

OpenEdge™ Development: Progress® 4GL Handbook

John Sadd

Expert Series

Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. This manual is also copyrighted and all rights are reserved. This manual may not, in whole or in part, be copied, photocopied, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Progress Software Corporation.

The information in this manual is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear in this document.

The references in this manual to specific platforms supported are subject to change.

Allegrix, A [Stylized], ObjectStore, Progress, Powered by Progress, Progress Fast Track, Progress Profiles, Partners in Progress, Partners en Progress, Progress en Partners, Progress in Progress, P.I.P., Progress Results, ProVision, ProCare, ProtoSpeed, SmartBeans, SpeedScript, and WebSpeed are registered trademarks of Progress Software Corporation or one of its subsidiaries or affiliates in the U.S. and/or other countries. A Data Center of Your Very Own, Allegrix & Design, AppsAlive, AppServer, ASPen, ASP-in-a-Box, BusinessEdge, Business Empowerment, Empowerment Center, eXcelon, Fathom, Future Proof, IntelliStream, ObjectCache, OpenEdge, PeerDirect, POSSE, POSSENET, ProDataSet, Progress Business Empowerment, Progress Dynamics, Progress Empowerment Center, Progress Empowerment Program, Progress for Partners, Progress OpenEdge, Progress Software Developers Network, PSE Pro, PS Select, SectorAlliance, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, Smart DataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, Technical Empowerment, Trading Accelerator, WebClient, and Who Makes Progress are trademarks or service marks of Progress Software Corporation or one of its subsidiaries or affiliates in the U.S. and other countries.

SonicMQ is a registered trademark of Sonic Software Corporation in the U.S. and other countries.

Vermont Views is a registered trademark of Vermont Creative Software in the U.S. and other countries.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Any other trademarks and/or service marks contained herein are the property of their respective owners.

This product includes Raster Imaging Technology copyrighted by Snowbound Software 1993-2000. Raster imaging technology by SnowboundSoftware.com

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>). Copyright © 1999 The Apache Software Foundation. All rights reserved (Xalan XSLT Processor) and Copyright © 2000-2002 The Apache Software Foundation. All rights reserved (Jakarta-Oro). The names "Apache," "Xerces," "Jakarta-Oro," "ANT," and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. Products derived from this software may not be called "Apache" or "Jakarta-Oro," nor may "Apache" or "Jakarta-Oro" appear in their name, without prior written permission of the Apache Software Foundation. For written permission, please contact apache@apache.org. Software distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

Portions of this software are copyrighted by DataDirect Technologies, 1991-2002.

This product includes software developed by Vermont Creative Software. Copyright © 1988-1991 by Vermont Creative Software.

This product includes software developed by IBM and others. Copyright © 1999, International Business Machines Corporation and others. All rights reserved.

This product includes code licensed from RSA Security, Inc. Some portions licensed from IBM are available at <http://oss.software.ibm.com/icu4j/>.

This product includes the RSA Data Security, Inc. MD5 Message-Digest Algorithm. Copyright ©1991-2, RSA Data Security, Inc. Created 1991. All rights reserved.

This product includes software developed by the World Wide Web Consortium. Copyright © 1994-2002 World Wide Web Consortium, (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University), <http://www.w3.org/Consortium/Legal/>. All rights reserved. This work is distributed under the W3C® Software License [<http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231>] in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.



December 2003

Product Code: 4632
Item Number: 98885; R10.0A

Acknowledgements

I'd like to thank all my colleagues who have reviewed this book and given valuable comments to add clarity to the text and also to point out where I had missed important information.

My thanks also go to the OpenEdge™ development, quality assurance, and documentation teams who have worked hard to provide me with invaluable assistance throughout the writing of this book.

Contents

| | |
|--|------------|
| Preface | Preface–1 |
| 1. Introducing the Progress 4GL..... | 1–1 |
| About the sample database..... | 1–2 |
| Getting started with the 4GL—the Progress snowplow | 1–3 |
| In the beginning...FOR EACH CUSTOMER | 1–5 |
| Starting your OpenEdge session..... | 1–6 |
| Writing your first procedure | 1–10 |
| Basic characteristics of the Progress 4GL | 1–13 |
| The 4GL is procedural..... | 1–13 |
| The 4GL is block-structured | 1–13 |
| A 4GL procedure consists of statements | 1–14 |
| The 4GL combines procedural, database, and user interface statements..... | 1–15 |
| Saving your test procedure | 1–16 |
| 2. Using Basic 4GL Constructs..... | 2–1 |
| Refining the data selection with a WHERE clause | 2–2 |
| Comparison operators | 2–3 |
| Using quotation marks..... | 2–4 |
| Creating nested blocks to display related data | 2–4 |
| Changing labels and formats | 2–8 |
| Using program variables and data types | 2–9 |
| Defining formats | 2–11 |
| Other variable qualifiers..... | 2–12 |
| Variable naming conventions | 2–13 |
| Placement of variable definitions..... | 2–14 |

| | |
|--|------------|
| Defining an IF-THEN-ELSE decision point | 2-15 |
| Using the Progress Unknown value | 2-16 |
| Using built-In 4GL functions | 2-17 |
| Progress 4GL Functions | 2-20 |
| Putting a calculation into your procedure | 2-24 |
| Arithmetic expressions and operands | 2-24 |
| Arithmetic built-in functions | 2-26 |
| Using the Intelligent Edit Control and its shortcuts | 2-28 |
| Getting to online help | 2-32 |
| Saving and compiling your test procedure | 2-35 |
| 3. Running Progress 4GL Procedures | 3-1 |
| Running a subprocedure | 3-2 |
| Using the Propath | 3-3 |
| Using external and internal procedures | 3-8 |
| Writing internal procedures | 3-9 |
| Assigning a value to a variable | 3-10 |
| When to use internal and external procedures | 3-13 |
| Adding comments to your procedure | 3-14 |
| 4. Introducing the OpenEdge AppBuilder | 4-1 |
| Starting the AppBuilder | 4-2 |
| Creating a new procedure and window | 4-4 |
| Adding fields to your window | 4-5 |
| Changing object names and titles | 4-7 |
| Saving a procedure from the AppBuilder | 4-8 |
| Running your procedure | 4-9 |
| Using the Query Builder | 4-10 |
| Adding a browse to your window | 4-13 |
| Using property sheets | 4-16 |
| Using the Section Editor | 4-18 |
| Looking at preprocessor values in the Code Preview | 4-20 |
| Adding buttons to your window | 4-24 |
| Defining a CHOOSE trigger for your button | 4-25 |
| Defining user interface events | 4-28 |
| Adjusting the layout of the buttons | 4-31 |

| | |
|--|------------|
| 5. Examining the Code the AppBuilder Generates | 5-1 |
| Viewing the entire sample procedure code | 5-2 |
| The Definitions section | 5-3 |
| Window, button, browse, and frame definitions | 5-4 |
| Creating the window | 5-8 |
| Defining triggers | 5-11 |
| Triggers as event-driven code | 5-12 |
| Looking at the main block | 5-13 |
| The internal procedures | 5-17 |
| Contrasting procedural and event-driven programs | 5-17 |
| Advantages of the AppBuilder file format | 5-20 |
| Looking ahead | 5-22 |
| Reusable components | 5-22 |
| User interface independence | 5-23 |
| Distributed applications | 5-23 |
| Dynamic programming | 5-23 |
| 6. Procedure Blocks and Data Access | 6-1 |
| Blocks and block properties | 6-3 |
| Procedure block scoping | 6-3 |
| Language statements that define blocks | 6-7 |
| DO blocks | 6-7 |
| FOR blocks | 6-15 |
| REPEAT blocks | 6-26 |
| Data access without looping – the FIND statement | 6-28 |
| 7. Record Buffers and Record Scope | 7-1 |
| Record buffers | 7-2 |
| Record scope | 7-6 |
| Generating a procedure listing file | 7-8 |
| Adding procedures to the test window | 7-18 |
| Defining h-OrderCalcs.p to calculate totals | 7-18 |
| Writing the BinCheck procedure to check inventory | 7-26 |
| Displaying the new fields in the window | 7-36 |
| 8. Defining Graphical Objects | 8-1 |
| Types of objects | 8-2 |
| Defining static objects | 8-3 |
| Using the VIEW-AS phrase for data representation objects | 8-4 |
| Defining objects that don't represent data values | 8-6 |
| Using and setting object attributes | 8-6 |
| Changing attribute values | 8-7 |
| Common attribute values for visual objects | 8-10 |
| Invoking object methods | 8-16 |

| | |
|---|-------------|
| Instantiating and realizing objects | 8-18 |
| Instantiating objects in a container | 8-18 |
| Realizing and derealizing objects | 8-23 |
| Using object events | 8-23 |
| User interface events | 8-24 |
| Defining triggers | 8-27 |
| Applying events in your application | 8-33 |
| 9. Using Graphical Objects in Your Interface | 9-1 |
| Using data representation objects | 9-2 |
| Modifying fill-in attributes in the Properties Window | 9-2 |
| Changing a fill-in field to an editor | 9-4 |
| Adding a toggle box to the window | 9-9 |
| Defining a slider | 9-12 |
| Objects that display a list of choices for a data value | 9-14 |
| Using a radio set to display a set of choices for a value | 9-15 |
| Using other types of objects | 9-18 |
| Using rectangles to organize objects | 9-18 |
| Using images to display pictures | 9-20 |
| Adding text to the window | 9-23 |
| Using menus | 9-26 |
| Menu bars | 9-26 |
| Defining a menu bar | 9-27 |
| Defining a menu bar for the sample window | 9-29 |
| Defining a pop-up menu | 9-39 |
| Character-mode considerations | 9-41 |
| Conclusion | 9-42 |
| 10. Using Queries | 10-1 |
| Why you use queries in your application | 10-2 |
| Queries versus block-oriented data access | 10-2 |
| Using queries to share data between procedures | 10-4 |
| Using queries to populate a browse | 10-5 |
| Defining and using queries | 10-5 |
| OPEN and CLOSE QUERY statements | 10-6 |
| GET statements | 10-8 |
| Closing a query | 10-10 |
| Determining the current number of rows in a query | 10-11 |
| Identifying the current row in the query | 10-16 |
| Repositioning a query | 10-20 |
| Extending the sample window to use the queries | 10-24 |
| Summary | 10-30 |

| | |
|--|-------------|
| 11. Defining and Using Temp-tables | 11-1 |
| Using temporary tables in your application | 11-2 |
| Progress work-tables | 11-3 |
| The temporary database for temp-tables | 11-3 |
| Defining a temp-table | 11-4 |
| Using a temp-table to summarize data | 11-8 |
| Using a temp-table as a parameter | 11-13 |
| Temp-table parameter syntax | 11-13 |
| Defining a procedure to return Order Lines | 11-15 |
| Using BUFFER-COPY to assign multiple fields | 11-16 |
| Using include files to duplicate code | 11-17 |
| Adding an Order Line browse to the Customer window | 11-19 |
| Summary | 11-24 |
| | |
| 12. Using the Browse Object..... | 12-1 |
| Defining a query for a browse | 12-3 |
| Planning for the size of the result set | 12-3 |
| Defining a browse | 12-4 |
| Changing the test window for the OrderLine browse | 12-7 |
| Enabling columns in the browse | 12-10 |
| Defining a single- or multiple-select browse | 12-12 |
| Browse selection and query interaction | 12-14 |
| Using calculated columns | 12-15 |
| Sizing a browse and browse columns | 12-18 |
| Programming with the browse | 12-19 |
| Browse events | 12-19 |
| Manipulating rows in the browse..... | 12-25 |
| Manipulating the browse itself | 12-33 |
| Browse style options | 12-45 |
| Resizable browse objects | 12-47 |
| Resizing the browse | 12-47 |
| Moving the browse | 12-48 |
| Resizing the browse column | 12-49 |
| Moving the browse column | 12-49 |
| Changing the row height of the browse.. | 12-49 |
| Additional attributes..... | 12-50 |
| User manipulation events | 12-50 |
| Using browse objects in character interfaces | 12-51 |
| Character browse modes | 12-51 |
| Control keys | 12-53 |
| Functional differences from the Windows graphical browse | 12-55 |
| Conclusion | 12-58 |

| | |
|---|-------------|
| 13. Advanced Use of Procedures in Progress | 13-1 |
| RETURN statement and RETURN-VALUE | 13-2 |
| Using persistent procedures | 13-3 |
| Running a procedure PERSISTENT | 13-5 |
| Using a persistent procedure as a run-time library | 13-10 |
| Useful procedure attributes and methods | 13-13 |
| Using a persistent procedure as shared code | 13-17 |
| Using a persistent procedure to duplicate code | 13-18 |
| Deleting persistent procedures | 13-24 |
| Examples: Communicating between persistent procedures | 13-29 |
| Shared and global objects in Progress procedures | 13-35 |
| Why you generally shouldn't use shared objects | 13-38 |
| Global shared objects in Progress | 13-40 |
| 14. Defining Functions and Building Super Procedures | 14-1 |
| User-defined functions | 14-2 |
| Defining a function | 14-3 |
| Making a forward declaration for a function | 14-4 |
| Using the AppBuilder to generate function definitions | 14-9 |
| Making run-time references with DYNAMIC-FUNCTION | 14-12 |
| Using super procedures in your application | 14-14 |
| Super procedure language syntax | 14-14 |
| Super procedure guidelines | 14-18 |
| Using session super procedures | 14-30 |
| Super procedure example | 14-31 |
| PUBLISH and SUBSCRIBE statements | 14-37 |
| Subscribing to an event | 14-38 |
| Publishing an event | 14-39 |
| Passing parameters | 14-39 |
| Canceling a subscription | 14-40 |
| PUBLISH/SUBSCRIBE example | 14-41 |
| Conclusion | 14-44 |
| 15. Handling Data and Locking Records | 15-1 |
| Overview of data handling statements | 15-3 |
| Record locking in Progress | 15-7 |
| Record locking examples | 15-7 |
| Making sure you release record locks | 15-16 |
| Lock table resources | 15-17 |
| Optimistic and pessimistic locking strategies | 15-18 |

| | |
|--|-------------|
| 16. Updating Your Database and Writing Triggers | 16-1 |
| Transactions in Progress | 16-2 |
| Transaction blocks | 16-2 |
| Building updates into an application | 16-3 |
| Defining database triggers | 16-22 |
| Database trigger guidelines | 16-23 |
| Database events | 16-24 |
| Trigger procedure headers | 16-25 |
| Accessing and defining triggers | 16-28 |
| Session triggers | 16-33 |
| General trigger considerations | 16-34 |
| 17. Managing Transactions | 17-1 |
| Controlling the size of a transaction | 17-2 |
| Making a transaction larger..... | 17-4 |
| Making a transaction smaller | 17-6 |
| Transactions and trigger and procedure blocks | 17-8 |
| Checking whether a transaction is active | 17-8 |
| The NO-UNDO keyword on temp-tables and variables | 17-9 |
| Using the UNDO statement | 17-11 |
| Using the UNDO statement in the sample procedure | 17-13 |
| Subtransactions | 17-23 |
| Transaction mechanics..... | 17-26 |
| Using the ON phrase on a block header | 17-27 |
| Handling the ERROR condition..... | 17-27 |
| ENDKEY condition | 17-31 |
| STOP condition | 17-31 |
| QUIT condition | 17-33 |
| Summary | 17-33 |
| 18. Using Dynamic Graphical Objects | 18-1 |
| Creating visual objects | 18-3 |
| Assigning object attributes | 18-3 |
| Assigning triggers to a dynamic object | 18-5 |
| Object handles | 18-5 |
| Static object handles | 18-7 |
| Managing dynamic objects | 18-9 |
| Deleting dynamic objects | 18-10 |
| Using widget pools | 18-12 |
| Using named widget pools | 18-14 |
| Using unnamed widget pools | 18-16 |
| Are widget pools static or dynamic objects?..... | 18-16 |
| Manipulating the objects in a window | 18-17 |
| Reading and writing object attributes | 18-21 |

| | |
|--|-------------|
| Identifying the columns of a browse | 18-22 |
| Using the CAN-QUERY and CAN-SET functions | 18-24 |
| Adding dynamic objects to a window | 18-26 |
| Using a buffer handle and buffer field handles | 18-28 |
| Populating a list at run time | 18-30 |
| Creating dynamic fields | 18-32 |
| Frame parenting | 18-32 |
| Object positioning | 18-32 |
| Object sizing | 18-32 |
| Label handling | 18-33 |
| Data handling | 18-33 |
| Data typing | 18-34 |
| Adding dynamic fields to the test window | 18-34 |
| Using multiple windows | 18-43 |
| Creating window families | 18-47 |
| Creating a dynamic browse | 18-49 |
| ADD-COLUMNS-FROM method | 18-51 |
| ADD-LIKE-COLUMN method | 18-52 |
| ADD-CALC-COLUMN method | 18-52 |
| Notes on dynamic browses and browse columns | 18-53 |
| Extending the sample procedure with a dynamic browse | 18-54 |
| Accessing the browse columns | 18-58 |
| Creating a dynamic menu | 18-59 |
| Creating a menu | 18-60 |
| Creating a submenu | 18-60 |
| Creating menu items | 18-61 |
| Navigating the hierarchy of menu handles | 18-61 |
| Adding a dynamic menu to the test window | 18-62 |
| Summary | 18-66 |
| 19. Using Dynamic Queries and Buffers | 19-1 |
| Using dynamic queries and query handles | 19-3 |
| Query methods and attributes | 19-3 |
| SET-BUFFERS method | 19-4 |
| ADD-BUFFER method | 19-5 |
| NUM-BUFFERS attribute | 19-6 |
| GET-BUFFER-HANDLE method | 19-6 |
| NAME attribute | 19-7 |
| QUERY-PREPARE method | 19-8 |
| PREPARE-STRING attribute | 19-9 |
| QUERY-OPEN method | 19-10 |
| Using the QUOTER function to assemble a query | 19-11 |
| QUERY-CLOSE method | 19-12 |
| QUERY-OFF-END attribute | 19-13 |

| | |
|--|-------------|
| IS-OPEN attribute | 19–13 |
| Navigation methods | 19–13 |
| Reposition methods | 19–15 |
| INDEX-INFORMATION attribute. | 19–15 |
| NUM-RESULTS and CURRENT-RESULT-ROW attributes | 19–16 |
| Cleaning up a dynamic query | 19–16 |
| Extending the sample window to filter dynamically | 19–16 |
| Defining a trigger block for multiple objects | 19–20 |
| Using the SELF handle to identify an object in a trigger | 19–21 |
| Using INDEX-INFORMATION and a MESSAGE with a yes/no answer . | 19–23 |
| Using dynamic buffers and buffer handles | 19–26 |
| Buffer handle attributes | 19–27 |
| Buffer handle methods | 19–28 |
| Specifying the lock or wait option as a variable | 19–30 |
| Extending the test window to use a buffer handle. | 19–30 |
| Cleaning up dynamic buffers | 19–36 |
| Special dynamic buffer considerations | 19–38 |
| Using BUFFER-FIELD objects | 19–39 |
| | |
| 20. Creating and Using Dynamic Temp-tables and Browses. | 20–1 |
| Creating and using dynamic temp-tables | 20–2 |
| Temp-table methods | 20–3 |
| Temp-table attributes | 20–9 |
| Temp-table buffer methods and attributes | 20–10 |
| Changing field attributes in a temp-table buffer | 20–12 |
| Cleaning up a dynamic temp-table | 20–14 |
| Extending the example to create and display records | 20–14 |
| Temp-table parameters | 20–17 |
| Using the TABLE form | 20–17 |
| Using the HANDLE form | 20–19 |
| Using the TABLE-HANDLE form. | 20–19 |
| Creating and using dynamic browses | 20–23 |
| Adding columns to the browse | 20–24 |
| Extending the test procedure with a dynamic browse | 20–26 |
| Adding columns to a static browse | 20–31 |
| Query methods for use with browses | 20–31 |
| The dynamic CALL object. | 20–32 |

| | |
|---|----------------|
| Building a comprehensive example | 20–33 |
| Creating a dynamic browse and customizing its display | 20–34 |
| Reading the database metaschema data | 20–36 |
| Populating a selection list dynamically | 20–37 |
| Creating a dynamic query | 20–37 |
| Creating a dialog box procedure | 20–39 |
| Making a call to an AppServer | 20–40 |
| Creating a dynamic temp-table | 20–42 |
| Using the SESSION handle to identify dynamic objects | 20–44 |
| Dynamic programming considerations | 20–45 |
| 21. Progress Programming Best Practices | 21–1 |
| Writing efficient procedures | 21–2 |
| Using NO-UNDO variables and temp-tables | 21–2 |
| Grouping assignments with the ASSIGN statement | 21–2 |
| Using BUFFER-COPY and BUFFER-COMPARE | 21–3 |
| Block-related tips | 21–4 |
| The CASE statement | 21–5 |
| Using arrays instead of lists | 21–7 |
| Using FOR FIRST versus FIND | 21–7 |
| The ETIME function | 21–7 |
| Using the CAN-FIND function | 21–9 |
| Dynamic programming tips | 21–10 |
| Using indexes properly | 21–14 |
| Hiding screen contents when making changes | 21–16 |
| Defining efficient queries and FOR EACH statements | 21–16 |
| Copying temp-tables as parameters | 21–20 |
| Setting your Propath correctly | 21–20 |
| Database and AppServer-related issues | 21–20 |
| Configuring your session using startup options | 21–22 |
| Memory management in Progress | 21–23 |
| Cleaning up dynamically allocated memory | 21–23 |
| Using widget pools | 21–27 |
| Making the best use of dynamic objects | 21–32 |
| Avoiding stale handles | 21–33 |
| Deleting persistent procedures | 21–34 |
| Deleting temp-table copies | 21–36 |
| Other object types | 21–37 |
| Conclusion | 21–38 |
| Conclusion to the book | 21–39 |
| Index | Index–1 |

Tables

| | | |
|-------------|--|-------|
| Table 2–1: | Comparison operators | 2–3 |
| Table 2–2: | Basic supported data types | 2–9 |
| Table 2–3: | Common format symbols | 2–11 |
| Table 2–4: | Variable qualifiers | 2–12 |
| Table 2–5: | Date functions | 2–20 |
| Table 2–6: | List functions | 2–21 |
| Table 2–7: | 4GL string manipulation functions | 2–22 |
| Table 2–8: | Supported arithmetic operands | 2–24 |
| Table 2–9: | Arithmetic built-in functions | 2–26 |
| Table 9–1: | iDiscPct attribute values | 9–12 |
| Table 12–1: | Browse and query interaction | 12–15 |
| Table 12–2: | Row mode control keys | 12–53 |
| Table 12–3: | Edit mode control keys | 12–54 |
| Table 18–1: | Window attributes, methods, and events | 18–45 |
| Table 20–1: | Temp-table parameter definitions | 20–21 |

Figures

| | | |
|--------------|--|------|
| Figure 2–1: | Result of running simple sample procedure | 2–6 |
| Figure 4–1: | The ADE desktop | 4–2 |
| Figure 4–2: | The AppBuilder main window | 4–2 |
| Figure 4–3: | The AppBuilder Palette | 4–3 |
| Figure 4–4: | Preprocessor definition in the Code Preview window | 4–21 |
| Figure 5–1: | The Definitions section | 5–3 |
| Figure 5–2: | Procedural top-down application flow | 5–18 |
| Figure 5–3: | Event-driven application flow | 5–19 |
| Figure 5–4: | Section Editor Insert menu | 5–20 |
| Figure 6–1: | Result of variable defined in main procedure only | 6–4 |
| Figure 6–2: | Result of variable defined in both main and subprocedures | 6–5 |
| Figure 6–3: | Result of variable defined in the subprocedure only | 6–6 |
| Figure 6–4: | Example of empty query result | 6–8 |
| Figure 6–5: | Result of looping with a DO block | 6–10 |
| Figure 6–6: | Example of looping with a DO block with initial value set to 1 | 6–11 |
| Figure 6–7: | Example DO WHILE loop result | 6–12 |
| Figure 6–8: | Joining records from more than one table | 6–17 |
| Figure 6–9: | Syntax error message | 6–18 |
| Figure 6–10: | Lowest Order number for each Customer | 6–21 |
| Figure 6–11: | Orders sorted by OrderDate | 6–22 |
| Figure 6–12: | Earliest Customer Order | 6–22 |
| Figure 6–13: | Specifying a different block | 6–24 |
| Figure 6–14: | Results of using the INSERT statement | 6–27 |
| Figure 6–15: | Result of a simple FIND procedure | 6–29 |
| Figure 6–16: | Result of variation on the simple FIND procedure | 6–30 |
| Figure 6–17: | Result of the simple FIND procedure using PostalCode | 6–30 |
| Figure 6–18: | Result of using the primary index | 6–32 |
| Figure 6–19: | Result of using the CountryPost index for record retrieval | 6–33 |
| Figure 6–20: | Result of forcing index selection | 6–34 |
| Figure 6–21: | Result of unique FIND | 6–35 |
| Figure 6–22: | Result of CAN-FIND function procedure | 6–36 |
| Figure 6–23: | Result of CAN-FIND function procedure without FIRST keyword | 6–37 |
| Figure 6–24: | CAN-FIND error message | 6–38 |
| Figure 6–25: | FIND FIRST Order result | 6–39 |
| Figure 7–1: | Comparing zip codes | 7–5 |
| Figure 7–2: | Result of record buffer Rule 1 example | 7–8 |
| Figure 7–3: | Invalid buffer references error message | 7–11 |
| Figure 7–4: | FOR EACH processing error message | 7–12 |
| Figure 7–5: | Conflicting table reference error message | 7–14 |
| Figure 7–6: | Customers with CreditLimits over 80000 – first result | 7–15 |
| Figure 7–7: | Customers with CreditLimits over 80000 – next result | 7–15 |
| Figure 7–8: | Raising buffer scope example result | 7–16 |
| Figure 7–9: | Raising buffer scope example 2 result | 7–17 |

| | | |
|---------------|---|-------|
| Figure 7–10: | Mismatched parameters error message | 7–19 |
| Figure 7–11: | Correct syntax information box | 7–20 |
| Figure 8–1: | Property sheet for a fill-in field | 8–14 |
| Figure 8–2: | Advanced Properties dialog box | 8–15 |
| Figure 8–3: | Frames for buttons | 8–22 |
| Figure 8–4: | Common button events | 8–24 |
| Figure 8–5: | Direct manipulation events | 8–25 |
| Figure 8–6: | Portable mouse events | 8–25 |
| Figure 8–7: | Developer events | 8–26 |
| Figure 8–8: | Keyboard Event dialog box | 8–26 |
| Figure 8–9: | Result of trigger example procedure | 8–30 |
| Figure 8–10: | Results of running the ChooseProc procedure | 8–31 |
| Figure 8–11: | Results of running the PersistProc procedure | 8–32 |
| Figure 8–12: | Result of typing into cFillFrom | 8–35 |
| Figure 9–1: | Properties Window | 9–4 |
| Figure 9–2: | Example submenu | 9–26 |
| Figure 9–3: | Sample menu information message | 9–38 |
| Figure 10–1: | Customers and Orders sample window | 10–3 |
| Figure 10–2: | Result of GET statement example | 10–9 |
| Figure 10–3: | Result of NUM-RESULTS example | 10–12 |
| Figure 10–4: | Result of CURRENT-RESULT-ROW example | 10–17 |
| Figure 10–5: | Result of RowID example | 10–22 |
| Figure 11–1: | First page result of h-InvSummary.p | 11–12 |
| Figure 12–1: | Result of ROW-DISPLAY event example | 12–22 |
| Figure 12–2: | Result of column event example | 12–23 |
| Figure 12–3: | Character browse in row mode | 12–51 |
| Figure 12–4: | Character browse in edit mode | 12–52 |
| Figure 12–5: | Browse widget online help topic | 12–58 |
| Figure 13–1: | A procedure call stack | 13–3 |
| Figure 13–2: | Call stack in InternalProc | 13–4 |
| Figure 13–3: | Message result of childproc.p | 13–7 |
| Figure 13–4: | Instantiating a persistent procedure | 13–8 |
| Figure 13–5: | PRO*Tools palette with Session icon selected | 13–11 |
| Figure 13–6: | Session Attributes PRO*Tool | 13–11 |
| Figure 13–7: | h-UsefulProc.p message | 13–13 |
| Figure 13–8: | Message box for h-mainsig.p | 13–16 |
| Figure 13–9: | Mainproc.p and Subproc.p | 13–37 |
| Figure 13–10: | Persistent procedure example | 13–39 |
| Figure 14–1: | Result of the ConvTemp.p procedure | 14–6 |
| Figure 14–2: | An object procedure and two super procedures | 14–19 |
| Figure 14–3: | Super procedure stack execution | 14–20 |
| Figure 14–4: | Super procedure stack execution without SEARCH-TARGET | 14–22 |
| Figure 14–5: | Running a procedure within a super procedure | 14–28 |
| Figure 14–6: | A global property in a super procedure | 14–29 |
| Figure 14–7: | Running the sample procedure with a super procedure | 14–34 |

