = ExportDatabaseXLS(Client, Path, Output)

   ' Close IDEA because we're done using it.
   Client.Quit

   ' Clear memory.
   Set Client = Nothing

   ' Open the resulting workbook.
   Workbooks.Open ExportName
End Sub
```

LISTING 16.3 Performing the Database Extraction in Excel

```
Function DirectExtraction(Client As Object, Path As String) _
   As String

   ' Define the output database name.
   Dim dbName As String
   dbName = Path & "ExtractForExcel.imd"
   DirectExtraction = dbName

   ' Delete the old file.
   DeleteOld dbName

   ' Create a database object.
   Dim Db As Object
   Set Db = Client.OpenDatabase(Path & _
      "Sample-Advanced Statistical Methods.imd")

   ' Create a task.
   Dim Task As Object
   Set Task = Db.Extraction
```

```
   ' Configure the task.
   Task.IncludeAllFields
   Task.AddExtraction dbName, "", "BRANCH == ""B"""

   ' Perform the task.
   Task.PerformTask 1, Db.Count

   ' Clear memory.
   Set Task = Nothing
   Set Db = Nothing

   ' Open the database in IDEA.
   Client.OpenDatabase (dbName)
End Function

Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub
```

The second difference is that you precede all IDEA calls with `Client`. For example, you don't call `OpenDatabase()` directly, you call `Client.OpenDatabase()`. The use of `Client` tells Excel that you want to work with IDEA, rather than internal Excel objects.

After you have access to the `Database` object, `Db`, other calls look precisely like the ones that you use in IDEA. For example, to create a `Task`, you simply call `Db.Extraction`, just as you would in IDEA because the `Db` object knows that it belongs to IDEA and not to Excel. The code configures the task as normal and then calls `PerformTask()`, just as it does in IDEA. When the code finishes using the objects, it clears them using exactly the same syntax as normal.

Notice that the code calls `Client.OpenDatabase()` with the name of the extracted database. Even though you can't see IDEA, it remains opened in the background and now it has the extracted database open. The `DirectExtraction()` function returns the name of the extracted database to the caller.

At this point, you have an extracted database in IDEA that contains precisely the information you need to work with in Excel. The only problem is that the data isn't in a format that Excel can understand, even though Excel indirectly requested that IDEA create the database. Listing 16.4 shows the next step of the process, which exports the data from IDEA into the `.xls` format.

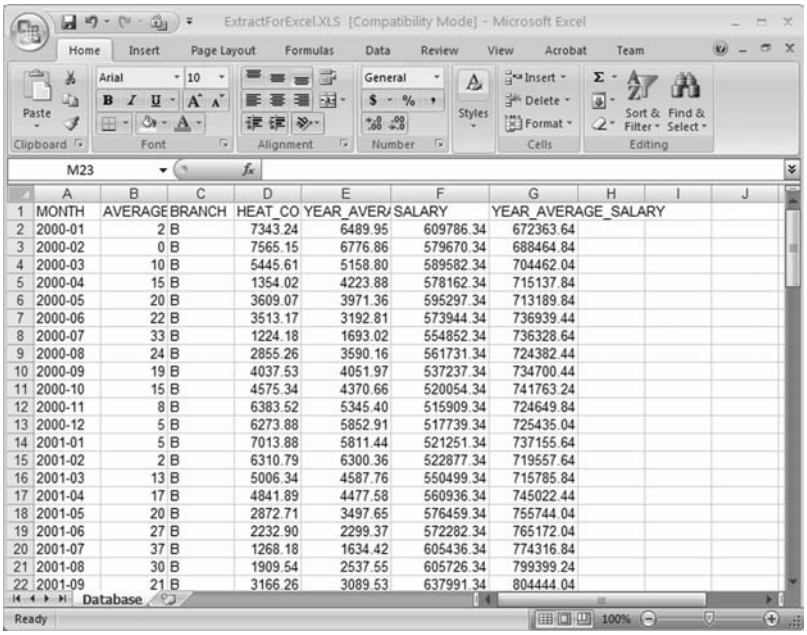
The code in Listing 16.4 begins by defining the output name of the `.xls` file. It then deletes the old file, if any, and opens the export database.

LISTING 16.4 Exporting the Data to .XLS Format in Excel

```
Function ExportDatabaseXLS(Client As Object, Path As String, _
   ExportDB As String) As String

   ' Define the exported worksheet name.
   Dim Output As String
   Output = Path & "ExtractForExcel.xls"
   ExportDatabaseXLS = Output

   ' Delete the old file.
   DeleteOld Output

   ' Create a database object.
   Dim Db As Object
   Set Db = Client.OpenDatabase(ExportDB)

   ' Create the task.
   Dim Task As Object
   Set Task = Db.ExportDatabase

   ' Configure the task.
   Task.IncludeAllFields
   Task.PerformTask Output, "Database", "xls", 1, Db.Count, ""

   ' Clear memory.
   Set Db = Nothing
   Set Task = Nothing
End Function

Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub
```

`Task` creation comes next. The configuration process opens the database and sets the task to export all of the fields. The call to `PerformTask()` outputs all the records in the extracted database to `ExtractForExcel.xls`. When the code finishes, it clears memory as usual. At this point, the data is in a form that Excel can use and the main subroutine, `AccessIDEA()`, shown in Listing 16.2, opens the `.xls` file so you can see it as shown in Figure 16.3.

There are cases where you don't need to make any changes to your IDEA code to use it within Excel (or another Office product). Listing 16.5 shows one such situation.



	A	B	C	D	E	F	G	H	I	J
1	MONTH	AVERAGE	BRANCH	HEAT_CO	YEAR_AVER	SALARY	YEAR_AVERAGE_SALARY			
2	2000-01	2	B	7343.24	6489.95	609786.34	672363.64			
3	2000-02	0	B	7565.15	6776.86	579670.34	688464.84			
4	2000-03	10	B	5445.61	5158.80	589582.34	704462.04			
5	2000-04	15	B	1354.02	4223.88	578162.34	715137.84			
6	2000-05	20	B	3609.07	3971.36	595297.34	713189.84			
7	2000-06	22	B	3513.17	3192.81	573944.34	736939.44			
8	2000-07	33	B	1224.18	1693.02	554852.34	736328.64			
9	2000-08	24	B	2855.26	3590.16	561731.34	724382.44			
10	2000-09	19	B	4037.53	4051.97	537237.34	734700.44			
11	2000-10	15	B	4575.34	4370.66	520054.34	741763.24			
12	2000-11	8	B	6383.52	5345.40	515909.34	724649.84			
13	2000-12	5	B	6273.88	5852.91	517739.34	725435.04			
14	2001-01	5	B	7013.88	5811.44	521251.34	737155.64			
15	2001-02	2	B	6310.79	6300.36	522877.34	719557.64			
16	2001-03	13	B	5006.34	4587.76	550499.34	715785.84			
17	2001-04	17	B	4841.89	4477.58	560936.34	745022.44			
18	2001-05	20	B	2872.71	3497.65	576459.34	755744.04			
19	2001-06	27	B	2232.90	2299.37	572282.34	765172.04			
20	2001-07	37	B	1268.18	1634.42	605436.34	774316.84			
21	2001-08	30	B	1909.54	2537.55	605726.34	799399.24			
22	2001-09	21	B	3166.26	3089.53	637991.34	804444.04			

FIGURE 16.3 Seeing the IDEA Database in Excel

LISTING 16.5 Some IDEAScript Code Works Without Modification

```
Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir(Filepath)

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
      MsgBox "Deleting old copy of " + PathCheck
      Kill Filepath
   End If
End Sub
```

Notice that this `DeleteOld()` subroutine looks precisely the same as the one used for most of the IDEA examples in the book. In fact, it's the same.

The introduction to this chapter states that you probably know enough right now to begin working with products other than IDEA and you can see now that this assumption is correct. There are ways to work with Word and Excel with only a little additional information.

Summary

This chapter has demonstrated some useful techniques for working with Excel and Word. Of course, Excel and Word provide a significant number of objects that you can interact with, so this is just the tip of the iceberg. It's possible to combine IDEA and Excel or Word in an amazing number of ways to obtain any outcome you require while analyzing data and present that output in a number of useful ways. The actual number of ways you can use the information in this chapter are only limited by your imagination.

By now, you have some good ideas on what you'd like to do with IDEA and other applications teamed up together. It never pays to go off without a plan though, so take some time to write down your ideas and then think about the best ways to implement them. You'll probably discover multiple ways to perform the same task, which is where experimentation comes into play. Work through your solutions to see which of them actually work as anticipated, then choose the best solutions for refinement. Take some time to work with subsets of your data so that you don't get weighed down waiting for results.

Chapter 17 moves on to some advanced database tasks you can perform with IDEA. You may find it hard to believe, but IDEA provides even more ways to interact with your data. For example, it's possible to create pivot tables to view your data in a new way and to look for gaps in the data that might signify problems that require more research. IDEA helps you summarize data and even join databases together so that you get a larger view of the information.

CHAPTER 17

Performing Data Analysis Tasks

Previous chapters in the book have shown you how to perform a wealth of common database tasks—everything from extracting information to creating special fields to hold your notes and data. However, IDEA can do a whole lot more. You might not need these special features every day, but it's good to know that they exist for those situations when you do. This chapter shows how to work with a number of advanced database features that include stratification, summarization, random record sampling, and gap detection. In addition, you consider how to work with pivot tables to view data in new ways, join databases to get the complete data picture, and work with monetary unit samples. Even if you won't use these features immediately, follow along with the examples so you become familiar with them. You never know when the need for such features will become apparent.

Performing Stratification

Stratification relies on the creation of value bands and placing the records from a database within these bands. Each value band represents a range of data and provides a means of categorizing the data in certain ways. You can stratify databases based on character, numeric, or date fields. IDEA supports up to 1,000 stratification bands. After you stratify the database, you can use the results for a number of tasks such as:

- Totaling the number of records in each band, then using the results to look for expected trends.
- Setting high and low cutoff values to test for exceptional items.
- Creating both a database and a result that contains the details of the band range each record falls within. You can then use this database for further analysis.

There are other ways to use the stratification results—see the IDEA Help file for additional ideas. Once you have the database stratified, you can obtain random samples from each band to use for further analysis.

Note

Make sure you check the IDEA Help file for restrictions and requirements on using Stratification. For example, all IDEA analysis is case sensitive. If your database has case irregularities, you need to extract a version that changes the case so that all entries are either uppercase or lowercase. In addition, you need to consider the effects of equations and @Functions. The "Extraction" section in Chapter 9 discusses extraction requirements. The "Using Extraction" section in Chapter 14 shows an extraction example.

Using Stratification

Stratification helps you create a view of your data. Creating a view lets you build graphs, drill down into the data, and perform other analysis tasks. However, there's no macro support for specific views. A database option allows you to do more after the stratification. A stratified database contains a special STRATUM field that shows which value band holds a particular record. Listing 17.1 shows the code used to create a stratification database.

LISTING 17.1 Stratifying the Database

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Delete the old file.
   DeleteOld Path & "Stratification-Sample-Customers.imd"

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path & "Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Set the task type.
   Set task = db.Stratification

   ' Configure the task.
   task.IncludeAllFields
   task.OutputDBName = "Stratification-Sample-Customers.imd"
   task.IncludeInterval = False
   task.ResultName = "Stratification"
   task.FieldToStratify = "CREDIT_LIM"
   task.AddFieldToTotal "CREDIT_LIM"
```

```
   ' Include support for random samples.
   task.SupportStratifiedRandomSample = True

   ' Create the required value bands.
   task.LowerLimit 1000.00
   task.AddUpperLimit 5000.00
   task.AddUpperLimit 10000.00
   task.AddUpperLimit 15000.00
   task.AddUpperLimit 20000.00
   task.AddUpperLimit 25000.00

   task.AddUpperLimit 30000.00
   task.AddUpperLimit 35000.00
   task.AddUpperLimit 40000.00
   task.AddUpperLimit 45000.00
   task.AddUpperLimit 50000.00
   task.AddUpperLimit 55000.00
   task.AddUpperLimit 60000.00
   task.AddUpperLimit 65000.00
   task.AddUpperLimit 70000.00
   task.AddUpperLimit 80000.00
   task.AddUpperLimit 90000.00
   task.AddUpperLimit 100000.00
   task.AddUpperLimit 200000.00
   task.AddUpperLimit 300000.00

   ' Perform the task.
   task.PerformTask

   ' Clear memory.
   Set task = Nothing
   Set db = Nothing

   ' Open the database.
   OpenDB(Path & "Stratification-Sample-Customers.imd")
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
   ' exist and we need to exit the routine.
   If Len(PathCheck) < 1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If

   ' Define a database object.
   Dim db As Database
```

(*continued*)

LISTING 17.1 (*Continued*)

```
   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function

Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub
```

The example begins by obtaining the current client working directory using `Client.WorkingDirectory`. It then deletes any existing database using the `DeleteOld()` function described in previous chapters. The example opens the `Sample-Customers.imd` database, which provides the source data for the stratified database, using the `OpenDB()` function described in previous chapters. After checking to make sure that the database is actually open, the code begins performing the `Stratification` task.

The task object is set to `db.Stratification`. In order to perform the task, the code must perform some configuration tasks. For this example, the code includes all of the fields found in the `Sample-Customers` database. You can choose not to include all of the fields by specifying individual fields using the `AddFieldToInc()` or `AddFieldToIncAt()` functions. The output database name is `Stratification-Sample-Customers.imd`. The `IncludeInterval` property determines whether the output database includes the upper and lower limit fields. `FieldToStratify` contains the name of the field to use to create the stratified output. The `AddFieldToTotal` field contains the name of the field you want totaled for each stratum.

Part of the configuration process is to determine whether you want to support random samples. If so, the code must set the `SupportStratifiedRandomSample` property to `True`. Otherwise, when you try to use the `StratifyRndSample` task to obtain a random sample, the code will complain about a lack of the SAM_RECNO field.

Once the code configures the task, it can begin defining value bands. The `LowerLimit()` function defines the lowest value that appears in the result. Each call to the `AddUpperLimit()` function adds a new band. Consequently, the first band is from $1,000.00 to $5,000.00. At this point, the code calls `PerformTask()` to create the result and output database. It then clears memory and opens the resulting database. Figure 17.1

Sample-Customers.IMD | Stratification-Sample-Customer...

	CUSTNO	COMPANY	FIRST_NAME	LAST_NAME	COUNTRY	STATUS	CREDIT_LIM	STRATUM	SAM_RECNO
1	21105	Fine Jewellery Inc.	CLAUDIA	BELAMA	GREENLAND	A	1000	1	124
2	21206	Wholesale Watches and Rings	ANDRE	HASHIYANA	NAMIBIA	A	1000	1	131
3	20065	Imperial Designers	OY	WING	HONG KONG	A	1500	1	48
4	10003	Diseños de la Vendimia	JOSE	ERNESTO	ARGENTINA	A	2000	1	2
5	11207	Barbados Jewellery Company	DENISE	KHAN	BARBADOS	A	2000	1	18
6	11301	Jewellery Now	MARINELA	HRISTOV	BULGARIA	A	2000	1	20
7	12203	Kara Jewels	OLGA	ARBELAEZ	COLOMBIA	A	2000	1	29
8	20008	Emitations	RAN	LAWSON	ENGLAND	A	2000	1	34
9	20039	Hong Kong Fine Jewellery	MARITA	PETERSEN	FAROE ISLANDS	A	2000	1	38
10	20414	Adrianna's Fine Jewels	ARIE	VERHOEF	UNITED KINGDOM	A	2000	1	73
11	20756	Yuli's	YUK	YEUNG	CHINA	A	2000	1	88
12	20764	Chinese Designers of Fine Jewellery	WING	WONG	CHINA	I	2000	1	89
13	20823	Gold Jewellery & Watches Inc.	MILANIO	CUADRA	ARGENTINA	A	2000	1	94
14	21092	Turquoise and Jewels	LEIF	ERIKSSON	GREENLAND	A	2000	1	122
15	21139	Lëtzebuergesch	CLAUDIA	SCHIFFERES	LUXEMBOURG	A	2000	1	125
16	21274	Michaela's Fine Pendants	GEORGA	GIMBEL	NORWAY	A	2000	1	139
17	21426	Lu-Pang's	SUSAN	AGBUYA	PHILIPPINES	A	2000	1	152
18	21646	Relojes Costa Rica	ROSA	SANTAMARIA	COSTA RICA	A	2000	1	161
19	30228	Joyería y cosas	MARCO	MARTINEZ	COSTA RICA	A	2000	1	164
20	30704	Homeland Jewellery	YUSUF	BRUUN	FINLAND	A	2000	1	167
21	40310	Jewelery On Time	RUPERT	CARLISLE	U.S.A.	A	2000	1	192
22	40317	Fancy Cuts	DONALD	SHANNON	U.S.A.	A	2000	1	195
23	40401	The Look	CAREY	WINSLOW	U.S.A.	A	2000	1	196
24	40605	Watches For All	CARLOS	FUENTES	U.S.A.	A	2000	1	201
25	40617	Impressions	DONNY	CHAMBERLAIN	CANADA	A	2000	1	204
26	40708	Fadi's	JUNE	NESRALLAH	CANADA	A	2000	1	206

FIGURE 17.1 Stratifying the Database Places Values within Preselected Value Bands

shows the result of this example. Notice the STRATUM field, which shows the band in which each of the records appear. In addition, because this example does support random samples, you see the SAM_RECNO field.

Using StratifyRndSample

Once you have a stratified database to use, you can obtain random samples from it for analysis purposes. Listing 17.2 shows how to use the `StratifyRndSample()` function to perform this task. (The example assumes that you created the `Stratification-Sample-Customers` database using the code in Listing 17.1.)

LISTING 17.2 Obtaining a Random Sample from a Stratified Database

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Delete the old file.
   DeleteOld Path & "StratRndSample-Sample-Customers.imd"

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path & "Stratification-Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If
```

(*continued*)

LISTING 17.2 (*Continued*)

```
   ' Set the task type.
   Set task = db.StratifyRndSample

   ' Configure the task.
   task.IncludeAllFields

   ' Define the number of samples for each band.
   task.StratifyOnBand 1,  10
   task.StratifyOnBand 2,  8
   task.StratifyOnBand 3,  3
   task.StratifyOnBand 4,  2
   task.StratifyOnBand 5,  1
   task.StratifyOnBand 6,  1
   task.StratifyOnBand 7,  1
   task.StratifyOnBand 8,  1

   ' Perform the task.
   task.PerformTask "StratRndSample-Sample-Customers.imd", "", _
     "CREDIT_LIM", 12848

   ' Clear memory.
   Set task = Nothing
   Set db = Nothing

   ' Open the database.
   OpenDB(Path & "StratRndSample-Sample-Customers.imd")
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
   ' exist and we need to exit the routine.
   If Len(PathCheck) < 1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

```
Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub
```

The example begins by performing the usual task of opening the source database and ensuring the destination database doesn't exist. The source database you choose must be stratified. If you try to perform the `StratifyRndSample` task on a non stratified database, the code will register an error.

The next step is to create the `StratifyRndSample` task and configure it. In this case, the only configuration requirement is to tell the task to include all of the data fields. As with most other tasks, you can also choose to include individual fields.

Configuring the random samples comes next. You use the `StratifyOnBand()` function to define the number of samples from each band. For example, the code obtains ten samples from band `1`. If you try to obtain more samples than the database can provide, the output simply contains all of the records for that band.

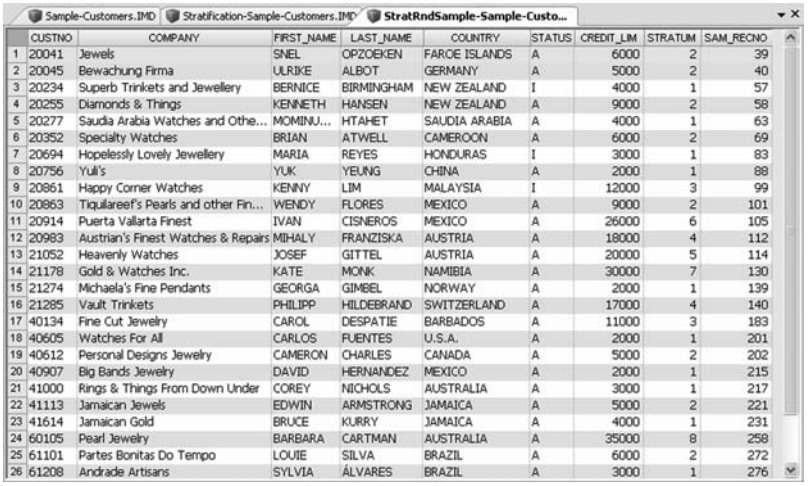
At this point, the code performs the task. You must supply the name of the output database, the stratification field, and the random seed used to obtain the samples. The description argument (the second argument in the list) isn't used. The example uses a hard-coded random seed. You can use any function that outputs a number for the random seed. Many developers use some type of time function, such as `Second(Now)` as the random seed. You can also use the `Rnd()` function to obtain a random number. However, because `Rnd()` outputs a `Double`, you must convert it to an Integer using code such as `CInt(Rnd * 10000)`. Figure 17.2 shows the output from this example.

Warning

It's important that you provide a unique random seed value each time you perform the `StratifyRndSample` task. Using a hard-coded seed will produce the same output every time, which means that the output isn't truly random and any analysis you perform will be faulty. The more complex the algorithm you use to produce the random seed, the better your results.

Performing Summarization

The `Summarization` task does much as the name suggests: it provides a summarization of data within a database. Precisely what the summarization means depends on



Sample-Customers.IMD | Stratification-Sample-Customers.IMD | StratRndSample-Sample-Custo...

	CUSTNO	COMPANY	FIRST_NAME	LAST_NAME	COUNTRY	STATUS	CREDIT_LIM	STRATUM	SAM_RECNO
1	20041	Jewels	SNEL	OPZOEKEN	FAROE ISLANDS	A	6000	2	39
2	20045	Bewachung Firma	ULRIKE	ALBOT	GERMANY	A	5000	2	40
3	20234	Superb Trinkets and Jewellery	BERNICE	BIRMINGHAM	NEW ZEALAND	I	4000	1	57
4	20255	Diamonds & Things	KENNETH	HANSEN	NEW ZEALAND	A	9000	2	58
5	20277	Saudia Arabia Watches and Othe...	MOMINU...	HTAHET	SAUDIA ARABIA	A	4000	1	63
6	20352	Specialty Watches	BRIAN	ATWELL	CAMEROON	A	6000	2	69
7	20694	Hopelessly Lovely Jewellery	MARIA	REYES	HONDURAS	I	3000	1	83
8	20756	Yuli's	YUK	YEUNG	CHINA	A	2000	1	88
9	20861	Happy Corner Watches	KENNY	LIM	MALAYSIA	I	12000	3	99
10	20863	Tiquilareef's Pearls and other Fin...	WENDY	FLORES	MEXICO	A	9000	2	101
11	20914	Puerta Vallarta Finest	IVAN	CISNEROS	MEXICO	A	26000	6	105
12	20983	Austrian's Finest Watches & Repairs	MIHALY	FRANZISKA	AUSTRIA	A	18000	4	112
13	21052	Heavenly Watches	JOSEF	GITTEL	AUSTRIA	A	20000	5	114
14	21178	Gold & Watches Inc.	KATE	MONK	NAMIBIA	A	30000	7	130
15	21274	Michaela's Fine Pendants	GEORGA	GIMBEL	NORWAY	A	2000	1	139
16	21285	Vault Trinkets	PHILIPP	HILDEBRAND	SWITZERLAND	A	17000	4	140
17	40134	Fine Cut Jewelry	CAROL	DESPATIE	BARBADOS	A	11000	3	183
18	40605	Watches For All	CARLOS	FUENTES	U.S.A.	A	2000	1	201
19	40612	Personal Designs Jewelry	CAMERON	CHARLES	CANADA	A	5000	2	202
20	40907	Big Bands Jewelry	DAVID	HERNANDEZ	MEXICO	A	2000	1	215
21	41000	Rings & Things From Down Under	COREY	NICHOLS	AUSTRALIA	A	3000	1	217
22	41113	Jamaican Jewels	EDWIN	ARMSTRONG	JAMAICA	A	5000	2	221
23	41614	Jamaican Gold	BRUCE	KURRY	JAMAICA	A	4000	1	231
24	60105	Pearl Jewelry	BARBARA	CARTMAN	AUSTRALIA	A	35000	8	258
25	61101	Partes Bonitas Do Tempo	LOUIE	SILVA	BRAZIL	A	6000	2	272
26	61208	Andrade Artisans	SYLVIA	ÁLVARES	BRAZIL	A	3000	1	276

FIGURE 17.2 Getting a Random Sample from the Stratified Database

the statistic you select. The example outputs a summary of minimum, maximum, and average credit limits by country from the `Sample-Customers` database. However, you can perform a wide variety of analysis types using this task. Listing 17.3 shows the code used for this example.

LISTING 17.3 Summarizing a Database

```
Sub Main
    ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Delete the old file.
   DeleteOld Path & "Summarization.imd"

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path & "Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Set the task type.
   Set task = db.Summarization

   ' Configure the task.
   task.AddFieldToSummarize "COUNTRY"
   task.AddFieldToTotal "CREDIT_LIM"
```

```
   task.OutputDBName = "Summarization.imd"
   task.CreatePercentField = False
   task.StatisticsToInclude = SM_COUNT + SM_MAX + SM_MIN + SM_AVERAGE

   ' Perform the task.
   task.PerformTask

   ' Clear memory.
   Set task = Nothing
   Set db = Nothing

   ' Open the database.
   OpenDB(Path & "Summarization.imd")
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
   ' exist and we need to exit the routine.
   If Len(PathCheck) < 1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function

Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub
```

The code begins by obtaining the current working directory, deleting the old output database (if any), and opening the `Sample-Customers` database. If the code doesn't detect any errors, it begins by creating the `Summarization` task.

The configuration process begins by defining the field to summarize using the `AddFieldToSummarize()` function. You can add as many fields as required to perform the analysis up to a maximum of eight. This example only requires one. The first field takes precedence, followed by the second, and so on. The `AddFieldToTotal()` function defines one or more fields to summarize. You can do a lot more than simply total the field (and may not even want to total it, as is the case for this example). The `OutputDBName` property tells where to output the summary. You can also create a percentage field when working with percentages. The `StatisticsToInclude` property contains one or more analyses to perform, as described in the following list:

- **SM_COUNT:** Counts the number of records for each summarized key.
- **SM_SUM:** Sums the values of the selected numeric field for particular summarization criteria.
- **SM_MAX:** Obtains the maximum value for each summarized key.
- **SM_MIN:** Obtains the minimum value for each summarized key.
- **SM_VARIANCE:** Computes the variance for each summarized key.
- **SM_AVERAGE:** Computes the average value of the records for each summarized key.
- **SM_STD_DEV:** Computes the standard deviation for each summarized key.

Sample-Customers.IMD | Summarization.IMD

	COUNTRY	NO_OF_RECS	CREDIT_LIM_MAX	CREDIT_LIM_MIN	CREDIT_LIM_AVERAGE
1	ARGENTINA	18	95000	2000	14167
2	AUSTRALIA	20	134000	2000	20550
3	AUSTRIA	5	50000	10000	22800
4	BARBADOS	5	15000	2000	7200
5	BELGIUM	5	23000	3000	11400
6	BERMUDA	8	42000	2000	14125
7	BRAZIL	6	20000	3000	7833
8	BULGARIA	2	6000	2000	4000
9	CAMEROON	2	28000	6000	17000
10	CANADA	12	68000	2000	20250
11	CHILE	2	14000	4000	9000
12	CHINA	5	95000	2000	22200
13	COLOMBIA	2	8000	2000	5000
14	COSTA RICA	5	5000	2000	3000
15	CZECH-REPUPLIC	2	8000	6000	7000
16	DENMARK	5	12000	3000	6600
17	ENGLAND	8	70000	2000	19625
18	FAROE ISLANDS	5	10000	2000	5200
19	FINLAND	5	3901000	2000	819600
20	FRANCE	11	86000	3000	30182
21	GERMANY	5	35000	3000	12000
22	GREENLAND	6	18000	1000	8500
23	GUYANA	4	7000	3000	5250
24	HONDURAS	7	110000	3000	19143
25	HONG KONG	5	19000	1500	9700
26	IRELAND	10	3500000	3000	377300

FIGURE 17.3 Performing a Database Summarization

To add a particular statistic, you simply add it to the property as shown in the code. Now that the task is configured, the code calls the `Perform()` task. The final steps are to clear memory and then display the database. Figure 17.3 shows the results of this example.