| | | |
|---------------|---|-------|
| Figure 14–8: | Variation on running the sample procedures with a super procedure | 14–36 |
| Figure 15–1: | Data handling statements | 15–4 |
| Figure 15–2: | h-findCustUser1.p procedure | 15–12 |
| Figure 15–3: | Result of two sessions running h-findCustUser1.p procedure | 15–12 |
| Figure 15–4: | SHARE-LOCK status message | 15–13 |
| Figure 15–5: | EXCLUSIVE-LOCK message | 15–14 |
| Figure 15–6: | NO-LOCK error message | 15–16 |
| Figure 17–1: | Example of creating and updating Customer records | 17–10 |
| Figure 17–2: | Example transaction scope | 17–16 |
| Figure 17–3: | Order Updates example | 17–17 |
| Figure 17–4: | Order Updates example message | 17–17 |
| Figure 17–5: | Order Updates example (after Fetch) | 17–18 |
| Figure 17–6: | Variation of transaction scope | 17–20 |
| Figure 17–7: | Another variation of transaction scope | 17–22 |
| Figure 17–8: | Third variation of transaction scope | 17–24 |
| Figure 17–9: | Example subtransaction | 17–25 |
| Figure 17–10: | FIND error message | 17–27 |
| Figure 17–11: | Example error message | 17–29 |
| Figure 17–12: | Example information message | 17–29 |
| Figure 17–13: | Procedure not found error message | 17–32 |
| Figure 17–14: | Example message for procedure not found condition | 17–32 |
| Figure 18–1: | Test button message | 18–6 |
| Figure 18–2: | Button handle error message | 18–7 |
| Figure 18–3: | Static button example result | 18–8 |
| Figure 18–4: | Enabled button example result | 18–8 |
| Figure 18–5: | Example button message | 18–10 |
| Figure 18–6: | Unknown table error message | 18–11 |
| Figure 18–7: | Static handle error message | 18–11 |
| Figure 18–8: | Button handle message | 18–14 |
| Figure 18–9: | Button handle message | 18–15 |
| Figure 18–10: | Another button handle message | 18–15 |
| Figure 18–11: | Relationships between objects in a window | 18–21 |
| Figure 18–12: | Updated sample window | 18–24 |
| Figure 18–13: | DATA-TYPE error message | 18–24 |
| Figure 18–14: | Result of LIST-QUERY-ATTRS function example | 18–26 |
| Figure 18–15: | ROW-MARKERS error message | 18–51 |
| Figure 18–16: | Result of COLUMN example | 18–59 |
| Figure 19–1: | Number of buffers message | 19–6 |
| Figure 19–2: | Names of buffers message | 19–7 |
| Figure 19–3: | Name of static query message | 19–7 |
| Figure 19–4: | PREPARE-STRING message | 19–9 |
| Figure 19–5: | Static OPEN query message | 19–11 |
| Figure 19–6: | Dynamic methods with OPEN query message | 19–11 |
| Figure 19–7: | Example QUOTER function message | 19–12 |
| Figure 19–8: | Result of query methods and attributes example | 19–14 |

| | | |
|---------------|--|-------|
| Figure 19–9: | Example LEAVE trigger | 19–20 |
| Figure 19–10: | Result of using the POSITION attribute | 19–41 |
| Figure 20–1: | Changed field attributes message | 20–13 |
| Figure 20–2: | Result of running h-dbbrowser.w | 20–33 |
| Figure 21–1: | Result of CASE statement example | 21–6 |
| Figure 21–2: | Result of ETIME function example | 21–8 |
| Figure 21–3: | Result of CAN-FIND function example | 21–9 |
| Figure 21–4: | Result of DELETE OBJECT example | 21–12 |
| Figure 21–5: | Result of more efficient DELETE OBJECT example | 21–13 |
| Figure 21–6: | Result of widget pool example | 21–28 |
| Figure 21–7: | Errors for unnamed widget pool example | 21–29 |
| Figure 21–8: | Result of named widget pool example | 21–30 |

Contents

Preface

This Preface contains the following sections:

- Purpose
- Audience
- Organization
- Typographical conventions
- Examples of syntax descriptions
- Example procedures
- OpenEdge messages

Purpose

This book is a comprehensive tour of all the essential elements of the Progress 4GL. By the time you complete it, you will be competent to build thoroughly modern applications that take advantage of most of the language's features. And you will use the language to build powerful business application logic, not just the toy demonstration windows you find in other language tutorials. Progress is not a toy. It's a tool for serious developers who have serious business problems to solve.

There are a lot more specifics to many of the language statements, keywords, and options than this book covers. Therefore, you should always refer to the product documentation for more information. Where the language is concerned, you should refer to the *OpenEdge Development: Progress 4GL Reference*. This three-volume set has a complete description of all of the 4GL's methods, attributes, and events in Volume I and an alphabetical listing of all the language keywords on Volumes II and III. Always look to it for more details on any topic covered in this book.

Audience

This book is written for two different kinds of audiences. The first group consists of existing Progress developers who might have created applications in earlier releases of the product, when many of the more recent extensions to the language were not yet available. These extensions include:

- Statements to define a Graphical User Interface (GUI).
- Event-driven constructs that allow the user of a GUI application to navigate through the code much more flexibly than in an older, top-down, character-mode application, developed in Progress fifteen or twenty years ago. Those older applications are still fully supported. You can compile and run them in the latest releases of the Progress product, but many new language statements and other features have made it possible to build modern applications in ways that were not possible before.

If you are already experienced in the 4GL, you will find some of the material, especially in the first couple of chapters, familiar. Feel free to skim through those sections quickly, but you should not skip even the first chapters altogether to avoid missing anything—there are changes and enhancements to the language at every level.

The second audience for this book consists of people completely new to Progress. The Progress 4GL remains by far the most powerful and comprehensive language for developing serious business applications. Combined with the Progress RDBMS™ database and the other components of the Progress OpenEdge™ product family, the Progress 4GL can support you in building and deploying your applications in ways that are matched by no other environment today. For those of you in this second group, you should take the time to learn enough about the language to understand some of the many ways it can help you in your development work. You probably need to have a background in some other programming language to catch on to Progress quickly, because the material might come at you fast and furious. Despite its many extraordinary and unique features, the Progress 4GL is basically a procedural programming language with most of the same constructs common to almost all such languages.

Organization

[Chapter 1, “Introducing the Progress 4GL,”](#)

Provides an introduction to the Progress 4GL, including an overview of its characteristics.

[Chapter 2, “Using Basic 4GL Constructs,”](#)

Describes the basics of writing Progress procedures.

[Chapter 3, “Running Progress 4GL Procedures,”](#)

Describes how you can work with and run different types of Progress 4GL procedures.

[Chapter 4, “Introducing the OpenEdge AppBuilder,”](#)

Introduces the AppBuilder's code generating features and pre-built components.

[Chapter 5, “Examining the Code the AppBuilder Generates,”](#)

Examines the underlying AppBuilder-generated code.

[Chapter 6, “Procedure Blocks and Data Access,”](#)

Describes how to integrating data-access logic into procedures using procedure blocks and scoping.

Chapter 7, “Record Buffers and Record Scope,”

Expands the discussion on integrating data-access logic into procedures with record buffers and record scope.

Chapter 8, “Defining Graphical Objects,”

Describes how to create visual objects for data display.

Chapter 9, “Using Graphical Objects in Your Interface,”

Describes how to add visual objects to a user interface procedure.

Chapter 10, “Using Queries,”

Describes how to define and use queries to pass result sets between application modules.

Chapter 11, “Defining and Using Temp-tables,”

Describes how to define and use temp-tables to pass result sets between application modules which is essential when creating distributed applications.

Chapter 12, “Using the Browse Object,”

Describes how to design and interact with static browse objects and the queries they represent.

Chapter 13, “Advanced Use of Procedures in Progress,”

Describes how to write and invoke procedures in addition to treating them as independent objects.

Chapter 14, “Defining Functions and Building Super Procedures,”

Describes how to create user-defined functions and build super procedures.

Chapter 15, “Handling Data and Locking Records,”

Describes data handling and record locks from the perspective of a distributed application.

Chapter 16, “Updating Your Database and Writing Triggers,”

Describes how to update your database and write triggers—topics you need to understand to build the business logic that is the heart of your application.

Chapter 17, “Managing Transactions,”

Describes how to manage database transactions.

Chapter 18, “Using Dynamic Graphical Objects,”

Adding flexibility to your application by creating visual objects and defining their attributes programmatically at run time.

Chapter 19, “Using Dynamic Queries and Buffers,”

Creating dynamic versions of data management objects such as queries and buffers.

Chapter 20, “Creating and Using Dynamic Temp-tables and Browses,”

Creating dynamic versions of data management objects such as temp-tables and the browse.

Chapter 21, “Progress Programming Best Practices,”

This final chapter introduces additional language constructs and programming techniques to help you build the best-performing applications you can.

Typographical conventions

This manual uses the following typographical conventions:

| Convention | Description |
|------------------------------------|--|
| Bold | Bold typeface indicates commands or characters the user types, or the names of user interface elements. |
| <i>Italic</i> | Italic typeface indicates the title of a document, provides emphasis, or signifies new terms. |
| SMALL, BOLD CAPITAL LETTERS | Small, bold capital letters indicate OpenEdge™ key functions and generic keyboard keys; for example, GET and CTRL . |
| KEY1-KEY2 | A hyphen between key names indicates a <i>simultaneous</i> key sequence: you press and hold down the first key while pressing the second key. For example, CTRL-X . |

| Convention | Description |
|-------------------------------|--|
| KEY1 KEY2 | A space between key names indicates a <i>sequential</i> key sequence: you press and release the first key, then press another key. For example, ESCAPE H . |
| Syntax: | |
| Fixed width | A fixed-width font is used in syntax statements, code examples, and for system output and filenames. |
| <i>Fixed-width italics</i> | Fixed-width italics indicate variables in syntax statements. |
| Fixed-width bold | Fixed-width bold indicates variables with special emphasis. |
| UPPERCASE fixed width | Uppercase words are Progress® 4GL language keywords. Although these always are shown in uppercase, you can type them in either uppercase or lowercase in a procedure. |
| Period (.) or colon (:) | All statements except DO, FOR, FUNCTION, PROCEDURE, and REPEAT end with a period. DO, FOR, FUNCTION, PROCEDURE, and REPEAT statements can end with either a period or a colon. |
| [] | Large brackets indicate the items within them are optional. |
| [] | Small brackets are part of the Progress 4GL language. |
| { } | Large braces indicate the items within them are required. They are used to simplify complex syntax diagrams. |
| { } | Small braces are part of the Progress 4GL language. For example, a called external procedure must use braces when referencing arguments passed by a calling procedure. |
| | A vertical bar indicates a choice. |
| ... | Ellipses indicate repetition: you can choose one or more of the preceding items. |

Examples of syntax descriptions

In this example, ACCUM is a keyword, and *aggregate* and *expression* are variables:

```
ACCUM aggregate expression
```

FOR is one of the statements that can end with either a period or a colon, as in this example:

```
FOR EACH Customer:  
    DISPLAY Name.  
END.
```

In this example, STREAM *stream*, UNLESS-HIDDEN, and NO-ERROR are optional:

```
DISPLAY [ STREAM stream ] [ UNLESS-HIDDEN ] [ NO-ERROR ]
```

In this example, the outer (small) brackets are part of the language, and the inner (large) brackets denote an optional item:

```
INITIAL [ constant [ , constant ] ]
```

A called external procedure must use braces when referencing compile-time arguments passed by a calling procedure, as shown in this example:

```
{ &argument-name }
```

In this example, EACH, FIRST, and LAST are optional, but you can choose only one of them:

```
PRESELECT [ EACH | FIRST | LAST ] record-phrase
```

In this example, you must include two expressions, and optionally you can include more. Multiple expressions are separated by commas:

```
MAXIMUM ( expression , expression [ , expression ] ... )
```

In this example, you must specify MESSAGE and at least one *expression* or SKIP [(n)], and any number of additional *expression* or SKIP [(n)] is allowed:

```
MESSAGE { expression | SKIP [ (n) ] } ...
```

In this example, you must specify { *include-file* , then optionally any number of *argument* or &*argument-name* = "argument-value" , and then terminate with }:

```
{ include-file
  [ argument | &argument-name = "argument-value" ] ... }
```

Long syntax descriptions split across lines

Some syntax descriptions are too long to fit on one line. When syntax descriptions are split across multiple lines, groups of optional and groups of required items are kept together in the required order.

In this example, WITH is followed by six optional items:

Syntax

```
WITH [ ACCUM max-length ] [ expression DOWN ]
  [ CENTERED ] [ n COLUMNS ] [ SIDE-LABELS ]
  [ STREAM-IO ]
```

Complex syntax descriptions with both required and optional elements

Some syntax descriptions are too complex to distinguish required and optional elements by bracketing only the optional elements. For such syntax, the descriptions include both braces (for required elements) and brackets (for optional elements).

In this example, ASSIGN requires either one or more *field* entries or one *record*. Options available with *field* or *record* are grouped with braces and brackets:

Syntax

```
ASSIGN { { [ FRAME frame ]
          { field [ = expression ] }
          [ WHEN expression ]
        }
      ...
    | { record [ EXCEPT field ... ] }
}
```

Example procedures

This manual provides numerous example procedures that illustrate syntax and concepts. Examples use the following conventions:

- They appear in boxes with borders.
- If a procedure is available online, its name appears above the box and starts with a prefix associated with the manual that references it:
 - i- — *OpenEdge Development: Programming Interfaces*, for example, i-ddeex1.p
 - h- — *OpenEdge Development: Progress 4GL Handbook*, for example, h-mainsig.p
 - r- — *OpenEdge Development: Progress 4GL Reference*, for example, r-dynbut.p

If the name does not start with a listed prefix, the procedure is not available online.

- If a procedure is not available online, it compiles as shown but might not execute for lack of completeness.

Accessing files in procedure libraries

Documentation examples are stored in procedure libraries, `prodoc.p1` and `prohelp.p1`, in the `src` directory where OpenEdge™ is installed.

You *must* create all subdirectories required by a library *before* trying to extract files from the library. You can see what directories and subdirectories a library needs by using the `PROLIB -list` command to view the contents of the library. See [OpenEdge Deployment: Managing 4GL Applications](#) for more details on the PROLIB utility.

Creating a listing of the procedure libraries

Creating a listing of the source files from a procedure library involves running PROENV to set up your OpenEdge environment, and running PROLIB.



To create a listing of the source files from a procedure library:

1. From the Control Panel or the **Progress** Program Group, double-click the **Proenv** icon.
2. The **Proenv** window appears, with the `proenv` prompt.

Running Proenv sets the DLC environment variable to the directory where you installed OpenEdge (by default, `C:\Program Files\Progress`). Proenv also adds the DLC environment variable to your PATH environment variable and adds the `bin` directory (`PATH=%DLC%;%DLC%\bin;%PATH%`).

3. At the `proenv` prompt, enter the following command to create the `prodoc.txt` text file, which contains the file listing for the `prodoc.p1` library:

```
PROLIB %DLC%\src\prodoc.p1 -list > prodoc.txt
```

Extracting source files from procedure libraries (Windows)

Extracting source files from a procedure library involves running PROENV to set up your OpenEdge environment, creating the directory structure for the files you want to extract, and running PROLIB.



To extract source files from procedure libraries:

1. From the Control Panel or the **Progress** Program Group, double-click the **Proenv** icon.
2. The **Proenv** window appears, with the proenv prompt.
3. At the proenv prompt, enter the following command to create the prodoc directory in your OpenEdge working directory (by default, C:\Progress\Wrk):

```
MKDIR prodoc
```

4. Create the langref directory under prodoc:

```
MKDIR prodoc\langref
```

5. To extract all examples in a procedure library directory, run the PROLIB utility. Note you must use double quotes because “Program Files” contains an embedded space:

```
PROLIB "%DLC%\src\prodoc.pl" -extract prodoc\langref\*.*
```

PROLIB extracts all examples into prodoc\langref.

To extract one example, run PROLIB and specify the file that you want to extract as it is stored in the procedure library:

```
PROLIB "%DLC%\src\prodoc.pl" -extract prodoc\langref/r-syshlp.p
```

PROLIB extracts r-syshlp.p into prodoc\langref.

Extracting source files from procedure libraries (UNIX)

Extracting source files from a procedure library involves running PROENV to set up your OpenEdge environment, creating the directory structure for the files you want to extract, and running PROLIB.



To extract source files from procedure libraries:

1. Run the PROENV utility:

```
install-dir/dlc/bin/proenv
```

Running proenv sets the DLC environment variable to the directory where you installed OpenEdge (by default, /usr/dlc). The proenv utility also adds the bin directory under the DLC environment variable to your PATH environment variable (PATH=\$DLC/bin:\$PATH).

2. At the proenv prompt, create the prodoc directory in your OpenEdge working directory:

```
mkdir prodoc
```

3. Create the proghand directory under prodoc:

```
mkdir prodoc/handbook
```

4. To extract all examples in a procedure library directory, run the PROLIB utility:

```
prolib $DLC/src/prodoc.pl -extract prodoc/handbook/*.*
```

PROLIB extracts all examples into prodoc/proghand.

To extract one source file (p-wrk1.p) from a procedure library (prodoc.pl), run PROLIB and specify the file you want to extract as it is stored in the procedure library:

```
prolib $DLC/src/prodoc.pl -extract prodoc/handbook/h-mainsig.p
```

PROLIB extracts `h-mainsig.p` into `prodoc/handbook`.

OpenEdge messages

OpenEdge displays several types of messages to inform you of routine and unusual occurrences:

- *Execution messages* inform you of errors encountered while OpenEdge is running a procedure; for example, if OpenEdge cannot find a record with a specified index field value.
- *Compile messages* inform you of errors found while OpenEdge is reading and analyzing a procedure before running it; for example, if a procedure references a table name that is not defined in the database.
- *Startup messages* inform you of unusual conditions detected while OpenEdge is getting ready to execute; for example, if you entered an invalid startup parameter.

After displaying a message, OpenEdge proceeds in one of several ways:

- Continues execution, subject to the error-processing actions that you specify or that are assumed as part of the procedure. This is the most common action taken after execution messages.
- Returns to the OpenEdge Procedure Editor, so you can correct an error in a procedure. This is the usual action taken after compiler messages.
- Halts processing of a procedure and returns immediately to the Procedure Editor. This does not happen often.
- Terminates the current session.

OpenEdge messages end with a message number in parentheses. In this example, the message number is 200:

```
** Unknown table name table. (200)
```

If you encounter an error that terminates OpenEdge, note the message number before restarting.

Obtaining more information about OpenEdge messages

On Windows platforms, use OpenEdge online help to obtain more information about OpenEdge messages. Many OpenEdge tools include the following Help menu options to provide information about messages:

- Choose **Help→Recent Messages** to display detailed descriptions of the most recent OpenEdge message and all other messages returned in the current session.
- Choose **Help→Messages** and then enter the message number to display a description of a specific OpenEdge message.
- In the Procedure Editor, press the **HELP** key or **F1**.

On UNIX platforms, use the Progress PRO command to start a single-user mode character OpenEdge client session and view a brief description of a message by providing its number.



To use the PRO command to obtain a message description by message number:

1. Start the OpenEdge Procedure Editor:

```
install-dir/dlc/bin/pro
```

2. Press **F3** to access the menu bar, then choose **Help→Messages**.
3. Type the message number and press **ENTER**. Details about that message number appear.
4. Press **F4** to close the message, press **F3** to access the Procedure Editor menu, and choose **File→Exit**.

Introducing the Progress 4GL

The Progress® 4GL is a high-level procedural programming language, developed to enable you to build almost all aspects of an enterprise business application, from the user interface to the database access and business logic. Over its twenty-year life span, the Progress 4GL has grown into a versatile and extraordinarily powerful tool, not only allowing you to use it to program applications, but to build many of the tools that you use to help create and support those applications.

This chapter provides an introduction to the Progress 4GL. It includes the following sections:

- [Getting started with the 4GL—the Progress snowplow](#)
- [In the beginning...FOR EACH CUSTOMER](#)
- [Basic characteristics of the Progress 4GL](#)
- [Saving your test procedure](#)

The Progress programming language is a fourth-generation language (4GL) because it has powerful statements and keywords that are specialized for building business applications. Single programming statements in the 4GL can do the work of dozens or possibly hundreds of lines of code in a standard 3GL, such as Visual Basic, Java, or C++. A single 4GL statement can bring data all the way from the application database to the user interface, or return a user's changes back to the database. Other statements let you program with great precision, even down to the level of extracting individual bits from a data stream. This flexibility is what gives the Progress 4GL its great power as a development language. Most of the development tools you use to develop OpenEdge™ applications are themselves written in the 4GL.

In its first releases, in the early 1980s, the Progress 4GL allowed developers to build character interface applications that ran on a wide variety of hardware platforms, including many varieties of UNIX, DOS, and some other operating systems no longer in use. Early Progress applications were, from the very first, fully portable between platforms so that a developer could simply move application programs from one type of machine or one type of display terminal to another with confidence that they would work correctly everywhere.

With the increasing presence of Microsoft Windows as a platform for graphical interfaces, the 4GL evolved to support those interfaces, with all their various visual controls, as well as the event-driven programming constructs needed for a menu-and-mouse-driven application. Today the 4GL continues to grow, with newer extensions to provide more and more dynamic definition of application components, as well as access to open technologies such as HTML and XML, and a host of other constructs to support an open application development and deployment environment.

And all the while, Progress 4GL-based applications can be brought from one release to the next largely without change. Progress provides a degree of compatibility and upward migration from one release to the next unmatched (unattempted, really) by any other high-level programming language.

About the sample database

This book uses one of the standard OpenEdge sample databases for its examples, the Sports2000 database. This is a simplified example of a database that might be used in a typical order entry application, with customers, orders, order lines, and other information for keeping track of customers and their orders. Some of the same example procedures and windows you will build are used and extended throughout the book as you are introduced to new programming concepts, but you will not be building a complete sample application. This is largely because, in the course of introducing so many separate language constructs and programming techniques, the book gives you more of a one-of-each experience that would not be typical of a standardized application. Once you've worked through the book, though, you will be able to use what you learn to build many kinds of comprehensive applications.

Getting started with the 4GL—the Progress snowplow

Have you ever learned to downhill ski?

The first thing the ski instructors teach you is how to do the snowplow by moving the tips of your skis close together so you can control your descent down the hill as you learn to keep your balance. (If you were *really* young when you learned, they probably talked to you about making a piece of pie, right?)

Then what is the *second thing* they teach you, after you can get down the hill safely? It is *never to snowplow again*, because you have to learn to ski with your skis *parallel* to ski well. The snowplow is just to get you started, to help you survive your first runs down the hill until you are ready to do things the right way.

This book uses a similar technique to show you how to use the Progress 4GL. The language and the tools that support it have evolved to the point where much of the work of building an application has been taken over by the development tools themselves. Because of this, the nature of the 4GL procedures you typically write has changed to take advantage of what the tools do for you, as well as to use the enhancements to the language that have appeared with each new release. So while it is important to learn the basics of how the language works and everything that you can do with it, once you understand these things you will want to let the OpenEdge tools generate much of your 4GL code for you, including many of the visual elements of the application and even some standard business logic.

One of the major development tools is the OpenEdge AppBuilder, a graphical design tool you can use both to build structured procedures for your 4GL code and to design and build windows, browses (or grids), menus, and the other parts of a GUI application. The AppBuilder is a code generator. When you lay out a window in the AppBuilder, it generates the 4GL code to define the window and all the controls that are in it. The AppBuilder can do a great deal more, defining not just graphical elements, but also the database access 4GL code needed to retrieve data from one or more database tables, display it, and update it, all without you having to write any code yourself at all.

Because the AppBuilder can generate this code for you, you won't have to write it yourself in the applications you develop. But you need to be aware of what that code looks like and what it does for you, so that you can understand and appreciate how the code you *do* write fills in the blanks where the tools don't do everything you need.

In Chapter 5, “[Examining the Code the AppBuilder Generates](#),” you’ll look at some of that AppBuilder-generated code as you learn how to define a graphical application. After that chapter, you won’t have reason to write that kind of code very often at all. This book shows you the basics of what the AppBuilder does and how to use it. Other manuals in the OpenEdge documentation set provide details on how to use the AppBuilder and related tools that are part of the OpenEdge Application Development Environment (ADE).

Another major tool is the Progress Dynamics™ development framework. Progress Dynamics is a comprehensive set of tools and supporting code to help you develop distributed, largely dynamic business applications. *Distributed* means that your applications can run worldwide, with a database server on one machine accessed by users working on many separate PCs or other workstations. They interact with the interface part of your application and exchange data with the server machine when it’s needed. *Dynamic* means that much of your application definition isn’t in procedure code at all. Rather, it is stored in a repository database that describes the whole user interface and even much of the logic of your application in abstract terms that can be presented to the user in different ways (for example as a Windows-style GUI interface or a Web-browser HTML interface) without any code changes to accomplish this.

Building a distributed application by yourself, let alone a *dynamic* distributed application, would definitely be the programming equivalent of a double-black-diamond ski run—a very advanced undertaking, not for the inexperienced or the faint of heart. But Progress Dynamics handles most of the work for you, generating application components that handle the large majority of the tasks your application requires. You are left to write 4GL procedures to define the specific business logic for your application, along with other customizations and extensions that you need. Because the framework does so much for you, much of the code you write in a snowplow application, such as you will create first in this chapter, won’t be the kind of code you write in a real application. For that reason, the first chapters of this book introduce some basic language concepts and constructs. Later chapters quickly move on to teach you to build code in the context of a real application.

There are several other books that teach you everything you need to know about the very powerful Progress Dynamics development environment. This book just provides you with enough guidance to help you build the things you need.

You can use what you learn from this book to build OpenEdge applications in a variety of ways, including using some of the newer constructs to extend and enhance an existing application written in an earlier version of the 4GL.

In the beginning...FOR EACH CUSTOMER

There's a prototypical Progress 4GL procedure often used as an example, one that couldn't be simpler, but that shows a lot about the power of the language. It's appeared on coffee mugs and tee shirts for two decades. Here it is:

```
FOR EACH Customer:  
    DISPLAY Customer.  
END.
```

You can't get much simpler than that, but it would take hours to explain in detail everything that this little procedure does for you. To summarize:

- **Customer** is the name of a table in the Sports2000 sample database that you'll connect to in a moment. The FOR EACH statement starts a block of code that opens a query on that database table and returns each record in the table, one at a time, in each iteration of the block.
- Each **Customer** record is displayed on the screen in turn. The code takes formatting and label information from the database schema definition and uses it to create a default display format for all the fields in the table. DISPLAY Customer means display all the fields in the table.
- As each record is displayed, the display moves down a row to display the next **Customer**. The effect is more like what you would see in a report rather than a browse or other grid control.
- The block of code—everything from the FOR EACH statement through the END statement—iterates once for each **Customer** record (hence the syntax FOR EACH). All the code in between (in this case just a DISPLAY statement) is executed for each customer retrieved.
- When the display gets to the bottom of the available display area, it automatically pauses, with a message prompting you to press the space bar to see the next set of rows.
- When you press the space bar, the display clears and a new set of rows appears.

- When Progress detects the end of the record set (all the **Customer** records in this case), it terminates the procedure with the message “Procedure complete. Press space bar to continue.”
- If you get tired of looking at Customers (there are several hundred in the table), you can press the **ESCAPE** key to terminate the procedure.

Starting your OpenEdge session

So where do you enter this little piece of code? First, you need to start a OpenEdge session, create and connect to a copy of the sample database the example uses, and then bring up the Procedure Editor.



To start your OpenEdge session:

1. From the Windows desktop **Start** menu, select the OpenEdge environment, using whatever name you gave to it when you installed it, and under that menu item, select the **Desktop** option. For example, select **Start→OpenEdge →Desktop**.

The **Application Development Environment (ADE) Desktop** window appears:



From here you can access all the basic OpenEdge development tools.

2. Choose the first icon, which looks like a book .

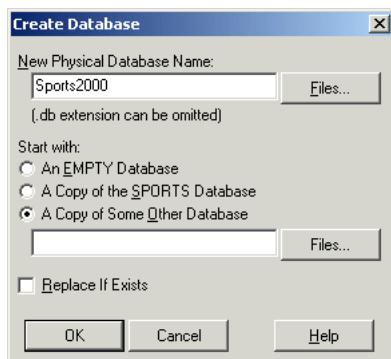
The **OpenEdge Data Dictionary Startup** dialog box opens. The Data Dictionary is where you define and manage all the tables and fields in your application database. To get you started, you can create your own copy of the Sports2000 database, which is the standard OpenEdge demo database. You need to copy the database so that your test procedures can make changes to it without modifying the version of it in your install directory.

3. In the **Dictionary Startup** dialog box, select the option to create a new database, then choose **OK**:

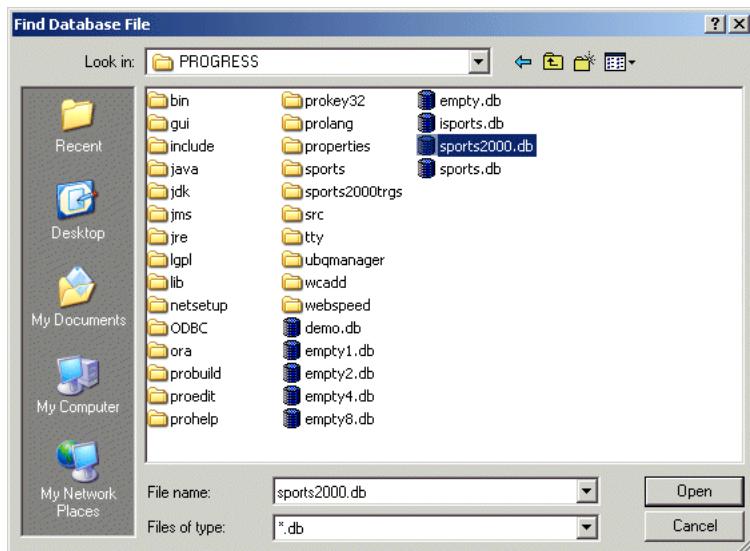


The **Create Database** dialog box appears and prompts you for the name of your copy of the database.

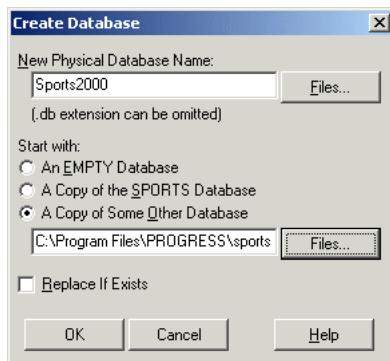
4. Type **Sports2000**. By default, OpenEdge creates a database named Sports2000 in your working directory. If you'd like the database to be somewhere else, you can choose the **Files** button next to the **New Physical Database Name** fill-in field to browse the directory structure for the right location for your database.
5. After you've entered your new database name, select the **Start with A Copy of Some Other Database** option, and then choose the **Files** button next to the fill-in field:



6. Locate the **Sports2000** database in your OpenEdge install directory, then double-click that entry or choose the **Open** button:



The pathname to the database is filled in when you return to the **Create Database** dialog box:



7. Choose **OK** to accept this pathname.

The **Connect Database** dialog box appears.

8. Make sure the **Physical Name** field shows **Sports2000** and the **Database Type** is Progress, then choose **OK**.

Because you created this database as part of the Data Dictionary startup, which needs to have a database connected before it lets you in, the Data Dictionary main window now opens. You can work with the Data Dictionary at a later time, but just to familiarize you with the database tables you'll be using in this chapter, go ahead and look at them here.



To familiarize yourself with the Sports2000 database tables:

1. Select the **Customer** table from the **Tables** list, then choose the **Fields** button  on the Dictionary's toolbar.

All the fields (or *columns*, to use the equivalent SQL terminology) in the table are shown in the **Fields** list to the right.

2. Scroll through the list to see all the fields in the table. You'll be displaying a few of these fields later, and using others to select a subset of the **Customers** for display.
3. Scroll down the list of **Tables**, then select the **Order** table.

You can see that, in addition to an **OrderNum** field, which gives each **Order** a unique number, there is also a **CustNum** field, which you will use in [Chapter 2, “Using Basic 4GL Constructs,”](#) to link, or join, the **Order** table to the **Customer** record with the same **CustNum** value.

There's a lot more to see in the Dictionary. If you want to wander around in the displays, go ahead, but just don't change anything! Any changes you make might affect steps later in this tutorial. If you make changes to tables or fields, this might invalidate records that are already in the sample database or keep some of your later example procedures from working.

4. To leave the Data Dictionary, select **Database→Exit** from the **Dictionary** menu.

Writing your first procedure

Your first Progress procedure will be a real snowplow exercise, something very simple just to get you started with the language. But before you enter this first procedure, you need to make an adjustment to the code—your first change!

There are a fair number of fields in the Customer table, so the resulting formatting of all the fields in a limited display area would be a mess. In fact, one of the fields, the **Comments** field, is so large that the Progress run-time engine (also known as the *interpreter*) will balk at displaying it without some guidance from you on how to format and position it.

So it's better to select just a few of the fields from the table to display, which is more manageable. Select the **CustNum** field, which is the unique customer number, the **Name** field, which is the customer name, and the **City** field, which is part of the customer's address.

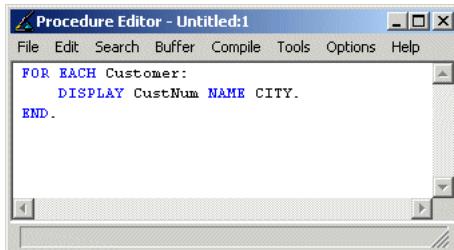
In the Progress 4GL, a list of fields is separated by spaces, so the new procedure becomes:

```
FOR EACH Customer:  
    DISPLAY CustNum Name City.  
END.
```



To enter this procedure in the Procedure Editor:

1. Return to the OpenEdge **Desktop** and choose the icon with the pen and inkwell .
2. In the **Procedure Editor** window that comes up, type your block of code:



You might notice the following about this code:

- The words FOR and EACH are automatically uppercased as you type them, and colored blue as well. This is because the Procedure Editor uses an intelligent edit control that recognizes Progress 4GL syntax. It not only does color-coding, but can also do autocompletion of common syntax, and can accept abbreviations of common language statements, to help speed up your programming. The editor also recognizes the FOR EACH statement as being the beginning of a block of code and automatically indents the DISPLAY statement, and then unindents the END statement to match the beginning of the block.
- You can type the table name **Customer** with a capital C, or type it in lower-case, or all in capitals, or however you wish. Progress is entirely case-insensitive when it comes to both language keywords and database table and field names. Uppercasing the keywords is just a convention to aid in code readability, but you could type **for each** or **For Each** as well and it would work just the same.
- Typing **CustNum** in mixed case is just a convention to aid readability.
- When you type the field name **Name**, it is uppercased and colored blue. This is because NAME is also a keyword in the 4GL. Progress is relatively forgiving in this way: if the Progress compiler, which processes the code you type in, can distinguish between a word that is a field name or table name and one that must be a keyword, it will let you use it that way. It's a good idea to avoid using keywords as table or field names in your database definition, though. For a list of all the Progress 4GL keywords, see Volume III of the *OpenEdge Development: Progress 4GL Reference*.

► To execute your procedure, press the **F2** key on your keyboard. This is the keyboard shortcut for the **RUN** command. Alternatively, you can select **Compile→Run** from the **Editor** menu.

The Procedure Editor shows the first set of customer numbers, names, and cities:

A screenshot of a Windows application window titled "Procedure Editor - Run". The window contains a table with three columns: "Cust Num", "Name", and "City". The data is as follows:

| Cust Num | Name | City |
|----------|----------------------|-------------|
| 1 | Lift Tours | Burlington |
| 2 | Upon Frisbee | Oslo |
| 3 | Hoops | Atlanta |
| 4 | Go Fishing Ltd | Harrow |
| 5 | Match Point Tennis | Boston |
| 6 | Fanatical Athletes | Montgomery |
| 7 | Aerobics valine Ky | Tikkurila |
| 8 | Game Set Match | Deatsville |
| 9 | Pihtiputaan Pyora | Pihtipudas |
| 10 | Just Joggers Limited | Ramsbottom |
| 11 | Keilailu ja Biljardi | Helsinki |
| 12 | Surf Lautaveikkoiset | Salo |
| 13 | Biljardi ja tennnis | Mantsala |
| 14 | Paris St Germain | Paris |
| 15 | Hoopla Basketball | Egg Harbor |
| 16 | Thundering Surf Inc. | Coffee City |
| 17 | High Tide Sailing | Immingham |
| 18 | Antin Metsastysase | Sysma |
| 19 | Buffalo Shuffleboard | Buffalo |
| 20 | Espoon Pallokeskus | Espoo |

At the bottom of the window, there is a message: "Press space bar to continue." with a small icon next to it.

► To see the next pages of **Customer** numbers, names and cities, press the **SPACE BAR** a few times. Press **ESCAPE** to terminate the display. Otherwise, the display terminates after the last page of **Customers** has been displayed.

This little three-line procedure certainly does a lot for you, but it doesn't produce an output display that you would put directly into an application. There are many qualifiers to the **DISPLAY** keyword in the statement that you can use to adjust its behavior and appearance, some of which you'll see later on in this chapter. Fundamentally, the **DISPLAY** statement is not intended to produce polished output by itself for a graphical application. There are other statements in the language you can use to define the windows, browses, toolbar buttons, and other features of a complete user interface. You'll get to know those in later chapters.

For the purposes of your introduction to the language, leave the display format as it is, so that you can see how the structure of the language itself is put together. That's the snowplow part of this exercise. You'll complete a little test procedure here that shows you a lot about the Progress 4GL, but which doesn't represent the way you will write really finished applications (just as snowplowing shows you a lot about skiing but doesn't represent how you will actually ski when you're an expert). To do that, you work together with the development tools to let the tools do a lot of the work of putting together the user interface for you. After only a few chapters, you'll be building complete application windows and all the logic that goes on behind them.

Basic characteristics of the Progress 4GL

You just wrote about the simplest starting procedure possible. What can you learn from it already?

The 4GL is procedural

Progress is a procedural programming language. That means that you write sets of language statements that you can save in individual, named procedures. The language statements are usually executed or processed in the order in which they appear in the procedure. In a simple procedure such as this one, the statements are executed as they appear. As you move further into building event-driven applications, where the user has a variety of ways to control the application by choosing buttons or making menu selections, you will learn about how to define trigger code that sets up blocks of statements to be executed later on when a particular action occurs.

The 4GL is block-structured

A Progress 4GL procedure is made up of blocks. The procedure itself is the main block of the procedure. There are a number of ways to define other blocks within the main procedure block. The FOR EACH statement and its matching END statement are one example of a nested block, in this case one that iterates through a set of database records and executes all the code in between for each record in the set. There are other block statements you can use for different purposes. Some of them are also iterating, and cause the block to be executed multiple times. Others simply define a set of statements to be executed together. You'll learn about all of these in later chapters.

A 4GL procedure consists of statements

A Progress 4GL procedure is made up of a sequence of language statements. Each statement has one or more 4GL keywords, along with other tokens such as database field names or variable names. A 4GL statement normally ends with a period. By convention, a statement that defines the beginning of a block, such as your FOR EACH Customer statement, can end with a colon instead. There's no significance to the end of the line in Progress. A single statement can span multiple lines, and if you wish, there can be multiple statements on a single line. There's no special syntax needed to break up a statement between lines, but if you have a single element of a statement, such as a quoted string, that is long enough that you need to make it span lines, then end the first line with a tilde character (~).

There must be at least one space (or other white space character such as a tab or new line) in between each token in the statement. Progress is not sensitive to additional white space. You can indent lines of code and put tabs between parts of a statement for readability, and none of this will affect how the statement executes.

Progress is case-insensitive. This means that you can write either a 4GL keyword or a reference to a variable name or database table or field name in any combination of uppercase and lowercase letters. As you have already seen, the intelligent Progress edit control defaults to a convention of capitalizing 4GL keywords. For a list of all the Progress keywords, you can look through the index in the Help Topics submenu of the Help menu (via any of the OpenEdge development tools).

When you define variables in a procedure, and when you define database tables and fields, you must give them names that begin with a letter. After that, the name can include letters, numeric digits, periods (.), hyphens (-), and underscores (_). When you save your 4GL procedure, you give it a name as well, and this follows the same naming rules. Note that, although the 4GL itself has many hyphenated keywords, you should *not* use hyphens in database table and field names, as well as procedure and variable names in your application. You should follow this guideline because other programming languages that you might want to combine with 4GL procedures often do not allow a hyphen as a character in a procedure or parameter name. And the SQL database access language, which you can also use to manage data in your OpenEdge database using other tools, does not allow hyphens in table and field names.

A good naming convention for all kinds of names in 4GL procedures is to use mixed case (as with the **CustNum** field) to break up a name into multiple words or other identifiable parts, using the capital letters to identify the beginning of each word or subpart within the name to improve readability of your code.

The 4GL combines procedural, database, and user interface statements

There are three basic kinds of statements in a 4GL program: procedural statements, database access statements, and user interface statements. Sometimes individual statements contain elements of all three. Your first simple procedure contains all three types, and illustrates the power of the language.

The `FOR EACH` statement itself can be considered procedural, because it defines an action within the program, in this case, repeating a block of statements up to the `END` statement.

But the `FOR EACH` statement is also a database access statement, because it defines a result set that the `FOR EACH` block is going to iterate over, in this case, the set of all **Customer** records in the database. This simple convention of combining procedural logic with database access is a fundamental and enormously powerful feature of the Progress 4GL. In another language, you would have to define a database query using a syntax that is basically divorced from the procedural language around it, and then have additional statements to control the flow of data into your procedure. In the 4GL, you can combine all these elements in a way that is very natural, flexible, and efficient, and relates to the way you think about how the program uses the data.

The `FOR EACH` block is also powerful because it transparently presents each record to your program in turn, so that the rest of the procedural logic can act on it. If you've written applications that use the SQL language for data retrieval, you can compare the Progress `FOR EACH` block with a program in another language containing embedded SQL statements, where the set-based nature of the SQL `SELECT` statement is not a good match to the record-by-record way in which your program normally wants to interact with the data.

The `DISPLAY` statement shows that user interface statements are also closely integrated with the rest of the program. Progress contains language statements not only to display data, but also to create, update, and delete records, and to assign individual values. All of these statements are fully integrated with the procedural language. In later chapters, you'll learn how to build your applications so that the procedures that control the user interface are cleanly separated from the procedures that manage the data and define your business logic.

Saving your test procedure

Before you move on to extend your first procedure to learn some of the other 4GL language basics, you should save it out to the operating system. To do this, give your code a procedure name. This follows the same rules as other names, and by convention has a filename extension of .p (for procedure).

- ▶ To save your procedure in the Procedure Editor, select **File**→**Save As**. Call the procedure `h-CustSample.p` and save the file to your working directory, or some other directory you've created.

Now you're ready to extend your procedure in various ways.

Using Basic 4GL Constructs

In Chapter 1, “Introducing the Progress 4GL,” you learned some of the basic characteristics of the Progress 4GL and created a small test procedure. In this chapter you will extend your test procedure by:

- Refining the data selection with a WHERE clause
- Creating nested blocks to display related data
- Changing labels and formats
- Using program variables and data types
- Defining an IF-THEN-ELSE decision point
- Putting a calculation into your procedure
- Using the Intelligent Edit Control and its shortcuts
- Getting to online help
- Saving and compiling your test procedure

Refining the data selection with a WHERE clause

So far you've been selecting all the **Customer** records in the database. Now you'll refine that selection to show only those **Customers** from the state of New Hampshire. There is a **State** field in the **Customer** table that holds the two-letter state abbreviation for states in the USA.

The Progress 4GL supports a **WHERE** clause in any statement that retrieves data from the database, which will be familiar to you if you have used SQL or other similar data access languages. The **WHERE** keyword can be followed by any expression that identifies a subset of the data. You'll learn a lot more about this in later chapters, but for now a simple expression is all you need.



To refine the data selection in your test procedure:

1. Add the following **WHERE** clause to the end of your **FOR EACH** statement:

```
FOR EACH Customer WHERE State = "NH":
```

2. Press **F2** to see the reduced list of **Customers**. The list should now use only a bit more than a page.
3. Add a sort expression to the end of your **WHERE** clause to sort the **Customers** in order by their **City**. The 4GL uses the keyword **BY** to indicate a sort sequence:

```
FOR EACH Customer WHERE State = "NH" BY City:
```

4. Press **F2** to run the procedure again to see the effects of your change.

Comparison operators

The equal sign is just one of a number of comparison operators you can use in 4GL expressions. [Table 2–1](#) provides a complete list.

Table 2–1: Comparison operators

| Keyword | Symbol | Explanation |
|----------|----------------|--|
| EQ | = | Equal to. |
| NE | <> | Not equal to. |
| GT | > | Greater than. |
| LT | < | Less than. |
| GE | >= | Greater than or equal to. |
| LE | <= | Less than or equal to. |
| BEGINS | Not applicable | A character value that begins with this substring. |
| MATCHES | Not applicable | <p>A character value that matches this substring, which can include wild card characters.</p> <p>The expression you use to the right of the MATCHES keyword can contain the wild card characters:</p> <ul style="list-style-type: none"> • An asterisk (*) represents one or more missing characters. • A period (.) represents exactly one missing character. |
| CONTAINS | Not applicable | <p>A database text field that has a special kind of index called a WORD-INDEX.</p> <p>The WORD-INDEX indexes all the words in a field's text strings, for all the records of the table, allowing you to locate individual words or associated words in the database records, much as you do when you use an Internet search engine to locate text in documents on the web.</p> |

Using quotation marks

You can use single quotation marks (‘) and double quotation marks (“) interchangeably to define a string constant. You must balance them properly, using the same type of quotation mark at the beginning and the end of the string. Use double quotes if your string contains a single quote, and vice versa, as in the following example:

```
DISPLAY "Always using the same type of quotes!".
DISPLAY 'Like this.'.
DISPLAY "But never like this!'.
```

Creating nested blocks to display related data

To review, the FOR EACH statement in your procedure creates a code block nested inside the implicit main block of the procedure itself. Now you will create yet another block nested inside of that, to display the **Order** records in the database for each New Hampshire **Customer**.

- To create a nested block, add another FOR EACH block inside the one you have, so that your procedure looks like this:

```
FOR EACH Customer WHERE State = "NH" BY City:
  DISPLAY CustNum Name City.
  FOR EACH Order OF Customer:
    DISPLAY OrderNum OrderDate ShipDate.
  END.
END.
```

This example shows the code indented so that the new block is visually nested in the outer block, which helps code readability. The intelligent editor should help you with this; if it doesn't get it quite right, make the effort to indent the code properly yourself.

First, look at the new FOR EACH statement. The keyword OF is a shorthand for a WHERE clause that joins the two tables together. When you looked at the two tables and their fields in the Dictionary, you saw that both tables have a **CustNum** field. This is the primary key of the **Customer** table, which means that each **Customer** is assigned a unique number for the **CustNum** field, and this is the primary identifier for the **Customer**. In the **Order** table, the **OrderNum** is the unique **Order** identifier, and its primary key. The **CustNum** field in the

Order table points back to the **Customer** the **Order** is for. It's a foreign key because it points to a record in another table. To retrieve and display the **Orders** for a **Customer**, you have to join the two tables on the **CustNum** field that they have in common. The full WHERE clause for this join would be: WHERE Customer.CustNum = Order.CustNum. This kind of syntax will be familiar to you if you've ever worked with SQL.

The WHERE clause is telling Progress to select those records where the **CustNum** field in one table matches the **CustNum** field in the other. In order to tell Progress which field is which, both are qualified by the table name, followed by a period.

In the Progress 4GL, you can use the syntax **Order OF Customer** as a shorthand for this join if the two tables have one or more like-named fields in common that constitute the join relationship, and those fields are indexed in at least one of the tables (normally they should be indexed in both). You can always use the full WHERE clause syntax instead of the OF phrase if you wish; the effect is the same, and if there is any doubt as to how the tables are related, it makes the relationship your code is using completely clear. In fact, the OF phrase is really one of those beginner shortcuts that makes it easy to write a very simple procedure but which really isn't good practice in a complex application, because it isn't clear just from reading the statement which fields are being used to relate the two tables. You should generally be explicit about your field relationships in your applications.

These simple nested FOR EACH statements accomplish something that would be a lot of work in other languages. To retrieve the **Customers** and their **Orders** separately, as you really want to do, you would have to define two separate queries using embedded SQL syntax, open them, and control them explicitly in your code. This would be a lot of work. For example, the straight forward single SQL query to retrieve the same data would be:

```
SELECT Customer.CustNum, Name, City, OrderNum, OrderData, ShipDate FROM  
Customer, ORDER WHERE State = "NH" AND Customer.CustNum = Order.CustNum;
```

This code would retrieve all the related **Customers** and **Orders** into a single two-dimensional table, which is not very useful: all the **Customer** information would be repeated for each of the **Customer's Orders**, and you would have to pull it apart yourself to display the information as header and detail, which is probably what you want.

By contrast, when you run your very simple Progress 4GL procedure, you get a default display that represents the data properly as master (**Customer**) and detail (**Order**) information, as shown in [Figure 2–1](#).



Figure 2–1: Result of running simple sample procedure

Progress automatically gives you two separate display areas, one for the **Customer** fields showing one **Customer** at a time, and one for the **Orders** of the **Customer**. These display areas are called *frames*. You'll learn more about Progress frames in later chapters.

Unlike in the first example, which displays a whole page of **Customers**, each time you press the **SPACE BAR**, Progress displays just the next **Customer** and its **Orders**. Why did Progress do this? The nested **FOR EACH** blocks tell Progress that there are multiple **Orders** to display for each **Customer**, so it knows that it doesn't make sense to display more than one **Customer** at a time. So it creates a small frame just big enough to display the fields for one **Customer**, and then separately creates another frame where it can display multiple **Orders**. The latter frame is called a *down frame*, because it can display multiple rows of data as it moves down the page. The top frame is actually referred to as a *one down frame* because it displays only a single row of data at a time.

You can control the size of the frames, how many rows are displayed, and many other attributes, by appending a **WITH** phrase to your statement.



To see how the WITH CENTERED phrase can affect your test procedure:

1. Add the words **WITH CENTERED** to the DISPLAY statement for the **Order** frame:

```

FOR EACH Customer WHERE State = "NH" BY City:
  DISPLAY CustNum Name City.
  FOR EACH Order OF Customer:
    DISPLAY OrderNum OrderDate ShipDate WITH CENTERED.
  END.
END.

```

2. Run the procedure again. The **Order** frame is centered in the default display window:



There are lots of frame attributes you can specify here (for a description of them, read the section on the **Frame Phrase** in the *OpenEdge Development: Progress 4GL Reference*). This book doesn't tell you much more about them, either now or later, because you won't use most of them in your GUI applications. These attributes are designed to help you define frames for a character mode application, where the interface is basically just a series of 24 or 25 lines of 80 characters each. For this kind of display format, a sequence of frames displayed one beneath the other is an appropriate and convenient way to lay out the screen. But in a GUI application, you instead lay out your display using a visual design tool, such as the OpenEdge AppBuilder, and it generates the code or data needed to create the user interface at run time. So this is another part of the snowplow exercise: this chapter shows you the basics of how the 4GL works and how it was designed, even though you will do most of your work a different way in your new applications.

Changing labels and formats

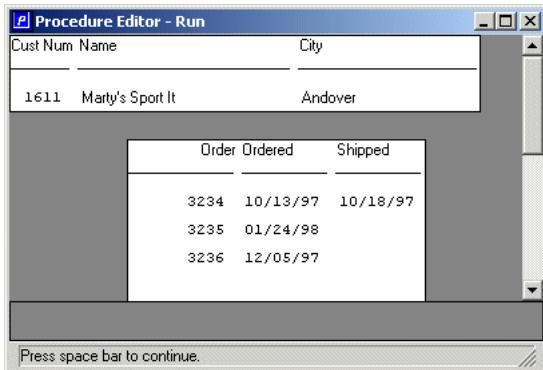
The default label for the **Order Number** field is **Order Num** (you define default labels in the Data Dictionary when you set up your database), and you might prefer that it just say **Order**. Also, the default display format for the **ShipDate** field is different from the format for the **OrderDate** field: one has a four-digit year and the other a two-digit year. You can change labels and default formats in your DISPLAY statement. If you add the **LABEL** keyword or the **FORMAT** keyword after the field name, followed by a string for the value you'd prefer, then the display changes accordingly.

- ▶ To make such changes to your test procedure, change the **OrderNum** **LABEL** to **Order** and the **ShipDate** **FORMAT** to **99/99/99**.

In the following example, each field has its own line in the code block, to make the code easier to read, and to emphasize doing that doesn't change how the procedure works. Everything up to the period is one 4GL statement:

```
FOR EACH Customer WHERE State = "NH" BY City:
  DISPLAY CustNum NAME City.
  FOR EACH Order OF Customer:
    DISPLAY OrderNum LABEL "Order"
      OrderDate
      ShipDate FORMAT "99/99/99" WITH CENTERED.
  END.
END.
```

- ▶ To view the results of these changes, **Run** your test procedure again:



Using program variables and data types

Like most programming languages, the Progress 4GL lets you define program variables for use within a procedure. Here is the basic syntax:

```
DEFINE VARIABLE varname AS datatype.
```

In this syntax, *varname* is the name of the variable, which must conform to the same rules as other names in the 4GL. (See the “[Variable naming conventions](#)” section on page 2–13 for details.)

Progress supports a range of data types. [Table 2–2](#) lists the basic ones. There are some other special data types available for more advanced programming, but these are enough to get you started.

Table 2–2: Basic supported data types

| Data type name | Default display format | Default initial value |
|----------------|------------------------|--|
| CHARACTER | X(8) | "" (the empty string) |
| DATE | 99/99/99 | ? (the Unknown value, which displays as blank for dates) |
| DECIMAL | ->>,>>9.99 | 0 |
| HANDLE | >>>>>9 | ? (the Unknown value) |
| INTEGER | ->,>>,>>9 | 0 |
| LOGICAL | yes/no | no |

Here are a few notes on Progress data types:

- All CHARACTER data in Progress variables, and in database fields in an OpenEdge database, is stored as variable length strings. You do not need to define the length of a variable, only its display format, which indicates how many characters (at most) are displayed to the user. The default display length for CHARACTER fields is 8 characters. The X symbol represents any printable character, and the 8 in parentheses effectively repeats that character, so that X(8) is the same as XXXXXXXX. The default value for a CHARACTER variable is the empty string.