Creating a Pivot Table

Pivot tables are interesting because you can use them to create a new summarized view of the existing data. For example, you could use a pivot table to summarize the number of elements by two topics. The example shown in Listing 17.4 provides such an example. It tells you how many customers have a certain credit limit within a particular country when using the `Sample-Customers` database.

LISTING 17.4 Creating a Pivot Table from a Database

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path & "Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Remove the old result.
   db.DeleteResultByName("Pivot Table")

   ' Set the task type.
   Set task = db.PivotTable

   ' Configure the task.
   task.ResultName = "Pivot Table"
   task.AddRowField "CREDIT_LIM"
   task.AddColumnField "COUNTRY"
   task.AddDataField "CUSTNO", "Customers by Credit Limit and Country", 1

   ' Perform the task.
   task.PerformTask

   ' Clear memory.
   Set task = Nothing
   Set db = Nothing
End Sub
```

(*continued*)

LISTING 17.4 *(Continued)*

```
Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
   ' exist and we need to exit the routine.
   If Len(PathCheck) < 1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

The code begins by opening the `Sample-Customers` database. Pivot tables aren't stored in a separate database—they appear in the Results area of the Properties window, so there's no need to delete an old database in this example. However, the code does use the `DeleteResultByName()` function to remove the old result.

The next step is to create the task and configure it. In order to create a pivot table, you must supply a row field, a column field, and a data field. The row and column fields create a matrix that defines the criteria used to compute some result that appears in the data area. In this case, the code creates a matrix that uses the CREDIT_LIM and COUNTRY fields. The result will show every credit limit for every country. The `AddDataField()` function requires three arguments: the field used to create the data, a label to use for the data, and the numeric value for the kind of calculation you want to perform as described in the following list.

1 - Sum
2 - Average
3 - Count
4 - Min
5 - Max

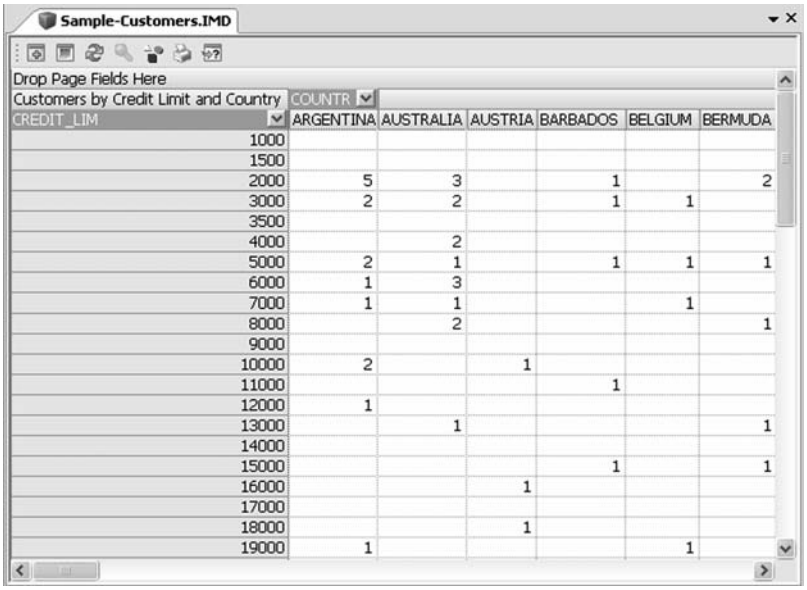


FIGURE 17.4 Pivot Tables Help You See Data in a New Way

At this point, the code performs the task. It then clears memory as normal. Figure 17.4 shows the pivot table created by this example. Notice that the rows and columns appear precisely as you expect. The data field is the area defined by the rows and columns. For example, Argentina has five customers that have a $2,000.00 credit limit.

If you double-click one of the entries, such as the cell presented for Argentina at a $2,000.00 credit limit, you see the individual database entries for that data, as shown in Figure 17.5. As you can see, it's easy to use a pivot table to get an overview of a particular trend and then drill down into the data you need.

A pivot table naturally works for two-dimensional data. However, what happens when you need to look at the data in three dimensions? In this case, you can add pages to the pivot table shown in Figure 17.4—the pages are the third dimension. You use the `AddPageField()` function to add pages to your pivot table.

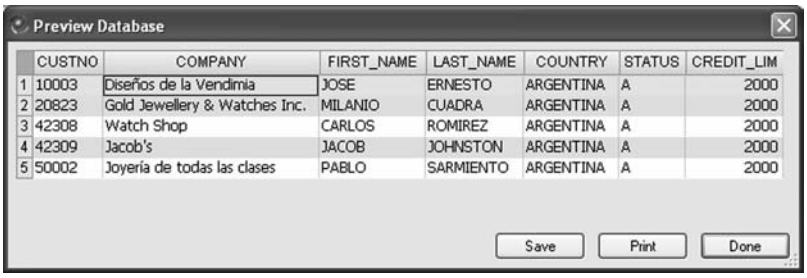


FIGURE 17.5 It's Also Possible to See the Details When Working with a Pivot Table

Employing Random Record Sampling Using RandomSample

For some types of analysis it's important to create a random sampling of the database. If you manually select your sample by flagging or creating a list of transactions to select, you can't be sure you have a valid random sample of the data. IDEA makes it easy to obtain a random valid statistical sample of any size. Listing 17.5 shows you how to perform this task.

LISTING 17.5 Obtaining a Random Sample Database

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Delete the old file.
   DeleteOld Path & "RandomSample.imd"

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path & "Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Set the task type.
   Set task = db.RandomSample

   ' Configure the task.
   task.IncludeAllFields

   ' Perform the task.
   task.PerformTask "RandomSample.imd", "", 31, 1, db.Count, 5421, False

   ' Clear memory.
   Set task = Nothing
   Set db = Nothing

   ' Open the database.
   Client.OpenDatabase ("RandomSample.imd")
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
```

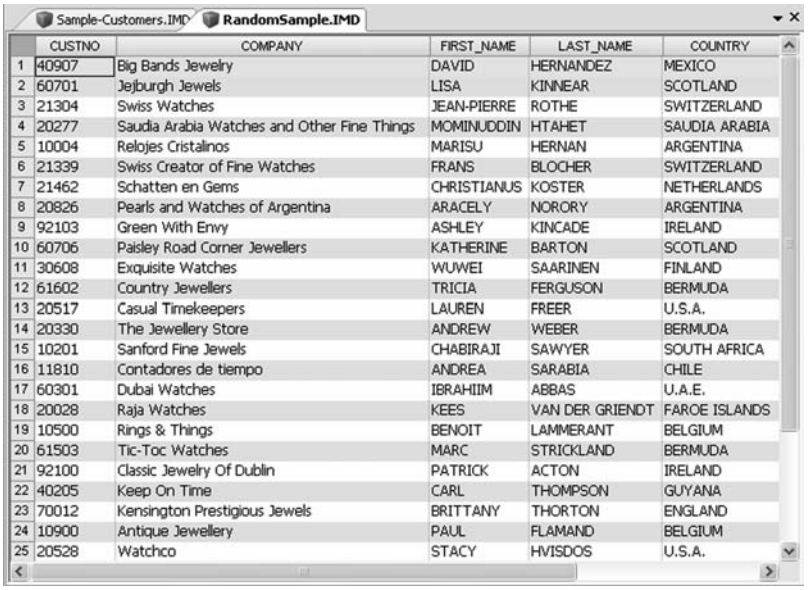
```
   ' exist and we need to exit the routine.
   If Len(PathCheck) < 1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function

Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub
```

The code begins by deleting the old `RandomSample` database and opening the `Sample-Customers` database. If there aren't any errors, the code proceeds to create the `RandomSample` task. In this case, the example outputs all of the fields in the database, but you can always choose to include just the fields you need in the output.

At this point, the code calls `PerformTask()`. The `PerformTask()` function requires the name of the database, the number of records required in the sample, the starting record, the ending record (normally obtained using the `Count` property of the database object), the random seed, and the duplicate record setting as input. The second argument, description, is always set to `""`.

This example uses a hard-coded seed value. As with any random sampling function, if you provide the same seed, you'll get the same result. Consequently, you need to generate a random integer value using a time value or something like `CInt(Rnd * 10000)`.

Selection methodology is important. If you set the duplicate record argument to `False`, as shown in the example, it means that a record can appear once at most, and the output records are guaranteed to be unique. When you set the duplicate record argument to `True`, every record has an equal chance of being selected every time IDEAScript makes a selection. Consequently, some records could appear more than once in the output.



Sample-Customers.IMD | RandomSample.IMD

	CUSTNO	COMPANY	FIRST_NAME	LAST_NAME	COUNTRY
1	40907	Big Bands Jewelry	DAVID	HERNANDEZ	MEXICO
2	60701	Jejburgh Jewels	LISA	KINNEAR	SCOTLAND
3	21304	Swiss Watches	JEAN-PIERRE	ROTHE	SWITZERLAND
4	20277	Saudia Arabia Watches and Other Fine Things	MOMINUDDIN	HTAHET	SAUDIA ARABIA
5	10004	Relojes Cristalinos	MARISU	HERNAN	ARGENTINA
6	21339	Swiss Creator of Fine Watches	FRANS	BLOCHER	SWITZERLAND
7	21462	Schatten en Gems	CHRISTIANUS	KOSTER	NETHERLANDS
8	20826	Pearls and Watches of Argentina	ARACELY	NORORY	ARGENTINA
9	92103	Green With Envy	ASHLEY	KINCADE	IRELAND
10	60706	Paisley Road Corner Jewellers	KATHERINE	BARTON	SCOTLAND
11	30608	Exquisite Watches	WUWEI	SAARINEN	FINLAND
12	61602	Country Jewellers	TRICIA	FERGUSON	BERMUDA
13	20517	Casual Timekeepers	LAUREN	FREER	U.S.A.
14	20330	The Jewellery Store	ANDREW	WEBER	BERMUDA
15	10201	Sanford Fine Jewels	CHABIRAJI	SAWYER	SOUTH AFRICA
16	11810	Contadores de tiempo	ANDREA	SARABIA	CHILE
17	60301	Dubai Watches	IBRAHIIM	ABBAS	U.A.E.
18	20028	Raja Watches	KEES	VAN DER GRIENDT	FAROE ISLANDS
19	10500	Rings & Things	BENOIT	LAMMERANT	BELGIUM
20	61503	Tic-Toc Watches	MARC	STRICKLAND	BERMUDA
21	92100	Classic Jewelry Of Dublin	PATRICK	ACTON	IRELAND
22	40205	Keep On Time	CARL	THOMPSON	GUYANA
23	70012	Kensington Prestigious Jewels	BRITTANY	THORTON	ENGLAND
24	10900	Antique Jewellery	PAUL	FLAMAND	BELGIUM
25	20528	Watchco	STACY	HVISDOS	U.S.A.

FIGURE 17.6 Getting a Random Sample from a Database

The code ends by clearing memory. It then opens the `RandomSample` database so you can see the results. Figure 17.6 shows the results from this example.

Performing Gap Detection

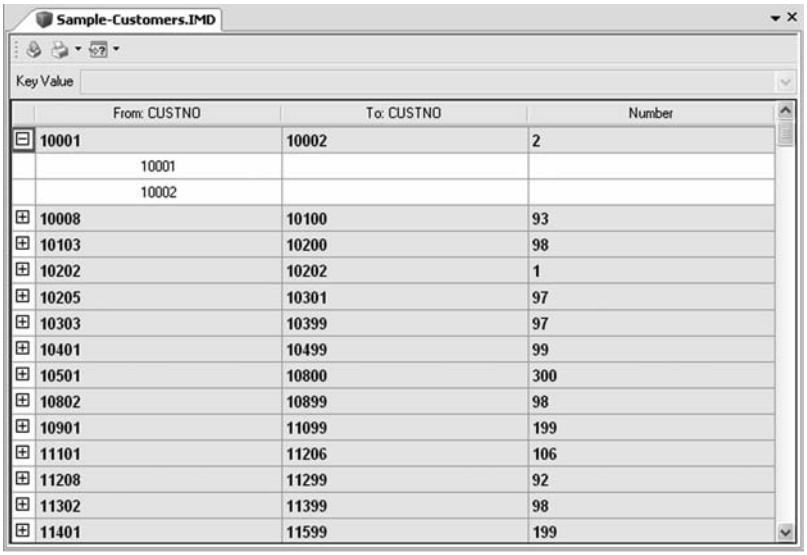
In many cases, you need to know whether the data in a database is complete—that nothing is missing. Gap Detection can help you quickly locate missing items within sequences of data. Listing 17.6 shows a typical example of using Gap Detection to find missing sequences of data.

LISTING 17.6 Detecting Gaps in Database Data

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   'Open the database.
   Dim db As Database
   Set db = OpenDB(Path & "Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If
```

```
   ' Remove the old result.
   db.DeleteResultByName("Gap Detection")

   ' Set the task type.
   Set task = db.Gaps

   ' Configure the task.
   task.FieldToUse =  "CUSTNO"
   task.Mask = "NNNNN"
   task.ResultName = "Gap Detection"

   ' Perform the task.
   task.PerformTask

   ' Clear memory.
   Set task = Nothing
   Set db = Nothing
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
   ' exist and we need to exit the routine.
   If Len(PathCheck) < 1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

The code begins by opening the `Sample-Customers.imd` database. If there aren't any errors, the code removes the existing result, if any, using `DeleteResultByName()`. You could also create an output database when performing Gap Detection, but the example uses a result because the missing elements are actually a bit easier to see, and you get the results faster. However, either approach works just fine.



From: CUSTNO	To: CUSTNO	Number
10001	10002	2
10001		
10002		
10008	10100	93
10103	10200	98
10202	10202	1
10205	10301	97
10303	10399	97
10401	10499	99
10501	10800	300
10802	10899	98
10901	11099	199
11101	11206	106
11208	11299	92
11302	11399	98
11401	11599	199

FIGURE 17.7 Seeing Gaps in the Database Content

The next step is to create the `Gaps` task and configure it. In order to perform Gap Detection, you must supply the name of a field to examine in the database and a mask for performing the analysis. A mask simply says how the data in the field is organized. The CUSTNO field is a series of five numbers, so the mask is `"NNNNN."` A mask consists of a series of Cs (normally for characters, but you can also use it for numbers), Xs (for ignore), and Ns (for numbers). For example, if a field had one character and four numbers in it, you'd use a mask of `"CNNNN."` The mask simply tells IDEAScript what kind of data to expect in the field.

At this point, the code calls `PerformTask()`. IDEAScript automatically displays the result on screen as shown in Figure 17.7. Notice that clicking the plus sign (+) next to any entry shows the complete list of missing elements. The final step, as usual, is to clear memory.

Checking Distribution Using SystematicSample

Systematic sampling creates a non random sample of the database by selecting records at specific intervals. For example, if your database has 300 records and you want 30 records in your sample, then the interval between records would be 10. The sample would select every tenth record regardless of what it contains. Listing 17.7 shows how to create a systematic sample.

LISTING 17.7 Looking for Distribution Issues

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Delete the old file.
   DeleteOld Path & "SystematicSample.imd"

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path & "Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Set the task type.
   Set task = db.SystematicSample

   ' Configure the task.
   task.AddFieldToInc "CUSTNO"
   task.AddFieldToInc "COMPANY"
   task.AddFieldToInc "FIRST_NAME"
   task.AddFieldToInc "LAST_NAME"
   task.AddFieldToInc "COUNTRY"
   task.AddFieldToInc "CREDIT_LIM"

   ' Perform the task.
   task.PerformTask "SystematicSample.imd", "", 31, 1, db.Count, 10

   ' Clear memory.
   Set task = Nothing
   Set db = Nothing

   ' Open the database.
   Client.OpenDatabase ("SystematicSample.imd")
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
   ' exist and we need to exit the routine.
   If Len(PathCheck) < 1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If
```

(*continued*)

LISTING 17.7 (*Continued*)

```
    ' Define a database object.
    Dim db As Database

    ' Open the database using the default client folder.
    Set db = Client.OpenDatabase(DBPath)

    '  Return the database object.
    OpenDB = db

    ' Clear the memory used by db.
    Set db = Nothing
End Function

Sub DeleteOld(Filepath As String)
    ' Determine if the file exists.
    Dim PathCheck As String
    PathCheck = Dir( Filepath )

    ' Delete the file if it exists.
    If Len(PathCheck) > 1 Then
    MsgBox "Deleting old copy of " + PathCheck
    Kill Filepath
    End If
End Sub
```

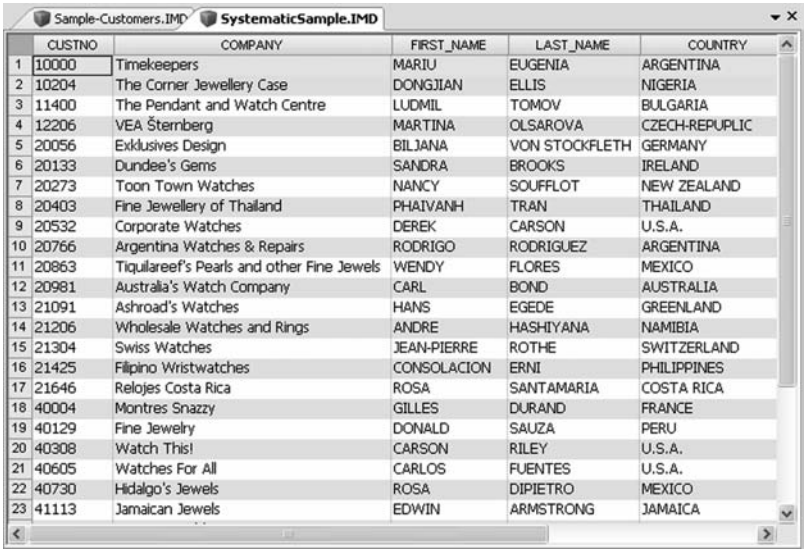
The example begins by deleting the old output database, `Systematic Sample.imd`. It then opens the `Sample-Customers` database. If there aren't any errors, the code begins creating the `SystematicSample` task. In this case, the code uses `AddFieldToInc()` to add individual fields to the output.

Once the task is configured, the code calls `PerformTask()` with the name of the output database, the number of records to output, the starting record, the ending record, and the interval between records. The second argument, description, isn't used, so you need to set it to `""`.

The code ends by clearing memory. It then opens the database. Figure 17.8 shows the output from this example. If you compare the output to the original database, you'll see the output contains every tenth record and that the records won't change each time you run the example code. In short, this is a non random sample.

Merging Databases

Companies often use huge databases that rely on a number of tables to hold the information required to run the business. The act of placing like data in individual tables is called normalization and you normally see these tables in a relational database. Unfortunately, normalized tables in a relational database are hard to analyze. In order for humans to understand the data, it's important to denormalize it by joining the required tables



Sample-Customers.IMD | SystematicSample.IMD

	CUSTNO	COMPANY	FIRST_NAME	LAST_NAME	COUNTRY
1	10000	Timekeepers	MARIU	EUGENIA	ARGENTINA
2	10204	The Corner Jewellery Case	DONGJIAN	ELLIS	NIGERIA
3	11400	The Pendant and Watch Centre	LUDMIL	TOMOV	BULGARIA
4	12206	VEA Šternberg	MARTINA	OLSAROVA	CZECH-REPUPLIC
5	20056	Exklusives Design	BILJANA	VON STOCKFLETH	GERMANY
6	20133	Dundee's Gems	SANDRA	BROOKS	IRELAND
7	20273	Toon Town Watches	NANCY	SOUFFLOT	NEW ZEALAND
8	20403	Fine Jewellery of Thailand	PHAIVANH	TRAN	THAILAND
9	20532	Corporate Watches	DEREK	CARSON	U.S.A.
10	20766	Argentina Watches & Repairs	RODRIGO	RODRIGUEZ	ARGENTINA
11	20863	Tiquilareef's Pearls and other Fine Jewels	WENDY	FLORES	MEXICO
12	20981	Australia's Watch Company	CARL	BOND	AUSTRALIA
13	21091	Ashroad's Watches	HANS	EGEDE	GREENLAND
14	21206	Wholesale Watches and Rings	ANDRE	HASHIYANA	NAMIBIA
15	21304	Swiss Watches	JEAN-PIERRE	ROTHE	SWITZERLAND
16	21425	Filipino Wristwatches	CONSOLACION	ERNI	PHILIPPINES
17	21646	Relojes Costa Rica	ROSA	SANTAMARIA	COSTA RICA
18	40004	Montres Snazzy	GILLES	DURAND	FRANCE
19	40129	Fine Jewelry	DONALD	SAUZA	PERU
20	40308	Watch This!	CARSON	RILEY	U.S.A.
21	40605	Watches For All	CARLOS	FUENTES	U.S.A.
22	40730	Hidalgo's Jewels	ROSA	DIPIETRO	MEXICO
23	41113	Jamaican Jewels	EDWIN	ARMSTRONG	JAMAICA

FIGURE 17.8 Verifying That the Database Data Is Distributed Properly

together. Because IDEA represents the individual tables found in a relational database as separate files, you actually need to merge databases to accomplish the task in IDEA. IDEA provides two techniques you can use to merge databases as described in the following sections.

Using JoinDatabase

Let's say that you want to know which customers made purchases and when they made them. In order to obtain this information, you must join the `Sample-Customers` database with the `Sample-Detailed Sales` database. Because the focus is on customers, you make the `Sample-Customers` database the primary database and the `Sample-Detailed Sales` database the secondary database. If the focus were on the sales, then you'd reverse the primary and secondary database assignments.

The method used to merge the databases is also important. You want to know which customers made purchases, which means that you must include all of the customers from the `Sample-Customers` database. IDEA supports a host of Join types, so you need to choose the right kind of join for your particular needs. The example lists all of the `Sample-Customers` database entries.

The join will entail some level of duplication. If nothing else, both the primary and secondary databases will include the same relationship field. In this case, the duplicated field is CUSTNO. You don't need two copies of CUSTNO in the output, so the secondary database output won't include the CUSTNO field. However, you must provide a matching field between the two databases. The CUSTNO field is the match between the `Sample-Customers` and `Sample-Detailed Sales` databases.

You may decide that you don't need other fields because they don't provide information necessary to your analysis. For the example, that means removing the STATUS and CREDIT_LIM fields from the primary database. The example also removes the SALESREP_NO field from the secondary database because the example is interested in what was sold, not who sold it.

You'd probably do more to make this example truly useful. For example, the `Sample-Detailed Sales` database includes just a PROD_CODE field. A second join would make it possible to include the product description in the output. Additional processing would make the example more complicated though, so the example shows just this first join. It's time to look at some code. Listing 17.8 shows the code used to join the `Sample-Customers` database with the `Sample-Detailed Sales` database.

LISTING 17.8 Joining Two Databases

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Delete the old file.
   DeleteOld Path & "Sample-Join.imd"

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path & "Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Set the task type.
   Set task = db.JoinDatabase

   ' Configure the task.
   task.FileToJoin "Sample-Detailed Sales.imd"
   task.AddMatchKey "CUSTNO", "CUSTNO", "A"

   ' Define the primary database output fields.
   task.AddPFieldToInc "CUSTNO"
   task.AddPFieldToInc "COMPANY"
   task.AddPFieldToInc "FIRST_NAME"
   task.AddPFieldToInc "LAST_NAME"
   task.AddPFieldToInc "COUNTRY"

   ' Define the secondary database output fields.
   task.AddSFieldToInc "INV_NO"
   task.AddSFieldToInc "INV_DATE"
   task.AddSFieldToInc "PROD_CODE"
```

```
   task.AddSFieldToInc "UNIT_PRICE"
   task.AddSFieldToInc "QTY"
   task.AddSFieldToInc "SALES_BEF_TAX"
   task.AddSFieldToInc "SALES_TAX"
   task.AddSFieldToInc "SALES_PLUS_TAX"

   ' Perform the task.
   task.PerformTask "Sample-Join.imd", "", WI_JOIN_ALL_IN_PRIM

   ' Clear memory.
   Set task = Nothing
   Set db = Nothing

   ' Open the database.
   Client.OpenDatabase ("Sample-Join.imd")
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
   ' exist and we need to exit the routine.
   If Len(PathCheck) < 1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function

Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub
```

The example begins by deleting the old output database if one exists. It then opens the primary database, but not the secondary database. When the tasks succeed, the code goes on to create the task.

The `JoinDatabase` task requires that you provide at least two inputs. The first, `FileToJoin`, defines the name of the secondary database. In this case, the code assumes the secondary database resides in the same working directory as the primary database. If the secondary database resides in another location, you must provide the full path to it. The second, `AddMatchKey()`, defines the names of the matching fields. The first argument contains the name of the field in the primary database. The second argument contains the name of the field in the secondary database, which doesn't have to match the name of the field in the primary database, but must match in values. The third argument is the sort order to use when matching the fields.

At this point, the code defines the fields to use from the primary and secondary databases. The code shows you how to include a selection of individual fields from both databases, which is the most common approach. However, you can use `IncludeAllP-Fields()` or `IncludeAllSFields()` to include all the fields from the primary or secondary databases respectively.

Tip

Further analyses will work faster if you include only the fields you actually require in the joined database. The less data that IDEA must search through to perform analysis tasks, the faster the analysis will complete. In addition, using fewer fields will also use system resources more efficiently. Using fewer resources will free resources for other tasks and speed overall system performance.

Now that the task is configured, it's time to call `PerformTask()`. The first argument contains the name of the output database—the one that holds the joined databases. The third argument contains the kind of join to perform. Table 17.1 provides a list of join types. You must select one of these values. The description (second) argument isn't required to perform the task, but must still be entered.

TABLE 17.1 Database Join Values

ID Value	Constant	Description
0	WI_JOIN_MATCH_ONLY	Matches only
1	WI_JOIN_ALL_IN_PRIM	All records in the primary database
2	WI_JOIN_ALL_REC	All records in both databases
3	WI_JOIN_NOC_SEC_MATCH	Records with no secondary match
4	WI_JOIN_NOC_PRI_MATCH	Records with no primary match

Sample-Customers.IMD | Sample-Join.IMD

	CUSTNO	COMPANY	FIRST_NAME	LAST_NAME	COUNTRY
1	10000	Timekeepers	MARIU	EUGENIA	ARGENTINA
2	10003	Diseños de la Vendimia	JOSE	ERNESTO	ARGENTINA
3	10004	Relojes Cristalinos	MARISU	HERNAN	ARGENTINA
4	10005	Clockwatcher	JUANMA	JUAN	ARGENTINA
5	10006	Contadores de tiempo de la estrella	MARIA	TERESA	ARGENTINA
6	10007	Perles de Tahiti	DIANE	BURROWS	SOUTH AFRICA
7	10101	Lord of the Rings and other Fine Jewellery	KEVIN	NICHOLSON-KNOWLES	SOUTH AFRICA
8	10102	Johnson Bancock Fine Collectibles	JENNIFER	DE FREITAS	SOUTH AFRICA
9	10201	Sanford Fine Jewels	CHABIRAJI	SAWYER	SOUTH AFRICA
10	10203	Ananzi Watches	KATHARINE	BURROWS	SOUTH AFRICA
11	10204	The Corner Jewellery Case	DONGJIAN	ELLIS	NIGERIA
12	10302	Trinkets & Things	MALINDA	JOHNSTON	NIGERIA
13	10400	Beljium Jewellery	FLORIN	GOOSSENS	BELGIUM
14	10500	Rings & Things	BENOIT	LAMMERANT	BELGIUM
15	10801	Fine Jewellers	ANNICK	VANDERVUST	BELGIUM
16	10900	Antique Jewellery	PAUL	FLAMAND	BELGIUM
17	11100	Clocks and other Time Tools	CRISTIAN	SUN	BELGIUM
18	11207	Barbados Jewellery Company	DENISE	KHAN	BARBADOS
19	11300	Personal Watch Designers	SAMUEL	GONSALVES	BARBADOS
20	11301	Jewellery Now	MARINELA	HRISTOV	BULGARIA
21	11400	The Pendant and Watch Centre	LUDMIL	TOMOV	BULGARIA
22	11600	The Crystal Watch Company	YVES	GODBOUT	CANADA
23	11702	Time Keepers	ANDREW	COLES	CANADA

FIGURE 17.9 Seeing the Bigger Picture Offered by Joined Databases

Once the task is performed, the code clears memory. It then opens the database containing the joined databases. Figure 17.9 shows the output from this example.

Using VisualConnector

A standard join is great for making a primary to secondary database connection. However, it isn't always the best way to create a merger that involves multiple databases. When you look at the `Sample-Detailed Sales` database, you notice that it has connections to three other databases: `Sample-Sales Representatives`, `Sample-Customers`, and `Sample-Inventory`. Creating all three connections using a simple join could prove a bit daunting, even though it's possible. Using `VisualConnector` makes things a lot easier by creating the multiple connections in one step. In addition, if you want to design a `VisualConnector` setup, you can use the graphic presentation to do it.

This example makes some assumptions about the merger. In this case, the emphasis is on the sales, not the other elements of the database. Consequently, the output doesn't include information from the `Sample-Sales Representatives` database such as COMMISSION and SALARY. The `Sample-Customers` database excludes CREDIT_LIM and STATUS from the output for the same reason. Likewise, you won't find STOCK_LEVEL from the `Sample-Inventory` database. The output also excludes duplicate fields. Figure 17.10 shows the Visual Connector: Create/Edit Relation window used for this example.

Now that you have some idea of the complexity of the database relationships in this example, let's look at some code. The example uses the code shown in Listing 17.9 to create the output database that represents the joined databases.

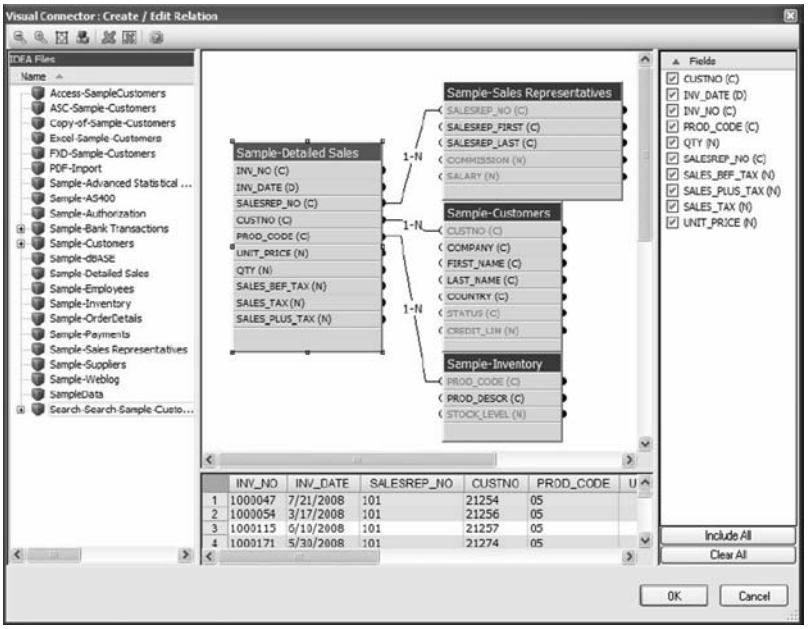


FIGURE 17.10 Seeing the Results of Using VisualConnector

LISTING 17.9 Working with VisualConnector

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Delete the old file.
   DeleteOld Path & "Sample-VisualConnector.imd"

   'Open the database.
   Dim db As Database
   Set db = OpenDB(Path & "Sample-Detailed Sales.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Set the task type.
   Set task = db.VisualConnector

   ' Add the required databases to the VisualConnector.
   id0 = task.AddDatabase ("Sample-Detailed Sales.imd")
   id1 = task.AddDatabase ("Sample-Sales Representatives.imd")
   id2 = task.AddDatabase ("Sample-Customers.imd")
   id3 = task.AddDatabase ("Sample-Inventory.imd")
```

```
   ' Set the master database.
   task.MasterDatabase = id0

   ' Create relations between the databases.
   task.AddRelation id0, "SALESREP_NO", id1, "SALESREP_NO"
   task.AddRelation id0, "CUSTNO", id2, "CUSTNO"
   task.AddRelation id0, "PROD_CODE", id3, "PROD_CODE"

   ' Configure the task.
   task.AppendDatabaseNames = False
   task.IncludeAllPrimaryRecords = True
   task.OutputDatabaseName = "Sample-VisualConnector.imd"

   ' Add all of the Sample-Detailed Sales database fields.
   task.AddFieldToInclude id0, "INV_NO"
   task.AddFieldToInclude id0, "INV_DATE"
   task.AddFieldToInclude id0, "SALESREP_NO"
   task.AddFieldToInclude id0, "CUSTNO"
   task.AddFieldToInclude id0, "PROD_CODE"
   task.AddFieldToInclude id0, "UNIT_PRICE"
   task.AddFieldToInclude id0, "QTY"
   task.AddFieldToInclude id0, "SALES_BEF_TAX"
   task.AddFieldToInclude id0, "SALES_TAX"
   task.AddFieldToInclude id0, "SALES_PLUS_TAX"

   ' Add the selected Sample-Sales Representatives fields.
   task.AddFieldToInclude id1, "SALESREP_FIRST"
   task.AddFieldToInclude id1, "SALESREP_LAST"

   ' Add the selected Sample-Customers fields.
   task.AddFieldToInclude id2, "COMPANY"
   task.AddFieldToInclude id2, "FIRST_NAME"
   task.AddFieldToInclude id2, "LAST_NAME"
   task.AddFieldToInclude id2, "COUNTRY"

   ' Add the selected Sample-Inventory field.
   task.AddFieldToInclude id3, "PROD_DESCR"

   ' Perform the task.
   task.PerformTask

   ' Clear memory.
   Set task = Nothing
   Set db = Nothing

   ' Open the database.
   OpenDB Path & "Sample-VisualConnector.imd"
End Sub
```

(*continued*)

LISTING 17.9 (*Continued*)

```
Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
   ' exist and we need to exit the routine.
   If Len(PathCheck) < 1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function

Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub
```

The example begins by deleting the old output database, `Sample-Visual Connector.imd`. It then opens `Sample-Detailed Sales.imd`, which is the primary database for this example. If the database opens without error, then the code creates the `VisualConnector` task.

This example relies on four databases in total as shown in Figure 17.10. The next step is to add all four databases using `AddDatabase()` and assigning each database an identifier. If you don't store the identifier, you'll find that it's impossible to manipulate the individual databases, so this step is especially important.

The `VisualConnector` task requires a master or primary database, which you set using the `MasterDatabase` property. The code completes the task configuration in the

Sample-Detailed Sales.IMD | **Sample-VisualConnector.IMD**

	INV_NO	INV_DATE	SALESREP_NO	CUSTNO	PROD_CODE	UNIT_PRICE	QTY	SALES_BEF_TAX
1	1000047	7/21/2008	101	21254	05	5.99	72	431.28
2	1000054	3/17/2008	101	21256	05	5.99	63	377.37
3	1000115	6/10/2008	101	21257	05	5.99	1209	7,241.91
4	1000171	5/30/2008	101	21274	05	5.99	250	1,497.50
5	1000199	3/18/2008	101	21285	05	5.99	435	2,605.65
6	1000219	4/25/2008	101	21304	05	5.99	360	2,198.33
7	1000254	3/4/2008	101	21330	05	5.99	700	4,193.00
8	1000256	5/29/2008	101	21339	05	5.99	250	1,497.50
9	1000448	6/19/2008	101	21340	05	5.99	8	47.92
10	1000617	12/22/2008	101	21341	05	5.99	168	1,006.32
11	1000666	9/1/2008	101	21342	05	5.99	250	1,497.50
12	1000732	9/26/2008	101	21395	05	5.99	63	377.37
13	1000766	12/15/2008	101	21400	05	5.99	58	347.42
14	1000772	6/30/2008	101	21402	05	5.99	101	604.99
15	1000852	12/22/2008	101	21403	05	5.99	57	341.43
16	1000001	6/24/2008	102	21425	02	25.95	13	337.35
17	1000002	7/14/2008	102	21426	03	35.15	70	2,460.50
18	1000032	6/19/2008	102	21450	03	35.15	22	773.30
19	1000048	2/26/2008	102	21462	05	5.99	972	5,822.28
20	1000070	2/5/2008	102	21464	05	5.99	1052	6,301.48
21	1000089	1/31/2008	102	21466	05	5.99	1200	7,188.00
22	1000090	3/25/2008	102	21467	05	5.99	85	509.15
23	1000111	3/18/2008	102	21490	05	5.99	49	293.51

FIGURE 17.11 Seeing the Results of Using VisualConnector

next three steps by setting the merger characteristics. Unlike the `JoinDatabase` task, the `VisualConnector` task doesn't rely on the special codes shown in Table 17.1 to configure the merger.

At this point, the code begins adding fields to the output using the `AddFieldTo-Include()` function, which requires a database identifier and field name as input. Notice that even if you want to include all of the fields from the primary database, you must list them individually. Using `IncludeAllFields()` will include all of the fields from all of the databases, which probably isn't what you want to accomplish. In almost every case, you'll list each field individually using the `AddFieldToInclude()` function, as shown in the example.

Now that the task is completely configured and the code has selected the required fields for output, it's time to call `PerformTask()`. After the task is complete, the code clears memory and then opens the output database. Figure 17.11 shows the output from this example.

Summary

This chapter has helped you understand the advanced IDEA features for working with databases. It's amazing just how many ways you can view and manipulate data in IDEA. These analysis features might not be an everyday requirement for you today, but they can become important tomorrow.

Sometimes you don't really know just how handy a feature is until you use it. That's the big thing to understand about this chapter—these features might not seem very useful until you start working with them with actual data. Take some time to try a few of the

more interesting features out now. As you work through the examples, consider how you might use the features for your own projects. Write the ideas down and then try them out as you have time. It's this extra effort that often makes a difference between getting a task done now and laboring over it for hours on end.

Chapter 18 takes a look at how you can work with charts and graphs in IDEA. The interesting thing about the next chapter is that automation makes it possible to use IDEA charts and graphs more efficiently, because you can update them much faster. Graphics speak volumes to most people—much more than tables of printed data. The information in Chapter 18 is your gateway to communicating more easily with others about the data you've analyzed. It also provides you with the means of creating more appealing reports and could even make it easier for you to see trends in data that you might not have seen in the past.

CHAPTER 18

Working with Charts and Graphs

Nothing speaks to other people in the same way graphics do. Unlike plain text, graphics help people see patterns in data by helping them visualize it. IDEA supports a number of different chart and graph types for different purposes. For example, a line graph presents data as a continuum, while a pie chart shows data as a part of a whole. This chapter helps you discover the basic charting that IDEA provides and how you can automate using it for just about any need.

Part of the purpose of graphics is to help analyze data. With this in mind, IDEA provides a number of analysis tools, only two of which appear in this chapter. Correlation is a measure of the strength and direction of a linear relationship between the values in two numeric fields in a database. In other words, Correlation helps you understand how the two fields compare. For example, it would be helpful to know if there's a correlation between the external temperatures of a building and the cost of heating. If heating costs rise without an associated decrease in external temperature, then you need to look for other sources of cost increases.

Trend Analysis shows whether data is generally increasing or decreasing over time (or even remaining the same) despite periodic spikes and troughs in individual values. Knowing the general trend in data helps you understand how the data might increase or decrease in value in the future. In fact, IDEA even helps you predict the future based on earlier trends!

Unlike other forms of analysis that you perform in IDEA, both Correlation and Trend Analysis are directly convertible into graphs or charts without any additional work on your part. The following sections describe the kinds of graphs and charts that IDEA supports and provides techniques for working with graphs and charts in code.

Choosing the Correct Chart or Graph

IDEA supports a number of chart and graph types to satisfy a variety of needs. The kind of chart or graph that you select determines the message that the output delivers. As previously mentioned, a pie chart conveys how particular data appears as part of a whole, while a line graph tends to show a continuum of information over time. In fact, IDEA supports these chart and graph types (the information displayed in Figures 18.1

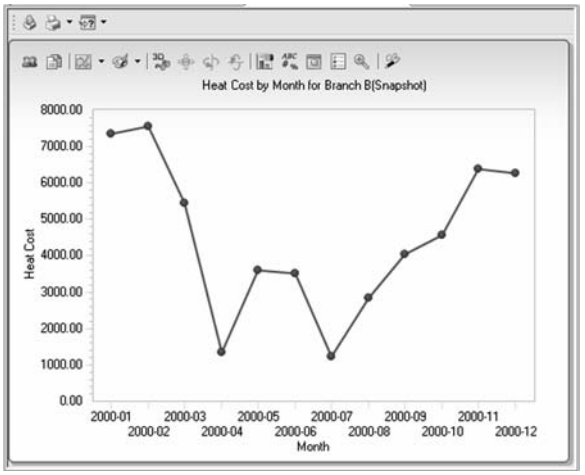


FIGURE 18.1 Line Graphs Emphasize Fluidity without Losing the Individual Data Points

through 18.6 comes from the same source so you can see how the same data looks in different forms—normally, you'd probably settle on one presentation for the data):

- **Line:** A graph that provides a continuum of data that emphasizes the fluidity of information from one data point to the next without losing the individual data points. Line graphs have a jagged look, as shown in Figure 18.1.
- **Bar:** A chart that emphasizes distinct data points and emphasizes differences between data points. Bar charts have a chunky look, like the one shown in Figure 18.2.

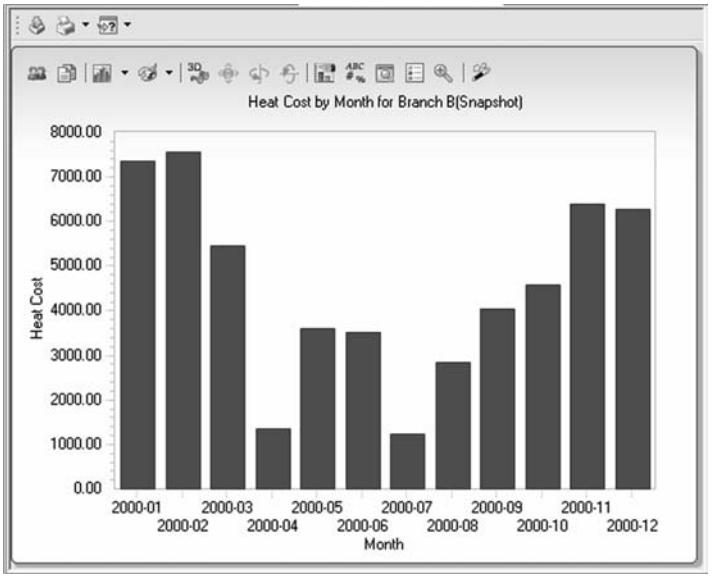


FIGURE 18.2 Bar Charts Emphasize Individual Data Points

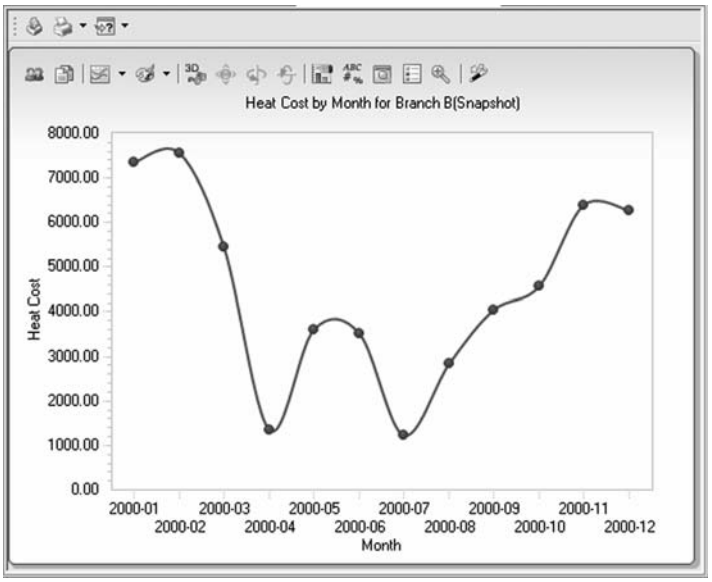


FIGURE 18.3 The Output Chart Is Simple, but Effective

- **Curve:** A graph that provides a continuum of data that emphasizes the data flow over individual data points. The curve may extend beyond data points in some cases to emphasize the flow, as shown in Figure 18.3.
- **Scatter:** A graph that places no emphasis on flow between data points and places the data points at the center. Using this kind of graph can help the viewer come to their own conclusions about data patterns as shown in Figure 18.4. A scatter graph can be good at illustrating clusters of data values.
- **Pie:** A chart that emphasizes the parts of a whole. For example, the monthly cost, as a percentage, for heat over an entire year. A pie chart normally includes a legend and can also include point labels, as shown in Figure 18.5.
- **Area:** A chart that emphasizes the amounts over a baseline (normally 0). The emphasis is on the area under the line, rather than the line itself as shown in Figure 18.6.

IDEA also supports the addition of a number of effects and additions for your charts and graphs. For example, you can add a 3D effect to your chart or graph. Figure 18.7 shows the same graph type as shown in Figure 18.2, but with a 3D effect. The 3D effect looks more eye-popping and dramatic, but can reduce the effectiveness of the presentation by obscuring the data to a degree.

Creating a Basic Graph

Theoretically, you can graph just about anything. If you want to graph the salaries of employees, the monthly flow of cash, or just about anything else in an organization,

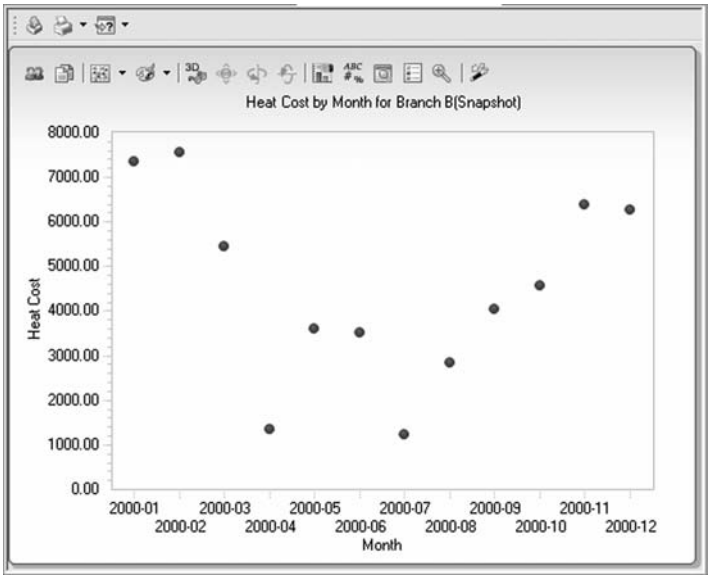


FIGURE 18.4 The Output Chart Is Simple, but Effective

you'll likely see new data patterns by doing so. In some cases, a graph or chart probably won't be quite as helpful—a graph of the credit limits of various customers probably won't tell you much unless you match it with some demographic, such as an area of the country. Most graphing and charting applications follow a pattern (although, you

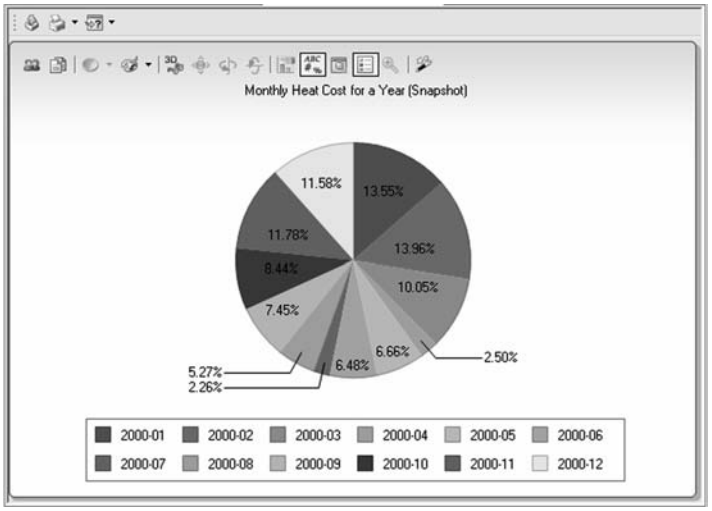


FIGURE 18.5 The Output Chart is Simple, but Effective

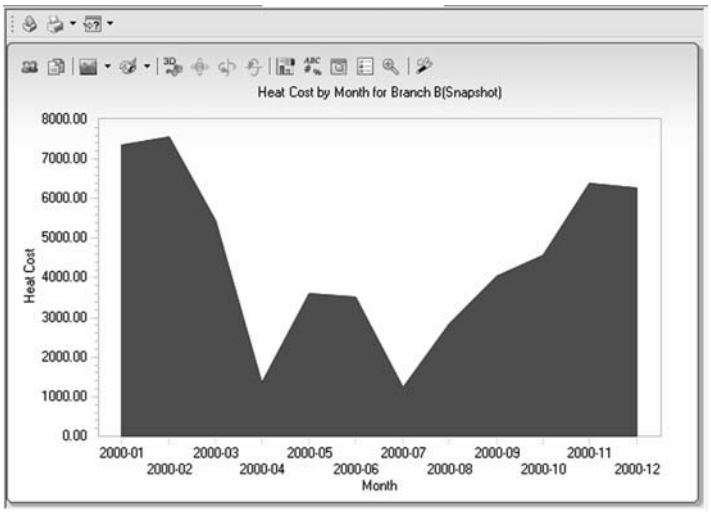


FIGURE 18.6 The Output Chart is Simple, but Effective

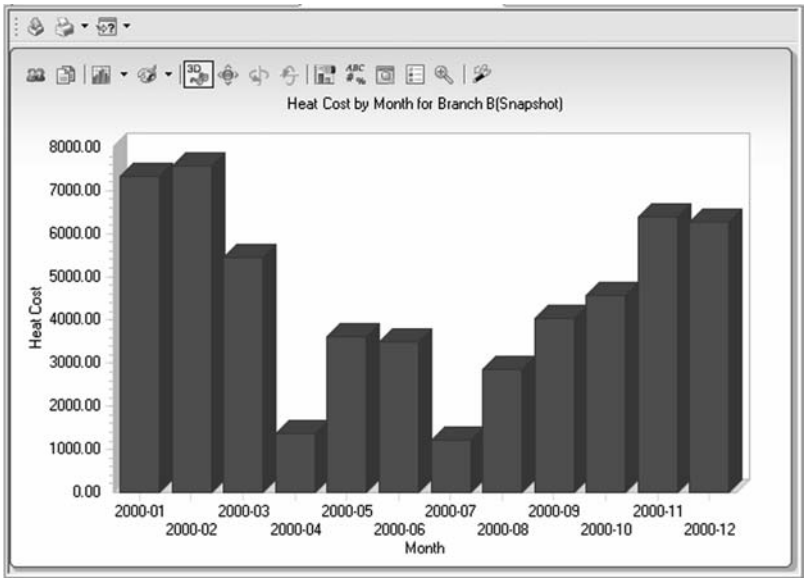


FIGURE 18.7 Using a 3D Effect Makes for an Exciting Presentation

certainly don't have to follow any pattern but the one you want to use). Here are the basic steps.

1. Determine how you want to display the data and what data you want to display.
2. Extract the required data from the main database and place it in another database to make it easier to work with.

3. Display the extracted data on screen using whatever charting or graphing technique you decide is best.

Depending on the complexity of the data, you might have to take additional steps to refine the data for display. The simpler you can make the source database, the easier it is to create a good graph or chart. Listing 18.1 shows the overview of the example application.

The code begins by obtaining the current working directory and then passing that information to a data extraction function, `IndexedExtraction()`. The `IndexedExtraction()` function returns a `Database` object that is then checked for errors. If this object is `Nothing`, then the extraction didn't work and the application exits.

When the data extraction is successful, the code passes the information to `ChartData()`, along with the working directory. `ChartData()` displays the data on screen. The code ends by clearing memory.

Now that you have the overview, let's look at some specifics. Listing 18.2 shows the `IndexedExtraction()` function.

The code begins by defining some constants used later in the code for determining the field type used for the data extraction. These constants are the same for every extraction, so you can copy them as shown in the book for your own applications.

Next, the code opens the database, `Sample-Advanced Statistical Methods.imd`. If the database is opened successfully, the code deletes the old extraction database, `GraphBFor2000.imd`. At this point, the code creates the `IndexedExtraction` task and begins configuring it.

LISTING 18.1 Creating a Chart Using ChartData

```
Sub Main
  ' Get the working directory.
  Dim Path As String
  Path = Client.WorkingDirectory

  ' Extract the data we want to graph from
  ' Sample-Advanced Statistical Methods.imd
  ' to GraphBFor2000.imd.
  Dim ExportDB As Database
  Set ExportDB = IndexedExtraction(Path)

  ' Check for errors.
  If ExportDB Is Nothing Then
     Exit Sub
  End If

  ' Chart the data found in GraphBFor2000.imd
  ChartData Path, ExportDB

  ' Clear memory.
  Set ExportDB = Nothing
End Sub
```

LISTING 18.2 Performing the Extraction

```
Function IndexedExtraction (Path As String)
  ' Constants used to determine field type.
  Const WI_IE_NUMFLD = 1
  Const WI_IE_CHARFLD = 2
  Const WI_IE_TIMEFLD = 3

  ' Open the database.
  Dim db As Database
  Set db = OpenDB(Path & "Sample-Advanced Statistical Methods.imd")

  ' Check for errors.
  If db Is Nothing Then
     Exit Sub
  End If

  ' Delete the old file.
  DeleteOld Path & "GraphBFor2000.imd"

  ' Create the task.
  Dim task As Task
  Set task = db.IndexedExtraction

  ' Configure the task.
  task.AddFieldToInc "MONTH"
  task.AddFieldToInc "HEAT_COST"
  task.Criteria = " BRANCH == ""B"""
  task.FieldToUse = "MONTH"
  task.FieldValueIs2 WI_IE_LTEQUAL, "2000-12", WI_IE_CHARFLD
  task.OutputFilename = "GraphBFor2000.imd"

  ' Perform the task.
  task.PerformTask

  ' Clear memory.
  Set task = Nothing
  Set db = Nothing

  ' Open the new database and
  ' return the extracted database object.
  Set IndexedExtraction = OpenDB(Path & "GraphBFor2000.imd")
End Function

Function OpenDB(DBPath As String) As Database
  ' Verify that the database exists.
  Dim PathCheck As String
  PathCheck = Dir( DBPath )

  ' If PathCheck has a 0 length, the database doesn't
  ' exist and we need to exit the routine.
  If Len(PathCheck) < 1 Then
     MsgBox "Couldn't file the file: " & DBPath
     Exit Function
  End If
```

(*continued*)

LISTING 18.2 (*Continued*)

```
  ' Define a database object.
  Dim db As Database

  ' Open the database using the default client folder.
  Set db = Client.OpenDatabase(DBPath)

  ' Return the database object.
  OpenDB = db

  ' Clear the memory used by db.
  Set db = Nothing
End Function

Sub DeleteOld(Filepath As String)
  ' Determine if the file exists.
  Dim PathCheck As String
  PathCheck = Dir( Filepath )

  ' Delete the file if it exists.
  If Len(PathCheck) >  1 Then
  MsgBox "Deleting old copy of " + PathCheck
  Kill Filepath
  End If
End Sub
```

It's important to keep the extracted database simple to make graphing or charting easier, so the example uses just two output fields, MONTH and HEAT_COST. The example is only interested in the B branch and the data for the year 2000 (as described as `MONTH <= "2000-12"`) by the `FieldToUse` property and `FieldValueIs2()` method.