- You can display dates in a variety of ways, including two- or four-digit years. To get a four-digit year, specify 99/99/9999 as your format. Note that changing the display format for a date does not change how it is stored in the database. Dates are stored in an internal format that can be converted to any of the display formats available to you.
- The DECIMAL data type supports a total of 50 digits of which up to 10 can be to the right of the decimal point. When you define a DECIMAL variable you can specify the number of positions to the right of the decimal point by adding the qualifier DECIMALS <n> to the DEFINE VARIABLE statement.
- Progress supports the automatic conversion of both DATEs and DECIMALs to the formats generally used in Europe and other parts of the world outside the US. If you use the European (-E) startup option, then all period or decimal characters in DECIMAL formats are converted to commas, and commas are converted to periods. In addition, the default DATE display becomes DD/MM/YY or DD/MM/YYYY instead of MM/DD/YY or MM/DD/YYYY.
- The HANDLE data type is used to store a pointer to a structure that represents a running Progress procedure or an object in a procedure such as a field or button. [Chapter 18, “Using Dynamic Graphical Objects,”](#) and [Chapter 19, “Using Dynamic Queries and Buffers,”](#) provide more information about how to use HANDLE variables.
- The maximum size of an INTEGER variable is 2G (slightly over a billion digits).
- A LOGICAL variable represents a yes/no or true/false value. You can specify any pair of literals you wish for the TRUE and FALSE values that are displayed to the user for a LOGICAL variable, with the TRUE or YES value first. However, it is not advisable to use LOGICALS to represent data values that aren’t really logical by nature, but which simply happen to have two valid values, such as Male/Female, because it might be unclear which of those the TRUE value represents.
- You will learn about the Progress Unknown value in the [“Using the Progress Unknown value”](#) section on page 2–16. The default display format for data that has the Unknown value is blank; for other data types it is a question mark.

Defining formats

The list of default formats for different data types introduced you to some of the format characters supported by Progress. [Table 2–3](#) provides a quick summary of the format symbols you’re most likely to use.

Table 2–3: Common format symbols

(1 of 2)

| This format character . . . | Represents . . . |
|-----------------------------|--|
| X | Any single character. |
| N | A digit or a letter. A blank is not allowed. |
| A | A letter. A blank is not allowed. |
| ! | A letter that is converted to uppercase during input. A blank is not allowed. |
| 9 | A digit. A blank is not allowed. |
| (n) | A number that indicates how many times to repeat the previous format character. |
| > | A leading digit in a numeric value, to be suppressed if the number does not have that many digits. |
| Z | A leading digit in a numeric value, to be replaced by a blank if the number does not have that many digits. |
| * | A leading digit in a numeric value, to be displayed as an asterisk if the number does not have that many digits. |
| , | A comma in a numeric value greater than 1,000. This is replaced by a period in European format. It is suppressed if it is preceded by a Z, *, or >, and the number does not have enough digits to require the comma. |
| . | A decimal point in a numeric value. This is replaced by a comma in European format. |

Table 2–3: Common format symbols

(2 of 2)

| This format character . . . | Represents . . . |
|-----------------------------|---|
| + | A sign for a positive or negative number. It is displayed as + for a positive number, and a – for a negative number. |
| – | A sign for a negative number. It is displayed as a – for a negative number. For a positive number it is suppressed if it is to the left of the decimal point in the format, and replaced by a blank if it is to the right of the decimal point. |

You can insert other characters as you wish into formats, and they are displayed as literal values. For example, the INTEGER value 1234 with the FORMAT \$>,>>>ABC is displayed as **\$1,234ABC**.

Other variable qualifiers

You can qualify a variable definition in a number of ways to modify these defaults. To do this, append one of the keywords listed in [Table 2–4](#) to the end of your variable definition.

Table 2–4: Variable qualifiers

| Keyword | Followed by |
|--------------|---|
| INITIAL | The variable's initial value. |
| DECIMALS | The number of decimal places for a DECIMAL variable. |
| FORMAT | The display format of the variable, enclosed in quotation marks. |
| LABEL | The label to display with the variable. (The default is the variable name itself.) |
| COLUMN-LABEL | The label to display with the variable when it is displayed as part of a column of values. (The default is the LABEL if there is one, otherwise the variable name.) |
| EXTENT | An integer constant. This qualifier allows you to define a variable that is a one-based array of values. You can then reference the individual array elements in your code by enclosing the array subscript in square brackets, as in myVar[2] = 5 . |

As an alternative to specifying all of this, you can use the following syntax form to define a variable that is LIKE another variable or database field you've previously defined:

```
DEFINE VARIABLE varname LIKEfieldname
```

In this case it inherits the format, label, initial value, and all other attributes of the other variable or field. This is another strength of the Progress 4GL. Your application procedures can inherit many field attributes from the database schema, so that a change to the schema is propagated to all your procedures automatically when they are recompiled. You can also modify those schema defaults in individual procedures.

There's one more keyword that you should almost always use at the end of your variable definitions, and that is NO-UNDO. This keyword has to do with how Progress manages transactions that update the database. When you define variables in your 4GL programs, Progress places them into two groups. Those that don't have the NO-UNDO qualifier are treated as though they were a database record with those variables as fields. If any of the variable values are modified during the course of a database transaction, and then the transaction is backed out, changes to the variable values made during the transaction are backed out as well, so that they revert to their values before the transaction started. This can be very useful sometimes, but in practice, most variables do not need this special treatment, and because Progress has to do some extra work to manage them, it is more efficient to get into the habit of appending NO-UNDO to the end of your variable definitions unless the variable is one that should be treated as part of a transaction. This places them into a second group where Progress manages them without the transaction support.

Variable naming conventions

There are no specific naming requirements for variables, but there are some recommended guidelines that will bring your own variables into line with the standards used in the OpenEdge development tools and their support code.

You should begin a variable with a lowercase letter (or sometimes two) to indicate the data type of the variable. This can help readers of your code to understand at a glance how a variable is being used. When you start doing more dynamic programming later on, it is very important to differentiate between a variable that represents a value directly, and one that is a handle to an object that has a value. Here are some recommended data type prefixes:

- **c** for CHARACTER
- **i** for INTEGER

- **d** or **de** for DECIMAL
- **da** for DATE
- **h** for HANDLE
- **I** for LOGICAL

The rest of the name should be in mixed case, where capital letters are used to identify different words or subparts of a name, as in **cCustName** or **iOrderCount**.

Placement of variable definitions

Where you place variables in your code also makes a difference. Progress does a single pass through the statements in a procedure to build the intermediary code that it uses to execute the procedure. Because a variable is a definitional element of a procedure and not a statement that is executed in sequence, it does not really matter where the variable definition appears.

However, because of the one-pass nature of the Progress syntax analyzer, the definition has to appear before the variable is used in the procedure. By convention, it is usually best for you to define all your variables at the top of a procedure, to aid in readability and to make sure that they're all defined before they're used.

For the next change to the test procedure, you will put to work several of the concepts you've just learned about. You will display a special value for each **Order** record. This task involves defining a variable with an initial value, writing an **IF-THEN** construct with a check for the Unknown value, and then using one of the many built-in Progress 4GL functions to extract a value to display. The value you will display is the **ShipDate** month of an **Order** expressed as a three-character abbreviation, such as JAN or FEB.

► To build up the list of possible values for the month, you need to define a CHARACTER variable to hold the list. Add the following variable definition to the top of your procedure:

```
DEFINE VARIABLE cMonthList AS CHARACTER NO-UNDO  
    INIT "JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC".
```

Defining an IF-THEN-ELSE decision point

You'll notice when you run your procedure that some of the ship dates aren't defined. Some of these **Orders** apparently have been on hold for a very long time! Let's construct a statement to branch in the code depending on whether the **ShipDate** field is defined or not. The 4GL, like most languages, has an IF-THEN construct for such decision points. It can also have an ELSE branch:

```
IF condition THEN { block | statement } [ ELSE { block | statement } ]
```

The *condition* is any expression that evaluates to true or false. Following the THEN keyword, you can place a single 4GL statement, or a whole set of statements that are all to be executed if the condition is true. The way to represent a block that simply groups a series of statements together that are all executed together is a DO-END block:

```
IF condition THEN  
DO:  
    statement  
    .  
    .  
    .  
END.
```

If you include the ELSE keyword followed by a statement or a block of statements, that branch is taken if the *condition* evaluates to false.

Using the Progress Unknown value

Although the **ShipDate** is displayed as being blank in these cases, the value stored in the database isn't a blank value, because this is a date, not a character value. If you were to add an expression to your code to compare the **ShipDate** to a blank value, you would get your very first Progress 4GL compiler error:

```
FOR EACH Customer WHERE State = "NH" BY City:  
    DISPLAY CustNum NAME City.  
    FOR EACH Order OF Customer:  
        IF ShipDate NE "" THEN  
            DISPLAY OrderNum LABEL "Order"  
                OrderDate  
                ShipDate FORMAT "99/99/99" WITH CENTERED.  
        END.  
    END.
```

Whenever you press the **F2** key to run your procedure, Progress performs a syntax validation. If it can't properly process the code you've written, you get an error such as this one. In this case, Progress sees that you're trying to compare a field defined to be a date with a character string constant, and these don't match. To correct this, you need to change the nature of your comparison.

The value stored in the database to represent a value that isn't defined is called the Unknown value. In 4GL statements you represent the Unknown value with a question mark (?). Note that you don't put quotation marks around the question mark; it acts as a special symbol on its own. It is not equal to any defined value. In the 4GL, you write a statement that looks as though you are comparing a value to a question mark, such as `IF ShipDate = ?`, but in fact the statement is asking if the **ShipDate** has the Unknown value, which means it has no particular value at all. The Unknown value is like the NULL value in SQL, and the expression `IF ShipDate = ?` is like the SQL expression `IF ShipDate IS NULL`.

Using built-in 4GL functions

To complete this latest change to the procedure, you need to define the statement that checks whether the **ShipDate** is Unknown, and then picks out the month from the date, using the **cMonthList** variable you defined just above, and converts it to one of the three-letter abbreviations in the list. Here is the whole statement:

```
IF ShipDate NE ? THEN
DISPLAY ENTRY(MONTH(ShipDate), cMonthList) LABEL "Month".
```

Now take a closer look at the elements in the DISPLAY statement. First there is a new keyword, ENTRY. This is the name of a built-in function in the 4GL. There are many such functions to do useful jobs for you, to save you the work of writing the code to do it yourself. The ENTRY function takes two arguments, and as you can see, those arguments are enclosed in parentheses. The first is an INTEGER value, which identifies an entry in a comma-separated list of character values. In this case it represents the month of the year, from 1 to 12. The second argument is the list that contains the entry the function is retrieving. In this case it is the variable you just defined.

Looking closer, you can see that the first of the two arguments to the function, MONTH(ShipDate), is itself another function. This function takes a Progress DATE value as an argument, extracts the month number of the date, and returns it. The returned value is an INTEGER from 1 to 12 that the ENTRY function then uses to pick out the right entry from the list of months in **cMonthList**. So if the month is May, the MONTH function returns 5 and the ENTRY function picks out the fifth entry from the list of months and returns it to the DISPLAY statement.

Here are some general observations about built-in functions:

- A function takes a variable number of arguments, depending on what the function requires. Some functions take no arguments at all (for example, the TODAY function, which returns today's date). Some functions have a variable number of arguments, so that one or more arguments at the end of the argument list are optional. For example, the ENTRY function can have an optional third argument, which is a character string representing a delimiter to use between the values in the list, if you don't want it to use a comma (,). Because the comma is the default delimiter, it is optional. You can't leave out arguments from the middle of the list, or specify them in a different order. Each of the arguments must be of the proper data type, depending on what the function expects.
- The arguments to a function can be constant values, variable names, database field names, or any expression involving these which evaluates to a value of the proper data type.

- Each function returns a value of a specific data type.
- You can nest functions to any depth in your code. The result of any function is returned up to the next level in the code.
- You can place functions anywhere within a statement where a value can appear. Because a function *returns* a value, it can appear only on the right-handside of an assignment statement. You can't use the MONTH function to assign the month value to a date variable, for example, or the ENTRY function to assign the value of an entry in a list. There are 4GL statement keywords in some cases to do those kinds of assignments.
- If you are displaying the result of a function or an expression involving a function, you can specify a LABEL or FORMAT for it, just as you can for a variable. The default LABEL for an expression is a text string representing the expression itself. The default format is the default for the function's data type. In this example you should add the label **Month** to the expression.

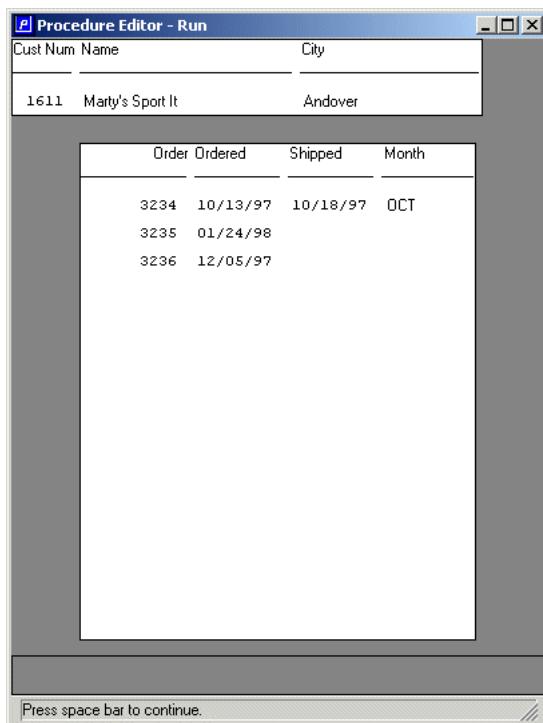


To display the month along with each Order:

1. Add the new statement with the function references into your procedure, inside the block of code that loops through the **Orders**:

```
DEFINE VARIABLE cMonthList AS CHARACTER NO-UNDO
  INIT "JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC".
FOR EACH Customer WHERE State = "NH" BY City:
  DISPLAY CustNum NAME City.
  FOR EACH Order OF Customer:
    IF ShipDate NE "" THEN
      DISPLAY OrderNum LABEL "Order"
      OrderDate
      ShipDate FORMAT "99/99/99" WITH CENTERED.
    IF ShipDate NE ? THEN
      DISPLAY ENTRY(MONTH(ShipDate), cMonthList) LABEL "Month".
    END.
  END.
```

2. To see the effect of the new code, rerun the procedure:



Note: Several separate 4GL DISPLAY statements contribute to the display of fields in a single line for each **Order**. This is one of the powerful and flexible characteristics of the 4GL. Progress can gather together a number of different statements in a procedure, some of which might be executed conditionally, and combine them together into a single operation such as this. This feature is generally not possible with other programming languages.

3. To save your procedure, press **F6**.

Progress 4GL Functions

This section provides a quick summary of some of the most useful functions available to you in the 4GL:

- [Date functions](#)
- [List functions](#)
- [4GL string manipulation functions](#)

Other sections of the book look at functions that convert data values and otherwise operate as part of data retrieval.

Date functions

[Table 2–5](#) describes functions that return or operate on data values.

Table 2–5: Date functions

| Function | Arguments | Returned value |
|----------|-----------|--|
| DAY | DATE | INTEGER — The day of the month. |
| MONTH | DATE | INTEGER — The month of the year. |
| YEAR | DATE | INTEGER — The year. |
| WEEKDAY | DATE | INTEGER — The day of the week, starting with 1 for Sunday. |
| TIME | none | INTEGER — The number of seconds since midnight. |
| TODAY | none | DATE — Today's date. |

Functions that convert data are discussed in later chapters. An example of one for now is the STRING function.



To see how you can use the STRING function to convert the time into a meaningful display:

1. In the Procedure Editor, select **File→New Procedure** to open a new procedure window.
2. Enter the following code, which applies the STRING function to the result of the TIME function, along with the special formatting characters HH (hour), MM (minute), and SS (second):

```
DISPLAY STRING(TIME, "HH:MM:SS").
```

3. Press F2:



List functions

The functions described in [Table 2–6](#) operate on lists. All the list functions are one-based, that is, the first element in a list is considered to be element 1, not 0.

Table 2–6: List functions

| Function | Arguments | Returned value |
|----------|--|--|
| ENTRY | <i>element</i> AS INTEGER, <i>list</i> AS CHAR, <i>delimiter</i> AS CHAR | CHARACTER — The <i>n</i> th element in a delimited list, where <i>n</i> is the value of the <i>element</i> argument. The <i>delimiter</i> is optional and defaults to comma. |
| LOOKUP | <i>element</i> AS CHAR, <i>list</i> AS CHAR, <i>delimiter</i> as CHAR | INTEGER — The numeric (one-based) location of the <i>element</i> within the list. The <i>delimiter</i> is optional and defaults to a comma. The function returns 0 if the <i>element</i> is not in the list. |

4GL string manipulation functions

The functions described in [Table 2–7](#) operate on character strings.

Table 2–7: 4GL string manipulation functions

(1 of 2)

| Function | Arguments | Returned value |
|-----------|--|--|
| FILL | <i>expression AS CHAR,</i> <i>repeat-count AS INTEGER</i> | CHARACTER — A character string made up of the <i>expression</i> repeated <i>repeat-count</i> times. |
| INDEX | <i>source AS CHAR, target AS CHAR, starting-point AS INTEGER</i> | INTEGER — The position of the first occurrence of the <i>target</i> string within the <i>source</i> string, relative to the <i>starting-point</i> . The <i>starting-point</i> is optional and defaults to 1 (the beginning of the string). |
| LEFT-TRIM | <i>string AS CHAR, trim-chars AS CHAR</i> | CHARACTER — The input <i>string</i> , with the <i>trim-chars</i> removed from the beginning of the string. The <i>trim-chars</i> argument is optional, and defaults to any <i>white space</i> (spaces, tabs, carriage returns, and line feeds). |
| LENGTH | <i>string AS CHAR, type AS CHAR</i> | INTEGER — The number of characters, bytes, or columns in the <i>string</i> . The <i>type</i> is optional and defaults to CHARACTER. Other possible values for this argument are RAW, which makes the function return the number of bytes in the string, and COLUMN, which causes it to return the number of display or print character-columns. These latter two <i>types</i> are useful for manipulating strings that might contain double-byte characters representing characters in non-European languages. |

Table 2–7: 4GL string manipulation functions

(2 of 2)

| Function | Arguments | Returned value |
|-----------------|--|--|
| R-INDEX | <i>source AS CHAR, target AS CHAR, starting-point AS INTEGER</i> | INTEGER — The position of the <i>target</i> string within the <i>source</i> string, but with the search starting at the end of the string rather than the beginning. The <i>position</i> , however, is counted starting at the left. This function is useful for returning the right-most (last) occurrence of the <i>source</i> substring. The <i>starting-point</i> is optional and defaults to 1. |
| REPLACE | <i>source AS CHAR, from-string AS CHAR, to-string AS CHAR</i> | CHARACTER — The <i>source</i> string with every occurrence of the <i>from-string</i> replaced by the <i>to-string</i> . |
| RIGHT-TRIM | <i>string AS CHAR. trim-chars AS CHAR</i> | CHARACTER — The input <i>string</i> with the <i>trim-chars</i> removed from the end of the <i>string</i> . The <i>trim-chars</i> argument is optional and defaults to all white space. |
| SUBSTRING | <i>source AS CHAR, position AS INTEGER, length AS INTEGER, type AS CHARACTER</i> | CHARACTER — The substring of the <i>source</i> string beginning at the <i>position</i> . The <i>length</i> is optional and specifies the (maximum) length to return. The default is the remaining length of the string starting at <i>position</i> . The <i>type</i> is also optional and has same values as for the LENGTH function. |
| TRIM | <i>string AS CHAR. trim-chars AS CHAR</i> | CHARACTER — The input <i>string</i> with the <i>trim-chars</i> removed from both the beginning and the end of the <i>string</i> . The <i>trim-chars</i> argument is optional and defaults to all white space. |

Putting a calculation into your procedure

The next change to your sample procedure is to perform a simple calculation and display a value based on the result. This section provides an introduction to representing arithmetic expressions in the Progress 4GL. It also discusses how to use some of the special built-in functions for advanced arithmetic operations.

Arithmetic expressions and operands

The Progress 4GL supports the set of arithmetic operands described in [Table 2–8](#). You can use these operands to define expressions. You might be familiar with them from other programming languages you have used.

Table 2–8: Supported arithmetic operands

| Symbol | Explanation |
|--------|---|
| + | Adds numeric values. Concatenates character strings. |
| - | Subtracts numeric values or date values. |
| * | Multiplies numeric values. |
| / | Divides numeric values. |

There's one special thing you need to know when you're writing expressions involving these operands. Because the Progress 4GL allows the use of a hyphen as a character in a procedure name, variable name, or database field name, it cannot recognize the difference between a hyphen and a minus sign used for subtraction, which are the same keyboard character. For example, there's no way for the syntax analyzer to tell whether the string ABC-DEF represents a single hyphenated variable or field name, or whether it represents the arithmetic expression ABC minus DEF, involving two fields or variables named ABD and DEF. For this reason, you have to put a space or other white space characters around the “-” character when you use it as a minus sign for subtraction of one number from another. Note that you *don't* have to insert a space after a minus sign that precedes a negative number, such as -25. For consistency, the other arithmetic operands also require white space. If you forget to put it in, you'll get an error, except in the case of the forward slash character. In the case of the slash, if you leave out the white space, Progress interprets the value as a date! So, for example, 5/6 represents May 6th, not a numeric fraction.

- To illustrate how to use arithmetic operands in the sample procedure, you need to determine whether the **CreditLimit** of the **Customer** is less than twice the outstanding **Balance**. If this is true, then you must display the ratio of **CreditLimit** to **Balance**. Otherwise you display the **Orders** for the **Customer**. Add the following code, just in front of the FOR EACH Order OF Customer statement that's already there:

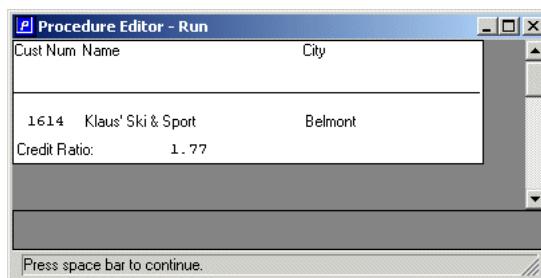
```
IF CreditLimit < 2 * Balance THEN
    DISPLAY "Credit Ratio:" CreditLimit / Balance.
ELSE FOR EACH Order OF Customer:
```

You can add parentheses to such an expression to make the grouping of terms explicit. Otherwise, Progress observes the standard rules of precedence. Multiplication and division are performed before addition and subtraction, and all such calculations are performed before a comparison operation.

The expression following the IF keyword compares the **CreditLimit** field from the current **Customer** record with two times the value of the **Balance** field. If the first value is less than the second, then the expression is true and the statement following the THEN keyword is executed, which displays the string expression **Credit Ratio:** followed by the value of the **CreditLimit** divided by the **Balance**.

The ELSE keyword is followed by the entire FOR EACH Order block, so that block of code, which displays all the **Orders** of the **Customer**, is skipped if the expression is true, and executed only if it is false.

- To see the result of this change, run your procedure again. For a **Customer** where the ratio is greater than or equal to 2, the **Orders** display as before. For a **Customer** where the ratio is less than 2, the new expression is displayed instead:



You might notice a couple of things about this display:

- Because the **CreditLimit** check is in the block of code where the procedure retrieves and displays **Customers**, it is displayed in the same frame as the **Customer** information. You can give names to frames to be more specific about the frame in which to display objects, as well as where in the frame each element is displayed. You'll learn more about naming frames in [Chapter 4, “Introducing the OpenEdge AppBuilder.”](#)
- Progress understands enough about what is going on here to clear and hide the **Order** frame if it's not being displayed for the current **Customer** (because the **CreditLimit** to **Balance** ratio is being displayed instead). This is part of the very powerful default behavior of the language.

Arithmetic built-in functions

[Table 2–9](#) describes some of the useful built-in functions that extend the basic set of numeric operands.

Table 2–9: Arithmetic built-in functions

(1 of 2)

| Function | Arguments | Returned value |
|----------|--|--|
| ABSOLUTE | <i>value</i> AS INTEGER or DECIMAL. | INTEGER or DECIMAL — The absolute value of the numeric value. |
| EXP | <i>base</i> AS INTEGER or DECIMAL, <i>exponent</i> AS INTEGER or DECIMAL. | INTEGER or DECIMAL— The result of raising the <i>base</i> number to the <i>exponent</i> power. |
| LOG | <i>expression</i> AS DECIMAL, <i>base</i> AS INTEGER or DECIMAL . | DECIMAL— The logarithm of the <i>expression</i> using the specified <i>base</i> . The <i>base</i> is optional; the natural logarithm, <i>base</i> (e), is returned by default. |
| MAXIMUM | two or more <i>expressions</i> AS INTEGER or DECIMAL . | INTEGER or DECIMAL— The largest value of the <i>expressions</i> . |
| MINIMUM | two or more <i>expressions</i> AS INTEGER or DECIMAL . | INTEGER or DECIMAL— The smallest value of the <i>expressions</i> . |
| MODULO | This function has the special syntax: <i>expression</i> MODULO <i>base</i> . | INTEGER— The remainder after division. The <i>expression</i> and <i>base</i> must be INTEGER values. |

Table 2–9: Arithmetic built-in functions

(2 of 2)

| Function | Arguments | Returned value |
|----------|---|---|
| RANDOM | <i>low-value</i> AS INTEGER, <i>high-value</i> AS INTEGER. | INTEGER — A random INTEGER value between the <i>low-value</i> and the <i>high-value</i> (inclusive). There is a Random (-rand) Progress startup option that determines whether a different set of values is returned for each OpenEdge session. |
| ROUND | <i>expression</i> AS DECIMAL, <i>precision</i> AS (positive) INTEGER. | DECIMAL — The DECIMAL <i>expression</i> rounded to the number of decimal places specified by the <i>precision</i> . The rounding is down for all values beyond the <i>precision</i> that are less than .5, and up for all higher values. |
| SQRT | <i>expression</i> AS INTEGER or DECIMAL. | DECIMAL — The square root of the <i>expression</i> . |
| TRUNCATE | <i>expression</i> AS DECIMAL, <i>precision</i> AS (non-negative) INTEGER. | DECIMAL — The <i>expression</i> truncated to the specified <i>precision</i> . |

Using the Intelligent Edit Control and its shortcuts

The first step in the next part of your sample procedure involves defining another program variable. You won't do too many of these before you might begin to get tired of typing the words **DEFINE VARIABLE** and **NO-UNDO** and so forth over and over again. In fact, you don't have to do all that typing at all.

You've already seen that when you use the Progress Procedure Editor to edit your Progress 4GL code, you are using an intelligent syntax editor that can perform many tasks for you. One of these is syntax completion. Using this feature can save you many keystrokes in your programming, helps to prevent typographical errors, and helps to impose a standard form to your code.



To define a new INTEGER variable for your procedure:

1. Position the cursor to the beginning of the line where you want the definition to go, following the variable definition you have already, then enter the letters **DVI**, or **DVIN**, which stand for *Define Variable INteger*.
2. Press the **SPACE BAR**. The following code appears:

```

Procedure Editor - C:\PROGRESS\WRK\CustSample.p
File Edit Search Buffer Compile Tools Options Help
DEFINE VARIABLE cMonthList AS CHARACTER NO-UNDO
    INIT "JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOW,DEC".
DEFINE VARIABLE   AS INTEGER   NO-UNDO. _____
FOR EACH Customer WHERE State = "NH" BY City:
    DISPLAY CustNum NAME CITY.
    IF CreditLimit < 2 * Balance THEN
        DISPLAY "Credit Ratio:" CreditLimit / Balance.
    ELSE FOR EACH Order OF Customer:
        DISPLAY OrderNum LABEL "Order"
            OrderDate
            ShipDate FORMAT "99/99/99" WITH CENTERED.
        IF ShipDate NE ? THEN
            DISPLAY ENTRY(MONTH(ShipDate), cMonthList) LABEL "Month".
    END.
END.

```

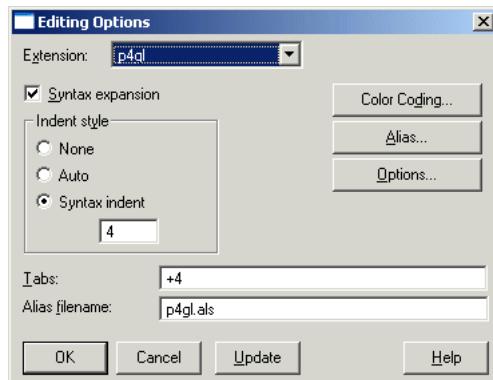
The text **DEFINE VARIABLE AS INTEGER NO-UNDO** is inserted for you, and the cursor is placed right where you need it so you can enter the variable name.

3. Type **iDays** as the variable name. You'll use this variable later.

Here are the abbreviations for other data types that you've learned:

- **DVI** — INTEGER
- **DVC** — CHARACTER
- **DVDE** — DECIMAL
- **DVDT** — DATE
- **DVH** — HANDLE
- **DVL** — LOGICAL

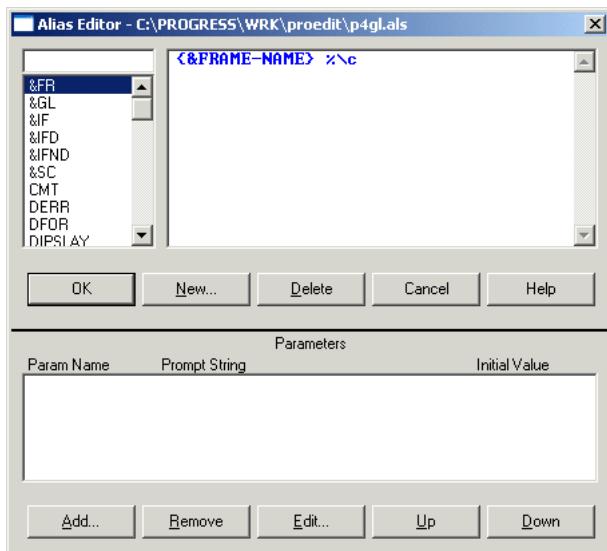
► To see all the abbreviations (also called *aliases*) and other editing options available to you, select **Options→Editing Options** from the Procedure Editor menu. The **Editing Options** dialog box appears:



Among other things, this dialog box specifies the following default settings:

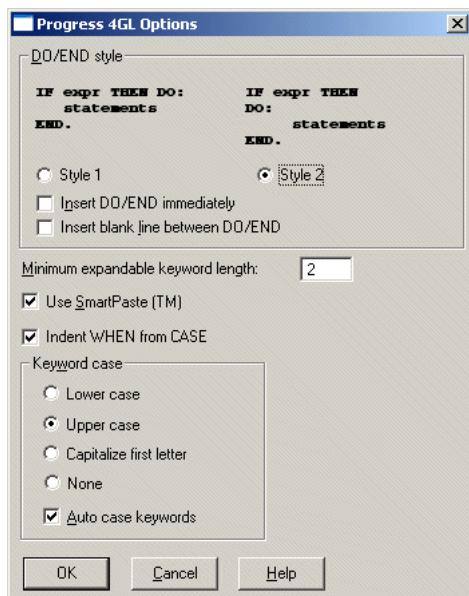
- The syntax expansion that turns **DVI** into **DEFINE VARIABLE AS INTEGER NO-UNDO** is turned on (the **Syntax expansion** toggle box).
- Lines of your program are indented based on the Editor's understanding of the statements you're entering (the **Indent style** selection).
- Tabs are set to 4-character intervals (the **Tabs** fill-in field).

- To see a list of all the aliases (also called *shortcuts*) that are defined, choose the **Alias** button. The **Alias Editor** dialog box appears:



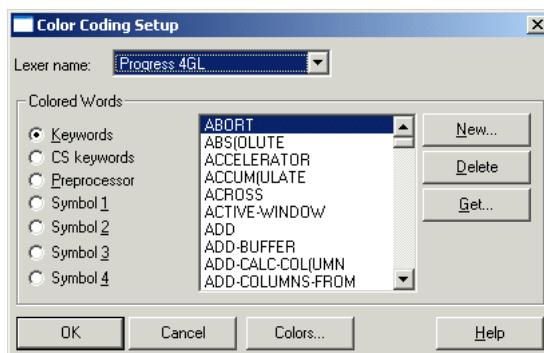
In this dialog box you see all the aliases for the **DEFINE VARIABLE** statement, along with a number of other useful ones, including one that automatically respells the (apparently) common typographical error DIPSLAY as DISPLAY. Aliases can be further abbreviated to any unique substring of the alias, which is why **DVI** works as well as **DVIN**. Note that you can also define new aliases of your own. They are stored in a file called **p4gl.als**.

- To peruse another set of options that you can customize, choose **OK** to return to the main **Editing Options** dialog box, then choose the **Options** button. The **Progress 4GL Options** dialog box appears:



This dialog box includes, among other options, the automatic uppercasing of keywords that you've seen in action. If this isn't your preference, you can turn it off.

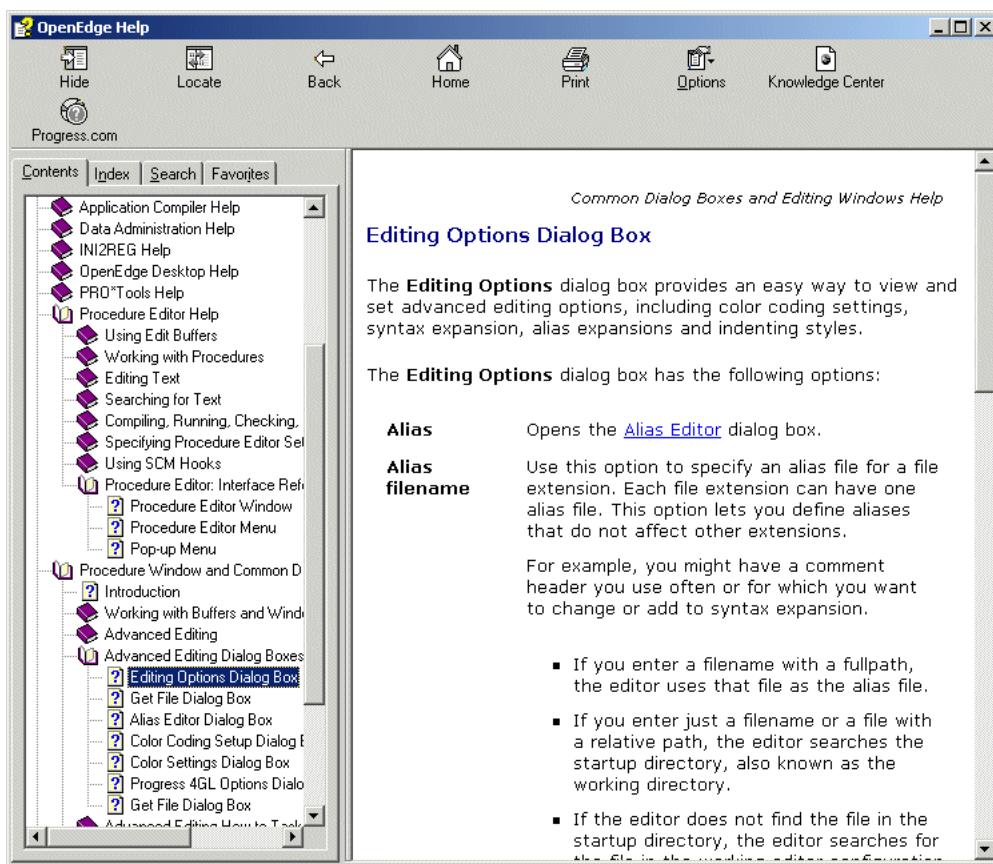
- To customize which keywords you want colored and what colors to use, choose **OK** to return to the **Editing Options** dialog box, then choose the **Color Coding** button. The **Color Coding Setup** dialog box appears:



If you choose the **Help** button on any of these dialog boxes, you can see detailed information on all the options available to you.

Getting to online help

You can access the OpenEdge Online Help system at any time just by pressing the **F1** key. Help is context-sensitive, so it always tries to come up initialized to a section of the help files that seems relevant to what you're doing. For example, if you press **F1** from within the **Editing Options** dialog box, you get help on the options in that dialog box:

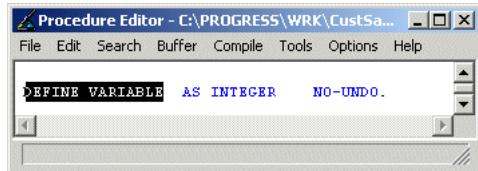


In a Progress editor, if you highlight one or more keywords in a procedure and press **F1**, you get help for that type of statement.

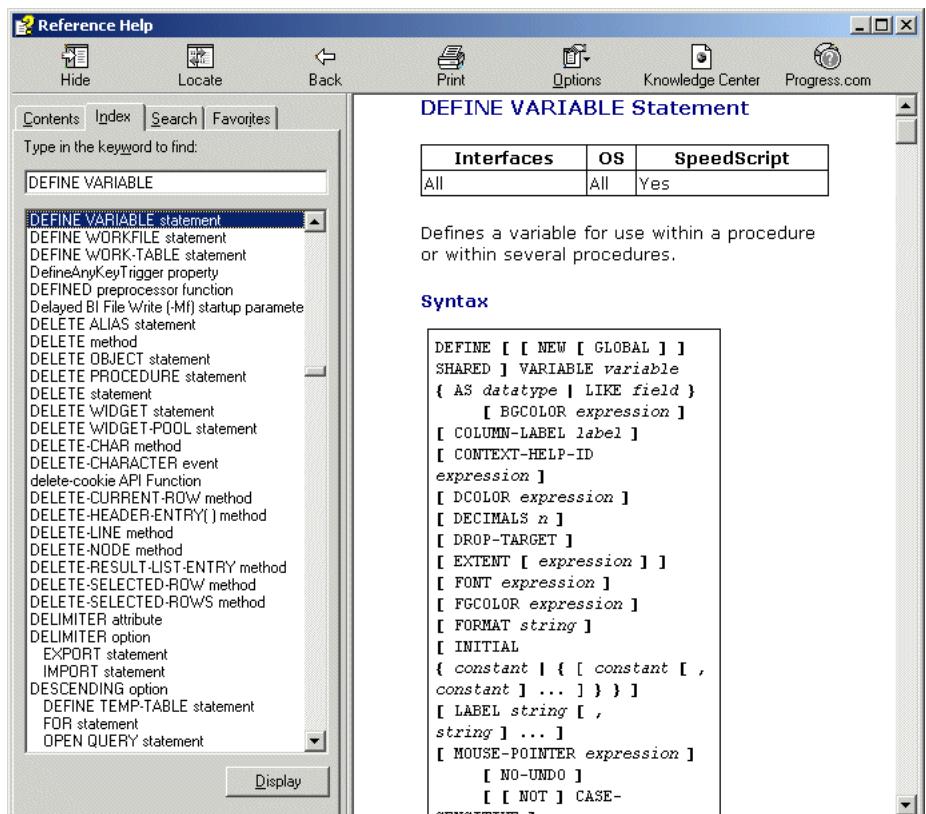


To try accessing online help:

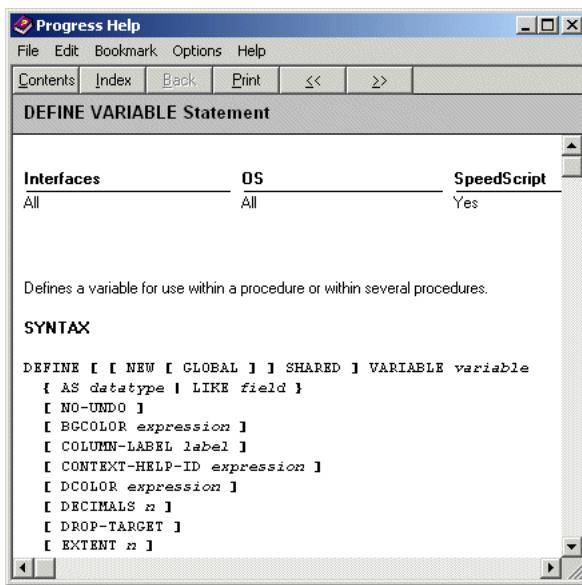
1. In your test procedure, highlight the keywords DEFINE VARIABLE, then press F1:



The online help window appears showing the help keyword index:



2. Choose **Display** to see the help text for this entry:



3. Choose the **Contents** button to access the available online help for all the OpenEdge tools, or choose the **Find** button to specify one or more key words to search for in the help text.

Saving and compiling your test procedure

You've written your first Progress 4GL procedure. As you make changes to it, you should resave it by selecting **File→Save** from the Procedure Editor menu or by just pressing **F6**.

This section examines what happens when you press the **F2** key or select **Compile→Run** from the menu to run your procedure:

1. When you tell Progress to run your procedure in this way, the Procedure Editor creates a temporary file from the 4GL code currently in the editor window and then passes this file to the Progress compiler.
2. The compiler performs a syntax analysis of the procedure and stops with one or more error messages if it detects any errors in your 4GL statements.
3. If your procedure is error-free, the compiler then translates or compiles your procedure into an intermediate form that is then interpreted by the Progress run-time interpreter to execute the statements in the procedure. This intermediate form is called r-code, for run-time code. It is not a binary executable but rather a largely platform-independent set of instructions (independent except for the user interface specifics, that is).
4. The Progress run-time interpreter reads and executes the r-code.

This ability to compile and run what's in the editor on the fly makes it easy for you to build and test procedures in an iterative way, making a change as you have done and then simply running the procedure to see the effects. Naturally, you want to save the r-code permanently when you have finished a procedure so that Progress can skip the compilation step when you run the procedure and simply execute the r-code directly. Generally it is just the compiled r-code that you deploy as your finished application.

To save the source procedure itself, use the standard **File→Save As** menu option in the Procedure Editor, which saves the contents of the editor out to disk and gives it a filename with a **.p** extension.

To compile a procedure that you have saved, you need to use the **COMPILE** statement in the 4GL. There's a **COMPILE** statement rather than just having a single key to press because the **COMPILE** statement in fact has a lot of options to it, including where the compiled code is saved, whether a cross-reference file is generated to help you debug your application, and so forth. For now, it is sufficient to know just the simple form of the statement you use to save your compiled procedure.

**To compile your test procedure:**

1. From the Procedure Editor, select **File→New Procedure Window**.

Another editor window appears that has all the same capabilities as the one with which you started out, except that not all the menu options are available from it. You can be working with any number of procedure windows at one time, and save them independently as separate named procedures. Or you can bring up a window just to execute a statement, as you are doing now.

2. In the new editor window, enter the following statement:

```
COMPILE h-CustSample.p SAVE.
```

3. Press **F2** or select **Compile→Run**. The procedure window disappears and almost immediately returns.

The compilation of `h-CustSample.p` is complete. If the compiler had detected any syntax errors in the procedure, you would have seen them and the procedure would not have compiled.

If you check your working directory you can see the compiled procedure. It has the `.r` extension and is commonly referred to as a `.r` file.

One option for the `COMPILE` statement is worth mentioning at this time. You can specify a different directory for your `.r` files. This is a good idea, especially when you get to the point where you are ready to start testing completed parts of your application. Use this syntax:

```
COMPILE source-procedure SAVE INTO directory.
```

If you need to recompile a large number of procedures at once, you can use a special tool to do that for you. This is available through the **Tools→Application Compiler** option on the Procedure Editor.

Now you've compiled your first procedure! In the next chapter, you learn how to run one procedure from another.

Running Progress 4GL Procedures

This chapter describes how you can work with and run different types of Progress 4GL procedures. It covers the following topics:

- [Running a subprocedure](#)
- [Using the Propath](#)
- [Using external and internal procedures](#)
- [Adding comments to your procedure](#)

Running a subprocedure

To run a procedure from within another 4GL procedure, you use the RUN statement. Like the COMPILE statement, the RUN statement has a lot of options, some of which you'll learn about in later chapters. For now a very simple form of the statement will do:

```
RUN procedure-name [ (parameters) ].
```

If you are running a procedure file like your h-CustSample.p, then the convention is to include the .p extension in the procedure name. For example, if you bring up a new **Procedure Window** again, you can enter this statement to run the sample procedure you've been working on, then press **F2**:



This is a little different than just pressing the **F2** key in the procedure window where you're editing h-CustSample.p. When you press **F2** or select **Compile→Run** you are compiling and running the current contents of the **Procedure Editor** window, which might be different from what you have saved to a named file. You can compile and run a procedure without giving it a name at all. Indeed, that's what you just did: you created a procedure containing a single RUN statement, and then simply compiled and executed it by pressing **F2**. This temporary procedure in turn ran the named procedure you had already saved and compiled.

Now, why did you put the .p extension on the RUN statement when you have already saved a compiled .r version of the file out to your directory? When you write a 4GL RUN statement with a .p extension on the procedure name, Progress always looks first for a compiled file of the same name, but with the .r extension. If it finds it, it executes it. If it doesn't, it passes the source file to the Progress compiler, which compiles it on the fly and then executes it. In this way, your application can be a mix of compiled and uncompiled procedures while you're developing it. If you always use the .p extension in your RUN statements, then your procedures are always found and executed whether they've been compiled or not. By contrast, if you specify the .r extension in a RUN statement, then Progress looks for only the compiled .r file, and the RUN fails if it isn't found.

Using the Propath

Before you continue on to look at the parameters that you can pass to a RUN statement, you should understand a little about how Progress searches for procedures. When you type **RUN h-CustSample.p**, how does Progress know where to look for it?

Progress uses its own directory path list, called the *Propath*, which is stored as part of its initialization (*progress.ini*) file. When you install OpenEdge, you get a default *progress.ini* file in the *bin* subdirectory under your install directory. You can also create a local copy of this file if you want to change its contents.

If you want to look at and maintain your Propath, there's a tool to do that for you. It's one of a number of useful tools you can access from the Procedure Editor.

- ▶ To access these tools from the Procedure Editor, select **Tools→PRO*Tools**. The **PRO*Tools** palette appears:



You can reposition and reshape the **PRO*Tools** palette.

- ▶ To retain the palette's new shape and position permanently:

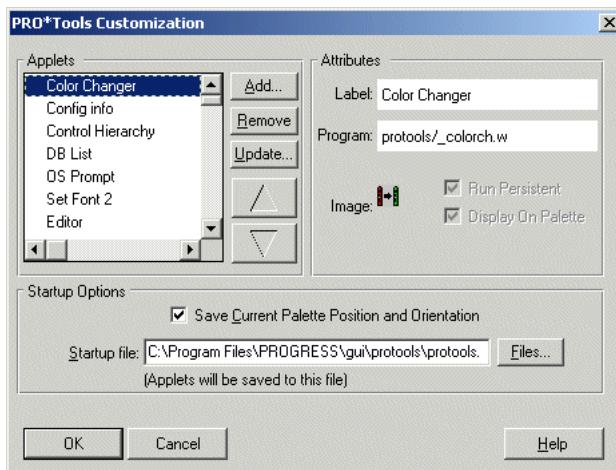
1. Right-mouse click on the **PRO*Tools** palette outside the area where icons are displayed. The **Menu Bar** pop-up menu item appears:



2. Select the **Menu Bar** pop-up item. A **File** menu appears at the top of the palette:



3. Select **File→Customize**. The **PRO*Tools Customization** dialog box appears:

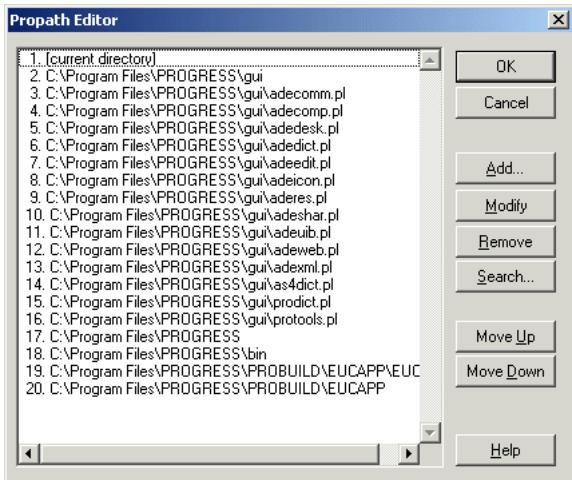


Using this tool you can add more useful tools of your own to the palette. For now, however, all you need to note is that the **Save Current Palette Position and Orientation** toggle box lets you save these settings permanently, so that the **PRO*Tools** palette always comes up the way you want each time you start OpenEdge.

4. Leave the **Save Current Palette Position and Orientation** toggle box checked on, then choose **OK**.

- To view and edit your Propath, choose the **Propath** icon  from the **PRO*Tools** palette.

The **Propath Editor** shows you a list of all the directories in which Progress looks, both to find all its own executable files and other supporting files, and to find your application procedures. Here is the default Propath you get when you install the product:



As you can see, Progress first looks in your current working directory (by default, C:/Program Files/OpenEdge). This is where it expects to find any procedures you have written and are trying to compile or run. After that it looks in the *gui* directory under your OpenEdge install directory. This is where it expects to find the compiled r-code for all the 4GL procedures that support your development and run-time environments. Remember that most of the OpenEdge development tools, including the Procedure Editor, are themselves written in the 4GL.

In the *gui* directory are a number of *procedure libraries*, each with a .pl filename extension. These are collections of many .r files, gathered together into individual operating system files. The *adecomm.pl* file, for example, is a library of many dozens of procedures that are common to the development tools, called the Application Development Environment (ADE). It's more efficient to store them in a single library because it takes up less space on your disk and it's faster to search.

Following these procedure library files is the install directory itself. This holds a few startup files as well as the original versions of the sports2000 database and other sample databases shipped with the OpenEdge products.

Next is the `bin` directory. This is where all the executable files are located, including all the supporting procedures for compiling and running your application, which are not written in the 4GL. Finally, there are a couple of directories that support building a custom OpenEdge executable.

Do not modify any of the directories below the current directory. If you try to, Progress resets the Propath, because it recognizes that the tools will stop running if it can't find all the pieces it needs. But you can add directories above, below, or in place of your current working directory. For example, if you save your r-code into a different directory using the `SAVE INTO` option on the `COMPILE` statement, then you must add this directory to your Propath.

When you `COMPILE` or `RUN` a procedure, you can specify a partial pathname for it relative to some directory that is in your Propath. For example, if you save something called `NextProc.p` into a subdirectory called `morecode`, you can run it using this statement:

```
RUN morecode/NextProc.p.
```

Note: You should always use forward slashes in your Progress code, even though the DOS and Windows standard is backslashes. Progress does the necessary conversions to make the statement work on a PC, but if some of your Progress 4GL code is written for another platform, such as UNIX, then it requires the forward slashes.

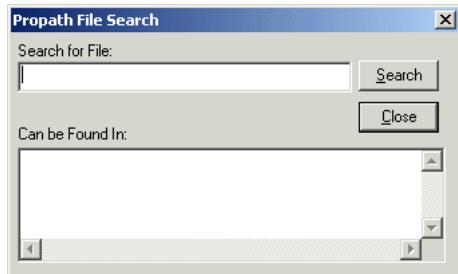
You can modify your Propath by using the **Add**, **Modify**, and **Remove** buttons in the **Propath Editor**.

If Progress is having difficulty locating a procedure that you've written, or if you have different versions of a procedure in different directories you can check to see how Progress searches for a particular procedure.

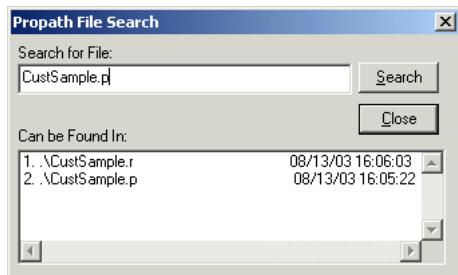


To verify which file Progress finds:

1. In the **Propath Editor**, choose the **Search** button. The **Propath File Search** dialog box appears:



2. Enter the name of the procedure, including the relative pathname if that's part of your RUN statement. Also include the same filename extension you are using in your code.
3. Choose the **Search** button. Progress displays all the versions of that file it can find, in the order in which it uses them. Thus the one at the top of the list is always the one Progress tries to execute. In this example, the display confirms that when you run h-CustSample.p, it chooses the .r file (if there is one) in preference to the source procedure:



If you save your r-code to a different directory (or set of directories) than you use for the source procedures, remember to construct your Propath in such a way that the r-code is found first if it is there, but that the source procedures are also found, at least when you are in development mode, without requiring a special relative pathname in your code to locate them.

In the OpenEdge install directories, for example, all the compiled 4GL procedures are under the `gui` directory. The corresponding source procedures are in an equivalent directory tree, but under an `src` directory. If you were to add the `src` directory to your Propath, then Progress could locate source procedures to compile and execute on the fly as needed. If you were to place the `src` directory *above* the `gui` directory in your Propath, then Progress would use all the source procedures and compile them on the fly because it finds them first. This would slow down your program execution dramatically. This is just an example of how, as you develop complex applications, you must be careful to arrange your Propath so that Progress can find the procedures it needs efficiently, without doing a lot of unnecessary searching or running uncompiled procedures.

Using external and internal procedures

A separate source procedure, such as `h-CustSample.p`, is known as an *external procedure*. This is to distinguish it from another kind of the procedure that you'll write as the next step in your sample. You can define one or more named entry points within an external procedure file, and these are called *internal procedures*. You run internal procedures in almost exactly the same way that you run external procedures, except that there is no `.p` filename extension on the internal procedure.

For your sample, you will run a procedure called `calcDays` that calculates the number of days between the **ShipDate** for each **Order** and today's date. In order for `calcDays` to do the calculation, you need to pass the **ShipDate** to the procedure, and get the count of days back out. Progress supports parameters for its procedures to let you do this. Follow these guidelines:

- In a RUN statement, you specify the parameters to the RUN statement in parentheses, following the procedure name. Use commas to separate multiple parameters.
- An INPUT parameter can be a constant value, an expression, or a variable or field name. An INPUT parameter is made available by value to the procedure you are running. This means that the procedure can use the value, but it cannot modify the value in a way that is visible to the calling procedure.
- An OUTPUT parameter must be a variable or field name. Its value is not passed to the called procedure, but is returned to the calling procedure when the called procedure completes.
- You can also define a parameter as INPUT-OUTPUT. This means that its value is passed in to the procedure, which can modify the value and pass it back when the procedure ends. An INPUT-OUTPUT parameter must also be a field or variable.

For each parameter in the RUN statement, you specify the type of parameter and the parameter itself. The default is INPUT, so you can leave off the type for INPUT parameters. You cannot have optional parameters in a Progress procedure. You must pass in a parameter name or value for each parameter the called procedure defines. They must be in the same order, and they must be of the same data type.



To run the **calcDays** procedure from your sample procedure:

1. Add the following RUN statement to your h-CustSample.p procedure, inside the FOR EACH Order OF Customer block, just before the END statement. Pass in the **ShipDate** field from the current **Order** and get back the calculated number of days, which uses the **iDays** variable you defined earlier when you learned to use the intelligent editor's aliases:

```
RUN calcDays (INPUT ShipDate, OUTPUT iDays).
```

2. Following this, still inside the FOR EACH Order block, write another statement to display the value you got back along with the rest of the **Order** information:

```
DISPLAY iDays LABEL "Days" FORMAT "ZZZ9".
```

Writing internal procedures

Now it's time to write the **calcDays** procedure. Put it at the end of **h-CustSample.p**, following all the code you've written so far.

Each internal procedure starts with a header statement, which is just the keyword PROCEDURE followed by the internal procedure name and a colon.