Once the task is configured, the code calls `PerformTask()` to create the extracted database, `GraphBFor2000.imd`. The code clears memory and returns the new database to the caller for further processing. Because the extracted database isn't already open, the code calls `OpenDB` to open it.

Now that you have some data to use for building a chart or graph, it's time to see how to perform that part of the task. Listing 18.3 shows how to create a basic graph given a relatively straightforward database. You can use about the same process for any chart or graph, but the example uses a line graph.

Notice that the example passes `Database` as one of the input arguments. Using this approach means that you can use the same chart or graph code for any number of `Database` objects. Not only does this mean that you won't have to write a number of subroutines that vary by small details, but all of your charts and graphs will have the same general appearance, making them look consistent.

The code begins by deleting the old result. It then creates the `ChartData` task and begins configuring it. The code shows the configuration for a basic single series line graph. You provide a chart title, an X-axis title, a Y-axis title, the number of series, the chart type, and whether to use special features such as 3D, legends, and grids. The essential input is `XFieldName`, `AddYFieldName` (you can use multiple Y-axis fields), and `ResultName`.

LISTING 18.3 Charting the Data

```
Sub ChartData (Path As String, db As Database)
   ' Remove the old result.
   db.DeleteResultByName("HeatCostByMonth")

   ' Create the task.
   Dim task as Task
   Set task = db.ChartData

   ' Configure the task.
   task.ChartTitle = "Heat Cost by Month for Branch B"
   task.XFieldTitle = "Month"
   task.YFieldTitle = "Heat Cost"
   task.SnapShot = True
   task.NoOfSeries = 1
   task.ChartType = 0
   task.Show3DChart = False
   task.ShowGrids = False
   task.Legend = False
   task.Criteria = ""
   task.NumOfRecords = 12
   task.XFieldName = "MONTH"
   task.AddYFieldName "HEAT_COST"
   task.ResultName = "HeatCostByMonth"

   ' Perform the task.
   task.PerformTask

   ' Clear memory.
   Set task = Nothing
   Set db = Nothing
End Sub
```

At this point, the code calls `PerformTask()`. It then clears memory. Figure 18.8 shows the output from this example.

Defining Analytical Charts

A standard graph or chart simply displays your data. Analytical charts perform extra processing to tell you something new about your data. The following sections show how to use two kinds of analytical charts: Correlation and Trend Analysis.

Using Correlation

Correlation helps you discover how two sets of data relate to each other. The Correlation topic in the IDEA help file explains this form of analysis in more detail. Listing 18.4 shows how to perform this task.

The code begins by opening the `Sample-Advanced Statistical Methods` database. When the opening succeeds, the code deletes the old `Correlation` result and creates the `Correlation` task.

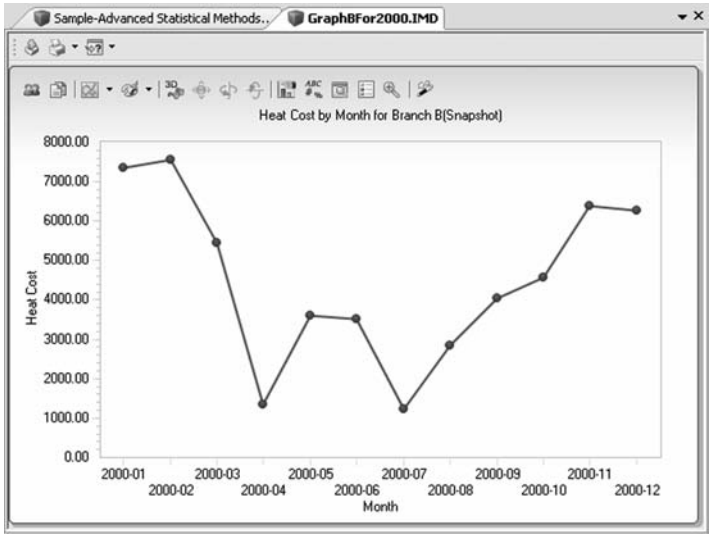


FIGURE 18.8 The Output Chart Is Simple, but Effective

`Task` configuration comes next. The code defines two correlation fields using the `AddFieldForCorrelation()` method. It then adds an audit unit using the `AuditUnitField` property and provides a `ResultName` property value of `Correlation`.

Once the task is configured, the code calls `PerformTask()`. It then clears memory and displays the tabular view, shown in Figure 18.9 on screen.

LISTING 18.4 Creating Charts with Correlation

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path & "Sample-Advanced Statistical Methods.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Remove the old result.
   db.DeleteResultByName("Correlation")

   ' Create the task.
   Dim task As Task
   Set task = db.Correlation
```

```
   ' Configure the task.
   task.AddFieldForCorrelation "AVERAGE_TEMP"
   task.AddFieldForCorrelation "HEAT_COST"
   task.AuditUnitField = "BRANCH"
   task.ResultName = "Correlation"

   ' Perform the task.
   task.PerformTask

   ' Clear memory.
   Set task = Nothing
   Set db = Nothing
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
   ' exist and we need to exit the routine.
   If Len(PathCheck) < 1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

The tabular view is nice, but it doesn't provide the same appeal as a graph. Fortunately, a single click of the leftmost button on the toolbar shown in Figure 18.9 displays the graphical view shown in Figure 18.10. If you're mystified by the output shown in Figures 18.9 and 18.10, the Examining the Correlation Results topic in the IDEA Help file explains the output in more detail.

Performing Trend Analysis

You see Trend Analysis used just about everywhere. Even on television, someone wants to show the overall direction of a number of data points using a simple line. Trend

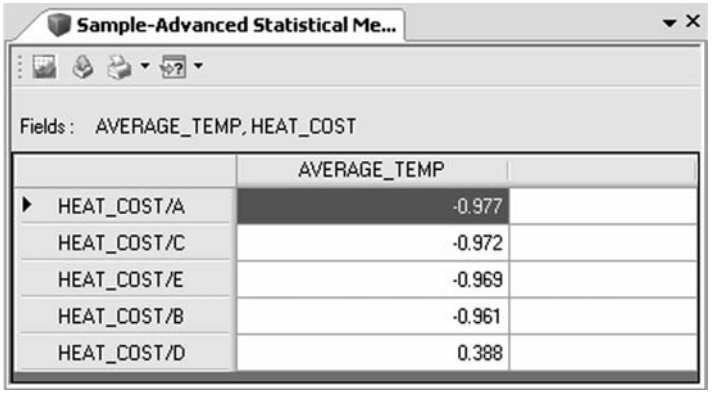


FIGURE 18.9 The Initial View of the Correlation Is Text-Based

Analysis gives you the bottom line. To discover more about Trend Analysis, view the Trend Analysis topic in the IDEA Help file. Listing 18.5 shows how to perform this task.

As with all of the other examples in this chapter, this example begins with the `Sample-Advanced Statistical Methods` database. In order to perform Trend Analysis, you begin by creating the `TrendAnalysis` task.

At this point, the code begins configuring the task by defining the `TrendField`, `RefField`, and `AuditUnitField` properties used to calculate the trend. You have the

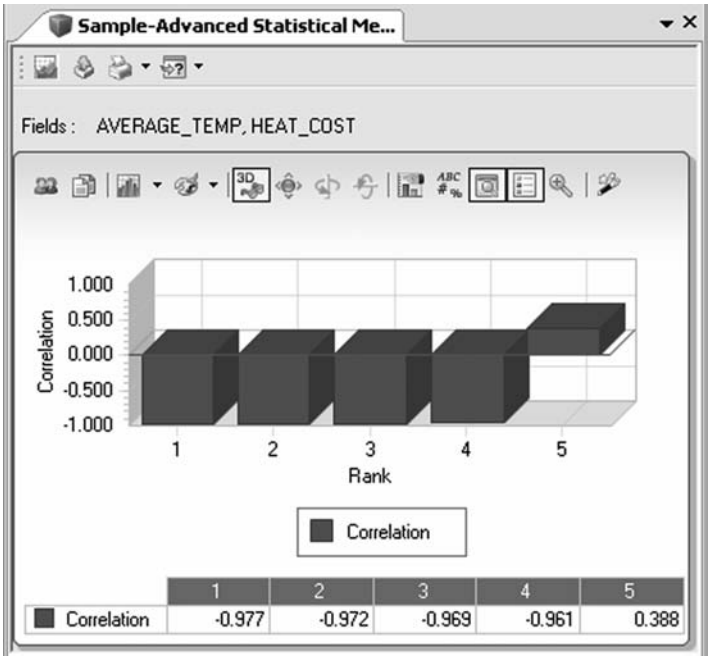


FIGURE 18.10 A Few Clicks Create a Graphical View of the Data

LISTING 18.5 Working with Trend Analysis

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path & "Sample-Advanced Statistical Methods.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Remove the old result.
   db.DeleteResultByName("Trend Analysis")

   ' Create the task.
   Set task = db.TrendAnalysis

   ' Configure the task.
   task.TrendField "HEAT_COST"
   task.RefField "YEAR_AVERAGE_HEAT_COST"
   task.AuditUnitField = "BRANCH"
   task.GenerateForecasts False, 96, 0
   task.TimeScale = 1
   task.CalendarValue = 1
   task.TimeScaleStartAndIncrement 1, 1
   task.ResultName = "Trend Analysis"

   ' Perform the task.
   task.PerformTask

   ' Clear memory.
   Set task = Nothing
   Set db = Nothing
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
   ' exist and we need to exit the routine.
   If Len(PathCheck) < 1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If
```

(*continued*)

LISTING 18.5 (*Continued*)

```
   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function
```

option of creating a forecast based on the Trend Analysis, but this example doesn't perform this task. The next step is to define the output characteristics using the `TimeScale`, `CalendarValue`, and `TimeScaleStartAndIncrement` properties. The output is placed in the Trend Analysis result.

The code calls `PerformTask()` once the configuration is complete. It then clears memory. Figure 18.11 shows the output from this example.

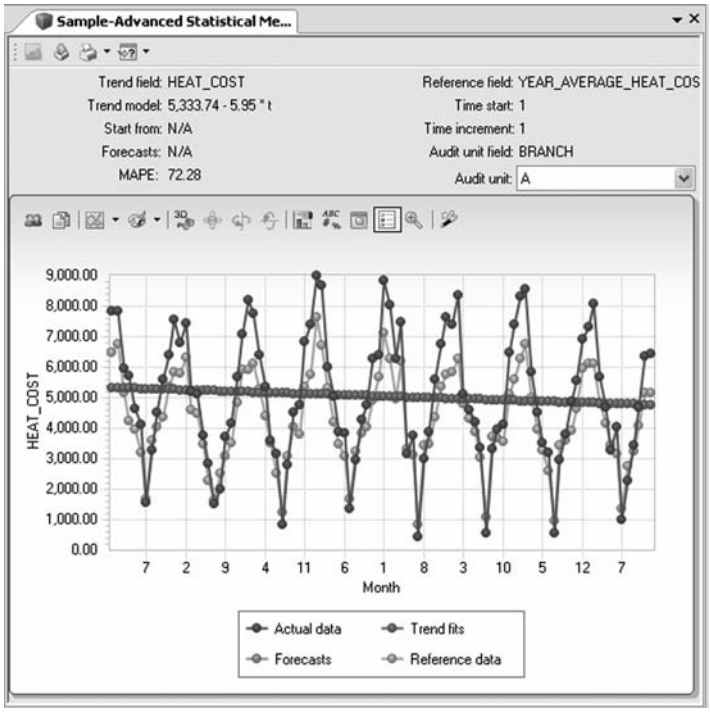


FIGURE 18.11 Considering How Trend Analysis Works

Summary

This chapter has demonstrated techniques for working with charts and graphs in code. At this point, you can see that automating the creation of charts and graphs that you use often will save considerable time and effort on your part because you won't have to repeat the manual process of creating them. Charts and graphs are important because they often communicate information better than mere words or tabular data. Of course, the one thing this chapter can't do is tap your imagination. Creative use of charts and graphs makes something that's good even better.

In order to get the most out of this chapter, you have to give yourself permission to play. Spend some time viewing your data in different ways, using different chart and graph types. Try extracting the data in different ways and creating the charts and graphs using varying techniques. You'll find that IDEA is extremely flexible in this regard and can produce precisely the message that you want the chart or graph to convey. If you find that you're in a quandary over precisely which chart or graph to use, try your ideas out on a few friends and see if they get the message you're trying to present.

Chapter 19 moves on to another presentation topic, reports. A picture might be worth a thousand words, but sometimes you need the abstract power of words to completely convey your ideas to other people. A good report, coupled with good graphics, can make your message clear and the value of your analysis plain to everyone. In short, this chapter is really just the beginning and Chapter 19 completes the process of helping you define usable methods for communicating with other people using IDEA.

CHAPTER 19

Defining Reports

Reports go hand-in-hand with graphics in conveying your findings to others. Chapter 18 describes how to create graphics using IDEA. This chapter discusses the report part of the equation.

IDEA makes it possible to output reports in two different forms: PDF and Microsoft Word (the Word output is actually in Rich Text Format, RTF, so any application that can read RTF can use this output). Each output format has advantages and disadvantages. For example, using PDF files makes it easy for everyone to view the report. However, letting others make changes to the report can be difficult unless you want to buy everyone a full copy of Adobe Acrobat.

The examples in this chapter show how to create output directly from a database. Of course, you may want to create customized output, which means writing code to output the custom material. Chapter 16 shows how to create such an application using Word. The following sections show how to create general report output using IDEA.

Defining a Report

Before you can output a report, you need to create one. This is one part of the report creation process that you must perform manually—there isn't any way to code reports. The following steps show you how to create the report used for the examples in this chapter. The same procedure works for any report you want to create.

1. Open the `Sample-Customers` database by double-clicking its entry in the **File Explorer**. You see the `Sample-Customers` database opened in IDEA.
2. Select **File** > **Print** > **Create Report**. You see the **Report Assistant** dialog box shown in Figure 19.1.
3. Select an orientation for the report. The example uses horizontal.
4. Choose between a new and existing report. The example is a new report.
5. Select options for the report. The example uses the default options of not allowing headers to span multiple lines, showing record numbers, and showing the database name in the report.

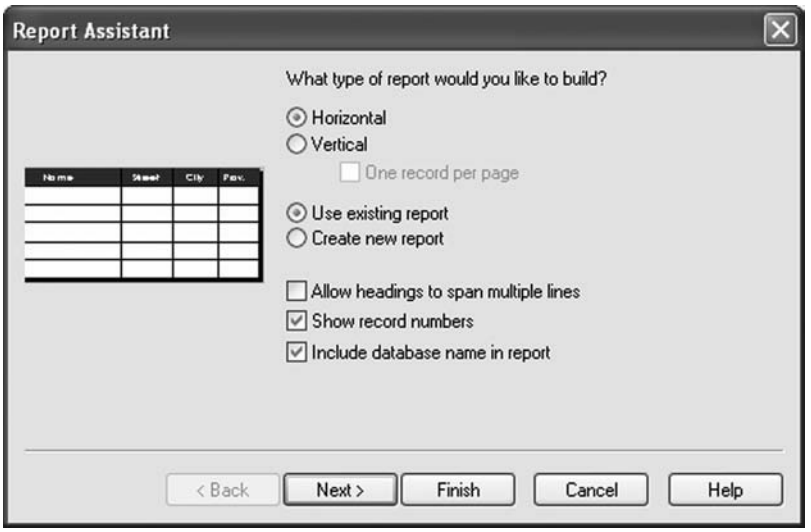


FIGURE 19.1 The Report Assistant Leads You Through the Process of Creating a Report

6. Click **Next**. You see the **Headings** page of the **Report Assistant** shown in Figure 19.2.
7. Define a font, alignment, and title for each of the headings that will appear in the report. The example uses the default font and alignment for each of the fields. It uses the following heading titles (in order of the fields shown in Figure 19.2): Customer Number, Company, First Name, Last Name, Country, Status, and Credit Limit.

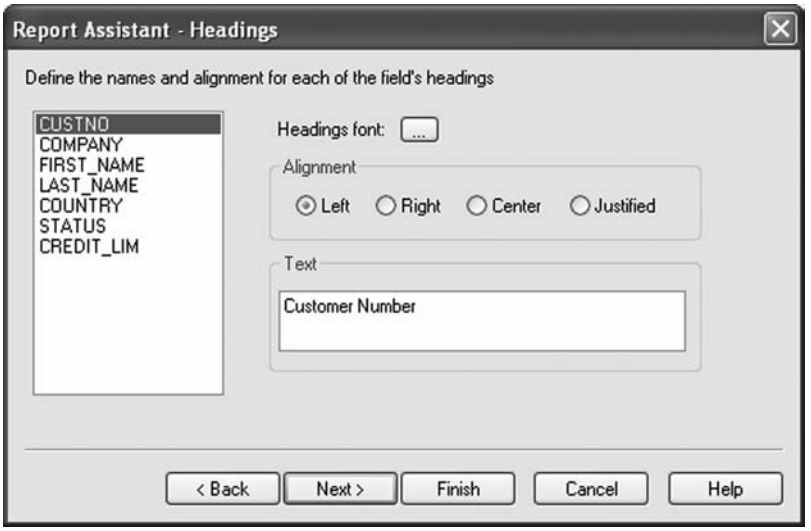


FIGURE 19.2 Define Headings for Each of the Fields You Wish to Include in the Output

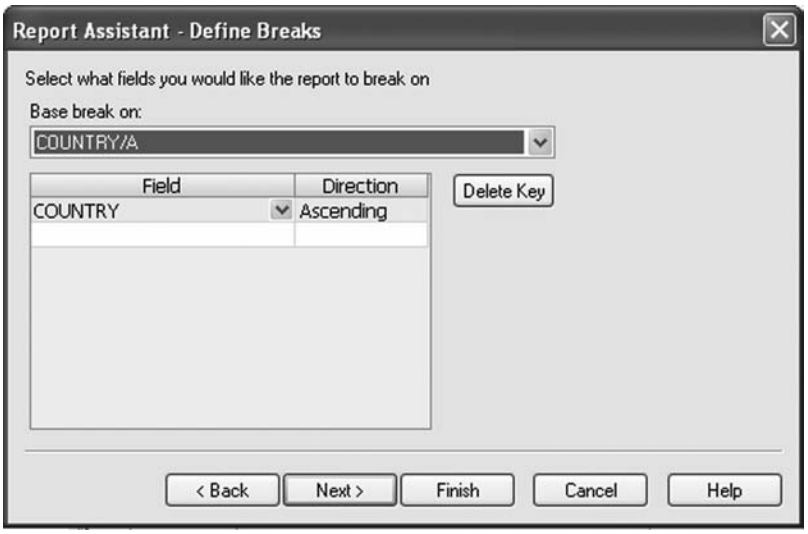


FIGURE 19.3 Create Breaks (Groups) for the Report

8. Click **Next**. You see the **Define Breaks** page of the **Report Assistant** as shown in Figure 19.3. The figure shows the break used for the example report.
9. Select a field name and sort order for each of the breaks you want to include in the report.
10. Click **Next**. You see the **Report Breaks** page of the **Report Assistant** shown in Figure 19.4. The figure shows the report break configuration for the example report.

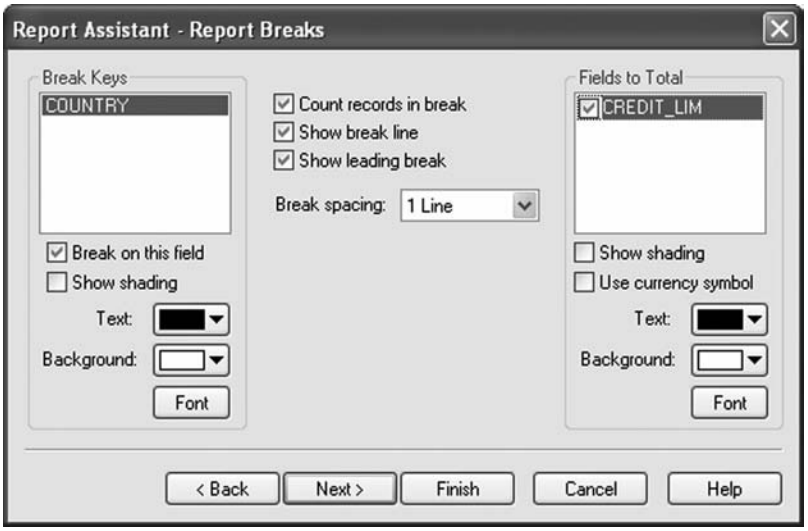


FIGURE 19.4 Define How You Want the Breaks to Appear in Your Report

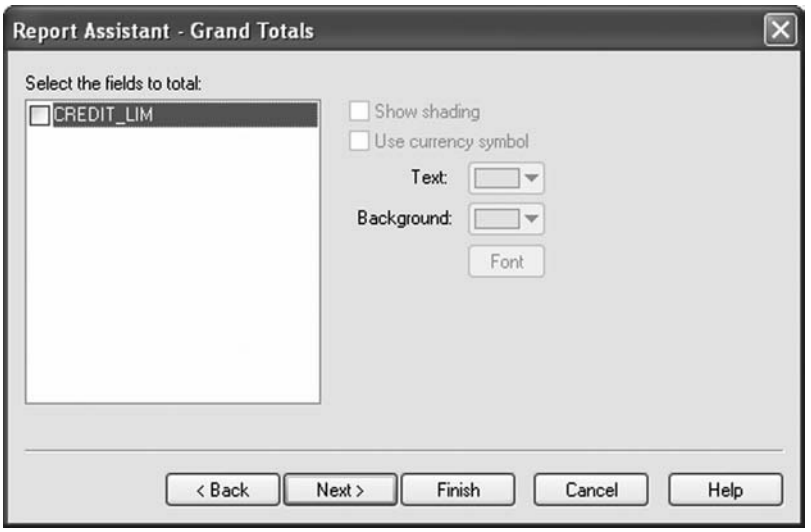


FIGURE 19.5 Add any Required Grand Totals to the Report

11. Specify how you want each of the break keys to appear in the report. You can define whether you want the break field to actually produce a break, the break coloring, and the break font. Each break can also use options such as counting the number of records in the break. If you want, you can also total the break and configure how the total appears.
12. Click **Next**. You see the **Grand Totals** page of the **Report Assistant** shown in Figure 19.5. IDEA automatically selects fields that could provide a grand total for

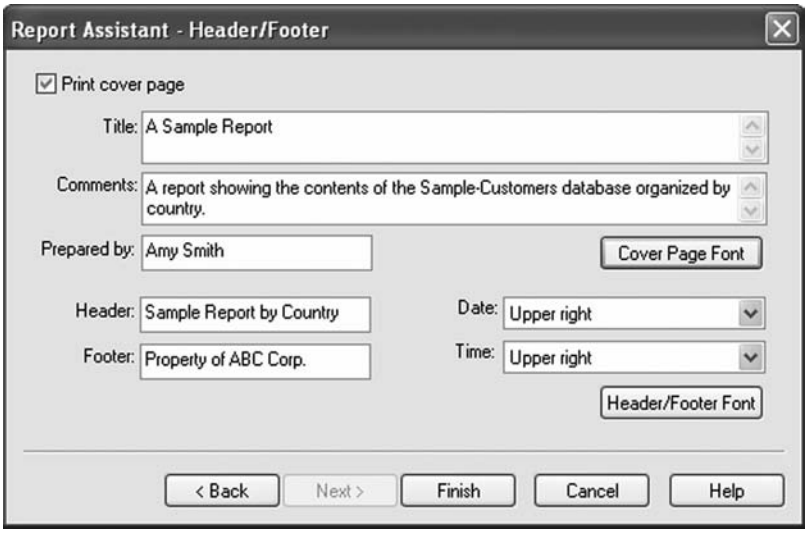


FIGURE 19.6 Modify the Report Title, Header, and Footer as Needed to Identify Your Report

you, even if these fields aren't part of the breaks you supplied earlier. The figure shows the grand totals configuration for the example report.

13. Provide any required grand totals for the report by checking the associated field. You can configure the grand totals to use shading and the currency symbol. It's also possible to change the text coloring (both foreground and background) and font.
14. Click **Next**. You see the **Header/Footer** page of the **Report Assistant** as shown in Figure 19.6. The figure shows the configuration for the example report.
15. Determine whether you want a cover page. If so, select the **Print cover page** option. Define values for the **Title, Comments**, and **Prepared by** fields. Modify the cover page font if desired.
16. Add header and footer details as desired by modifying the **Header, Footer, Date**, and **Time** fields. You can modify the header and footer font, if desired.
17. Click **Finish**. IDEA asks whether you want to preview the report you created. If you preview the report, you can ensure it contains everything you want. Figure 19.7

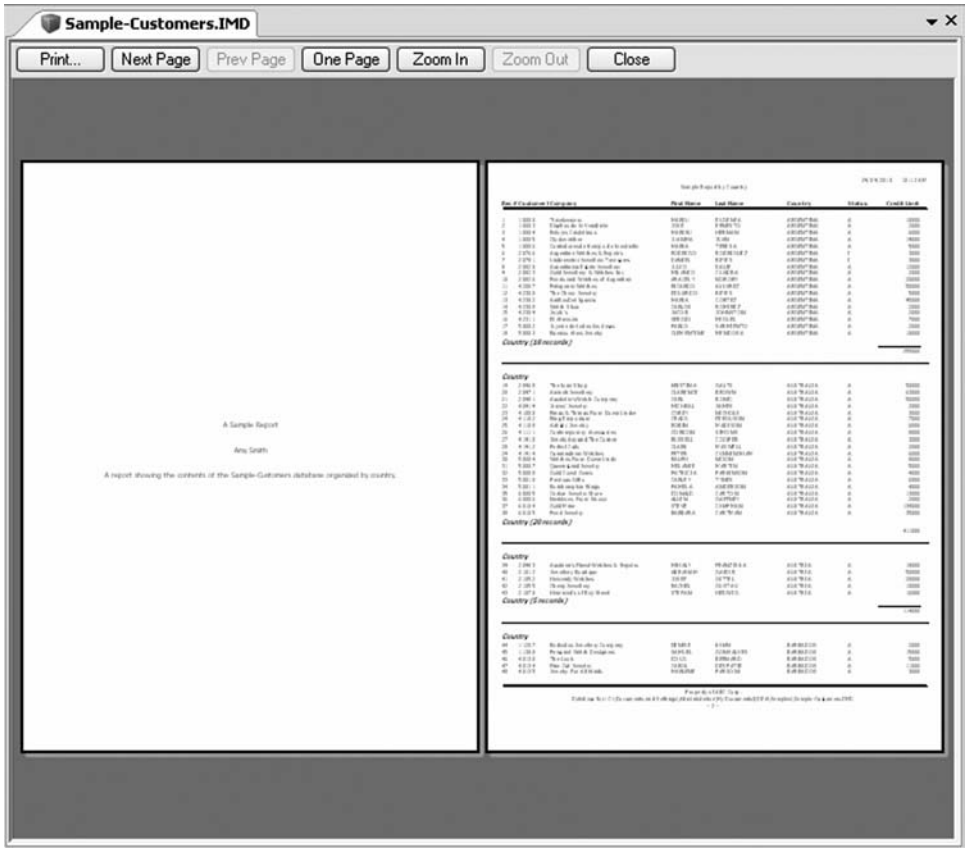


FIGURE 19.7 The Example Report Preview Shows How the Settings You Made in the Report Assistant Affect Report Content and Appearance

shows how the example report appears in preview mode. You can use the single page view to see more details. It's also possible to zoom in and zoom out as needed.

At this point, you can save the report as a view by selecting **View** > **Views** > **Save View**. Enter a name for the view in the **File name** field of the **Save As** dialog box and click **Save**. Saving the report as a view makes it possible to select from multiple reports later. Reports aren't saved in the **Properties** window, so you do need to save them as views if you want to recall them later.

There isn't any way to programmatically select a report. To use a report later, select **View** > **Views** > **Open View**. Select the view you want to use in the Open dialog box and click **Open**. If you want to see the report before you print it, select **File** > **Print** > **Print Preview**. After you finish using the report, return IDEA to the default view by selecting **View** > **Views** > **Reset**.

Outputting Data in PDF Format

Using PDF format for your report has a number of advantages. The biggest advantage is that you can find a version of Adobe Acrobat for just about any platform made, which means that it doesn't matter what kind of computer your viewer uses. A PDF file tends to provide a more professional view than other kinds of report formats. It's relatively easy to add content if you have the full version of Adobe Acrobat (but not as easy as using Word). You can also protect the finished document from changes. A PDF file also tends to be smaller than a Word document, so it requires less time to transfer using e-mail and other techniques. There are many other reasons to use PDF files, but these are the main reasons to use it for an IDEA database. Listing 19.1 shows how to create a report in PDF form.

LISTING 19.1 Creating a PDF from a Database

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path & "Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Delete the old file.
   DeleteOld Path & "Sample-Customers.pdf"

   ' Set the output filename.
   db.FilenameForPublishing = Path & "Sample-Customers.pdf"
```

```
   ' Perform the task.
   db.PublishToPDF

   ' Open the report for viewing.
   Shell "cmd.exe /c " & Chr(34) & Path & _
      "Sample-Customers.pdf" & Chr(34)
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
   ' exist and we need to exit the routine.
   If Len(PathCheck) < 1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function


Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) > 1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub
```

The example begins by opening the `Sample-Customers` database. If the database opens without error, the code deletes any old copy of the report in the output directory and begins creating the new report.