Following this you need to define any parameters the procedure uses. The syntax for this is very similar to the syntax for the **DEFINE VARIABLE** statement. In place of the keyword **VARIABLE**, use the keyword **PARAMETER**, and precede this with the parameter type—**INPUT**, **OUTPUT**, or **INPUT-OUTPUT**. Note that the keyword **INPUT** is not optional in parameter definitions. Here's the declaration for the `calcDays` procedure and its parameters. The parameter names start with the letter *p* to help identify them, followed by a prefix that identifies the data type as **DATE** or **INTEGER**:

```
PROCEDURE calcDays:  
  DEFINE INPUT  PARAMETER pdaShip  AS DATE      NO-UNDO.  
  DEFINE OUTPUT PARAMETER piDays   AS INTEGER    NO-UNDO.
```

Now you can write 4GL statements exactly as you can for an external procedure. If you want to have variables in the subprocedure that aren't needed elsewhere, then define them following the parameter definitions. Otherwise you can refer freely to variables that are defined in the external procedure itself. You'll take a much closer look at variable scope and other such topics later. Be cautious when you use variables that are defined outside a procedure unless you have a good reason for using them, because they compromise the modularity and reusability of your code. For example, if you pull your procedures apart later and put the `calcDays` procedure somewhere else, it might break if it has a dependency on something declared outside of it. For this reason, you pass the calculated number of days back as an **OUTPUT** parameter, even though you could refer to the variable directly.

Assigning a value to a variable

The `calcDays` procedure just has a single executable statement, but it's one that demonstrates a couple of key new language concepts—assignment and unknown value.

You're probably familiar with language statements that assign values by using what looks like an equation, where the value or expression on the right side is assigned to the field or variable on the left. Progress does this too, and uses the same equal sign for the assignment that is used for testing equality in comparisons. However, you can use the keyword **EQ** only in comparisons, not in assignments.

In this example, you want to subtract the **ShipDate** (which was passed in as the parameter `pdaShip`) from today's date (which, as you have seen, is returned by the built-in function **TODAY**) and assign the result to the **OUTPUT** parameter `piDays`.

To make this change in your sample procedure, add the following statement:

```
piDays = TODAY - pdaShip.
```

This is simple enough, but it won't work all the time. Remember that some of the **Orders** have not been shipped, so their **ShipDate** has the Unknown value. If you subtract an Unknown value from TODAY (or use an Unknown value in any other expression), the result of the entire expression is always the Unknown value. In this case you want the procedure to return 0. To do this you can use a special compact form of the IF-THEN-ELSE construct as a single 4GL expression, appearing on the right side of an assignment. The general syntax for this is:

```
result-value = IF logical-expression  
    THEN value-if-true ELSE value-if-false.
```

This syntax is more concise than using a series of statements of the form:

```
IF logical-expression THEN result-value = value-if-true.  
ELSE result-value = value-if-false.
```

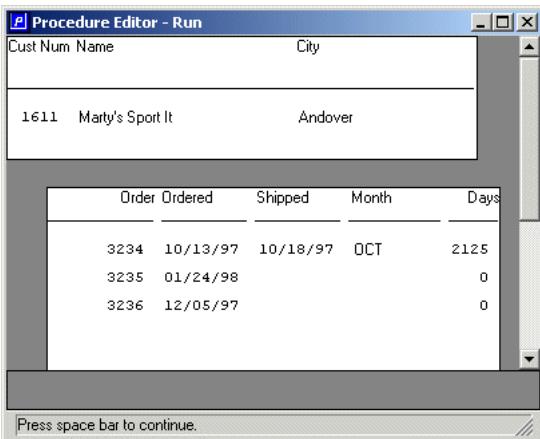
For your purposes you can refine the assignment statement in this way to allow for Unknown values:

```
piDays = IF pdaShip = ? THEN 0 ELSE TODAY - pdaShip.
```

Finally, you end each internal procedure with an END PROCEDURE statement. The keyword PROCEDURE is optional, but is always a good idea as it improves the readability of your code:

```
END PROCEDURE.
```

► To see the calculated days displayed, **Run** the procedure once more:



In summary, you define an internal procedure just as would an external procedure, with the exception that each internal procedure must start with a PROCEDURE header statement and end with its own END PROCEDURE statement. An external procedure uses neither of these.

You RUN an internal procedure just as you would an external procedure, except that there is no filename extension on the procedure name. Here are a couple of important related points:

First, it is possible for you to run an external procedure by naming it in a RUN statement without any filename extension, for example, RUN CustSample. In this case, Progress searches for these forms of the procedure in this order:

1. An internal procedure named `CustSample`.
2. An external compiled file named `CustSample.r`.
3. An external source procedure named `h-CustSample.p`.

To do this, however, would be considered very bad form. When a person reading Progress code sees a RUN statement with no filename extension, it is natural for that person to expect that it is an internal procedure. There is rarely a good reason for violating this expectation.

Second, and even more exceptional, is that because a period is a valid character in a procedure name, it would be possible to have an *internal* procedure named, for example, `calcDays.p`, and to run it under that name. You should never do this. Always avoid confusing yourself or others who read your code.

When to use internal and external procedures

The decision as to when to use a separate external procedure file versus when to make an entry point for an internal procedure in a larger file is largely a matter of style and application organization. It's a good idea to group together in a single procedure file related small procedures that call one another or that are typically called by multiple other procedures. On the other hand, large procedures that perform complex operations on their own should probably be independent external procedure files. Keep in mind that when you need to run an internal procedure, Progress first needs to load the entire procedure file that contains it, and this can consume excessive memory if you make your procedure files extremely large. Later in [Chapter 13, “Advanced Use of Procedures in Progress,”](#) you’ll learn about persistent procedures and how you can preload a file that contains many internal procedures that other procedures need to call.

Adding comments to your procedure

The final step in this exercise is to add some comments to your procedure to make sure you and everyone else can follow what the code does. In Progress you begin a comment with the characters /* and end it with */. A comment can appear anywhere in your procedure where white space can appear (that is, anywhere except in the middle of a name or other token). You can put full-line or multi-line comments at the top of each section of code, and shorter comments to the right of individual lines. Just make sure you use them, and make them meaningful to you and to others. The intelligent editor colors comments in green by default. The editor can also type the comment symbols for you, if you type **CMT** and press the **SPACEBAR** in the editor window. Here's the final procedure with a few added comments:

h-CustSample.p

```
/* h-CustSample.p- shows a few things about the Progress 4GL */

DEFINE VARIABLE cMonthList AS CHARACTER NO-UNDO
  INIT "JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC".
DEFINE VARIABLE iDays      AS INTEGER    NO-UNDO. /* used in calcDays proc */

/* First display each Customer from New Hampshire: */
FOR EACH Customer WHERE state = "NH" BY City:
  DISPLAY CustNum NAME City.
  /* Show the Orders unless the Credit Limit is less than
     twice the balance. */
  IF CreditLimit < 2 * Balance THEN
    DISPLAY "Credit Ratio:" CreditLimit / Balance .
  ELSE FOR EACH Order OF Customer:
    DISPLAY OrderNum LABEL "Order"
      OrderDate ShipDate FORMAT "99/99/99" WITH CENTERED.
    /* Show the month as a three-letter abbreviation, along with the
       number of days since the order was shipped. */
    IF ShipDate NE ? THEN
      DISPLAY ENTRY(MONTH(ShipDate), cMonthList) LABEL "Month".
    RUN calcDays (INPUT ShipDate, OUTPUT iDays).
    DISPLAY iDays LABEL "Days" FORMAT "ZZZ9".
  END.
END.

PROCEDURE calcDays:
  /* This calculates the number of days since the Order was shipped. */
  DEFINE INPUT  PARAMETER pdaShip  AS DATE      NO-UNDO.
  DEFINE OUTPUT PARAMETER piDays   AS INTEGER   NO-UNDO.

  piDays = IF pdaShip = ? THEN 0
            ELSE TODAY - pdaShip.
END PROCEDURE.
```

You've learned a tremendous amount about the Progress 4GL in the course of writing a procedure barely over twenty lines long, which retrieves and merges data from two different database tables, and performs a number of extra calculations and formatting operations along the way. These first chapters have tried to provide you with some insight into the power of the Progress 4GL. Feel free to experiment with the language statements and functions you've encountered.

In the next chapters, you'll look at another major OpenEdge development tool, the AppBuilder, and use it to generate the code you need to create an application window that looks a lot closer to the kind of graphical interface you might expect in your applications. Later chapters then introduce you to a world of creating applications with almost no code at all!

4

Introducing the OpenEdge AppBuilder

In the earlier chapters of this book, you learned how to use the Progress Procedure Editor to write your first Progress 4GL procedure. This chapter introduces you to the OpenEdge™ AppBuilder, a tool for designing visual interfaces and for structuring more complex procedures and the code they contain.

As with the earlier chapters, this is a bit of a snowplow exercise. You'll use the AppBuilder to generate the 4GL code to define a window and its contents, to give a graphical look to your sample procedure.

This chapter includes the following sections:

- [Starting the AppBuilder](#)
- [Creating a new procedure and window](#)
- [Using the Query Builder](#)
- [Using the Section Editor](#)
- [Adding buttons to your window](#)

Starting the AppBuilder

To bring up the AppBuilder, you need to close down the Procedure Editor if it's running, then select the **AppBuilder** icon on the OpenEdge Desktop, as shown in [Figure 4–1](#).



Figure 4–1: The ADE desktop

Alternately, you can bring up the AppBuilder directly from the Windows desktop, by selecting the AppBuilder menu option under the OpenEdge startup group.

The AppBuilder main window has its own menu and toolbar:, as shown in [Figure 4–2](#).



Figure 4–2: The AppBuilder main window

In addition, the AppBuilder has a separate **Palette** window where you can select various kinds of both visual objects and nonvisual data management objects to add to your application. You'll look at just a few of these objects in this chapter. You can learn all about the AppBuilder in [OpenEdge Development: AppBuilder](#).

Figure 4–3 shows the icons for the Palette objects you’ll be using in this chapter. You can learn some of the others by floating the mouse over an icon and looking at its tooltip.

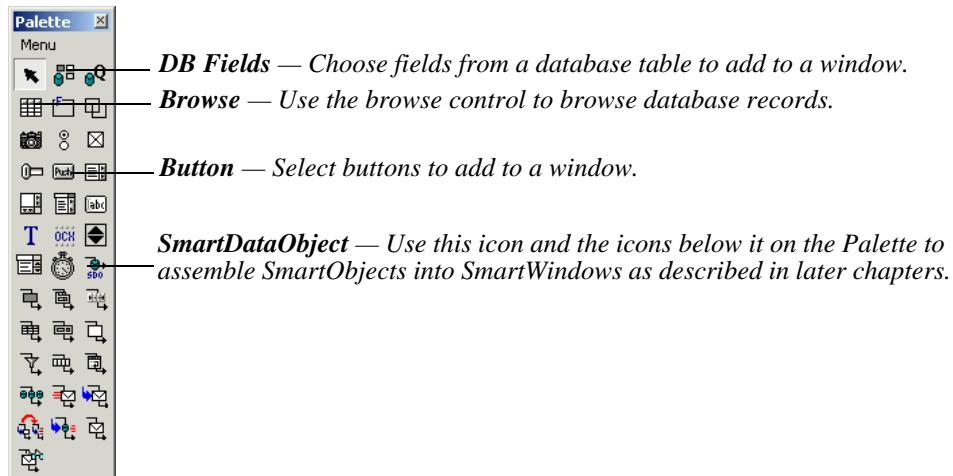


Figure 4–3: The AppBuilder Palette

You can reshape and reposition the **Palette** window as you like. To save the **Palette** window’s position, select **Menu→Options→Save Palette** from the **Palette** window.

In addition to the **Palette** window, the **Pro*Tools** window also comes up when you start the AppBuilder.

In this chapter you’ll build a window that displays some fields from the **Customer** table, as well as a browse that shows the **Orders** for the current **Customer**. So this is like the example you built in the first chapter, except for two differences:

- By choosing graphical objects to build your window, you get a much better interface than you got using the defaults provided by the **DISPLAY** statement in the previous chapters, one that looks more like a real application window.
- Because the AppBuilder generates most of the supporting 4GL code for you, you actually have to write even less code than you did in previous chapters.

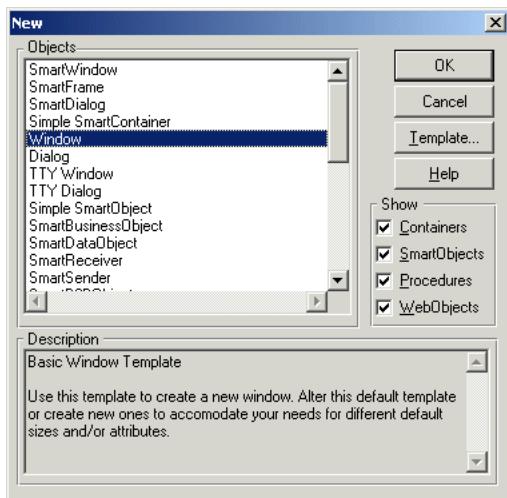
Creating a new procedure and window

This section describes how to create a new sample procedure and window.

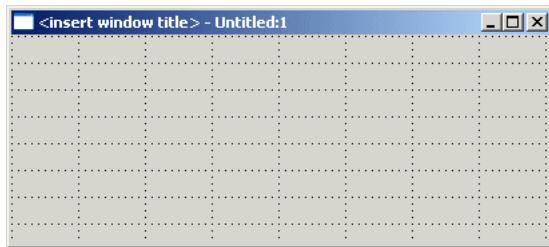


To create the window that holds all the other objects:

1. Choose the **New** icon  on the toolbar or select **File→New** from the menu.
2. From the **New** dialog box, select the option that is just called **Window**, then choose **OK**:



A design window appears for you to work with:



You can resize the design window by grabbing one corner with the mouse and moving it.

Think of this window as corresponding to the new Progress 4GL procedure you will write. The code in that procedure, most of it generated by the AppBuilder, creates the window at run time, along with all the other objects it contains. A single 4GL procedure could certainly create multiple windows, but it is good to begin thinking in terms of a single procedure creating and managing a single visual object, whether it is a window or, later on, an intelligent version of one of the objects inside the window. This is the recommended way to structure your applications. It provides the framework you need to manage things easily.

Adding fields to your window

As you add visual objects to the window, the AppBuilder generates code to populate them with data automatically.

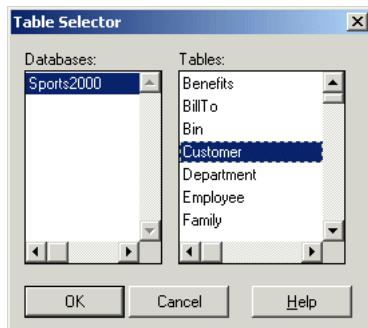


To add fields to your window:

1. Display some fields from the **Customer** table by choosing the **DB Fields** icon  on the Palette, and then double-click on your design window. The AppBuilder doesn't support true drag and drop between the Palette and the design window, so use two distinct clicks.

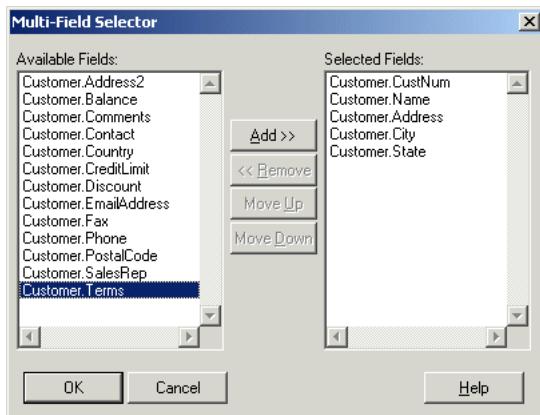
Because you've requested database fields, the **Table Selector** dialog box appears to let you select the database table that contains the fields.

2. Select **Customer** from the table list for the **Sports2000** database, then choose **OK**:

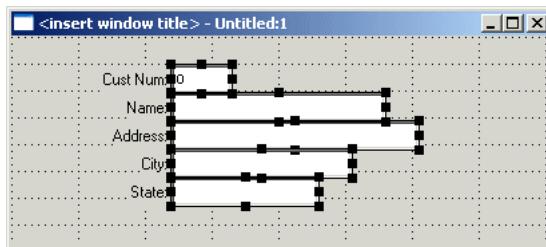


The **Multi-Field Selector** dialog box appears, showing all the fields in the **Customer** table.

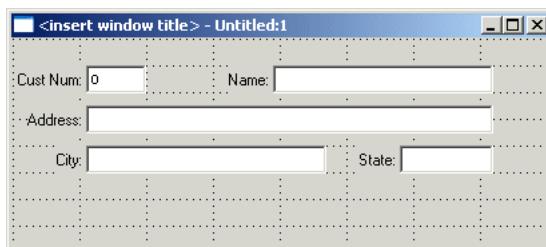
3. Select some fields by double-clicking on them in the **Available Fields** list or by selecting one or more fields and choosing the **Add** button. For example, select the **CustNum**, **Name**, **Address**, **City**, and **State** fields:



4. Choose **OK**. All the fields appear in your design window in a default layout:



5. To arrange the fields in a more useful way, first click in the window background to deselect the fields, and then click on each individual field and drag it to a more visually sensible location:



In [Chapter 2, “Using Basic 4GL Constructs,”](#) you saw how Progress displays objects in frames. When you create a window in the AppBuilder, it creates a default frame for you that fills the whole design window, and all the other objects, such as your **Customer** fields, are contained in this frame. If you click on the background of the design window, that is, in any part of the window that shows the layout grid, then you’re selecting that default frame, and it appears in the AppBuilder main window as the selected object. Logically enough, its name is **DEFAULT-FRAME**.

Changing object names and titles

This section describes how to change the names and titles of objects in your window.



To change the name and title of the default frame:

1. Click on the frame, then change the value of the **Object** field in the AppBuilder main window to **CustQuery**:



The name **CustQuery** is appropriate because the AppBuilder, by default, creates a database query for the fields in the frame, and gives it the same name as the frame. You’ll see just ahead what this default query does for you.

2. If you want to select another object such as a field, click on it.
3. To make a change to the window itself (for example, to edit its title), press **CTRL-Click** (that is, hold the **CTRL** key down and click the left mouse button).

The AppBuilder has defined a default name for the window of **C-Win**, which you can see in the **Object** field. Change this to **CustWin**. As you’ll see later, the object name is the name of the Progress 4GL variable that holds the handle or pointer to the window’s definition.

4. Modify the title in the **Title** fill-in field in the AppBuilder main window, to change it from the default window title of **<insert window title>** to something meaningful, such as **Customers and Orders**. The text in the **Title** field is what displays in the user interface of the window.

Saving a procedure from the AppBuilder

Before moving on, you should save your procedure.



To save your sample window:

1. Select **File→Save As** from the AppBuilder menu.
2. Give your window procedure the name **h-CustOrderWin1.w**.

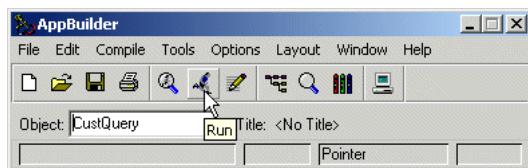
You can save it into the same directory as your **h-CustSample.p** procedure. The AppBuilder automatically gives the procedure a filename extension of **.w** instead of **.p**. For historical reasons, the **w** stands for *window*, even though you can create much more in the AppBuilder than windows. Although there is no absolute requirement that you observe this naming convention, it is strongly recommended that you do. The AppBuilder gives a default **.w** extension to any procedure it generates that has any user interface element at all, and also to some nonvisual procedures that can be selected and added to a window as part of its logic. Thus the **.w** extension can be taken to mean any user interface or other window procedure generated by, and therefore readable by, the AppBuilder. This designation can help you to keep your source procedures straight. Note that all Progress source procedures, regardless of their extension, compile into a file with a **.r** extension.

You should get into the habit right now of creating every procedure you write, except for the most trivial, in the AppBuilder rather than the Procedure Editor. This is part of the snowplow again. Chapter 1, “[Introducing the Progress 4GL](#),” showed you the Procedure Editor to get you started, but now you shouldn’t generally use it. There are templates available from the AppBuilder’s **New** dialog box for many kinds of procedures, including a *structured procedure* template that organizes code that doesn’t define any visual elements. By the end of this chapter the advantages of the AppBuilder format should be clear.

Running your procedure

You can already run your window to see what it looks like.

- To run your procedure as you did from the Procedure Editor, press **F2** or select **Compile→Run**. You can also choose the **Run** icon in the AppBuilder main window:

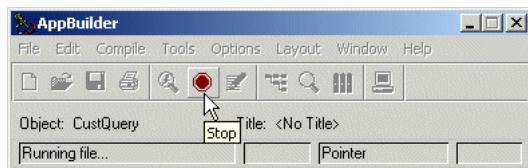


Here's what you see when you run (and resize) your window:



Not only do you get your window with its title and the **Customer** fields, but the AppBuilder has already generated code to open a query on the **Customer** table, retrieves the first **Customer**, and displays it for you. You'll look at the code that does this just a little later. But first, you need to complete the window.

- To stop the window, click the **Stop** icon, which replaced the **Run** icon:



Using the Query Builder

You can easily customize this default data retrieval by using a tool called the Query Builder, where you can define a WHERE clause and other elements of a query much as you defined the FOR EACH statement in [Chapter 2, “Using Basic 4GL Constructs.”](#)

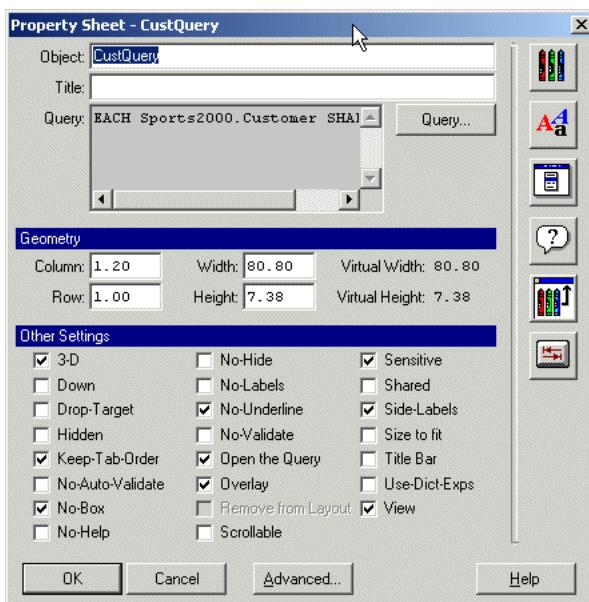


To bring up the Query Builder for the CustQuery frame:

1. Double-click on the frame background. Alternatively, select the **Object Properties** button from the toolbar:

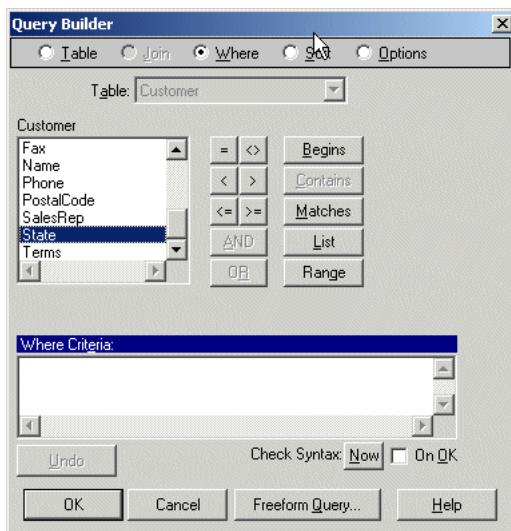


The property sheet for the frame appears, where you can define various frame properties:



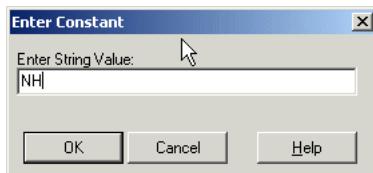
2. Choose the **Query** button to bring up the Query Builder.
3. To define a WHERE clause of `State = "NH"`, select the **Where** radio set button from the set of options at the top of the Query Builder.

4. Select **State** from the list of **Customer** fields:



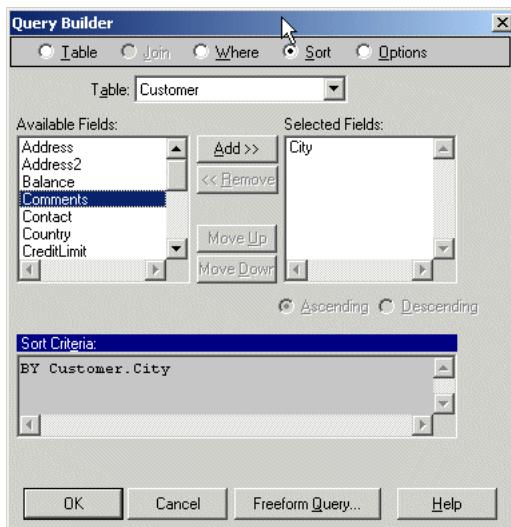
5. Choose the equal sign button $=$.

This brings up the **Enter Constant** dialog box where you can enter the **State** value **NH**. The Query Builder provides the quotation marks for you:



6. Choose **OK** to return to the Query Builder.
7. To add a sort clause BY **City** to your query, select the **Sort** radio set button.

8. Select the **City** field by double-clicking on it or by selecting it and choosing the **Add>>** button:



9. Choose **OK** to leave the Query Builder.
10. Choose **OK** again to leave the frame's property sheet.
11. **Run** your window again.

The first **Customer** is displayed again, but this time it is the first **Customer** in the **State** of New Hampshire, sorted by **City**:



Adding a browse to your window

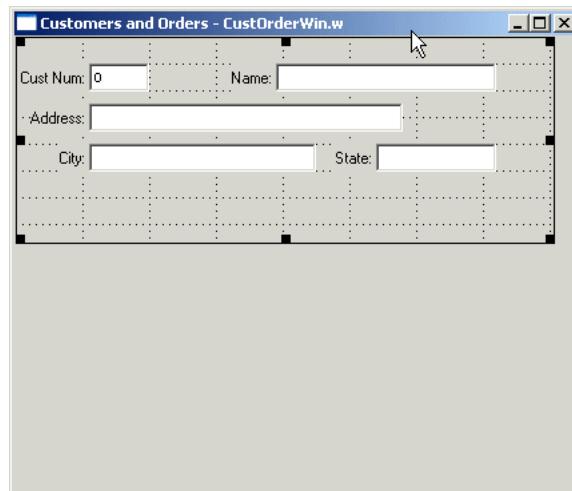
Next you're going to add a browse control to the window that displays **Orders**.



To add a browse to your window:

1. If you resized the window to make it smaller, expand it again to make room for the browse.

As you do this, the window's frame, which displays as a layout grid, should expand automatically with the window. It's possible that the frame and the window can get out of sync if you inadvertently resize the frame alone (by grabbing its resize handles) rather than the window (by grabbing the corner of the window). If this happens, the layout grid lines won't fill the whole window, like this:

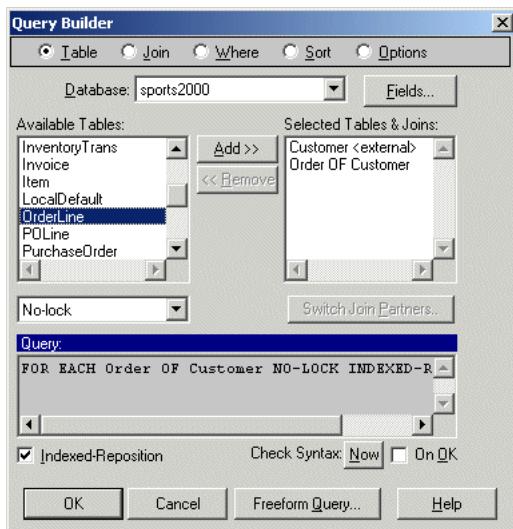


To correct this, grab the frame's lower-right resize handle and drag it down so that it fills the window. If you inadvertently try to drop any visual objects directly onto the window itself, the AppBuilder won't let you, because all the objects must be contained in, or *parented to*, the frame.