Creating the report consists of two steps. First, you must define the output location using the `FilenameForPublishing` property. Second, you call `PublishToPDF()`.

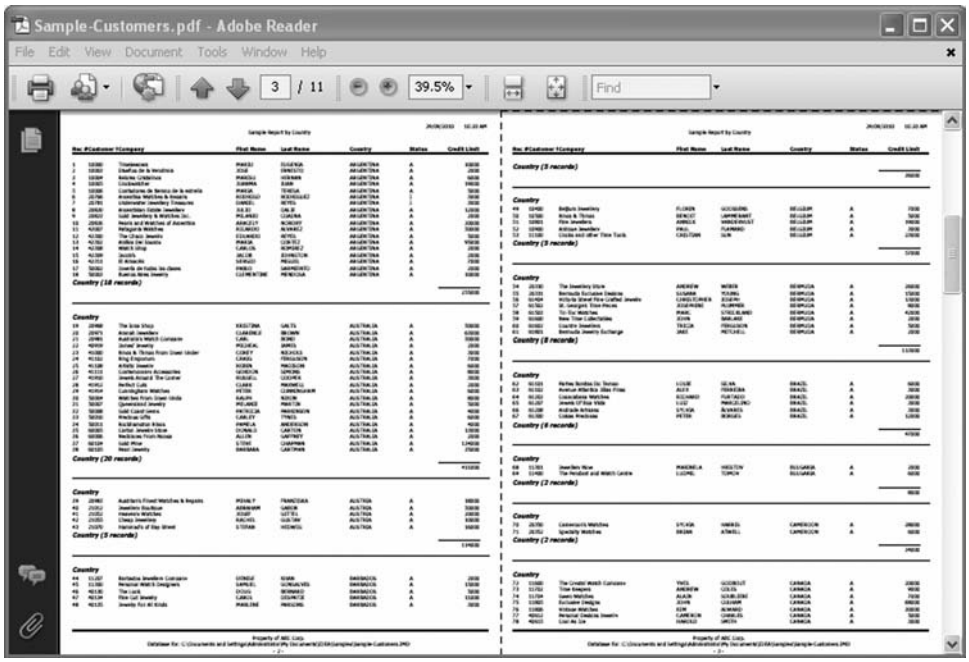


FIGURE 19.8 The PDF Format Provides an Aesthetically Pleasing Way to Present Database Information

The `Shell()` function makes it possible to display the resulting report automatically. Figure 19.8 shows the resulting report.

Outputting Data in Word Format

Reports created in Word tend to provide the best means of sharing information that you want others to comment on and possibly modify. It's a great way to create a report for a group scenario because Word provides a flexible way for multiple authors to contribute and keep track of who is contributing what. It's also possible to easily add graphics and other touches using Word that might be difficult when working with other report formats. Of course, the downside to using Word is that everyone must have a full copy in order to participate in the editing process. Listing 19.2 shows how to create a report using Word.

As with the PDF report, the code begins by opening the database and deleting the old report. After the code completes these steps, it supplies the new report name using the `FilenameForPublishing` property. The code then calls `PublishToMicrosoftWord()` to output the report. The `Shell()` function lets you open the report immediately for viewing and possible modification.

LISTING 19.2 Creating a Word Document from a Database

```
Sub Main
   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path & "Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Delete the old file.
   DeleteOld Path & "Sample-Customers.doc"

   ' Set the output filename.
   db.FilenameForPublishing = Path & "Sample-Customers.doc"

   ' Perform the task.
   db.PublishToMicrosoftWord

   ' Open the report for viewing.
   Shell "cmd.exe /c " & Chr(34) & Path & _
      "Sample-Customers.doc" & Chr(34)
End Sub

Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
   ' exist and we need to exit the routine.
   If Len(PathCheck) < 1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db
```

(*continued*)

LISTING 19.2 (*Continued*)

```
   ' Clear the memory used by db.
   Set db = Nothing
End Function

Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) > 1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub
```

When the `Shell()` command executes, you should expect to see the Convert File dialog box shown in Figure 19.9 when you have Word set to tell you about file conversions (always a good idea for security purposes). Simply select the **Rich Text Format (RTF)** option in the **Convert file from** field and click **OK** to perform the conversion.

Once the conversion process is complete, you can work with the document using any view that Word supports. Figure 19.10 shows the Print Layout format.

If someone doesn't have Word, but still wants to review the report, they can use WordPad. The WordPad view of the information isn't as nice as the one provided in Word, but it does work. Figure 19.11 shows the same report (using an `.rtf` extension) in WordPad.

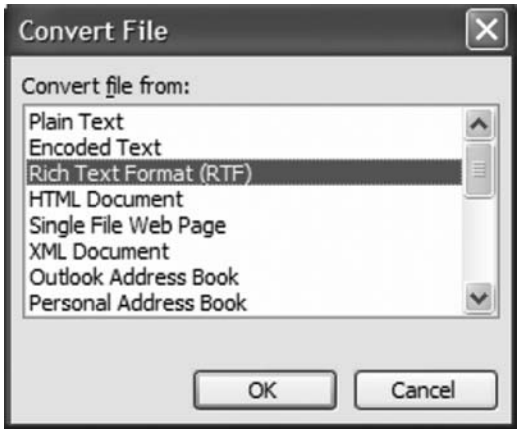
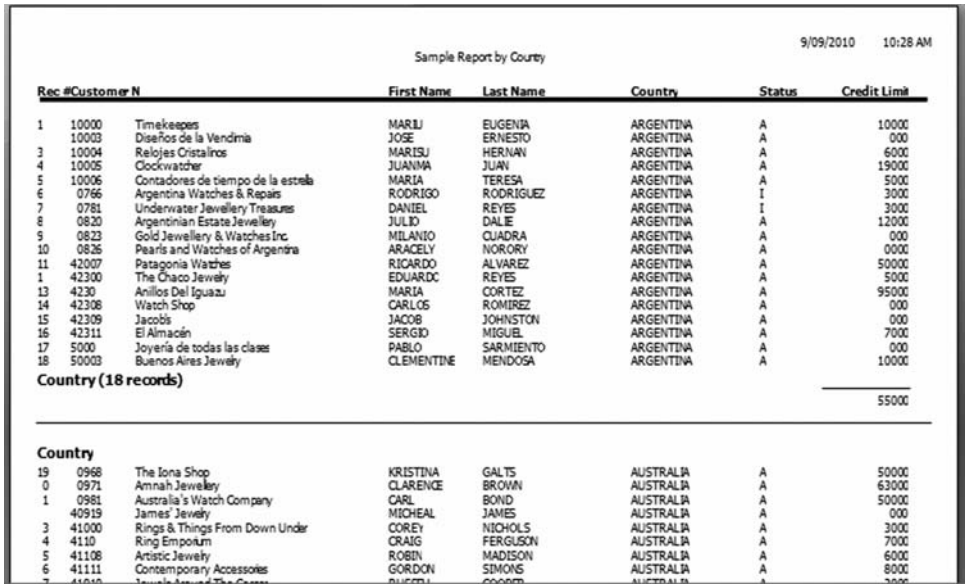


FIGURE 19.9 Tell Word to Convert the File from an RTF Document



9/09/2010 10:28 AM

Sample Report by Country

Rec #	Customer N		First Name	Last Name	Country	Status	Credit Limit
1	10000	Timekeepers	MARIU	EUGENIA	ARGENTINA	A	10000
	10003	Diseños de la Vendimia	JOSE	ERNESTO	ARGENTINA	A	000
3	10004	Relojes Cristalinos	MARISU	HERNAN	ARGENTINA	A	6000
4	10005	Clockwatcher	JUANMA	JUAN	ARGENTINA	A	19000
5	10006	Contadores de tiempo de la estrela	MARIA	TERESA	ARGENTINA	A	5000
6	0766	Argentina Watches & Repairs	RODRIGO	RODRIGUEZ	ARGENTINA	I	3000
7	0781	Underwater Jewellery Treasures	DANIEL	REYES	ARGENTINA	I	3000
8	0820	Argentinian Estate Jewellery	JULIO	DALIE	ARGENTINA	A	12000
9	0823	Gold Jewellery & Watches Inc.	MILANIO	CUADRA	ARGENTINA	A	000
10	0826	Pearls and Watches of Argentina	ARACELY	NORORY	ARGENTINA	A	0000
11	42007	Patagonia Watches	RICARDO	ALVAREZ	ARGENTINA	A	50000
1	42300	The Chaco Jewelry	EDUARDC	REYES	ARGENTINA	A	5000
13	4230	Anillos Del Iguazu	MARIA	CORTEZ	ARGENTINA	A	95000
14	42308	Watch Shop	CARLOS	ROMIREZ	ARGENTINA	A	000
15	42309	Jacobs	JACOB	JOHNSTON	ARGENTINA	A	000
16	42311	El Almacén	SERGIO	MIGUEL	ARGENTINA	A	7000
17	5000	Joyería de todas las clases	PABLO	SARMIENTO	ARGENTINA	A	000
18	50003	Buenos Aires Jewelry	CLEMENTINE	MENDOSA	ARGENTINA	A	10000
Country (18 records)							55000
Country							
19	0968	The Iona Shop	KRISTINA	GALTS	AUSTRALIA	A	50000
0	0971	Amnah Jewellery	CLARENCE	BROWN	AUSTRALIA	A	63000
1	0981	Australia's Watch Company	CARL	BOND	AUSTRALIA	A	50000
	40919	James' Jewelry	MICHEAL	JAMES	AUSTRALIA	A	000
3	41000	Rings & Things From Down Under	COREY	NICHOLS	AUSTRALIA	A	3000
4	4110	Ring Emporium	CRAIG	FERGUSON	AUSTRALIA	A	7000
5	41108	Artistic Jewelry	ROBIN	MADISON	AUSTRALIA	A	6000
6	41111	Contemporary Accessories	GORDON	SIMONS	AUSTRALIA	A	8000

FIGURE 19.10 Word Provides an Advantage in Making the Data Easy to Work With

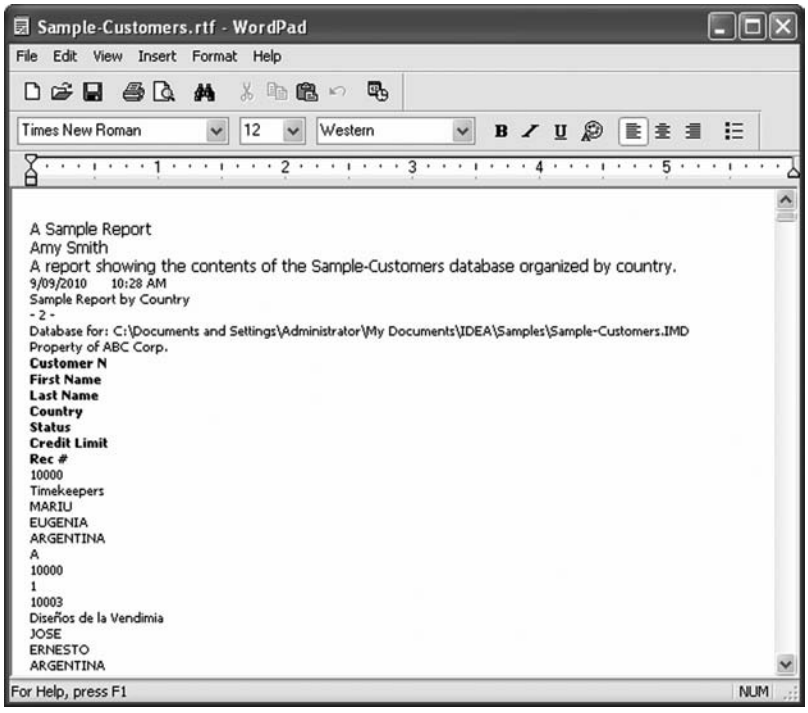


FIGURE 19.11 WordPad Offers an Alternative Way to View the Word Output of IDEA

Summary

This chapter has demonstrated how to output IDEA databases in various report formats. Choosing the right format is essential if you want to ensure others will be able to use the reports as intended. If you really want to create custom output, then you need to review the Word example in Chapter 16.

Now that you know how to create your own reports, it's time to give them a try. Use the techniques in this chapter to output the same database in PDF and Word format. Spend some time working with the reports in different ways to determine which formats work best for specific needs.

Chapter 20 discusses the important topic of database security. IDEA already protects your data to a certain extent by making the original databases uneditable. However, as you add custom fields, you'll want to protect their content because this data is editable. In addition, you want to ensure your applications perform as expected by ensuring someone can't input unanticipated data that crashes the application and causes it to do unexpected things.

CHAPTER 20

Considering Database Security

IDEA already provides some safeguards for data that you work with. For example, you can't modify the original data in a database. It's possible to add new fields of your own data, extract the data to a new location, and perform other manipulations, but the original data is pretty much inaccessible except for reading. That means that someone with ill intent can't contaminate the original data—at least, not without a whole lot of effort that's well outside the programming capabilities of most people. That's the good news, and what good news it is! Just knowing you can always go back to the original data should provide some reassurance to you immediately.

However, there are all kinds of other ways in which people can cause problems for you. In many cases, these issues aren't even caused purposely, but are the result of simple human error. Whether application security is purposely breached or benignly bungled, the issue is that somehow application integrity or data is compromised or damaged. Security actually covers quite a bit of ground, but this chapter focuses on some simple ways of making your application significantly more secure without a lot of work on your part or the use of exotic techniques.

Warning

Some people will try to tell you that it's possible to provide complete protection of your software and data. The fact is that if someone really wants to get into your system, they'll likely find a way to do it given enough time and skill. Security measures prevent accidental breaches and discourage less skilled intruders from breaking in. The idea is to put up enough of a wall that the intruder decides to find someone less prepared than you are. The best security is someone who is prepared and constantly looking for breaches. The human monitoring factor is exceptionally important for secure systems.

Considering Programmatic Data Security

Most of the security holes in applications today are the result of bugs of some sort. That's why you often receive an update for an application to fix the security errors that it

contains. In many cases, the bug is one of omission. The code works fine normally. However, if some extraordinary circumstance occurs, the code may do something unexpected. The simple act of adding a check for the extraordinary condition fixes the problem, but you need to know to add the check in many cases and it isn't always easy to predict where code will break.

Fortunately, IDEAScript already protects you from a good many of these programmatic security issues. For example, you won't have to worry about *buffer* (an area of memory used to store data) checks because IDEA manages memory for you.

You may have noticed in previous chapters that the example code provides a check to ensure that a database actually opened after the code made a request. Here's a code snippet showing the code in question.

```
' Get the working directory.
Dim Path As String
Path = Client.WorkingDirectory

' Open the database.
Dim db As Database
Set db = OpenDB(Path & "Sample-Customers.imd")

' Check for errors.
If db Is Nothing Then
   Exit Sub
End If
```

This check helps ensure that the application doesn't do anything odd if the database doesn't open for whatever reason. It also relieves the application user about having to think about an error, should one occur. In short, while this check's main purpose is to proactively protect the application from a missing database, the end result is a kind of security check that also ensures that no one outside the application can force the application to do something unexpected. Programmatic security often takes this form in IDEAScript—it prevents the application from working in an unexpected way, and provides a suitable alternative action when errors do occur.

Security also means being proactive about reducing unnecessary user interaction. For example, in many situations, the code deletes an existing database or result, rather than rely on the user to tell IDEA that it's ok to do so. Here's an example of a deletion found in Chapter 19:

```
' Delete the old file.
DeleteOld Path & "Sample-Customers.pdf"
```

By keeping control over how the application works, you improve application security. Any time you give the user a decision to make, you risk receiving incorrect input that could cause the application to malfunction. Use the features of IDEAScript to maintain control whenever possible and to control user interaction at all times to maintain application security.

Choosing the Correct Data Type

Data type is something that many developers don't define. In fact, in IDEAScript, you can literally use variables without ever declaring any of them. However, taking this approach is error prone and could lead to all kinds of security problems. A simple variable name misspelling could cause your code to crash and finding the source of the problem is incredibly difficult when your application becomes complex. In fact, if you find that you tend not to declare your variables before you use them, the best security change you can add to your code is to include: `Option Explicit` at the beginning of every application. Using `Option Explicit` forces you to declare your variables. You may find that declaring all your variables is time consuming, but you'll find that your applications work much better if you do and you'll close potential security holes in the process.

Choosing the correct data type is important when writing IDEAScript. However, IDEAScript often covers up problems in your data type selections. For example, the following code may look incorrect, but it runs just fine. In fact, the message box displays an output value of 8.

```
Sub Main
   Dim MyName As String
   MyName = 3
   MsgBox VarType(MyName)
End Sub
```

You know from Listing 5.7 that the output of 8 equates to the String data type. So, what happened in this code snippet? IDEAScript has automatically converted the Integer input to a String. Of course, you may ask why you should bother then with data type if IDEA will perform the required conversions correctly. The answer is that you want to ensure you know what the code is doing. If you looked at the code in the snippet and thought that it was incorrect, then you can better understand why using the correct data type is essential. Table 20.1 shows a listing of the most common data types in IDEAScript (see Chapter 5 for additional details).

The Object data type can be especially hard for some developers to understand and work with. When you create objects, such as a `Task` or `Database`, it might be tempting to simply use Object. Indeed, in some cases, such as when working with an external application, you must use Object. However, whenever possible, use the actual object type to declare your variables. Again, the purpose of taking the extra time to do this is to make your code easier for you to understand—especially after you've put it down for a while and don't remember how it works.

Validating Data

Most people associate security with building a wall. However, security is often more an issue of simple vigilance. In fact, you can view it as a matter of helping. Validating

TABLE 20.1 Common Data Types in IDEAScript

Type	VarType( ) Output
Empty	0
Null	1
Integer	2
Long	3
Single	4
Double	5
Currency	6
Date	7
String	8
Object	9
Error	10
Boolean	11
Variant	12
DataObject	13

data is one place in which you're helping, more than hindering (as a wall would do). By validating data, you ensure the application runs properly and can help users with mistakes long before they become an error. For example, if the user enters "three" instead of "3," you can provide a message that tells the user to type a numeric value rather than a word. Such a message is helpful. More importantly, it's specific, whereas an error message might not be very specific at all.

There are all kinds of data validation checks you could perform if you were using another language. Given the forgiving nature of IDEAScript and the kinds of applications you're writing, you don't have to become mired in all kinds of exotic checks. In fact, if you perform the checks described in the following three sections, you'll find that there are actually very few ways in which a user can cause your application to crash by accidentally (or purposely) providing the wrong kind of data to your application.

Verifying the Data Range

The first check you can perform on data is to check the data range. For example, a user could reasonably enter a value of 10 in a dialog box, but if you're expecting a number in the range of 0 through 9, then a value of 10 is incorrect. IDEA can't check for this kind of error. All that IDEA knows is that the user supplied an Integer. Listing 20.1 shows an example of checking both data type and data range.

LISTING 20.1 Checking Data Type and Range

```
Sub Main
   Dim Value As Integer
   Dim KeepTrying As Boolean
   KeepTrying = True
   Dim Response as Integer
```

```
   ' Keep trying to obtain correct input.
   Do While KeepTrying

      ' Handle incorrect input type errors.
      On Error GoTo HandleError

      ' Obtain required input from the user.
      Value = InputBox("Type a number between 1 and 9")

' This label is where the code resumes after a type error.
AfterTypeError:

      ' Check the data input range.
      If Not (Value >  0 And Value <  10) Then

         ' The input is incorrect.
         Set Response = MsgBox( _
            "You need to type a numeric value between 1 and 9. " _
            & "Keep trying?", _
            MB_YESNO Or MB_ICONEXCLAMATION, _
            "Incorrect Input")

         ' If the user doesn't want to try again, exit.
         If Response = IDNO Then
            Exit Sub
         End If

      Else
         ' We have good data, so no need to keep trying.
         KeepTrying = False
      End If
   Loop

   ' Display the result.
   MsgBox "You typed: " & CStr(Value)
   Exit Sub

HandleError:

   ' If this is a type error.
If Err.Number = 13 Then

      ' Set Value to an incorrect numeric value.
      Value = 0

      ' Tell the user what went wrong.
      MsgBox "Please type a number, not text.", _
         MB_ICONEXCLAMATION, "Incorrect Input Type"
```

(*continued*)

Listing 20.1 (*Continued*)

```
      ' Return to the original code so that it loops and
      ' asks the user to input the data again.
      GoTo AfterTypeError

   ElseIf Err.Number = 6 Then

      ' Set Value to an incorrect numeric value.
      Value = 0

      ' Tell the user what went wrong.
      MsgBox "The number you typed is too large. " _
         & "Use a number between 0 and 9", _
         MB_ICONEXCLAMATION, "Input Too Large"

      ' Return to the original code so that it loops and
      ' asks the user to input the data again.
      GoTo AfterTypeError

   End If
End Sub
```

This example assumes that you must have valid input in a specific range from the user. You could probably place this code in a function so you wouldn't need to write it more than once. The example shows the technique in a straightforward manner to make it less confusing. Of course, the key to getting correct input is to provide a good prompt, like the one shown in Figure 20.1.

The code begins with a loop that basically says that it will continue asking for user input until the user supplies correct input or decides to give up. The user could do a

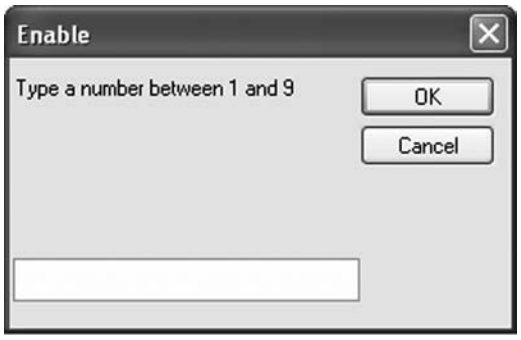


FIGURE 20.1 Good Prompts Will Usually Help You Obtain Good Input

FIGURE 20.2 Provide an Error Message That Tells the User About the Specific Error

number of things here:

- Not input anything at all
- Input something other than a number, such as a String
- Input a numeric value that's too large or too small
- Input a numeric value

If the user inputs nothing at all, the code will react as if the user has input a value of 0 and there's range code to handle that problem. Inputting something other than a number will create a type mismatch, which is where the error handler comes in to play. You must catch the error in such a way that the user has a chance to correct the problem. Likewise, the user could input a value that's way too large and causes an overflow (error 6). Of course, inputting a numeric value in the correct range is the expected action. The code is expecting an Integer. If the user inputs something like 1.1, the code simply truncates the value to an Integer, so even this contingency is handled.

The error handler code starts with the `HandleError:` label. It checks for a type mismatch error, `13`, and then takes corrective measures. First, it sets `Value` to 0 so that it doesn't fall within the required range and the user gets another chance to enter a correct value. Second, it displays an error message like the one shown in Figure 20.2. Finally, the code returns to the `AfterTypeError` label to resume processing. The processing for an overflow is the same, but notice that the code uses a different error message in this case so the user knows what action to take.

Range checking is simple in this case. All you need to do is determine whether `Value` fits within the specified range using a simple `If...Then`. When the user supplies incorrect input, the code displays a message box telling the user what is needed. Of course, you can make this message box as detailed as needed. The code offers an option for exiting so the user can go ask a supervisor for help or simply give up as shown in Figure 20.3.

This example ends by showing the typed value. Of course, a production application will do more.

Checking Data Length

When you work with numeric input, the user can only enter a value that's so large. If the number is too large, the user will see an overflow error (that you'll hopefully catch—see Listing 20.1 for details). Text input can be of any length and users can include all sorts

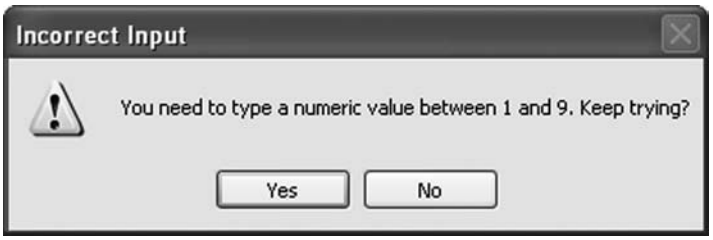


FIGURE 20.3 In Some Cases, You Want to Provide the User with a Way Out to Avoid Frustration

LISTING 20.2 Verifying the Data Length

```
Sub Main
   Dim Response As String

   ' Get the user input.
   Response = InputBox("Type your name:", "Name Input", _
      "John Smith")

   ' Check the length of the response.
   If Len(Response) >  40 Then
      MsgBox "The name you typed is too long."
   Else
      MsgBox "You typed: " & Response
   End If
End Sub
```

of things that don't belong. In fact, there are many security problems that are the result of unrestricted text length issues.

A user who can input text of any length will almost certainly do so—if for no other reason, than to see what happens. Listing 20.2 shows the simple technique you can use to check for data length. Using this approach makes it possible to limit what the user can input without restricting the actual characters. This is a simple example—Listing 20.1 shows the type of looping you could employ to give the user another chance to provide the correct input.

This example shows one other method you should employ to ensure good input. Notice that `InputBox()` includes a title and a default value. A default value is helpful because it provides the user with an example of what you want as shown in Figure 20.4. Examples are always a good feature to include when you can.

Methods for Checking Data Content

Plan your application as much as possible. If you ask the user for input, try to figure out a list of discrete values for the input. If you can come up with specific values, then you can check the input for those values. For example, when the user inputs a state name, you can check against a specific list of states.

FIGURE 20.4 Provide an Example of the Kind of Input You Want Whenever Possible

It's important to remember that you have an array of string manipulation functions. You can check for partial strings within a string when combinations of terms are allowed as part of the user input. The best way to perform this task is to rely on the `InStr()` function and check for an output value greater than 0. Using this approach, you can even perform partial string checking, such as looking for particular area codes within a telephone number. Partial string checking is helpful even when you can't check an entire string for content.

Don't assume that all content checking is of the white-listing variety, where you allow certain terms. You can also perform black-listing checks where you look for terms in a string that are disallowed. These negative checks can help you filter input where there are a lot of good values that a user can input, and fewer restricted values.

Sometimes you need to check for input variety. You may not want to let the user input the same value multiple times. In this case, the `StrComp()` function comes to the rescue. You can use this function to compare an input against all of the previous inputs to ensure the new value is unique.

Protecting Dialog Boxes

Dialog boxes present one of the greatest security dangers to your application because dialog boxes are your application's method of interacting with the user. (Of course, the user can also interact with your application through custom fields in a database.) Chapter 12 describes how to work with dialog boxes. It also describes all the controls that are available to you when creating a dialog box. A good dialog box is one that:

- Provides the user with great prompts.
- Uses appropriate controls.
- Includes good input examples.
- Presents few methods of providing incorrect input.