2. Choose the **Browse** icon  on the Palette and then click in the space under the **Customer** fields.

The **Query Builder** dialog box appears again to help you define a database query to populate this browse. You'll see that the **Customer** table has already been selected and marked as **<external>**, because by default the AppBuilder expects you to display **Orders** of the current **Customer**, and the **Customer** fields and their query are external to your browse.

3. Select **Order** from the **Available Tables** by double-clicking on it or by selecting it and choosing the **Add>>** button:



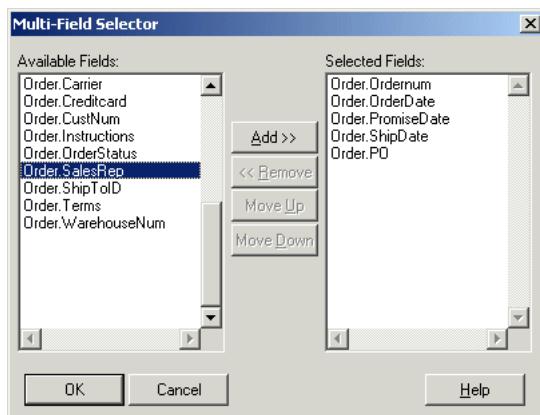
The Query Builder has constructed a query **FOR EACH Order OF Customer**. This is the same query you created in the sample procedure in [Chapter 1, “Introducing the Progress 4GL.”](#)

Next you need to select some fields to display in the browse.

4. Choose the **Fields** button. This brings up the **Column Editor**.
5. Choose the **Add** button to bring up the **Multi-Field Selector** dialog box.

This dialog box displays fields from both the **Customer** table and the **Order** table. It doesn't make much sense to put **Customer** fields into the browse, because they always have the same value for every **Order** of a given **Customer**.

6. Scroll down to the **Order** fields and select some:

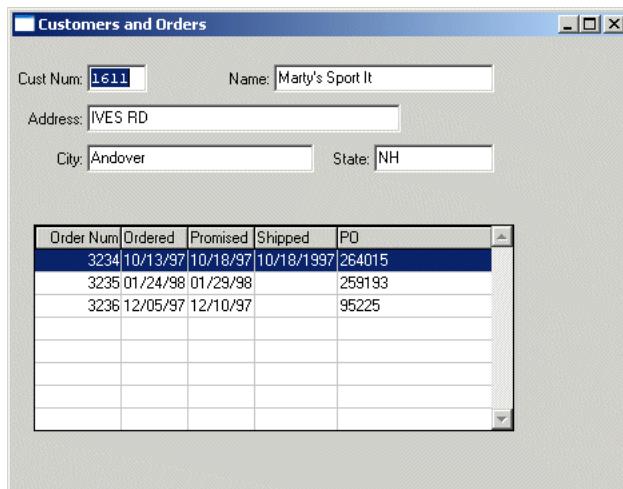


7. Choose **OK** when you're done.

Back in the **Column Editor**, there are various field attributes you could set to enable fields or change their label, and so forth.

8. For now just choose **OK** again. This returns you to the **Query Builder**.
9. Choose **OK** once more. This brings you back to the design window, where the AppBuilder has added the browse for you.

10. Resize the browse so that all the fields you selected appear. You can also make it taller to display as many rows as you wish.
 11. Press **F2** again to rerun your window:



Now the AppBuilder has constructed another query to select just those **Orders** for the current **Customer**, opened it, and used it to populate your browse. And you haven't written a line of 4GL code yet. This is an example of the power of the OpenEdge development tools.

Using property sheets

Every type of object you place onto a window has a property sheet where you can set various property values for it. This section does not describe property sheets in detail, but shows you how to bring up just one so that you know how they work.

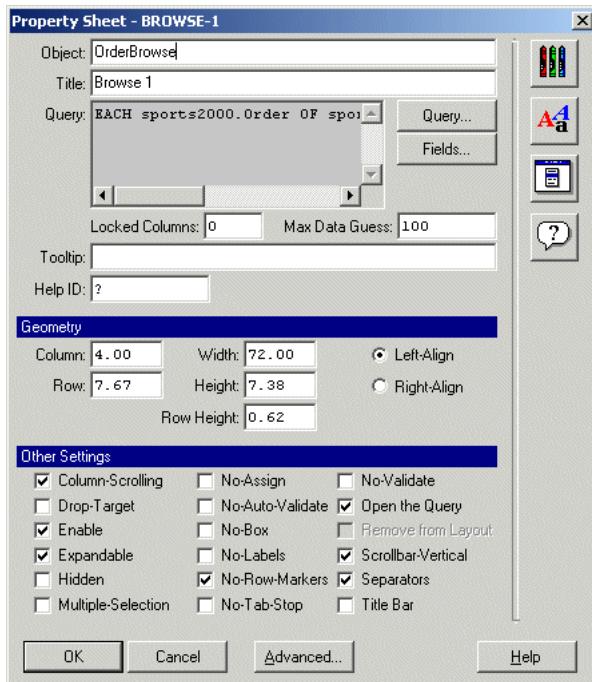


To set a property value for the browse:

- Double-click on the browse control in your design window. Alternatively, select the browse and then click on the **Object Properties** button in the AppBuilder toolbar.

The property sheet for the object comes up, in this case for the browse.

2. Change the **Object** name from **Browse-1** to **OrderBrowse**. This will make some of the generated code you look at later a little more readable:



3. Choose **OK** to exit the property sheet.

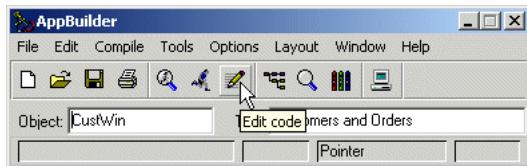
Using the Section Editor

Now you're ready to look at some of the code the AppBuilder has generated for you.



To view the code the AppBuilder created:

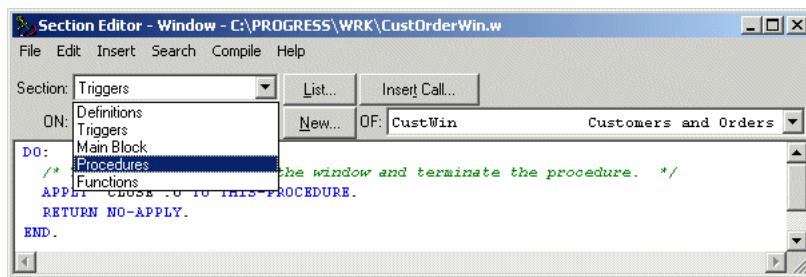
1. Stop your window, and then choose the **Edit Code** icon in the main window:



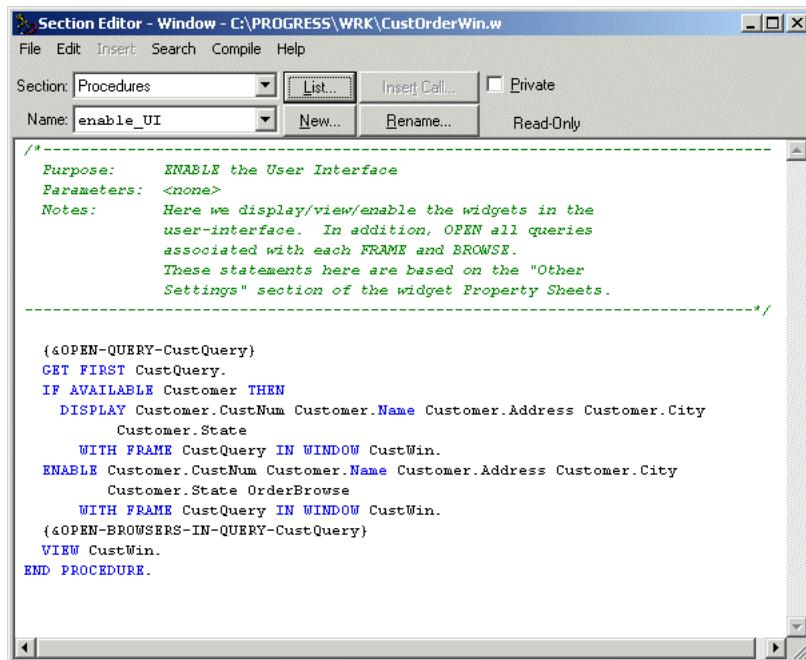
The **Section Editor** appears. This is where you examine and write all your 4GL code when you're using the AppBuilder. If you drop down the **Section** list, you see that there are five types of sections the **Section Editor** maintains for you:

- **Definitions** — A section at the top of your procedure where you can write variable definitions and other statements that are needed by the whole procedure.
- **Triggers** — Blocks of code that execute when an event occurs (for example, when a user chooses a button). You'll write some triggers of your own just ahead.
- **Main Block** — The part of the 4GL code that is executed as soon as the procedure starts up. You'll look at the main block of this sample procedure below and see what it does for you.
- **Procedures** — Internal procedures, exactly like the one you wrote in [Chapter 3, “Running Progress 4GL Procedures.”](#)
- **Functions** — User-defined functions are like internal procedures, but they return a value to the statement that uses them, just as the built-in 4GL functions you used in [Chapter 2, “Using Basic 4GL Constructs”](#) do.

2. Select **Procedures** from the list of section types:



3. From the list of internal procedures (the **Name** drop-down list), select **enable_UI**:



The AppBuilder generates this code to get your window started. This code does all the things that make the window, and then the data, appear.

Note that the AppBuilder has generated the header statement `PROCEDURE enable_UI` for you, but doesn't show it except to show the procedure name. Also note that the procedure is marked **Read-Only**. That's because the AppBuilder created it and doesn't want you to make any changes to it directly. If you change any objects in the design window, it changes the code for you.

When you create new procedures of your own, you can edit them in the **Section Editor** just as you did in the Procedure Editor.

Next you take a look at this code to see what it's doing for you. A word of warning before you start: Because this is AppBuilder-generated code, some of it isn't as readable as it might be if you wrote it by hand. The whole idea is that you almost never need to look at this code. But you're going to do that here so that you understand what's going on behind the scenes.

Looking at preprocessor values in the Code Preview

The first statement already looks pretty strange:

```
{&OPEN-QUERY-CustQuery}
```

What's *that* all about? A name enclosed in braces and preceded by an ampersand tells you this is a Progress *preprocessor* value. You are probably familiar with preprocessor values from other languages you've used. It is nothing more than a substitution string. It's defined up near the beginning of the procedure file, and everywhere it occurs in the 4GL code, the Progress syntax analyzer replaces it with the string it's been defined to represent. That way, you (or the AppBuilder) can define a commonly used string of code once and use it multiple times in your procedure.

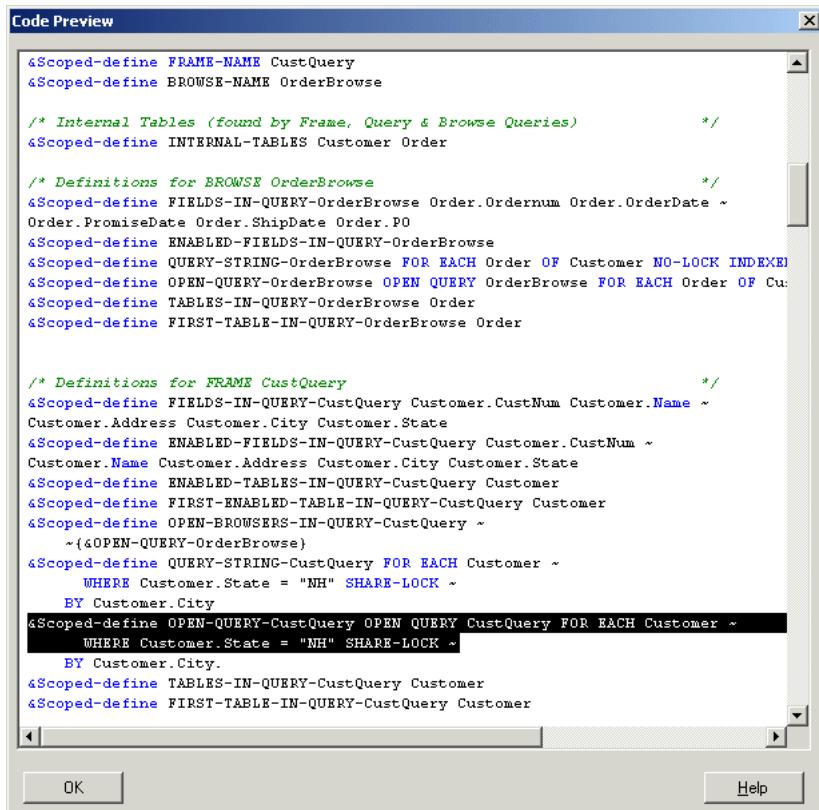
The AppBuilder generates the definition of this preprocessor value. Normally you don't need to look at it, so it's in a special section that isn't displayed along with the rest of your code sections.



To look at the preprocessor definition:

1. Return to the AppBuilder main window and press **F5** or select **Compile→Code Preview** from the menu. The **Code Preview** dialog box appears and shows all the code for the whole procedure.

2. Scroll down in this dialog box to the section marked **Preprocessor Definitions** to find the definition of OPEN-QUERY-CustQuery, as shown in [Figure 4–4](#).



The screenshot shows the 'Code Preview' window with the title bar 'Code Preview'. The main area contains a large amount of preprocessor code. A specific section is highlighted with a black rectangular selection, showing the definition of the OPEN-QUERY-CustQuery macro. The code is color-coded for syntax highlighting, with blue text for keywords like &Scoped-define and green text for comments. The highlighted section starts with '&Scoped-define OPEN-QUERY-CustQuery OPEN QUERY CustQuery FOR EACH Customer ~ WHERE Customer.State = "NH" SHARE-LOCK ~ BY Customer.City.' Below this, the code continues with more definitions for other queries and tables.

```

Code Preview

&Scoped-define FRAME-NAME CustQuery
&Scoped-define BROWSE-NAME OrderBrowse

/* Internal Tables (found by Frame, Query & Browse Queries)
   */
&Scoped-define INTERNAL-TABLES Customer Order

/* Definitions for BROWSE OrderBrowse
   */
&Scoped-define FIELDS-IN-QUERY-OrderBrowse Order.Ordernum Order.OrderDate ~
Order.PromiseDate Order.ShipDate Order.PO
&Scoped-define ENABLED-FIELDS-IN-QUERY-OrderBrowse
&Scoped-define QUERY-STRING-OrderBrowse FOR EACH Order OF Customer NO-LOCK INDEXED
&Scoped-define OPEN-QUERY-OrderBrowse OPEN QUERY OrderBrowse FOR EACH Order OF Cu-
&Scoped-define TABLES-IN-QUERY-OrderBrowse Order
&Scoped-define FIRST-TABLE-IN-QUERY-OrderBrowse Order

/* Definitions for FRAME CustQuery
   */
&Scoped-define FIELDS-IN-QUERY-CustQuery Customer.CustNum Customer.Name ~
Customer.Address Customer.City Customer.State
&Scoped-define ENABLED-FIELDS-IN-QUERY-CustQuery Customer.CustNum ~
Customer.Name Customer.Address Customer.City Customer.State
&Scoped-define ENABLED-TABLES-IN-QUERY-CustQuery Customer
&Scoped-define FIRST-ENABLED-TABLE-IN-QUERY-CustQuery Customer
&Scoped-define OPEN-BROWSERS-IN-QUERY-CustQuery ~
~(OPEN-QUERY-OrderBrowse)
&Scoped-define QUERY-STRING-CustQuery FOR EACH Customer ~
      WHERE Customer.State = "NH" SHARE-LOCK ~
      BY Customer.City
&Scoped-define OPEN-QUERY-CustQuery OPEN QUERY CustQuery FOR EACH Customer ~
      WHERE Customer.State = "NH" SHARE-LOCK ~
      BY Customer.City.
&Scoped-define TABLES-IN-QUERY-CustQuery Customer
&Scoped-define FIRST-TABLE-IN-QUERY-CustQuery Customer

```

Figure 4–4: Preprocessor definition in the Code Preview window

&Scoped-define means that this definition is scoped to just this procedure file, rather than being available to any other source files that are associated with this one. From this line you see that the preprocessor OPEN-QUERY-CustQuery is defined to represent the text OPEN QUERY CustQuery FOR EACH Customer SHARE-LOCK. In later chapters you'll explore the differences between defining and opening a query on a table and just retrieving the data using a FOR EACH statement by itself. For now it's enough to understand that the query named CustQuery defines the set of data to be retrieved, the OPEN QUERY statement starts the retrieval, and then other statements that you'll see next actually walk through the data row by row. The query definition itself is in another part of the AppBuilder-generated code, and just says DEFINE QUERY CustQuery FOR Customer.

Positioning within the query

The first statement opens that **Customer** query and sets you up to retrieve and display each of the **Customer** records in turn. Now look at the next statement:

```
GET FIRST CustQuery.
```

This statement retrieves the first **Customer** record using the query **CustQuery**.

Moving on, you see a **DISPLAY** statement, which should be familiar to you. It has an **IF-THEN** construct in front of it that you haven't seen before, but like most Progress 4GL statements, it's self-explanatory. If the query turns out to be empty, or if you've reached the end of it, then you don't want to try to display anything. The phrase **IF AVAILABLE Customer** checks for that:

```
IF AVAILABLE Customer THEN
    DISPLAY Customer.CustNum Customer.Name Customer.Address Customer.City
        Customer.State
    WITH FRAME CustQuery IN WINDOW CustWin.
```

Because a field name might occur in more than one table used in the procedure, the AppBuilder puts the table name in front of each field name, as in **Customer.CustNum**. This is a good practice for you to follow in the code you write unless your field names are all guaranteed to be unique.

The end of the **DISPLAY** statement has a qualifier you didn't see in previous chapters either, **WITH FRAME CustQuery**. But you can certainly tell what it does. [Chapter 2, “Using Basic 4GL Constructs,”](#) explains that Progress uses frames to define different display areas, and that each field and block of code is associated with a frame. If you don't name the frames yourself, you get default frames. If you need to, you can give each frame a name and specify that frame in a **DISPLAY** statement. Then Progress knows exactly where each displayed field or other object should go. You renamed your window's default frame **CustQuery** earlier, and here you see that name being used. And remember the AppBuilder uses the same name for the frame's default query.

The second qualifier is understandable, too. If your procedure happened to define more than one window, then you would have to tell it which frames go in which window. Here the code makes that explicit, by appending the phrase **IN WINDOW CustWin**.

So apart from these qualifiers this statement should look familiar. Now look at the next statement:

```
ENABLE Customer.CustNum Customer.Name Customer.Address Customer.City  
Customer.State OrderBrowse  
WITH FRAME CustQuery IN WINDOW CustWin.
```

Not surprisingly, this statement enables each of the fields for data entry, so that you could make changes to a record and then save it. Also it enables the browse control itself so that you can scroll it up and down. At this time, you won't actually add a **Save** button to this window to let you save changes (you'll do that in [Chapter 16, “Updating Your Database and Writing Triggers”](#)).

Next is another preprocessor value:

```
{&OPEN-BROWSERS-IN-QUERY-CustQuery}
```

If you do a **Code Preview** again (or refer to [Figure 4-4](#)) you can see that this value represents the statement OPEN QUERY OrderBrowse FOR EACH Order OF Customer NO-LOCK INDEXED-REPOSITION. You'll get to details like NO-LOCK versus SHARE-LOCK and what INDEXED-REPOSITION means in [Chapter 12, “Using the Browse Object.”](#) Otherwise, you should recognize this as the query you built in the Query Builder when you dropped the browse control onto the window.

The final executable statement is:

```
VIEW CustWin.
```

This statement tells Progress at run time to make the window visible.

And finally, as you learned in the [Chapter 3, “Running Progress 4GL Procedures,”](#) every internal procedure must end with an END PROCEDURE statement:

```
END PROCEDURE.
```

This is all the code that's needed to make all the things happen that you saw when you ran the window. It opens two different queries representing a parent-child relationship between **Customers** and **Orders**. It displays and then enables the **Customer** fields and the **Order** browse. And it views the window itself. All this happens in about ten lines of 4GL code. And you didn't have to write any of it yourself.

Adding buttons to your window

When your window comes up, it displays the first **Customer** and its **Orders** automatically. This is very nice, but of course you'd like to see other **Customers** as well. You can extend your window so that you can walk through the **Customers** in sequence and see the **Orders** for each **Customer** as you go. To do this, you'll add some buttons to the window and then define triggers for them. *Triggers* are blocks of 4GL code that are executed when the user chooses a button.

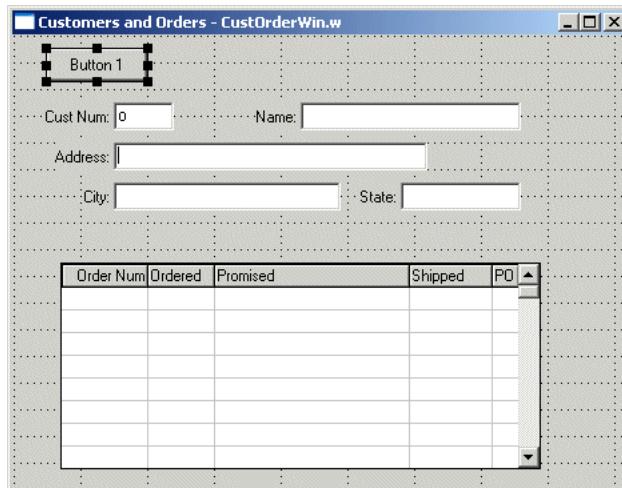
- To make room for some buttons, grab all the fields in your window by clicking with the mouse to the upper-left of the first field, and then dragging the mouse to draw a rectangle around all the fields. Drag them down slightly so that there's room at the top of the window.



To add a button to your window:

1. Choose the **Button** icon  on the **AppBuilder Palette** and then click again near the top of the design window.

The AppBuilder draws a button for you, gives it a default name of **Button-1**, and labels it **Button 1**:



2. In the AppBuilder main window, change the name to **BtnFirst** and the label to **First**:



You see that the AppBuilder changes the button label in the design window for you when you tab out of the **Label** fill-in field.

Defining a **CHOOSE** trigger for your button

Now you need to write some code for Progress to execute when the user chooses the **First** button. You can cheat a little here and copy some code you've already looked at, because it uses some query syntax that you won't read about in detail until a later chapter.



To define a CHOOSE trigger for your button:

1. Choose the **Edit Code** button to bring up the **Section Editor**, and go back to the procedure `enable_UI`.
2. Copy the two statements (five lines of code in all) starting with `GET FIRST CustQuery`. Select the text by clicking and dragging the mouse over it, then press **CTRL-C**:

Section Editor - Window - C:\PROGRESS\WRK\CustOrderWin.w

File Edit Insert Search Compile Help

Section: Procedures List... Insert Call... Private

Name: enable_UI New... Rename... Read-Only

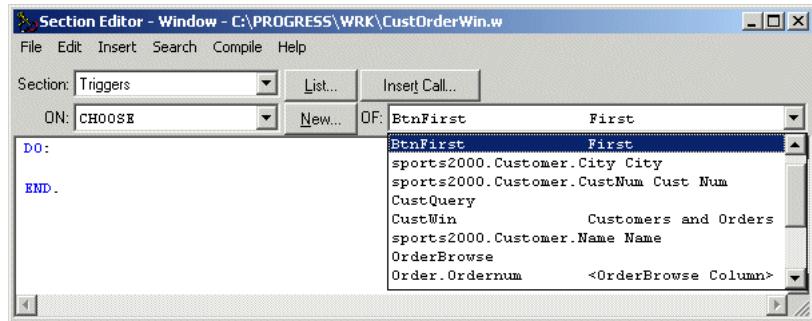
```

/*
  Purpose:      ENABLE the User Interface
  Parameters:   <none>
  Notes:        Here we display/view/enable the widgets in the
                user-interface. In addition, OPEN all queries
                associated with each FRAME and BROWSE.
                These statements here are based on the "Other
                Settings" section of the widget Property Sheets.
*/
(*OPEN-QUERY-CustQuery)
GET FIRST CustQuery.
IF AVAILABLE Customer THEN
  DISPLAY Customer.CustNum Customer.Name Customer.Address Customer.City
    Customer.State
    WITH FRAME CustQuery IN WINDOW CustWin.
ENABLE BtnFirst Customer.CustNum Customer.Name Customer.Address Customer.City
  Customer.State OrderBrowse
  WITH FRAME CustQuery IN WINDOW CustWin.
(*OPEN-BROWSERS-IN-QUERY-CustQuery)
VIEW CustWin.
END PROCEDURE.

```

Why do you select just these lines? The preprocessor value in the first line of the procedure opens the **Customer** query, and since it's already open when the user chooses a button, you don't need to do that again. Likewise, the fields and other objects in the window are enabled, so you don't need to do that again either. (By the way, you might notice that your new **BtnFirst** button has already been added to the list of things to enable.) And the window has been viewed, so you don't need that statement either. However, the statement before that, which opens the **Order** query, you do need, so you'll go back and get that in a moment.

3. Drop that same code into the trigger to execute when the user chooses the **First** button. You can get the **Section Editor** to move to that place in the procedure in two ways:
- Back in the design window, select the **First** button by clicking on it, and then select the **Edit Code** icon again.
 - In the **Section Editor**, select **Triggers** as the **Section** type. This might bring up the correct trigger for you. If it doesn't, then select **BtnFirst** from the drop-down list labeled **OF**:



In either case you get a starting point for some 4GL code defined within a DO-END block. You can see that the block of code is defined to execute ON CHOOSE OF BtnFirst.

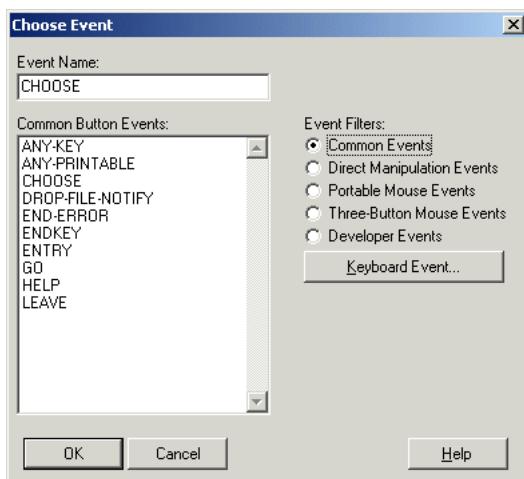
Defining user interface events

The CHOOSE event is the one you want to capture. An *event* is an action that happens while the application is running that Progress can intercept for you. Some events are associated with particular kinds of user interface objects, such as the CHOOSE event for a button. Others are defined for mouse clicks and other mouse events.



To define an event for the button:

1. Choose the **New** button in the **Section Editor** to see a list of all the events that you can define for the current object:



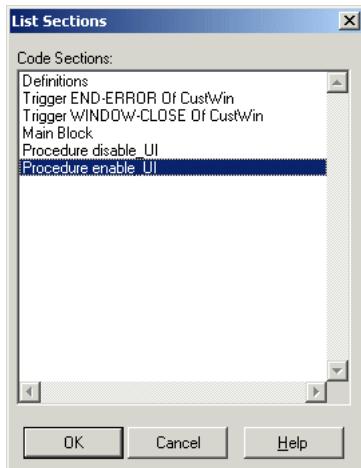
The CHOOSE event is the most common event for a button, so that's the event the **Section Editor** offers you by default.

2. Choose **Cancel** to exit the **Choose Event** dialog box and return to the **Section Editor**.
3. In the **Section Editor**, press **CTRL-V** or select the menu item **Edit→Paste** to paste the code you copied from the enable_UI procedure into your trigger.