The most insecure control that you can use with a dialog box is the `EditBox` control. Every other control provides at least a modicum of control over user input. Obviously,

you always want to validate user input using the techniques described in the "Validating Data" section, but it's especially important to do so when working with the `EditBox` control because it allows freeform input.

Make sure you consider the requirements for read versus write access. Some users will need to change sensitive information, while others should only read it. In order to make information read-only, you'll need to use the `StaticText` control, instead of the `EditBox` control, which means creating separate versions of the dialog box for users with different needs. The extra work is well worth the benefits you'll receive by protecting the data from abuse.

User input isn't the only potential problem with dialog boxes. The prompts you provide can also cause problems. A prompt that provides too much information, such as sensitive information to personnel without the proper credentials, is just as bad as one that lets the user input the wrong kind of data. If your application offers access to sensitive information, place the information on a separate dialog box and offer it only to those who have the proper credentials.

Summary

This chapter isn't meant to provide a comprehensive look at security. If someone really means to cause damage to your system, they'll find a way to do it. What this chapter does provide is some ideas you can use to keep the less-skilled intruder at bay and reduce the potential for accidental errors that could result in data loss. This chapter also focuses on IDEA; you need to provide security for your machine as a whole and other applications as well in order to consider your system relatively safe.

Take time to review security. Of course, you'll want to secure any applications you create using IDEAScript, but also take time to review security on your system as a whole. The security of your IDEA data is only as strong and as safe as the weakest link in your system's security. Intruders don't care how they get into your system—they only care that they can.

Chapter 21 looks at another important topic for the IDEAScript developer, debugging. Looking for errors in your application is important. IDEA will capture a number of errors for you when it starts the application or sometimes it finds them as the application runs. However, there are many sorts of errors that IDEA won't find because these errors require human interpretation. For example, IDEA won't know that the output from an equation is incorrect—it simply knows that it provided the result that you requested.

CHAPTER 21

Debugging Your Application

Unless your application is extremely simple, it's likely that it'll have a bug or two in it at some point in time. In fact, the more complex the application, the greater the likelihood that it contains bugs. No matter how carefully you code your application, bugs are likely to slip in unnoticed (and definitely unwanted). As a consequence, you'll spend some amount of time debugging your application, which is removing the bugs.

Fortunately, IDEA provides you with tools to locate and squash bugs in your application. Bug removal isn't automatic, but it's possible with a little time and effort using these tools. The following sections describe the kinds of bugs you'll encounter and the techniques you can use to find them. Once found, fixing a bug is as simple as making a small change to your code.

Understanding the Kinds of Application Errors

Before you go any further, it's important to understand that all application writers have problems with bugs. It's not just the novice who experiences problems with bugs—everyone sees a bug now and then. It's true that experience does help to reduce the number of bugs, but bugs are simply part of writing an application.

Bugs can happen in a number of ways. Some bugs are relatively easy to fix, while others require considerable effort on your part to locate and fix. The following sections describe four kinds of bugs that you'll encounter when writing applications of any type.

Considering Syntax Errors

A syntax error occurs when the wording of your code is wrong. Syntax errors are relatively common and can sometimes be hard to find. In fact, there are three common sources of syntax errors:

- Misspelled words
- Improper use of reserved keywords
- Incorrect punctuation

An example of a misspelled word is a variable name that you've spelled in two different ways—`MyVariable` is different from `MyVaraible`. Using `Option Explicit` will save you a lot of hours of searching for this particular kind of error. If you don't declare the variable, IDEA will tell you about it. The variable you didn't declare may very well be a misspelled variable name. Of course, you can also misspell keywords, but IDEA will always find these errors when it tries to compile your application.

You may accidentally try to use a reserved keyword as a variable name. For example, if you want to create a variable for an input string, you might try to name it `InStr`. Of course, `InStr()` is the name of a function, so you can't use it as a variable name. IDEA provides two clues about improper use of keywords. First, if you see IntelliSense help when you type the variable name, you know that it's a keyword. Second, IDEA also displays keywords in a different color, which makes them stand out.

IDEA doesn't use a lot of punctuation, but it does use some. For example, if you typed `MsgBox "Hello World'` IDEA would register an error because you used the incorrect punctuation at the end of the line of code—you needed to use a double quote. Fortunately, IDEA tends to find punctuation errors very quickly. In addition, IDEA provides a color change to the text when you enclose it with the second double quote. If you don't see the color change, then there's a punctuation error to consider.

Considering Compile Errors

The computer doesn't speak your language. In fact, the computer speaks a very strange language of 1s and 0s. Most people sort of know that fact, but it's important when it comes to you telling the computer what to do. The computer can't speak your language and you can't speak its language. An intermediary called the *interpreter* takes the instructions that you write in IDEAScript and *compiles* them into a form that the computer can understand. Compiling is an act of translating what you say into what the computer can understand.

When the interpreter reads your code and sees something it doesn't understand, it displays an error message like the one shown in Figure 21.1. In this case, the code says, `MsgBox Hello World`. Notice that the double quotes are missing around `Hello World`, so the interpreter has no idea that `Hello World` is the string you want displayed on screen. As the dialog box in Figure 21.1 points out, the compile error is a syntax error—a mistake in the way you've issued the command to the computer.

FIGURE 21.1 The Interpreter Tells You When It Doesn't Understand Something You Write

Notice that Figure 21.1 includes a line number. The line number tells you precisely which line of the source code file contains the error. IDEA also places the cursor on the errant line so that you don't have to look for it.

You can also count on the interpreter to tell you when something is missing. For example, an `If...Then` statement requires a `Then` clause to work. If you were to leave out the `Then` clause, as in the following code:

```
If MyVar = True
   MsgBox "Hello"
End If
```

the compiler would tell you that there's a syntax error using the same dialog box as shown in Figure 21.1. Of course, the line number would reflect the line on which the `Then` clause is missing.

There's a wealth of other errors that the interpreter will find for you. For example, if you were to type `MsgBox()` as a single line of code, the interpreter would again display the dialog box shown in Figure 21.1. This time there are actually two errors. First, you don't use the parenthesis unless you plan to use the output from `MsgBox()`. Second, `MsgBox()` really should display a message of some sort. Consequently, to fix this error, you might type something like `MyVar = MsgBox("My Message")` instead.

The interpreter is very good at its job, but sometimes it misses errors. For example, let's say you write `MsgBox Hello` as the code to execute. In this case, the interpreter won't complain because it assumes that there's a variable somewhere named `Hello` and that it's empty. As a result, you'll see a blank dialog box that contains nothing at all. The way around this problem is to use `Option Explicit` in your code. When you use `Option Explicit`, you see the error message shown in Figure 21.2 because the interpreter can now say that it doesn't understand the instruction you provided.

Considering Run-Time Errors

The interpreter can find the vast majority of your syntax errors during the compile process. However, some errors aren't known until the program runs. For example, your application might rely on a file that doesn't actually exist on the hard drive. The interpreter can't find this error until it actually runs your application, making the error a run-time error.

FIGURE 21.2 Use Option Explicit to Make It Easier for the Interpreter to Help You Find Problems in Your Code

Many run-time errors are easy to find, even if they aren't necessarily easy to fix. If a file that you need is missing, then you either need to add it to the hard drive or find an alternative file. However, run-time errors fall into a special category that you can't necessarily fix completely. Someone besides you could remove the file and break your application. Therefore, you have to be proactive about potential run-time errors and use error trapping to handle them, in addition to making sure that your application is as bug-free as possible. The "Adding Error Trapping to Your Application" section in Chapter 6 tells you how to add error trapping to your application so that it can better handle run-time errors.

Considering Semantic Errors

The most difficult of all bugs to find is the semantic error, because IDEA can't help you locate it. A semantic error is one in which the code is perfectly acceptable, but the result isn't what you intended. You might also hear this kind of error called a logic error because it often reflects a problem in the logic you used to write the application.

A common semantic error is to use the wrong kind of structure to process data. You might have meant to use a `Do...While` structure, but used a `Do...Until` structure instead. The code will end up running the wrong number of times or may even end up running the loop an infinite number of times. The `For...Next` structure is especially prone to problems when the application writer asks it to run the wrong number of times. Logic errors often require that you step through and fix your code one line at a time, and that can be very time consuming.

Some semantic errors are extremely subtle. For example, if you write an equation as 1 + 2 * 3, the result is 7. However, you might really have wanted 9 as the output. In this case, you have to add parentheses to get the right result: (1 + 2) * 3.

Tip

Semantic errors can be so hard to find that application writers often ask other people to help them find the error. You've been looking at your code for hours on end, so you could possibly look right at an error and never see it. A fresh pair of eyes can often look at the code and see the error without any trouble. If you run into an especially difficult problem and you have another application writer you can talk to, ask them to look at your code as a kind of sanity check.

The best way to avoid semantic errors is to plan your application out carefully. Write down precisely what you want to do. Use a chart to show the flow of information in your application, just as you would use a chart to show how to perform any other task. Write out the steps used to perform the task. In short, think the application through to make sure you understand it completely before you begin writing code. Most professional developers follow this very process because it saves so much time and effort debugging the application later.

Running and Stopping the Application

Before you can begin debugging your application, you must know how to start and stop it. You've already started applications when working with the other examples in the book. All you need to do is press F5, click the Run Script button (the right-pointing blue triangle), or select Debug > Run.

Unless you set breakpoints (see the "Using Breakpoints" section for details), the application will run as it normally would. If the application has a dialog box in it, it will stop after it displays the dialog box and wait for your input. Consequently, you won't normally see the Stop Debugging option on the Debug menu or the Stop Debugging button enabled. Fortunately, you can still stop the application in most cases (when you're doing something in the scripting engine, versus running a task). Simply press Ctrl+Break to stop the application. The application will actually end; you won't be able to interact with it further until you restart it. However, using Ctrl+Break can help you stop an application that has ended up in a continuous loop.

At some point, you'll pause the application to examine it. When you have the application paused, you can stop it by clicking the Stop Debugging button, pressing Shift+F5, or selecting Debug > Stop Debugging.

This chapter relies on an example with some errors in it. In other words, don't rely on the output of the example until you fix it! Listing 21.1 shows the errant code.

The code in this example is a classic. It uses a technique called recursion to compute the factorial (`n!`) of the number input by the user. A factorial is the number multiplied

LISTING 21.1 The IAmBroken Example

```
Option Explicit

Sub Main
   ' Create a variable to hold the input.
   Dim Factor As Integer

   ' Get the number from the user.
   Factor = InputBox("Provide a number between 0 and 15")

   ' Obtain the factorial of the number.
   Dim Result As Integer
   Result = Factorial(Factor)

   ' Display the result on screen.
   MsgBox "The factorial (n!) of " & CStr(Factor) & " is " _
      & CStr(Result)
End Sub

Function Factorial(ByVal Value As Integer)
   ' Handle the special case of 0.
   If Value = 0 Then
      Factorial = 1
      Exit Function
   End If
```

(*continued*)

LISTING 21.1 (*Continued*)

```
   ' Perform a normal factorial process.
   If Value = 1 Then

      ' When the factorial reaches 1, we have
      ' computed the final value.
      Factorial = 0
      Exit Function
   Else
      ' Keep calling factorial with lower values.
      Value = Value * Factorial(Value - 1)
      Factorial = Value
      Exit Function
   End If
End Function
```

by all of the values leading up to that number. For example, 3! is 1 * 2 * 3 or 6. Likewise, 5! is 1 * 2 * 3 * 4 * 5 or 120.

Using Breakpoints

Breakpoints are special indicators in the IDEAScript Editor window that tell IDEA to pause the application at a certain point. After the application pauses, you can examine it for potential problems. A breakpoint always appears on an executable line of code, not a comment or the beginning or ending of a subroutine or function. You can easily spot breakpoints by the red circle at the left side of the window, as shown in Figure 21.3.

Figure 21.3 shows three breakpoints. You use these breakpoints later in this chapter to work with the application. The following sections describe how to set and remove breakpoints as needed so that you can start and pause the application as needed.

Setting Breakpoints

Before the application will pause at a particular point, you must set a breakpoint. To perform this task, place the insertion point (the text cursor) on the line of code you want to examine, and then click the Toggle Breakpoint button, press F9, or select Debug > Set/Clear Breakpoint. You should see the red circle appear in the left margin, as shown in Figure 21.3.

Removing Individual Breakpoints

In some cases, you may place a breakpoint in the wrong place or simply not need it anymore. You can remove an individual breakpoint by placing the insertion point at the line that has the breakpoint in the left margin (the red circle shown in Figure 21.3), and then click the Toggle Breakpoint button, press F9, or select Debug > Set/Clear Breakpoint. You should see the red circle disappear in the left margin.

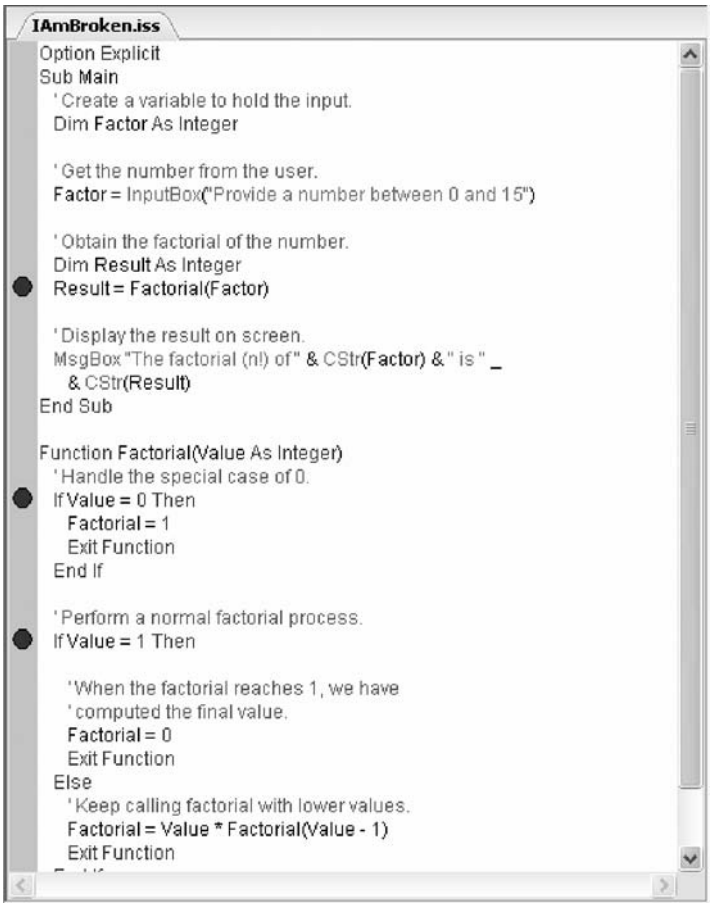


FIGURE 21.3 Breakpoints Tell IDEA When to Pause Your Application So That You Can Examine It

Removing All Breakpoints

When you finish debugging the application, you'll probably want to see it run at full speed, which means removing all of the breakpoints. To remove all the breakpoints in the application, click the Remove All Breakpoints button or select Debug > Remove All Breakpoints.

Stepping through the Application

Pausing the application means that it has stopped executing statements. The application is still active and can still perform work, but you must tell IDEA to perform the next step. IDEA provides two buttons on the toolbar for this task: Step Into and Step Over. Both of these buttons tell IDEA to perform the next step of the instructions, but they do it in

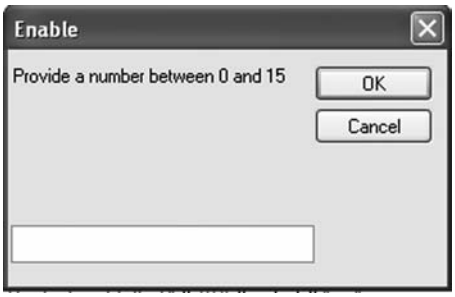


FIGURE 21.4 The Example Prompts You to Type a Value between 0 and 15

different ways. The following sections tell you about the two buttons and explain the differences between them.

Stepping into a Call

When you click the Step Into button, press F8, or select Debug > Step Into, you tell IDEA to perform the next step of the instructions. If the next step is a call to a subroutine or a function found in your code, then IDEA steps into that subroutine or function so that you can see it at work. Otherwise, if the code doesn't call a subroutine or function or that subroutine or function is external to the code, then IDEA simply performs the required task and moves onto the next line of code.

Let's say that you've stepped through a subroutine or function called by another subroutine or function and you're at the `End Sub` or `End Function` line of code. The next click of either Step Into or Step Over will take you out of the current subroutine or function and back into the subroutine or function that made the call. You aren't stuck in the current subroutine or function forever. Of course, the topmost subroutine in most cases is `Main()`. When you reach `End Sub` in this case, the application ends.

It's important to understand how Step Into works. The following steps take you through a partial scenario with the example application shown in Listing 21.1. Make sure you set just the first breakpoint for the example, as shown in Figure 21.3, using the techniques described in the "Using Breakpoints" section.

1. Click the **Run Script** button. You see the prompt dialog box shown in Figure 21.4.
2. Type **0** and click **OK**. At this point, IDEA pauses the application at the first breakpoint, as shown in Figure 21.5. You can tell that the application is paused at the breakpoint because you can see the yellow arrow in the red circle. The yellow arrow (the instruction pointer) shows the current instruction—the one that IDEA will execute next.
3. Click the **Step Into** button. Notice that the instruction pointer is now at the first line of the `Factorial()` function, as shown in Figure 21.6. As you can see, IDEA stepped into the `Factorial()` function. If you click the **Step Into** button again, the instruction pointer will move to the `Factorial = 1` line of code. Clicking the **Step**

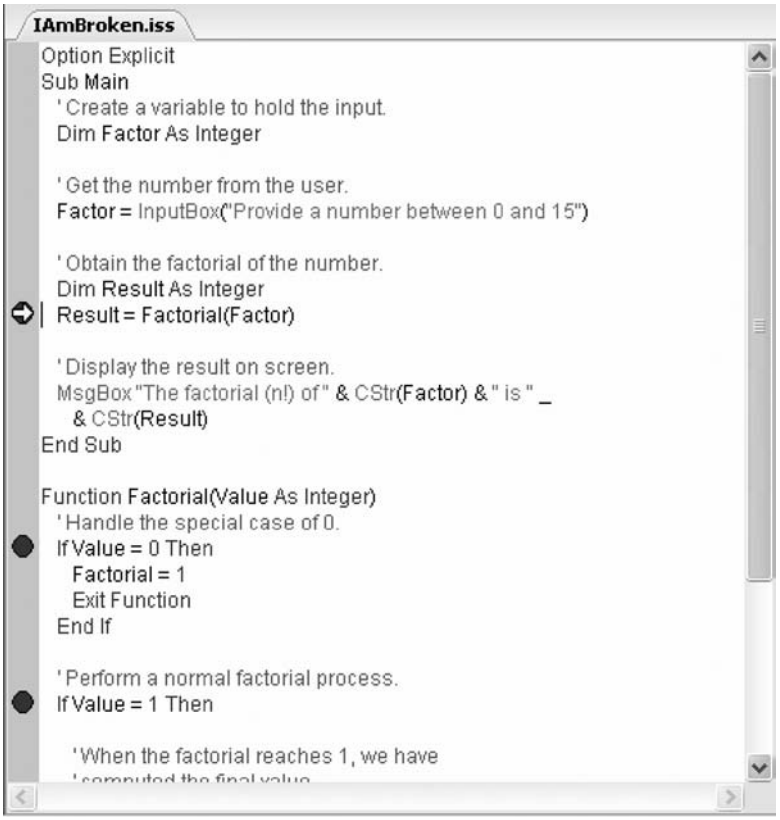


FIGURE 21.5 IDEA Stops the Example at the First Breakpoint

Into button again will move to the next line of code and so on until the application is finished.

4. Click the **Stop Debugging** button. Notice that IDEA stops precisely where it is in the application and ends application execution.

Stepping over a Call

Sometimes, you don't want to see all of the details of application execution, even if IDEA can access the code for a subroutine or function. In this case, you can click the Step Over button, press Shift+F8, or select Debug > Step Over to execute the instruction without stepping into the code (see the "Stepping into a Call" section for details). Use the following steps to see how Step Over differs from Step Into.

1. Click the **Run Script** button. You see the prompt dialog box shown in Figure 21.4.
2. Type **0** and click **OK**. At this point, IDEA pauses the application at the first breakpoint, as shown in Figure 21.5.

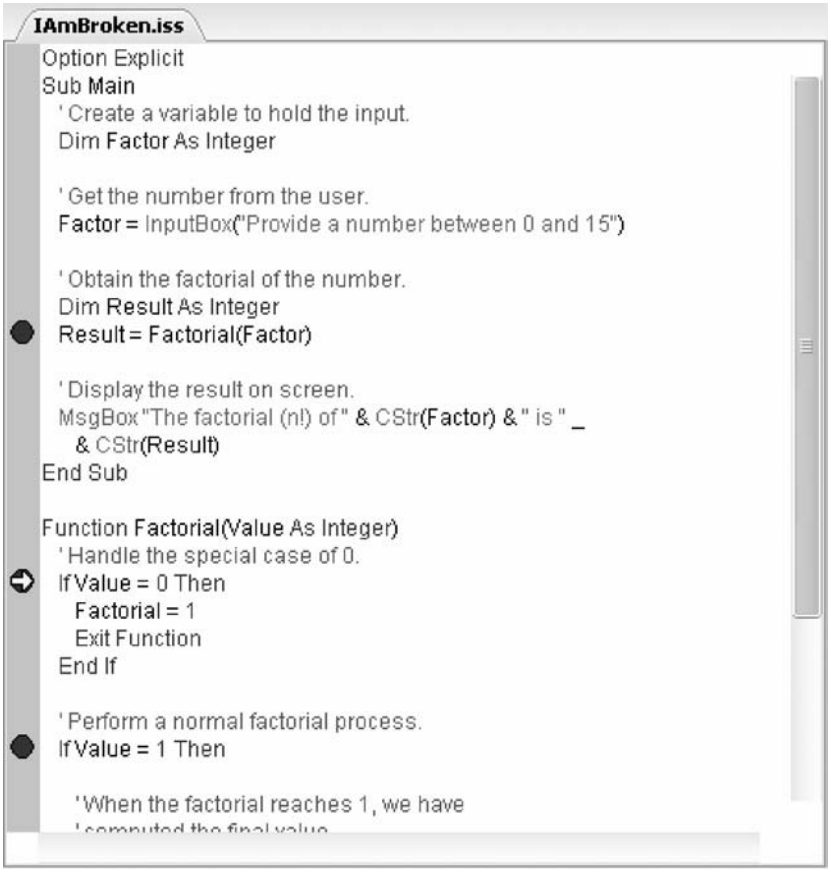


FIGURE 21.6 This Figure Shows the Results of Clicking Step Into

3. Click the **Step Over** button. Notice that IDEA doesn't move the instruction pointer into the `Factorial()` function, even though the `Factorial()` function code exists in the file. Instead, IDEA moves the instruction pointer directly to the next line of code in `Main()`, as shown in Figure 21.7.
4. Click the **Stop Debugging** button. As before, IDEA immediately stops application execution.

Using the Watch Window

The Watch window tells you about the content of variables as the application executes. You need to know what the variables contain so that you can detect problems in your code. If a variable contains a value that you didn't expect, then you can figure out where the code went wrong. Stepping through code to see where the code goes and tracking the values in variables are the two ways most developers use to discover how their code

FIGURE 21.7 This Figure Shows the Results of Clicking Step Over

actually works, rather than the way they thought it would work. The following sections tell you how to use the Watch window to work with variables.

Understanding the Problem

The example program is broken. To see the first problem that the Watch window will help you fix, run the application without any breakpoints in place. The first time you run the application, type **0** in the prompt dialog box shown in Figure 21.4. Figure 21.8 shows the result, a value of 1, which is correct.

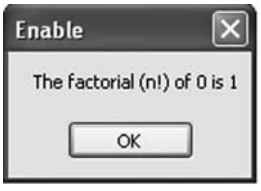


FIGURE 21.8 The Result of 0! Is Correct

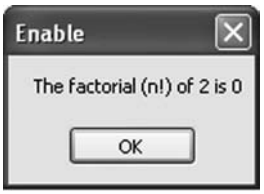


FIGURE 21.9 The Result of any Other Input Is Incorrect

Now, try a different number between 1 and 15 in the prompt dialog box. Any value you type results in an output of `0`, as shown in Figure 21.9. Clearly something is wrong with the application.

Creating a Quick Watch

A Quick Watch is a simple way of seeing the current value of a variable. The "Understanding the Problem" section describes a problem with the example code. If you look at the code in Listing 21.1, you might suspect that `Value` in the `Factorial()` function is the problem, so let's use a Quick Watch of `Value` and see what happens. The following steps show you how to find and fix the first error in the example application. (In order to follow this section, make sure you set all of the breakpoints shown in Figure 21.3 using the techniques described in the "Using Breakpoints" section.)

1. Click the **Run Script** button. You see the prompt dialog box shown in Figure 21.4.
2. Type **2** and click **OK**. At this point, IDEA pauses the application at the first breakpoint, as shown in Figure 21.5.
3. Click the **Step Into** button. The instruction pointer moves to the first instruction in the `Factorial()` function.
4. Double-click on any occurrence of `Value` in the `Factorial()` function to highlight it. Click the **Quick Watch** button, press **Ctrl+W**, or select **Debug** > **Quick Watch**. You see the current contents of `Value`, as shown in Figure 21.10. At this point, `Value` contains 2 as you might expect.

FIGURE 21.10 Value Contains the Correct Information Initially

FIGURE 21.11 The Section Call to Factorial() Also Shows the Correct Value

5. Click the **Close** button to close the **Quick Watch** dialog box. Click the **Step Into** button. Notice that the instruction pointer moves past the initial `If...Then` statement and onto the second `If...Then` statement because the conditions for the first `If...Then` statement weren't satisfied.
6. Click the **Step Into** button. Notice that the code moves directly to the first statement after the `Else` clause. This is the first executable statement that passes the requirements of the`If...Then...Else` statement.
7. Click the **Step Into** button. Notice that you're now at the top of the `Factorial()` function again. This is the result of recursion—a technique that makes it possible to write complex code in a very small amount of space. Let's check the content of `Value`.
8. Double-click `Value`, and then click the **Quick Watch** button. Figure 21.11 shows that `Value` now contains `1`, which is the correct value because the previous iteration of `Factorial()` called itself with `Value – 1`.
9. Click **Close** to close the **Quick Watch** dialog box. Click the **Step Into** button twice. Because `Value` is `1` this time, the first part of the second `If...Then` statement is true.
10. Click the **Step Into** button twice. Let's look at the content of `Value` now. You should be at the `Factorial = Value` line of code.
11. Double-click `Value`, and then click the **Quick Watch** button. Figure 21.12 shows that `Value` is now incorrect as shown in Figure 21.12. After the call to

FIGURE 21.12 Suddenly, Value Contains the Wrong Information

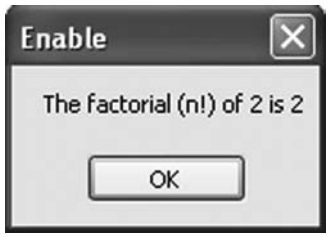


FIGURE 21.13 The Change at Least Partially Fixes the Application

FIGURE 21.14 Larger Values Still Cause Problems

`Factorial()`, `Value` should contain a value of `2`, not `0`. Something happened in that second call to `Factorial()`!

12. Click **Close** to close the **Quick Watch** dialog box. Click **Stop**. Let's look at that code. The line, `Factorial = 0` is causing the problem. OK, let's change it to `Factorial = 1`. Now, run the example again and use a value of 2. Suddenly, the example begins to work, as shown in Figure 21.13.

The problems with the example application aren't over with this simple change. Run the application again and use a value of `10`. Figure 21.14 shows that the output is incorrect. Using a value of `10` as input results in an overflow. In fact, you can only calculate values up to `7` without error, so something else is wrong.

Typing a Variable into the Watch Window

Using a Quick Watch is fine if you only want to monitor the content of a variable every once in a while. However, it quickly becomes monotonous when you need to monitor a variable over a longer term. In this case, using the Watch window itself is the best way to accomplish the task. To add a variable to the Watch window, you can type its name directly into the window or drag and drop the variable into the window. You can also use the **Copy** button in the Quick Watch dialog box to add the selected variable to the Watch Window. The following steps describe how to use the Watch window for debugging:

1. Set breakpoints at the following lines of code:

```
Result = Factorial(Factor)
If Value = 1 Then
Value = Value * Factorial(Value - 1)
```

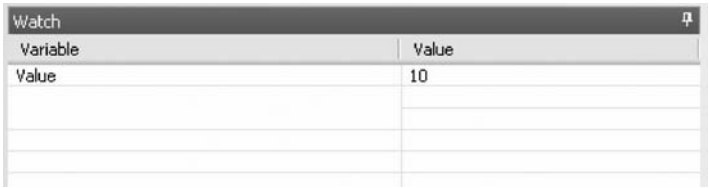


FIGURE 21.15 The Watch Window Provides Continuous Variable Monitoring

2. Click the **Run Script** button. You see the prompt dialog box shown in Figure 21.4.
3. Type **10** and click **OK**. At this point, IDEA pauses the application at the first breakpoint, as shown in Figure 21.5.
4. Click the **Run Script** button. Notice that IDEA moves the instruction pointer directly to the second breakpoint, the one at the `If Value = 1 Then` line of code.
5. Type **Value** into the **Watch** window. You'll see the current content of `Value` as shown in Figure 21.15. As you can see, the content is correct.
6. Click the **Run Script** button. The instruction pointer stops at the third breakpoint and shows that `Value` still contains `10`—no surprises here.
7. Keep clicking the **Run Script** button. You'll see `Value` count down as it recurses through the code. Eventually, `Value` will contain `1`.
8. Click the **Step Into** button and start following the code execution. You'll see how the code begins multiplying the values to calculate the factorial. Remember, the factorial is 1 * 2 * 3. . .
9. Keep clicking the **Step Into** button and monitoring the Watch window. Eventually, you'll see something that should tell you why `10!` isn't working, as shown in Figure 12.16. Notice that the content of `Value` is getting close to the 32,768 limit of an Integer. The data type used for the example is too small—we need a Long!
10. Change the data type of `Factor`, `Result`, and `Value` to a Long. You'll find now that you can successfully calculate values between `0!` and `12!`.

Unfortunately, there's still one more error in the application. Notice that the original prompt tells the user to enter a value between `0` and `15`. It turns out that a Long can only hold values up to `12!` The result of `13!` is 6,227,020,800, which is greater than the 2,147,483,648 storage capacity of a Long. Consequently, the final fix for this application is to change the prompt to accurately reflect the capabilities of the application. It's essential

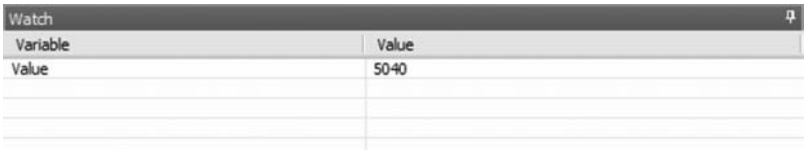


FIGURE 21.16 Careful Checks of Your Code Will Show When a Data Type Is Too Small

to remember that bugs can just as easily exist in the user interface as they can in your code.

Relying on Message Boxes

There are times when you'll need to rely on an old standby that application writers of all types have used for many years, the message box. Of course, at one time the message box was actually something else, but the idea is that you output some text that tells you what your application is doing. Using message boxes makes it possible to quickly see what's going wrong with your application without a lot of digging.

Note

You want to make sure you remove the diagnostic message boxes before you release the application into a production environment. Nothing is more embarrassing than a message box saying, "Remember to fix this error!" popping up when a user experiences a problem.

You can use message boxes in several different ways. The first is to display the value of variables. You might need to know the actual value of one or more variables when the application is running at full speed. Of course, you aren't limited to displaying the variable content in its raw form—you can always manipulate the variable using any of the functions that IDEA provides or functions that you've created yourself.

The second way to use message boxes is to see where you're at in the application. You might display a message box that has the name of the function that the application has visited. Application writers often include the line number of the message box or a bit of code to remind them of where the message box is within the code. Sometimes, the application ends up in a place you didn't expect because of a logic error, and using message boxes is a very good way to locate this kind of error.

To use a message box, you just place it in your code as you would any other dialog box. Diagnostic message boxes are seldom fancy—they contain just the message and rely on the OK button. You might simply write `MsgBox MyVar` to display the current value of `MyVar`. The precise message content is up to you, but it should always reflect the diagnostic needs of your application.

Of course, you might wonder why you need to use message boxes at all when you have all of the other techniques in this chapter at your disposal. The problem is that single stepping through an application doesn't always reveal the problem—sometimes you need to run through the application at full speed to see a problem. In some cases, you might also need to create special test cases to fully understand a problem. The debugger might not provide everything you need. Normally, you want to start the debugging process using the debugger, but always remember that you have the message box alternative when you need it.

Summary

This chapter has described what bugs are and demonstrated debugging techniques. Debugging is one of those tasks that every application writer faces at some point and everyone seems to universally hate. It would be nice if applications could fix themselves, but that just won't happen anytime soon (if ever). Getting rid of bugs requires human insights and intuition.

Now that you've had some experience with bugs, take some time to practice a bit. Use the sample application from this chapter to locate bugs and fix them. After you fix the bugs, introduce new bugs into the example to see how they look when you run the application. If you know of someone else who is working with IDEAScript, ask them to introduce a bug or two into the example application so that you can find them. The only way to build skills as a bug hunter is to practice.

Chapter 22 takes a look at project management tasks. As you begin building more applications, you'll want to manage them to make it easier to upgrade them as needed. In some cases, you may find that you're working with a group, so it becomes especially important to manage the application as a project. Most failed application building attempts are due to poor or non-existent project management, so this chapter is especially important for larger tasks.

CHAPTER 22

Performing Project Management Tasks

For many people who write simple applications using IDEAScript, the idea of it becoming a project is so far-fetched that they'd never even consider it. It's a fact that many IDEAScript applications won't start as projects. In fact, they'll start off as something simple that you write for your own use. You won't even consider use by anyone else as a priority and may even be tempted to hide your creation away. The problem with applications is that other people do eventually see them. If you've come up with something interesting, these gawkers will soon become users of your application. Word of mouth will eventually get the application into the hands of others and suddenly, you have a project. It happens all the time. In fact, many professional developers started by writing applications they planned to use only for their own needs. After a few applications, these hobbyists became full-time developers.

You may never become a full-time developer and your application may only warrant a few positive comments. However, it's important to always consider that your application might become popular and write it accordingly. A few extra minutes of planning can save you a lot of headaches tomorrow.

There are other reasons to work with your application as if it were a project. The most important personal reason is that people tend to forget. The code you fully understand today might look foreign tomorrow. Documenting your project is the only way you can ensure that your hard work today won't be wasted tomorrow when you need to make changes.

Creating a plan of some sort also makes it easier to write the application. Just as it takes time to write clear instructions for another human, writing instructions for the computer requires clear thinking. Jotting down some notes and writing a step-by-step procedure can save you a lot of time and effort. More importantly, it helps you think through a task that might be so ingrained that you really don't know how you do it.

This chapter won't make you into a full-time developer or a software designer. This is a simple chapter on some techniques you can use to plan, document, and manage your application. Using the techniques in this chapter will make your life easier by reducing the work you need to perform and enhancing the application you do create.

Creating a Plan for Your Application

Every plan begins with an idea. You see something you want to automate and you begin thinking about just what that would mean. It's at this point that you should start writing things down because many great ideas end up getting lost. Write what you want to do and why you think it's a good idea. Most importantly, come up with goals for your idea—what you plan to accomplish. These three elements are the starting point of most plans. The following sections provide more information about creating an application plan.

Defining the Basics

Once you have your idea down on paper, start thinking about what the idea requires to be completed. For example, what resources do you need in the form of databases and other data? Write down everything you can about what the idea needs to work. Don't be afraid to continue updating your budding plan as you think things through. Write everything down so you don't forget things later.

An important part of your plan is a procedure. Write down every step you use to perform the task. Don't try to write the procedure from memory. Perform each step and then write it down immediately. Most people abbreviate tasks in their minds as they become familiar with them. Eventually, they leave out entire steps because they've performed the task so many times that they perform these steps without even thinking.

Once you have a written procedure, try to perform the task by just using the steps you've written down. No cheating! If you find yourself thinking that you should be doing something in addition to the step, write it down too. Keep performing the task until your written procedure is precise enough that anyone could perform the task using just the procedure.

Developing the Application Shell

You can begin creating your application at this point by performing the procedure you've just written down using the Macro Recorder. When you get done, run the resulting macro. If it produces the rough results you wanted, then you have a great start on your application. Now you can add all those bells and whistles you wanted into the application.

Adding Bells and Whistles

The basic application shell is a usable application. It performs the procedure that you normally use for the task. However, automation often requires more than the basics—these additions are commonly called bells and whistles. For example, your task may require that you handle some variables, which means adding optional processing to the application. You need to think about how this optional processing will take place. IDEAScript offers two major methods for adding optional processing to your application.

- **Automatic Processing:** You can add code to your application to perform optional processing based on the environment, files, or other criteria. For example, if you only process certain files on Tuesdays, then you can add code to detect the day of the week and automatically perform the optional processing when it's a Tuesday.
- **User-Defined Processing:** You can create dialog boxes (also called forms) for your application that ask the user for direct input. Although some forms provide a means of data input, you can also use them to control application processing. In fact, there's nothing to stop you from creating a form with a menu of tasks to perform.

It's important to add bells and whistles one at a time, then test your application. If you try to add too much at once, you'll find that the result is hard to debug and you might never get it working properly. In addition, by adding a single new element at a time, you can decide when your application has reached a useful conclusion, rather than becoming bloated and unwieldy.

Of course, you can add much more than basic processing to your application. For example, you could add features for outputting reports, charts, and graphs as needed. It's possible to add functionality to send data to other applications. You've seen all of these sorts of additional processing presented throughout the book. It's time to decide which of these items make sense for your particular application.

Keeping Track of Application Files

Given the purpose of IDEA, your IDEAScript application is going to have a number of files associated with it. One of the major causes of problems for your application is going to be files that are missing, corrupted, the wrong version, or simply in the wrong place. Make a list of every file used by your application. Your list should include the following file types:

- Application
- Database
- Configuration
- External application data
- Other data (such as XML)
- Report
- Graphics

It's not enough just to list the files, however. You also need to consider the location and uses of those files. In fact, you should create a table that has six headings: File Name, Type, Location, Use, Version, and Description. Complete the table entries for each of the files so that you know everything about them.

The location of files is especially important. The location depends on the environment in which you work and the use of the files. For example, you might use databases found in a shared folder when working in a group setting, but on your own drive when working alone. Configuration files might be found in two places. A configuration file

containing personal settings will likely appear in your `My Documents` (or simply `Documents`) folder, while a configuration file containing overall application settings might appear in a shared folder. Your application could also rely on data produced by other applications. Such data might even appear somewhere on an intranet. In short, you need to consider precisely where files appear as part of creating your application so that you don't burden the user with trying to find the files.

One of the issues that many application writers fail to consider is file versioning. You might initially write an application that includes few configuration settings. As the application becomes more complex, the configuration file might become more complicated as well, which means that the old version of the configuration file won't contain sufficient information to run the application. It's a good idea to create a methodology for determining the file version as part of the application, so that your application can automatically reject files that are the wrong version.

Working within a Group

In some cases, you'll work within a group environment when using IDEA. Groups make it easier for people to coordinate their efforts. It also makes it possible for someone to take a vacation without a loss of support for a particular data analysis scenario. In fact, there are many reasons for working in a group—the particular reason for your organization to work in a group is less important for the purposes of this book than the fact you're working in a group.

From a project management perspective, you must consider two aspects of the group. The first is that everyone is sharing a common application to perform analysis tasks using IDEA. In this case, you must consider the dynamics of shared file use within the group when creating your application. It's important to keep group data separate from personal data. For example, everyone is likely to share databases, but no one's likely to need access to your personal configuration settings.

The separation of group from personal is essential. Of course, the term group can encompass several levels. For example, your application might need to consider organizational and workgroup requirements, where organizational needs might entail general requirements for the organization as a whole. An organization might require that all workgroups within it use standardized reports, so your application might need access to this organizational-level data obtained from a shared organizational-level folder.

The second aspect is application development. Some organizations use teams of application developers to good effect. Relying on a team development process reduces application development time. In addition, it safeguards the application from knowledge drain. If one person leaves the organization, someone else knows how the application works. It's important to coordinate your efforts with other application writers when working in a group development environment. This book doesn't include full information about group development dynamics, but you can find a number of good books on the topic. For example, *C# Design and Development* by John Mueller provides a full discussion of techniques that you can use for team development in the first few chapters.

Even though later chapters of this book are C#-specific, the first half of the book provides a good guide to working in a team environment for anyone (and any language).

Documenting Your Application

There's absolutely nothing worse than trying to modify software you don't understand. Without proper documentation, software becomes a maze of language elements that most professionals spend months trying to analyze and sometimes never fully understand. In fact, sometimes even the experts give up—it's actually easier to rewrite the software from scratch, rather than try to understand it. Software without documentation becomes an albatross around the neck of anyone designated to maintain it. Even if you write the software and understand it fully today, there's a very good chance that you won't remember how things work tomorrow when you need to make modifications. In short, the most important task in managing your application project is the documentation.

Documentation comes in a number of forms. If you started at the beginning of this chapter and have worked your way through the discussion to this point, you already have some documentation for your application. For one thing, the plan you create is part of the application documentation because it tells everyone what you wanted to accomplish with the application and the precise process that the application is supposed to follow. Likewise, the list of application files documents the resources that the application requires to work properly. These sources of information are irreplaceable components of the application documentation.

However, even a simple application requires a little more documentation than what you've put together so far. In fact, there are three pieces of documentation that every application should include:

- Application comments
- Code documentation
- A user guide

With these three pieces of additional documentation, anyone can return to the application at some point and make essential changes and updates to it. The following sections describe each of these forms of documentation further.

Considering the Importance of Comments

You've seen a wealth of examples in this book. It's important to note that all of the examples include comments, lots of them. The comments are superfluous—they tell you what the code is expected to do. In some cases, there are actually more lines of comments than there are lines of code. Comments provide line-by-line documentation of the code. Whenever you write code, you must include comments with it or the code is worthless. Tell yourself (and everyone else) these facts:

- Precisely what the code is supposed to do
- Why you chose to perform the task in a particular way

- Use of variables within the code
- External modules required to work with the code
- Alternatives you considered using and why you didn't use them
- Bugs you fixed in the code and what caused them

Not all of these documentation elements appear with every line of code. You only need to provide those elements that make sense in a particular situation. The point is to document the code fully so that you don't have to guess about how it works later on. Otherwise, you could find your weekends ruined as you try to figure out why a piece of code isn't working or how to incorporate essential changes.

The examples in the book don't include some other forms of comments that you'll commonly find in production code. You may also want to include these types of information as part of your comments:

- Purpose of the application as a whole
- Purpose of the individual subroutines and functions
- Changes (version information) for the application
- Changes for individual subroutines and functions
- Inputs and outputs for each subroutine and function
- Important dates for application development
- Contact information
- Copyright information
- Any other pertinent information

Providing Written Documentation

Large applications have pages, sometimes entire books, of written documentation. You don't have to go to quite such lengths in providing written documentation for your application. In fact, you've already developed some of the written documentation in other sections of this chapter. Here are items that you normally want to include as part of the written documentation for your application:

- Application plan
- Application file listing
- Group planning information
- Screenshots of forms in a storybook format that explains typical application flows
- Subroutine and function descriptions
- Amplification of code comments as necessary
- Contact information for each of the application writers
- Plans for future application updates
- Current bug listing
- Bug fix information
- Update and version information

The important thing is to make the written documentation as complete as possible. Everything you forget to write down now is something you'll probably forget about later. Unfortunately, it's those forgotten bits of information that often prove the most useful later in fixing bugs. The little bits of knowledge that don't appear in the written documentation often cause work for those who follow you and sometimes ensnare you as well, because you'll remember that there was some detail you needed to know, but you won't remember what it is.

Creating a User Guide

No one should have to guess how to use your application, including you. Some people make a living at looking at something and figuring out how it works. Unfortunately, those people are a rare commodity—most of us need a little more help. Your application should include user documentation that explains how to use the application in detail. This information can also help developers who are trying to discover why things work the way they do in your application. Good user documentation includes the following elements:

- An introduction that provides an explanation of the application's purpose, who the application is written for, how the user documentation is organized, what the user needs to use the application, and conventions (styles, note types, and other presentation elements) used throughout the user documentation
- An introductory chapter that describes the basic user interface
- Chapters that discuss how to perform each task the application can perform
- Screenshots of each dialog box and a complete discussion of what task the dialog box accomplishes (use callouts as necessary to make the form elements clear)
- Descriptions of each report, graph, or chart the application can produce, along with samples of the output
- Tips, notes, and warnings about application usage
- Information about common errors that the user might make
- A listing of errors that the application might display, as well as a description of how to avoid the errors
- Resources that the user needs when using the application
- Contact information for application support
- Guidelines on where to find additional helpful application information

Summary

This chapter is about simple project management. You won't write the next major Windows application using these techniques, but you can write a great application using IDEAScript. The emphasis of this chapter is on ease of development and reduced time maintaining your application. If you follow these techniques, you'll find that your time with IDEAScript is significantly more enjoyable, and everyone wants to enjoy their work.

It's time to put some of what you've discovered in this chapter to work. By the time you reach this chapter, you've probably envisioned a few things you want to do with IDEAScript. Start by jotting your ideas down. Spend some time writing down the steps you actually follow to perform a task. Refine your notes by using your procedure to perform the task. Once you have a plan in place, start developing your application and fully document it as you do. Enjoy your IDEAScript experience!

Chapter 23 is the final chapter of the book. You know that IDEA supports Visual Script as an alternative to writing code using IDEAScript. Although Visual Script is a great idea, you might find that it's a bit limited and you want to move to IDEAScript. Fortunately, you can take all of the work you've already invested in Visual Script with you. Chapter 23 shows you how to convert your Visual Script macro to an IDEAScript macro.

CHAPTER 23

Converting Visual Script to IDEAScript

Visual Script provides a convenient method for creating simple macros that you don't plan to customize in any way. Even though the result is quite simple, it does work. For some people, Visual Script is all they'll ever need because they aren't looking for anything complicated. However, you might start with Visual Script and find that it doesn't quite provide everything you need. When this happens, you can always convert your Visual Script macro to an IDEAScript macro, then add the customizations you require using the techniques described in this book. The following sections tell you how to convert a Visual Script macro to an IDEAScript macro so that you can customize the result as needed.

Considering the Benefits of Using IDEAScript

Visual Script is a good way to start scripting, but if you're like most people, you'll eventually want more. IDEAScript can perform just about any task that you can imagine and probably a bit more. There are some significant benefits to moving from Visual Script to IDEAScript:

- **Flexibility:** IDEAScript lets you control precisely how the task is performed.
- **Forms:** Obtaining information from the user is important for many tasks. Using forms makes it possible to do more with the application.
- **External Application Access:** In many cases, you must interact with other applications to complete a task. IDEAScript makes it significantly easier to work with other applications.
- **Data Manipulation:** Sometimes data doesn't come in the form you need. The data manipulation features provided by IDEAScript make it possible to change the form of the data into something you can use.
- **Repetitive Processing:** You may need to perform a task more than once, but may not know precisely how many times. The looping features of IDEAScript make it easy to describe how many times to perform a task based on environmental or other conditions.

- **Modification:** It's very easy to modify IDEAScript applications so that they perform tasks precisely the way you want them performed.
- **Custom Data Formats:** Data doesn't always come in a neat package. You might need to perform custom data conversion in order to read the data into (and out of) IDEA.
- **Code Safety and Reliability:** Examples throughout this book have shown how to add safety features to your code that makes it more bulletproof. For example, you've seen code that handles situations where someone has removed a file you need.
- **Security:** If your application requires security features, then using IDEAScript is the only way to go. IDEAScript makes it possible to address all of the security concerns described in Chapter 20.

Performing the Conversion

The actual conversion process is quite easy. Let's say you begin with the Visual Script macro shown in Figure 23.1.

This very simple macro opens the `Sample-Customers` database, performs an Indexed Extraction that creates a database with all of the companies in Argentina,

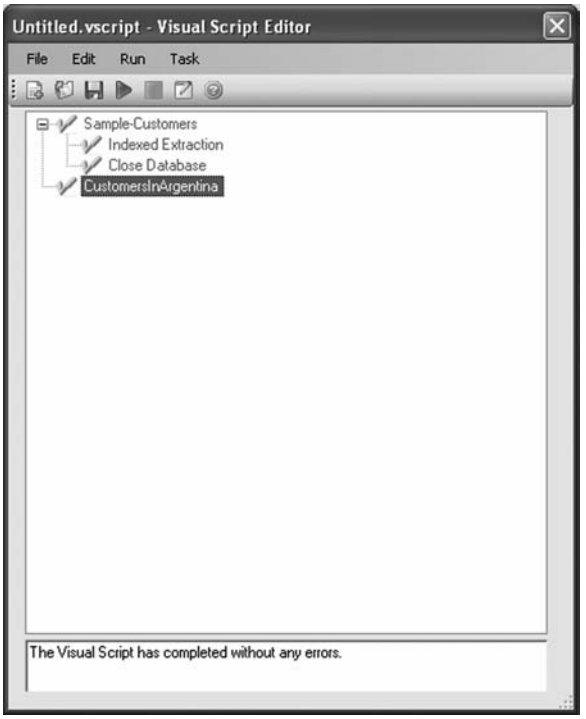


FIGURE 23.1 A Simple Visual Script Macro to Use for Demonstration Purposes

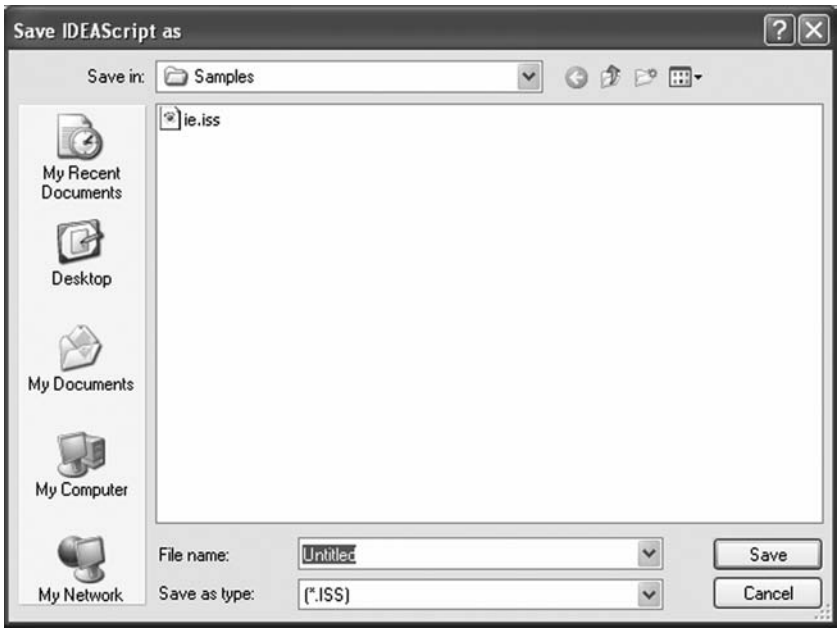


FIGURE 23.2 Tell IDEA Where You Want to Store the New IDEAScript Macro

and then closes the `Sample-Customers` database. Finally, the macro opens the `CustomersInArgentina` database that contains the results so that you can see them.

If you try running this macro, you'll see that it performs precisely as predicted. When the `CustomersInArgentina` database already exists, you'll need to tell IDEA to overwrite it. In short, the macro behaves precisely as if you were performing the task manually, which means you have to monitor the macro as it runs.

Let's say that this is a long-running macro and you want to add code that automatically deletes existing databases as needed. In order to add this code, you must first convert the macro into an IDEAScript macro. These steps show you how to take this first step.

1. Open the Visual Script macro (if it isn't already open).
2. Run the Visual Script macro to ensure it works as anticipated.
3. Select **File** > **Convert to IDEAScript**. You see the **Save IDEAScript as** dialog box shown in Figure 23.2.
4. Choose a location to store the IDEAScript macro. Type a name for the IDEAScript macro in the **File name** field. Click **Save**. IDEA saves the new IDEAScript macro in the location you specify.
5. Close the **Visual Script Editor**.

That's it! As you can see, the conversion process is very simple.

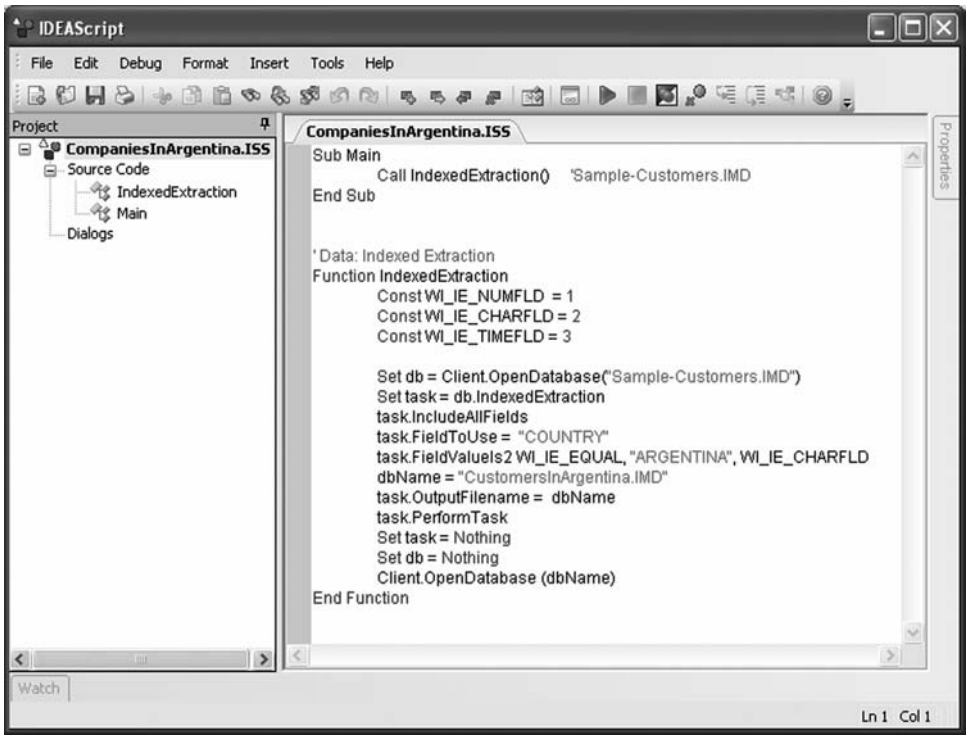


FIGURE 23.3 The Resulting IDEAScript Macro Is Very Simple

Making Changes and Saving the Result

At this point, you have a rough IDEAScript macro that you can modify as you see fit. The IDEAScript macro that you see generated from the Visual Script macro is very much like the one you'd get from the Macro Recorder. Figure 23.3 shows the initial results of the conversion process.

Make sure the organization of the code makes sense. If you have just one function in the resulting macro, move the code from the macro into `Main()`. Repeated functions may simply require that you add arguments to a single function and then use those arguments to modify the function's output.

After you organize the code, you'll want to document it. Look at what the code is doing and add comments that describe it. Use the examples throughout the book as a starting point for this process, but add comments that make sense to you. Take your time. This part of the process is very important because otherwise, you'll spend a lot of time later trying to figure out what the macro was supposed to do. Before you do anything else, test the macro to make sure it still works as anticipated. Small changes can sometimes produce the worst bugs, and you want to catch any bugs as early as possible.

Now you can start augmenting the code. For example, you could add the `OpenDB()` function and `DeleteOld()` subroutine used in other chapters of this book. If you need to add security features, now would be the time to add them. Add one feature at a time. Test the application after every change you make because you really don't want to have to spend hour upon hour debugging your application. Listing 23.1 shows the modified result for this example.

LISTING 23.1 Performing an Indexed Extraction

```
Sub Main
   ' Constants used to describe the field type used for the index.
   Const WI_IE_NUMFLD  = 1
   Const WI_IE_CHARFLD = 2
   Const WI_IE_TIMEFLD = 3

   ' Get the working directory.
   Dim Path As String
   Path = Client.WorkingDirectory

   ' Open the database.
   Dim db As Database
   Set db = OpenDB(Path & "Sample-Customers.imd")

   ' Check for errors.
   If db Is Nothing Then
      Exit Sub
   End If

   ' Delete the old file.
   dbName = "CompaniesInArgentina.imd"
   DeleteOld Path & dbName

   ' Create the task.
   Set task = db.IndexedExtraction

   ' Configure the task.
   task.IncludeAllFields
   task.FieldToUse =  "COUNTRY"
   task.FieldValueIs2 WI_IE_EQUAL, "ARGENTINA", WI_IE_CHARFLD
   task.OutputFilename =  dbName

   ' Perform the task.
   task.PerformTask

   ' Clear memory.
   Set task = Nothing
   Set db = Nothing

   ' Open the resulting database.
   OpenDB(Path & dbName)
End Sub
```

(*continued*)

LISTING 23.1 (*Continued*)

```
Function OpenDB(DBPath As String) As Database
   ' Verify that the database exists.
   Dim PathCheck As String
   PathCheck = Dir( DBPath )

   ' If PathCheck has a 0 length, the database doesn't
   ' exist and we need to exit the routine.
   If Len(PathCheck) <  1 Then
      MsgBox "Couldn't file the file: " & DBPath
      Exit Function
   End If

   ' Define a database object.
   Dim db As Database

   ' Open the database using the default client folder.
   Set db = Client.OpenDatabase(DBPath)

   '  Return the database object.
   OpenDB = db

   ' Clear the memory used by db.
   Set db = Nothing
End Function

Sub DeleteOld(Filepath As String)
   ' Determine if the file exists.
   Dim PathCheck As String
   PathCheck = Dir( Filepath )

   ' Delete the file if it exists.
   If Len(PathCheck) >  1 Then
   MsgBox "Deleting old copy of " + PathCheck
   Kill Filepath
   End If
End Sub
```

The code shown in Listing 23.1 is similar to the code shown in Figure 23.3, but it's a lot easier to read and understand. The code works much like many of the other examples in the book. It begins by creating some constants that define the field types used for the index. The code then obtains the working directory, opens the `Sample-Customers` database, deletes the existing `CompaniesInArgentina` database if it exists, and begins creating the extraction if everything works as anticipated.

The actual task begins with task configuration. The example includes all of the fields found in the `Sample-Customers` database as part of the output. The index relies on the COUNTRY field being equal to ARGENTINA. The `OutputFilename` property contains the name of the output database.

Sample-Customers.IMD | CustomersInArgentina.IMD

	CUSTNO	COMPANY	FIRST_NAME	LAST_NAME	COUNTRY
1	10000	Timekeepers	MARIU	EUGENIA	ARGENTINA
2	10003	Diseños de la Vendimia	JOSE	ERNESTO	ARGENTINA
3	10004	Relojes Cristalinos	MARISU	HERNAN	ARGENTINA
4	10005	Clockwatcher	JUANMA	JUAN	ARGENTINA
5	10006	Contadores de tiempo de la estrella	MARIA	TERESA	ARGENTINA
6	20766	Argentina Watches & Repairs	RODRIGO	RODRIGUEZ	ARGENTINA
7	20781	Underwater Jewellery Treasures	DANIEL	REYES	ARGENTINA
8	20820	Argentinian Estate Jewellery	JULIO	DALIE	ARGENTINA
9	20823	Gold Jewellery & Watches Inc.	MILANIO	CUADRA	ARGENTINA
10	20826	Pearls and Watches of Argentina	ARACELY	NORORY	ARGENTINA
11	42007	Patagonia Watches	RICARDO	ALVAREZ	ARGENTINA
12	42300	The Chaco Jewelry	EDUARDO	REYES	ARGENTINA
13	42302	Anillos Del Iguazu	MARIA	CORTEZ	ARGENTINA
14	42308	Watch Shop	CARLOS	ROMIREZ	ARGENTINA
15	42309	Jacob's	JACOB	JOHNSTON	ARGENTINA
16	42311	El Almacén	SERGIO	MIGUEL	ARGENTINA

FIGURE 23.4 The Output Is an Extraction of the Companies in Argentina

At this point, the code calls `PerformTask()` to create the output. The code clears memory and then opens the extracted database. Figure 23.4 shows the output from the example.

As you can see, converting your Visual Script macro does save you time and effort because you don't have to figure out how to write the initial code. However, you must perform some post-processing to create a useful IDEAScript application. In this case, the example performed these additional steps:

1. Reorganized the code to make it easier to read.
2. Added comments to document the code.
3. Augmented the code to include `OpenDB()` and `DeleteOld()`.

Summary

This chapter has shown you how to convert Visual Script macros to IDEAScript macros. While Visual Script is a very practical way to create limited macros that do just the basics, many people find that it isn't quite enough to satisfy their needs. The conversion process in this chapter helps you retain the investment in time that you've already made to create the Visual Script macro and create your first IDEAScript application much more quickly.

Now that you know how to perform the conversion, give it a try. Create a basic Visual Script macro as shown in this chapter, and then perform the conversion on it. Add

comments to the result and begin customizing the resulting IDEAScript macro. You'll find that the conversion process is both fast and easy.

Congratulations! You've reached the end of the book and now have a very good understanding of how to use IDEAScript to perform an amazing array of tasks. Of course, practice makes perfect when it comes to scripting. Take some time to work through the examples in the book. Play with the code and give yourself a chance to discover all the wonders of IDEAScript.

About the Author

John Mueller is a freelance author and technical editor. He has writing in his blood, having produced 84 books and more than 300 articles to date. The topics range from networking to artificial intelligence and from database management to heads-down programming. Some of his current books cover topics that include Windows power optimization, Windows Server 2008 GUI, and Windows Server 2008 Server Core. He has also written a programmer's guide that discusses the new Office Fluent User Interface (RibbonX). His technical editing skills have helped more than 63 authors refine the content of their manuscripts. John has provided technical editing services to both *DataBased Advisor* and *Coast Compute* magazines. He's also contributed articles to the following magazines: *CIO.com*, *DevSource*, *InformIT*, *Informant*, *DevX*, *SQL Server Professional*, *Visual C++ Developer*, *Hard Core Visual Basic*, *asp.netPRO*, *Software Test and Performance*, and *Visual Basic Developer*.

When John isn't working at the computer, he enjoys spending time in his workshop crafting wood projects or making candles. On any given afternoon, you can find him working at a lathe or putting the finishing touches on a bookcase. He also likes making glycerine soap, which comes in handy for gift baskets. You can reach John on the Internet at JMueller@mwt.net. He is also setting up a website and blog at www.mwt.net/~jmueller/; feel free to look and make suggestions on how he can improve it.

About the Website

This book includes a companion website, which can be found at www.wiley.com/go/IDEAScript. This website includes:

- All IDEAScript examples from the book
- FAQs related to IDEAScript
- IDEAScript training information

The password to enter this site is: Mueller.

Index

Absolute locations, 319
Accelerator keys, 222
Access data
  exporting, 313–315
  importing, 310–312
  importing and exporting data, 310–315
Adding databases, 135–138
Advanced math, 198–200
Advanced math functions, 199
Advanced math in applications, 200
AdvanceFindTask, 171–173
Aging, 173, 201–205
Aligning controls, 233
Alignment with grid, 235
Analysis
  about, 200
  aging, 201–205
  BenfordsLaw, 203–205
Analytical charts
  correlation, 399–401
  trend analysis, 401–404
Append database, 135–138
AppendDatabase, 173
Application building
  executable files, 19
  Windows Explorer, 20
Application documentation
  comments, 451–452
  user guide, 453
  written documentation, 452–453
Application errors
  compile errors, 430–431
  run-time errors, 431–432
  semantic errors, 432
  syntax errors, 429–430
Application features, 5
Application files organization, 449–450
Application parts, 40–41
Application shell development, 448
Applications. *See also* Hello World application; IDEAScript applications; planning for application; structured applications; toolbar and menu applications
  advanced math in, 200
  defined, 9, 39
  documentation, 451–455
  features of, 5
  methods to create structured, 41–45
  running and stopping, 433–434
  stepping through, 435–438
Applications, creation of
  copying from help, 43–44
  copying from history, 42–43
  editor window, 43
  recording tasks, 41–42
Applications, working with other
  Excel Object Model, 350–351
  IDEA running from Excel, 355–359
  vs. Visual Basic for Applications (VBA), 349–350
  Word Object Model, 350–351
  Word running from IDEA, 351–354
  summary, 359–360
Arguments, 49
Arrays
  about, 207–208
  copying data between, 212–213

Arrays (*continued*)
creating and using, 208–212
summary, 213–214
Automation, 1–3
Avoiding errors, 107–112

Basic controls
about, 218–219
aligning controls, 233
alignment with grid, 235
Button, 221–222
CancelButton, 222
centering controls on dialog box, 235
CheckBox, 225
ComboBox, 230–231
control sizing, 233–234
DropDownComboBox, 231–232
EditBox, 226–227
GroupBox, 228
ListBox, 228–230
OKButton, 222
RadioButton, 222–225
Select tool, 219–221
spacing, horizontal and vertical, 234–235
StaticText, 225–226
visual appearance, 232–234
Basic design, 448
Basic dialog box, 236–238
Basic graphs, 393–399
Basic math, 195–198
Basic search
character fields, 251–254
date fields, 256–258
numeric fields, 255–256
Beep, 238–242
Bells and whistles, 448–449
Benefits, 2–3
BenfordsLaw, 173–174, 203–205
Binary files, 322
Binary operator, 82
Binding, 57
Binding and application to menu, 57–58
Binding and application to toolbar, 59
Black-listing checks, 427
Boolean numbers, 74–75
Boolean operators, 258–261
Breakpoints
removing all, 435
removing individual, 434
setting, 434
Buffer, 334, 420
Button, 221–222

Calculations, 4
Call, 48
Caller, 215
Calling function from subroutine, 54–56
CancelButton, 222
Case sensitivity, 362
Categories, 220
Centering controls on dialog box, 235
Character fields, 251–254
ChartData, 174–175
Charts and graphs
analytical charts, 399–404
basic graphs, 393–399
selection of, 391–393
summary, 405
ChDir, 318–321
ChDrive, 321–322
CheckBox, 225
Classes, 167
Client elements
Database objects, 168–170
NewDatabase objects, 170
NewRecord objects, 170
NewTableDef objects, 170
RecordSet objects, 168–170
TableDef objects, 168–170
TableManagement objects, 168–170
Close, 322–326
Code readability, 56–57
ComboBox, 230–231
Comments
application documentation, 451–452
defined, 39
CompareDB, 138–140, 175
Comparing databases, 138–140
Compile errors, 430–431

Compiling, 430
Conditional statement, 93
Conditional statement and loops
  For . . . Next, 99–100
  avoiding errors, 107–112
  Conditions with Do Loops, 102–106
  Do Loop . . . While, 104–105
  Do Until . . . Loop, 105
  Do While . . . Loop, 103–104
  For Each . . . Next, 100–101
  On Error, 107–109
  error trapping, 106–112
  Errors, 109–112
  If . . . Then . . . Else statement, 93–95
  options, choosing between select case, 95–99
  redirecting macro flow with GoTo, 112
  summary, 112–113
Conditions with Do Loops, 102–106
Constant, defined, 65–66
Constants, 41, 50
Control sizing, 233–234
Control variables, 243–244
Copying databases, 157
Copying from help, 43–44
Copying from history, 42–43
Correlation, 175–176, 391, 399–401
Creating new, 161–164
CurDir, 326
CurDir$, 326
Currency values, 75
Custom data types, 89–91
Custom dialog boxes
  about, 235–236
  basic dialog box, 236–238
  Beep, 238–242
  control variables, 243–244
  DlgEnable, 243–244
  DlgFocus, 242–243
  DlgValue, 243–244
  DlgVisible, 243–244
Custom searches, 268

Data analysis
  distribution checking, 378–380
  GapDetection, 376–378
  merging databases, 380–389
  pivot table, 371–373
  RandomSample, 374–376
  stratification, 361–367
  summarization, 367–371
  SystematicSample, 378–380
  summary, 389–390
Data content checking, 426–427
Data conversion, 79–82
Data extraction
  Extraction, 283–285
  KeyValueExtraction, 285–288
  TopRecordsExtraction, 288–290
Data importing and exporting, 5
Data length, 425
Data range validation, 422–425
Data type, 67
Data type effect on code, 67–68
Data type selection
  data type effect on code, 67–68
  performing data conversion, 79–82
  working with Boolean numbers, 74–75
  working with currency values, 75
  working with date/time values, 76–78
  working with numbers, 73–74
  working with scientific values, 75
  working with strings, 68–72
  working with variant data, 78–79
Data validation
  about, 421–422
  data content checking, 426–427
  data length, 425
  data range validation, 422–425
  dialog boxes, 427–428
  summary, 428
Database closing, 133
Database commitment, 131–133
Database components modification, 129–131
Database criteria, 125
Database history, 120–122
Database objects, 168–170

Database parts
  about, 115
  Fields, 116
  Indexes, 117–118
  Records, 117
  Tables, 116
Database security
  data validation, 421–427
  programmable, 419–420
Database sorting, 127–129
Databases, 4
Databases, working
  adding databases, 135–138
  append database, 135–138
  CompareDB, 138–140
  comparing databases, 138–140
  ExportDatabase, 145–150
  exporting databases, 145–150
  Fields, 150–152
  keys, 140–145
  Records, 150–157
  Tables, 157–158
  summary, 164
Date fields, 256–258
Date/time values, 76–78
DEBUG menu, 32
Debugging
  application errors, 429–432
  breakpoints, 434–435
  defined, 26
  message boxes, 444
  running and stopping applications, 433–434
  stepping through applications, 435–438
  watch window, 438–444
  summary, 445
Default action using Case Else, 97–98
Definition files, 298–299
Delimited files, 296–297
Diagnostic message boxes, 444
Dialog box editor, 25–26
Dialog boxes, 41, 427–428
Dialog boxes, interactive. *See* interactive dialog boxes
Dialog tools, 25
Dim statement, 230
Dir, 326–327
Directory removal, 345
Distribution checking, 378–380
DlgEnable, 243–244
DlgFocus, 242–243
DlgValue, 243–244
DlgVisible, 243–244
Do Loop . . . While, 104–105
Do Until . . . Loop, 105
Do While . . . Loop, 103–104
Docked window, 28
Documentation, 451–453
DropDownComboBox, 231–232
DupKeyDetection, 141–143, 176
DupKeyExclusion, 144–145, 176
Duplicate keys, 141–143

EDIT menu, 32
EditBox, 226–227
Editing code, 49–51
Editor window, 13, 15–16, 21–23, 27, 43
Efficiency in searching, 249–250
Empty variable, 64
End sub, 15
EOF, 327–329
Error trapping, 106–112
Errors, 107–112. *See also* application errors
Events, 168
Excel data
  exporting, 308–310
  importing, 306–308
  importing and exporting data, 306–310
Excel Object Model, 350–351
Executable files, 19
ExportDatabase, 145–150, 176–177
Exporting
  Access data, 313–315
  Excel data, 308–310
  PDF data, 292–294
  text data types, 302–306
Exporting databases, 145–150
External code, 41
External data, 41

External variables, 346–348
Extraction, 177, 283–285

Field statistics, 122–125
Fields, 116, 150–152
File extensions, .iss vs. .ise files, 17–18
File format, 317–318
File IO features
  ChDir, 318–321
  ChDrive, 321–322
  Close, 322–326
  CurDir, 326
  CurDir$, 326
  Dir, 326–327
  EOF, 327–329
  FileCopy, 330–331
  FileLen, 331–333
  FreeFile, 333–334
  Get, 334–336
  GetAttr, 336–341
  Input, 341–342
  Input$, 341–342
  Kill, 342
  Line Input #, 342
  LOF, 342
  MkDir, 342–343
  Name, 343
  Open, 343–344
  Print, 344
  Print #, 345
  Put, 345
  RmDir, 345
  SeeAttr, 346
  Seek, 346
  Write #, 346
FILE menu, 31
File versioning, 450
FileCopy, 330–331
FileLen, 331–333
Files
  external variables, 346–348
  file format, 317–318
  file IO features, 318–346
  summary, 348
Fixed length files, 297–298
Flat files, 141
Focus, 242
For . . . Next, 99–100
For Each . . . Next, 100–101
FORMAT menu, 32–33
Formatting data, 85–89
FreeFile, 333–334
Functions. *See also* subroutines and functions
  actions of, 40
  advanced math functions, 199
  calling function from subroutine, 54–56
  defined, 16
  execution sequence, 40
  line continuation character use in, 51

GapDetection, 376–378
Gaps, 177–179
Get, 334–336
GetAttr, 336–341
GetFileStatisitics, 331
Global variables, 40–41, 56
Glue code, 49
Group, working within a, 450–451
GroupBox, 228

Handles, 322
Hello World application
  .iss vs. .ise files, 17–18
  about, 15
  saving macros, 17
  sending macros, 19
  typing in Editor window, 16–17
HELP menu, 33–34
History, 179

IDEA database system overview, 118–119
IDEA databases
  closing the database, 133
  database commitment, 131–133
  database components modification, 129–131
  database criteria, 125
  database history, 120–122

IDEA databases (*continued*)
database parts, 115–118
database sorting, 127–129
field statistics, 122–125
IDEA database system overview, 118–119
indexing, 125–127
opening for use, 119–120
summary, 133–134
IDEA interface, 4
IDEA Object Model
about, 165
client elements, 168–171
objects, 166–168
task object model, 170–192
summary, 193
IDEA running from Excel, 355–359
IDEAScript
application features, 5
applications, 5
automation, 1–3
benefits of, 2–3
calculations, 4
data importing and exporting, 5
databases, 4
IDA interface, 4
introduction, 1–8
macros, 3–6
skill assessment, 7–8
skill level priorities, 6–7
vs. Visual Basic for Applications (VBA), 349–350
summary, 7–8
IDEAScript applications
application building, 19–20
Hello World application, 15–19
macro types, 9–11
Visual Script Editor, 11–15
summary, 20
IDEAScript Editor
about, 21–27
dialog box editor, 25–26
dialog tools, 25
editor, 22–23
menus and toolbars, 29–37
project, 23–24
properties, 24–25
watch, 26–27
Windows, hiding and viewing, 26–27
If . . . Then . . . Else combination, 94–95
If . . . Then . . . Else statement, 93–95
If . . . Then . . . ElseIf combination, 94
If . . . Then alone, 93–94
Import and export features
ImportAccess, 272
ImportAS400, 272
ImportdBASE, 272–274
ImportExcel, 275
ImportXML, 275, 276
PublishToMicrosoftWord, 275, 277–282
PublishToPDF, 282
ImportAccess, 179–180, 272
ImportAS400, 180, 272
ImportdBASE, 272–274
ImportExcel, 180–181, 275
ImportExcel2007, 180–181
Importing
Access data, 310–312
Excel data, 306–308
PDF data, 290–292
text data types, 299–302
Importing and exporting data
Access data, 310–315
data extraction, 283–290
Excel data, 306–310
import and export features, 271–282
PDF data, 290–294
text data, 294–306
summary, 315
ImportXML, 181, 275, 276
Index, 182
IndexedExtraction, 182
Indexes, 117–118
Indexing, 125–127
Input, 341–342
Input$, 341–342
INSERT menu, 33
Integer, 67
Interacting with records, 155–157

Interactive dialog boxes
about, 215–216
basic controls, 218–232
best practices, 216–218
creating, 216–218
custom dialog boxes, 235–244
layout design, 218
pictures, 244–247
summary, 247
Interpreter, 430

JoinDatabase, 182–183, 381–385

Keys
defined, 135, 140
DupKeyDetection, 141–143
DupKeyExclusion, 144–145
duplicate keys, 141–143
locating records, 144–145
working with, 140–145
KeyValueExtraction, 183–184, 285–288
Kill, 342

Language Browser, 44
Lasso technique, 221
Layout design, 218
Line communication character, 51
Line Input #, 342
ListBox, 228–230
Local variables, 41
Locating records, 144–145
LOF, 342
Logic error, 432
Loops, 93

Macro recorder
about, 45
editing code, 49–51
starting macros, 46–47
stopping macros, 47
testing results, 51–52
viewing results, 47–49
Macro types
in IDEAScript, 10
in Visual Script, 10
Macros
described, 2, 9
how to use, 3–6
redirecting macro flow with GoTo, 112
saving, 17
sending macros, 19
Main, 15
Mathematical tasks
advanced math, 198–200
analysis, 200–205
basic math, 195–198
summary, 205–206
Memory leak, 120, 325
Menus and toolbars
about, 29–30
DEBUG menu, 32
EDIT menu, 32
FILE menu, 31
FORMAT menu, 32–33
HELP menu, 33–34
INSERT menu, 33
standard toolbar, 34–37
TOOLS menu, 33
Merging databases
about, 380–381
JoinDatabase, 381–385
VisualConnector, 385–389
Message boxes, 444
Methods, 167–168
Methods to create structured applications, 41–45
MkDir, 342–343
MUSCombinedEvaluate, 184–185
MUSEvaluate, 185–186
MUSExtraction, 186

Name, 343
Native fields, 150
NewDatabase objects, 170
NewRecord objects, 170
NewTableDef objects, 170
Null, 78
Numbers, 73–74
Numeric fields, 255–256

Objects
about, 166–167
classes, 167
defined, 165
events, 168
methods, 167–168
properties, 168
OKButton, 222
Open, 343–344
Open Database Connectivity (ODBC), 314–315
Opening database for use, 119–120
Operators, 82–85
Option explicit, 66
Options, choosing between select case conditional statement and loops, 95–99
default action using Case Else, 97–98
selective case alternative, 98–99
silent case, 96–97
Outputting data in PDF format, 412–414
Outputting data in Word format, 414–417

Path, 119
PDF data
exporting, 292–294
importing, 290–292
Pictures, 244–247
Pivot table, 371–373
PivotTable, 186–187
Planning for application
application files organization, 449–450
application shell development, 448
basic design, 448
bells and whistles, 448–449
Primary key, 141
Print, 344
Print #, 345
Private variables, 41, 56
Proactive checking, 282
Programmable database security, 419–420
Project, 23–24
Project management tasks
about, 447
application documentation, 451–453
planning for application, 448–449
working within a group, 450–451
summary, 453–454
Properties, 24–25, 168, 220
Proximity searches, 264–268
PublishToMicrosoftWord, 275, 277–282
PublishToPDF, 282
Put, 345

Quick watch, 440–442

RadioButton, 222–225
Random seed, 367
RandomSample, 187, 374–376
Record Definition File (RDF), 298
Record pointer, 155
Record set, 153–155
Recording tasks, 41–42
Records
defined, 117
individual records, 155–157
record set, 153–155
working with, 152–157
RecordSet objects, 168–170
Redirecting macro flow with GoTo, 112
Reference count, 281
Relative locations, 319
Removing all breakpoints, 435
Removing individual breakpoints, 434
Reports
definition of, 407–412
outputting data in PDF format, 412–414
outputting data in Word format, 414–417
summary, 418
RmDir, 345
Rubberband technique, 221
Running and stopping applications, 433–434
Run-time errors, 431–432

Scientific values, 75
Scope, 56
Scripting, 2
Searching
basic search, 251–258
Boolean operators, 258–261

built-in features for, 251–268
custom searches, 268
efficiency in, 249–250
proximity searches, 264–268
wildcards, 262–264
summary, 269
SeeAttr, 346
Seeds, 367
Seek, 346
Select tool, 219–221
Selective case alternative, 98–99
Semantic errors, 432
Sending macros, 19
Setting breakpoints, 434
Short-cut-keys, 222
Silent case, 96–97
Skill assessment, 7–8
Skill level priorities, 6–7
Sort, 187–188
Spacing, horizontal and vertical, 234–235
Spaghetti code, 112
Speed keys, 222
Standard toolbar, 34–37
Starting macros, 46–47
StaticText, 225–226
Stepping into a call, 436–437
Stepping over a call, 437–438
Stepping through applications
about, 435–436
stepping into a call, 436–437
stepping over a call, 437–438
Stopping macros, 47
Stratification
about, 361–362
uses of, 188, 361–365
StratifyRndSample, 188–189, 365–367
Strings, 68–72
Structured applications
about, 39–40
application parts, 40–41
code readability, 56–57
macro recorder, 45–52
methods to create, 41–45
subroutines and functions, 52–56
to toolbar and menu, 57–60
Structures, 93
Sub, 15
Subroutine creation, 52–54
Subroutines and functions
calling function from subroutine, 54–56
scope, 56
subroutine creation, 52–54
Summarization, 189–190, 367–371
Syntax errors, 429–430
SystematicSample, 190, 378–380

Tab separated value (TSV) files, 295
TableDef, 157–161
TableDef objects, 168–170
TableManagement, 161–164, 190
TableManagement objects, 168–170
Tables
copying databases, 157
creating new, 161–164
elements of, 116
TableDef, 157–161
TableManagement, 161–164
Task, defined, 165
Task object models
about, 170–171
AdvanceFindTask, 171–173
aging, 173
AppendDatabase, 173
BenfordsLaw, 173–174
ChartData, 174–175
CompareDB, 175
correlation, 175–176
DupKeyDetection, 176
DupKeyExclusion, 176
ExportDatabase, 176–177
Extraction, 177
Gaps, 177–179
History, 179
ImportAccess, 179–180
ImportAS400, 180
ImportExcel, 180–181
ImportExcel2007, 180–181
ImportXML, 181
Index, 182
IndexedExtraction, 182

Task object models (*continued*)
JoinDatabase, 182–183
KeyValueExtraction, 183–184
MUSCombinedEvaluate, 184–185
MUSEvaluate, 185–186
MUSExtraction, 186
PivotTable, 186–187
RandomSample, 187
Sort, 187–188
Stratification, 188
StratifyRndSample, 188–189
Summarization, 189–190
SystematicSample, 190
TableManagement, 190
TopRecordsExtraction, 191
TrendAnalyss, 191–192
VisualConnector, 192
Testing results, 51–52
Text data, 294
Text data types
definition files, 298–299
exporting, 302–306
fixed length files, 297–298
formatting, 295
importing, 299–302
tab separated value (TSV) files, 295
Toolbar and menu applications
binding and application, 57–59
binding and application to menu, 57–58
binding and application to toolbar, 59
summary, 60–61
TOOLS menu, 33
TopRecordsExtraction, 191, 288–290
Trend analysis, 391, 401–404
TrendAnalysis (task), 191–192
Type, 63

Unary operators, 82
User guide, 453

Variables
defined, 40, 64–65
Global variables, 40–41, 56
local (private), 41, 56
Variables and constants
about, 63
constant, defined, 65–66
option explicit, 66
variable, defined, 64–65
Variant data, 78–79
Vertical fields, 150
Viewing results, 47–49
Visual appearance, 232–234
Visual Basic for Applications (VBA) vs. IDEAScript, 349–350
Visual Script conversion to IDEAScript
conversion, 456–457
IDEAScript benefits, 455–456
making changes and saving results, 458–461
summary, 461–462
Visual Script Editor, 11–15
VisualConnector, 192, 385–389

Watch, 26–27
Watch window
about, 438–439
quick watch, 440–442
typing variables into quick watch, 442–444
understanding the problem, 439–440
White-listing checks, 427
Wildcards, 262–264
Windows, hiding and viewing, 26–29
Windows Explorer, 20
Windows Scheduler, 20
Word Object Model, 350–351
Word running from IDEA, 351–354
WordPad, 417
Workflow, 2
Working with data
custom data types, 89–91
data type selection, 67–82
employing operators, 82–85
formatting data, 85–89
variables and constants, 63–67
summary, 91–92
Write #, 346
Written documentation, 452–453