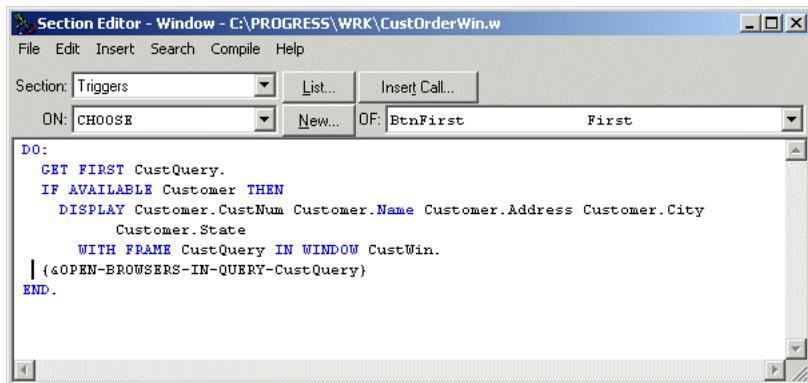


To go back and get the statement that opens the Order query for you:

1. Reselect the enable_UI procedure. You can do this by selecting **Procedures** from the **Section** type list. Or you can choose the **List** button, which brings up a list of all editable sections:



2. Copy the statement `{&OPEN-BROWSERS-IN-QUERY-CustQuery}` from enable_UI, go back to your trigger block, and paste it at the end of the block:



This is the code that positions to the first **Customer**, displays it, and opens a query to display its **Orders**.

**To repeat these steps for a Next button:**

1. Choose the **Button** icon on the **Palette**.
2. Drop another button onto your design window next to the **First** button.
3. Rename the button **BtnNext** and change its label to **Next**.
4. Copy the block of code from the **First** button and paste it into the same section for the **Next** button.
5. Change the first statement to **GET NEXT CustQuery**. This advances to the next record in the set of **Customers** in New Hampshire when you choose the button.
6. Save your procedure by selecting **File→Save** or by choosing the **Save** icon in the AppBuilder toolbar, and then rerun it.

Now you should be able to walk through the New Hampshire **Customers**. As you display each one by choosing the **Next** button, you see that its **Orders** are displayed in the browse.

**To finish adding buttons to your window:**

1. Add two more buttons to your window called **BtnPrev** and **BtnLast**, with the labels **Prev** and **Last**.
2. Copy the same block of code into the trigger sections for these two buttons.
3. Change the **GET FIRST** statement to **GET PREV** for the **BtnPrev** trigger, and to **GET LAST** for the **BtnLast** trigger.

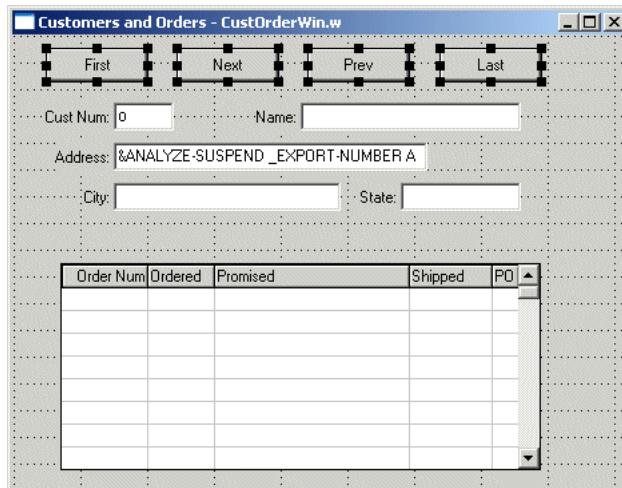
Adjusting the layout of the buttons

Now that you have all your buttons in a row, you can take advantage of the AppBuilder's layout options to line them up and space them out properly.

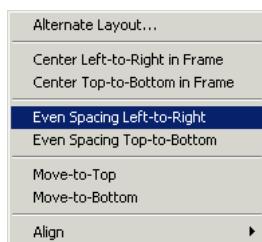


To select all four buttons and align them:

1. Select all the buttons at once, either by drawing a rectangle around them with the mouse or by selecting each one in turn by pressing **CTRL+LEFT-.MOUSE-CLICK**.



2. Select the **Layout** menu in the AppBuilder window:



Here you can see a number of methods to adjust the layout of your window.

If you select **Even Spacing Left-to-Right**, for instance, the AppBuilder adjusts the spacing in between the buttons to even it out.

3. Experiment with some of these options if you like. The **Alternate Layout** option is rather advanced, and you should leave this one until later. It lets you define different layouts of your window for different user interface types, or even for different user groups or other special customizations.

If you expand the **Align** item, you see a number of ways in which you can align a group of selected fields, buttons, or other objects in your window:



4. Run your window once again to see all your buttons in action.

This chapter gives you an overview of the OpenEdge AppBuilder. In the next chapter, you'll examine the code that the AppBuilder generates for you.

5

Examining the Code the AppBuilder Generates

In the previous chapter you didn't write much code, but rather let the AppBuilder do it for you. This might seem inappropriate in a book about how to learn to program in the Progress 4GL, but there's a reason for this approach. You need to be familiar with the syntax the AppBuilder generates for you when you put together a window in this way, but you needn't get into the habit of writing much code like this yourself. The tools are there to do this for you, and you should take advantage of them.

Having said that, in this chapter you'll look at more of the code the AppBuilder generated for your window, so that you understand how it works. By doing this, you can learn some new 4GL statements used to define visual objects and database queries. But then this chapter explains some of the ways in which this won't be typical of the complete applications you write. As you progress through the book, you'll learn about how the OpenEdge development tools can help you create applications where there is no compiled 4GL code for the user interface at all! But to work in that mode, you'll need to learn a little more about dynamic programming in Progress.

So this chapter provides a grounding in the static definitions that are the starting point for the more dynamic object definitions you'll use in later chapters. This chapter includes the following sections:

- [Viewing the entire sample procedure code](#)
- [Contrasting procedural and event-driven programs](#)
- [Looking ahead](#)

Viewing the entire sample procedure code

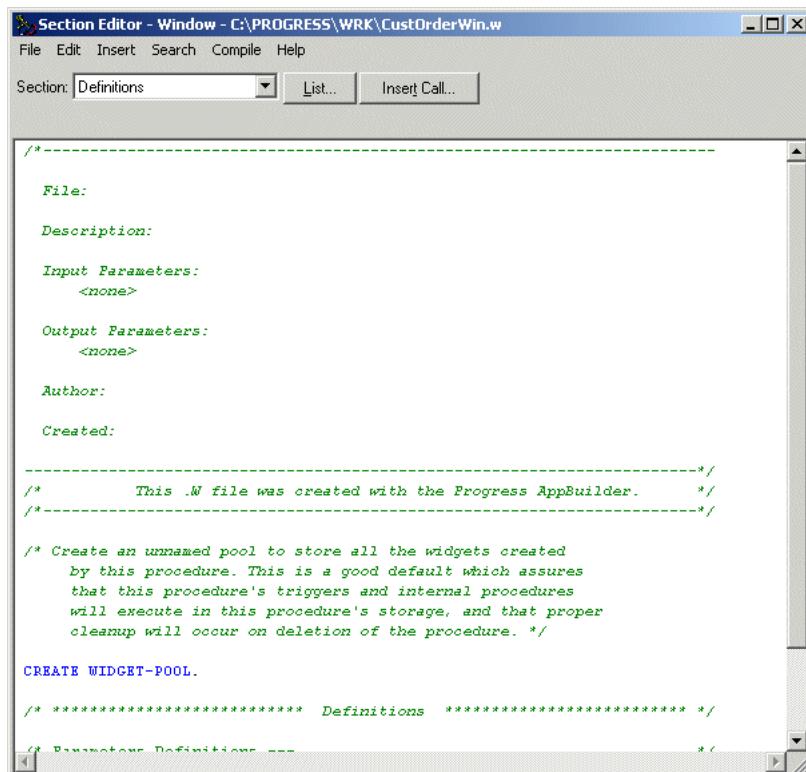
In the “[Using the Section Editor](#)” section on page 4–18, you saw some of the procedural code that gets the window up and running, such as the `enable_UI` procedure, but not the code that defines all the objects in the window. Take a look at that now. As before, since the AppBuilder generates and maintains that code for you based on how you lay out objects in the design window, it doesn’t show the code to you in the Section Editor.

- To view the entire procedure, you can select either **Compile→Code Preview** again or **File→Print** from the AppBuilder’s main menu to print a hard copy of the entire procedure.

Now look through the entire procedure to see some of the syntax that defines the window and its contents. If you look at a printed listing of the procedure file, you’ll notice numerous `ANALYZE-SUSPEND` and `ANALYZE-RESUME` preprocessors mixed in with the code. The AppBuilder uses these to separate out code it has generated and maintains for you from code you can add to the procedure yourself. These are stripped out of the **Code Preview** listing for you, making it a little easier to read. Just try to look beyond these things for the time being to learn what the statements are that really define the window.

The Definitions section

The top of the procedure file is the **Definitions** section. This section starts out as an empty template. It provides a space where you can enter documentation information about your procedure. You can access this section by selecting it as the **Section** type in the **Section Editor**, as shown in [Figure 5–1](#).



The screenshot shows the 'Section Editor - Window' application window. The title bar displays the path 'C:\PROGRESS\WRK\CustOrderWin.w'. The menu bar includes File, Edit, Insert, Search, Compile, and Help. A toolbar below the menu has buttons for 'Section' (set to 'Definitions'), 'List...', and 'Insert Call...'. The main text area contains the following code:

```
/*-----  
File:  
  
Description:  
  
Input Parameters:  
<none>  
  
Output Parameters:  
<none>  
  
Author:  
  
Created:  
  
*      This .W file was created with the Progress AppBuilder.  
*/-----  
  
/* Create an unnamed pool to store all the widgets created  
   by this procedure. This is a good default which assures  
   that this procedure's triggers and internal procedures  
   will execute in this procedure's storage, and that proper  
   cleanup will occur on deletion of the procedure. */  
  
CREATE WIDGET-POOL.  
  
/* ***** Definitions ***** */  
/* Parameters Definitions --- */
```

Figure 5–1: The Definitions section

It's a good idea to fill this section in and keep it up to date.

There's just one statement generated for you in this section: CREATE WIDGET-POOL. Objects that make up an application are sometimes called *widgets*. Progress uses *widget pools* to provide memory management for all of the objects that are created in a procedure, including the frames, fields, and buttons in your window as well as some data management objects such as queries. This statement gives you an unnamed widget pool for everything in the procedure. When the procedure finishes executing, all of the memory its objects use is released. This is a good default, but when you get to more advanced programming, you learn how to create named widget pools that have a lifetime you define for them. For the most part you don't have to worry about them.

Below the CREATE WIDGET-POOL statement is an area where you can define variables and parameters for your procedure. These variables are scoped to the external procedure. That is, any variables you define here are available for use throughout the procedure file, including any internal procedures it contains. You'll learn a lot more about scoping in [Chapter 6, “Procedure Blocks and Data Access.”](#)

Continuing your look through the **Code Preview** or printed listing, the next section of the file has all the **Preprocessor Definitions** the AppBuilder uses to define queries, field lists, and other syntax that can be used in other AppBuilder-generated code throughout the procedure. You've seen this before and looked at a few specific definitions in the “[Looking at preprocessor values in the Code Preview](#)” section on page 4–20.

Window, button, browse, and frame definitions

The next section is labeled **Control Definitions**. These are definitions for the objects in your window.

Defining a handle variable for the window

First is a definition for the window itself, or rather for a handle used to point to the window's definition:

```
DEFINE VAR CustWin AS WIDGET-HANDLE NO-UNDO.
```

There are a couple of things to note about this statement before you examine what a handle really does:

1. You can see that the AppBuilder has abbreviated the keyword **VARIABLE** to **VAR**. You can abbreviate many Progress 4GL keywords in this way, but you cannot use arbitrary abbreviations. That is, each keyword definition in the language also defines the acceptable abbreviation of the keyword. If you want to confirm the minimum abbreviation for a keyword or find out the kind of statements in which it is used, you can always look it up in the Keyword Index at the back of the third volume of the *OpenEdge Development: Progress 4GL Reference*. Not all keywords have abbreviations. You should generally *not* abbreviate any keywords. Typing keywords in full eliminates any chance of a conflict with a future keyword that starts the same way, and generally makes your code more readable. As described in the “Using the Intelligent Edit Control and its shortcuts” section on page 2–28, the Edit Control provides aliases for you to eliminate manually typing many of them in full.
2. The variable *CustWin* is defined as a **WIDGET-HANDLE**. The term *widget* applies to all sorts of things, including visual objects such as fields and buttons, as well as other things that can have handles, including queries and even procedures themselves. **WIDGET-HANDLE** is a synonym for the keyword **HANDLE**, because a single Progress data type accommodates all these different kinds of objects. Since there is really just one **HANDLE** data type, and since it is used for more than just objects that you might consider widgets, this book always uses the keyword **HANDLE**.

In the “[Creating the window](#)” section on page 5–8, you’ll look a little more about what the handle does when you get to the code that uses it to create the window.

Defining a button

Next is another kind of **DEFINE** statement. This one defines not a variable, but the first of your buttons:

```
DEFINE BUTTON BtnFirst
  LABEL "First"
  SIZE 15 BY 1.14.
```

There is a separate DEFINE statement for each different type of object you can have in your application. You can learn all the particular attributes you can set for each kind of object from the *OpenEdge Development: Progress 4GL Reference* or the online help, but they're similar in form. The DEFINE statement first names the object (**BtnFirst** in this case) and then has a list of whatever attributes you want to define for the object and their values. In this case the AppBuilder has defined the button's LABEL to be **First**, because this is what you set it to in the design window. The SIZE is a standard size the AppBuilder defaults, which you can change by resizing the button in the design window.

The following three statements are the definitions for the other three buttons in the window.

Defining a query

After the DEFINE BUTTON statement, the next statements define the two queries your procedure uses:

```
DEFINE QUERY OrderBrowse FOR  
    Order SCROLLING.  
  
DEFINE QUERY CustQuery FOR  
    Customer SCROLLING.
```

These statements define the query objects that your code later opens using the specific WHERE clause you defined in the Query Builder. The **Order** query got its name by default from the browse that displays its data, which you'll see next. The **Customer** query got its name from the frame it's in. You'll learn more about queries in Chapter 10, "Using Queries."

Defining a browse

Next is the statement to define your **Order** browse:

```
DEFINE BROWSE OrderBrowse  
    QUERY OrderBrowse NO-LOCK DISPLAY  
        Order.OrderNum FORMAT "zzzzzzzz9":U  
        Order.OrderDate FORMAT "99/99/99":U  
        Order.PromiseDate FORMAT "99/99/99":U  
        Order.ShipDate FORMAT "99/99/9999":U  
        Order.PO FORMAT "x(20)":U  
    WITH NO-ROW-MARKERS SEPARATORS SIZE 65 BY 6.67 ROW-HEIGHT-CHARS .57  
    EXPANDABLE.
```

This code first names the query the browse is defined for, and then the list of fields to display, along with their formats. Finally there's a list of attributes for the browse itself, in a phrase starting with the keyword **WITH**. You can set these and other attributes in the Browse property sheet you looked at in the “[Using property sheets](#)” section on page 4–16.

Defining a frame

Next is a section marked **Frame Definitions**, where the window's one frame is defined:

```
DEFINE FRAME CustQuery
  BtnFirst AT ROW 1.48 COL 8
  BtnNext AT ROW 1.48 COL 24.6
  BtnPrev AT ROW 1.48 COL 41.2
  BtnLast AT ROW 1.48 COL 58
  Customer.CustNum AT ROW 3.38 COL 13.4 COLON-ALIGNED
    VIEW-AS FILL-IN
    SIZE 9 BY 1
  Customer.Name AT ROW 3.38 COL 37 COLON-ALIGNED
    VIEW-AS FILL-IN
    SIZE 32 BY 1
  Customer.Address AT ROW 4.57 COL 13 COLON-ALIGNED
    VIEW-AS FILL-IN
    SIZE 37 BY 1
  Customer.City AT ROW 5.76 COL 13 COLON-ALIGNED
    VIEW-AS FILL-IN
    SIZE 27 BY 1
  Customer.State AT ROW 5.76 COL 47 COLON-ALIGNED
    VIEW-AS FILL-IN
    SIZE 22 BY 1
  OrderBrowse AT ROW 7.67 COL 13
  WITH 1 DOWN NO-BOX KEEP-TAB-ORDER OVERLAY
    SIDE-LABELS NO-UNDERLINE THREE-D
    AT COL 1 ROW 1
    SIZE 86 BY 13.67.
```

Here the code defines the position of each object in the frame. The exact position of the objects depends on how you laid them out in the design window. The four buttons are all at row position 1.48, counting in full character units.

Next come the fields from the **Customer** table. There are no **DEFINE** statements for these because the field definitions are taken automatically from the Data Dictionary definitions for the **Customer** fields.

Then the frame definition places the browse control at column 13 of row 7.67 .

Finally, the frame definition has its own WITH clause, where it sets the following frame attributes:

- **1 DOWN** — You'll remember from [Chapter 2, “Using Basic 4GL Constructs,”](#) that the kind of Progress frame you get from a FOR EACH block with a DISPLAY statement in it is a down frame that displays multiple records in a report-like format. Frames in a graphical application are typically *one down* frames, which display only one instance of the objects defined for the frame. In this case, the browse control is a single GUI object that takes the place of the multi-line down frame in the older interface style that's designed for character terminals.
- **NO-BOX, OVERLAY, NO-UNDERLINE, THREE-D** — These all define various visual characteristics of the frame and are self-explanatory.
- **KEEP-TAB-ORDER** — This attribute keeps language statements such as the ENABLE statement you saw in enable_UI from changing the tab order of the fields.
- **AT COL 1 ROW 1** — This position is relative to the window the frame is in. The objects in the frame are positioned relative to the frame (their container), and the frame is positioned relative to its container (the window).
- **SIZE 86 BY 13.67** — This is the size of the whole frame in characters.

For more information on any of the frame attributes, see their descriptions under the **Frame Phrase** entry of the [OpenEdge Development: Progress 4GL Reference](#).

Following the **Frame Definition** section are the **Procedure Settings**, which are specially formatted comments with information the AppBuilder uses internally. You should never edit special sections like this one.

Creating the window

Now things are going to get a little more interesting. The next part of the generated code, labeled **Create Window**, has a new type of statement in it that requires a brief explanation of a very basic Progress 4GL concept, that of static versus dynamic objects.

Static objects

Your procedure contains several DEFINE statements in it for a variable, a browse, and four buttons. These are called *static objects* because they are fully defined in the 4GL syntax that names them. Because of this, the Progress compiler knows most everything it needs to know about them when it compiles the procedure. This allows the compiler to store a complete

structure in the r-code that defines each static object. At run time the interpreter scans the r-code and builds each object based on its definition.

Dynamic Objects

The window is a different kind of object—a *dynamic* object. You use a CREATE statement rather than a DEFINE statement for it. When you create the object you associate the object with a HANDLE variable that must already be defined. It is for this that the DEFINE VARIABLE CustWin statement you saw earlier was coded. The handle acts as a pointer to a structure that describes the object, but it's different from the structure resulting from a DEFINE statement in that the structure is only built up at run time, as the program is executing. This allows you to set some or all of the object's attributes based on program conditions. So, in effect, the CREATE statement creates an empty shell to be filled in with the object's description, and the handle points to that shell. Here's the CREATE statement for your window:

```
CREATE WINDOW CustWin ASSIGN
  HIDDEN          = YES
  TITLE           = "Customers and Orders"
  HEIGHT          = 13.67
  WIDTH           = 86
  MAX-HEIGHT     = 16
  MAX-WIDTH      = 86.2
  VIRTUAL-HEIGHT = 16
  VIRTUAL-WIDTH   = 86.2
  RESIZE          = yes
  SCROLL-BARS    = no
  STATUS-AREA     = no
  BGCOLOR         = ?
  FGCOLOR         = ?
  KEEP-FRAME-Z-ORDER = yes
  THREE-D         = yes
  MESSAGE-AREA    = no
  SENSITIVE       = yes.
```

Around this code is an IF-THEN-ELSE statement:

```
IF SESSION:DISPLAY-TYPE = "GUI":U THEN
  CREATE WINDOW CustWin ...
ELSE {&WINDOW-NAME} = CURRENT-WINDOW.
```

This rather cryptic-looking sequence effectively says: “If you’re running in the GUI environment, as opposed to on a character device, then create the new window `CustWin`, which will appear as its own identifiable display space on the desktop. Otherwise use the default (and only) window that’s always there for character environments.”

This `DISPLAY-TYPE` test is the answer to a question that might have popped into your head:

Why has the AppBuilder made the window dynamic when everything else is static?

There is no `DEFINE WINDOW` statement in the 4GL, only the `CREATE WINDOW` statement. And this, in turn, is because Progress 4GL procedures are designed to be compilable for different environments largely without change, including graphical and character environments. There’s only one “window” in a character environment, and that is the entire display device. So your code can never ask a character device to create another window. Thus, to have the same code compile and run in both GUI and character, creating the window the GUI environment requires must be conditional. And that is exactly what dynamic objects are for: to let your procedures decide at run time what objects to create and what attributes to give them.

To set a dynamic object’s attributes, you normally reference the handle in later program statements. However, in this case the `CREATE WINDOW` statement itself has an `ASSIGN` phrase that sets all the window’s attributes to their proper initial values. In principle, you can define a dynamic object by associating it with a handle, and then set and reset its attributes as needed.

Setting attributes of dynamic objects

The next part of the procedure consists of mostly internal comments for the AppBuilder’s benefit, but there’s one executable statement in it, and you’ll look at it to learn one or two more things about dynamic objects:

```
IF SESSION:DISPLAY-TYPE = "GUI":U AND VALID-HANDLE(CustWin)
THEN CustWin:HIDDEN = no.
```

This statement means: “If the session’s Display Type is GUI and the window handle named `CustWin` is valid, then set the window’s `HIDDEN` attribute to `no`.”

The `VALID-HANDLE` built-in function tests to see if the structure the handle points to has been properly associated with an object.

Will the handle be valid? It should be because the procedure created the window that uses it just above this. But in your own procedures, you can use handles long after they’re defined. You must always make sure that they point to valid structures before you use them.

This language statement demonstrates that in the 4GL you can set the attributes of a dynamic object after the CREATE statement by using a reference to the object's handle, followed by a colon, followed by the attribute name. You can read attributes in the same way, as in:

```
IF CustWin:HIDDEN = no THEN
  .
  .
  .
```

Using dynamic objects is a very powerful way to write general-purpose procedures that can handle a whole set of variations on any common pattern in your application, whether it is windows with different titles, sizes, and contents, or browses on different queries with different columns displayed. You'll learn a lot more about these dynamic language constructs in later chapters.

Defining triggers

Skip down to the part of the code marked **Control Triggers**. Here you'll find the trigger blocks you defined for the buttons. Before these button triggers, there are a couple of standard AppBuilder-generated triggers to capture window events. Take a look at the second window trigger block:

```
ON WINDOW-CLOSE OF CustWin /* Customers and Orders */
DO:
  /* This event will close the window and terminate the procedure. */
  APPLY "CLOSE":U TO THIS-PROCEDURE.
  RETURN NO-APPLY.
END.
```

Note: The :U tag that follows the quoted string tells the compiler to leave this string out of its list of strings that might be sensible to translate into other human languages. Since the word CLOSE is just part of the program logic and not something a user would ever see or want to see in a different language, it should never be translated.

WINDOW-CLOSE is one of those events like the CHOOSE event for a button. Remember that each type of object has its own set of events it can capture. WINDOW-CLOSE, logically enough, is the event that a window receives when it is closed, for example, by choosing the standard close-window box in the corner of the window.

The code then cascades this event down to the running procedure itself, by applying the CLOSE event to the procedure. There's a special built-in function that always holds the handle of the currently running procedure: THIS-PROCEDURE. The APPLY statement makes an event happen just as a user action would. The event in this case is the CLOSE event for the procedure. Keep this in your mind for a few moments, and you'll soon see what the CLOSE event does.

The RETURN NO-APPLY statement just means: "Skip whatever action was in the queue of user interface actions, because we're getting out anyway."

The button triggers

Next come the four button triggers you defined yourself. Just look at the first of them to confirm the syntax of the ON statement:

```
ON CHOOSE OF BtnFirst IN FRAME CustQuery /* First */
DO:
  GET FIRST CustQuery.
  IF AVAILABLE Customer THEN
    DISPLAY Customer.CustNum Customer.Name Customer.Address Customer.City
      Customer.State
    WITH FRAME CustQuery IN WINDOW CustWin.
    {&OPEN-BROWSERS-IN-QUERY-CustQuery}
END.
```

When you define triggers in the **Section Editor**, it masks the syntax of the ON statement somewhat by putting the event name and the object name into fill-ins that you can select. It also automatically adds the IN FRAME qualifier for you, based on which frame contains the object.

Triggers as event-driven code

There's one important thing to note about the trigger blocks. In [Chapter 1, "Introducing the Progress 4GL,"](#) you learned that language statements are generally executed or processed in the order in which they appear in the procedure. Now you need to think about the difference between statements being *executed* and statements merely being *processed*. Definitional statements such as the DEFINE VARIABLE and DEFINE BUTTON statements aren't executed at all. They just tell Progress to set up the structures they define for later use. The trigger blocks, however, are executable blocks of code, but they aren't executed when they're first encountered. They are just set up to be executed when the event they are defined for occurs. You can think of the Progress compiler marching down through the procedure and defining the r-code for each executable statement in sequence. When it encounters a trigger block, it sets aside a special part of the r-code with those statements in it, keyed by the event that triggers the code.

This concept is fundamental to the nature of OpenEdge applications. These applications are called *event-driven*, because to a large extent the procedures in the application just set up blocks of code, and even entire procedures, to be available in memory when user-initiated events call for them.

Looking at the main block

So far most of the code you've looked at has been set-up code. It is either definitions of objects and variables or triggers to execute when events happen later on.

Finally you've arrived at the part of the procedure that is actually executed when you run the procedure. This is the *main block* of the procedure. Note that *main block* is not a Progress 4GL syntax element or a required structure in any way. It is really just a convention observed by the **Section Editor**. After all the definitions and triggers, you write the executable code for the procedure itself.

The main block of an AppBuilder-generated procedure that creates a window is entirely standard. You can add more initialization code to the block if you wish, but normally you won't want to remove what it already does. A few of these statements won't mean a lot to you yet, but for now take a quick look at this block to see a bit more about how events are set up for a typical GUI application.

The first statement simply assigns the value for a built-in function you saw earlier, the CURRENT-WINDOW. As the comment on the statement explains, this determines defaults for proper parenting of dialog boxes and frames to the windows with which they are associated:

```
/* Set CURRENT-WINDOW: this will parent dialog-boxes and frames.      */
ASSIGN CURRENT-WINDOW          = {&WINDOW-NAME}
      THIS-PROCEDURE:CURRENT-WINDOW = {&WINDOW-NAME}.
```

The next statement is more interesting. Remember that back in the Trigger section there is a trigger on the WINDOW-CLOSE event, then another trigger for when the user clicks on the close button in the upper-right corner of a window. This event in turn applies the CLOSE event to the executing procedure itself:

```
ON WINDOW-CLOSE OF CustWin /* Customers and Orders */
DO:
  /* This event will close the window and terminate the procedure. */
  APPLY "CLOSE":U TO THIS-PROCEDURE.
  RETURN NO-APPLY.
END.
```

Now you have come to the next step in this cascading series of actions:

```
ON CLOSE OF THIS-PROCEDURE
  RUN disable_UI.
```

When the user chooses the close button in the window, a trigger applies a CLOSE event to the procedure. Now here in the main block is another trigger block that defines the action for that CLOSE event, namely to run an internal procedure called `disable_UI`. This AppBuilder-generated procedure is the counterpart of `enable_UI`. It deletes the C-Win structure that defines the window and then deletes the procedure itself.

So now the AppBuilder has set up all the actions necessary to clean up when the window and the procedure that created it are no longer needed.

The next statement, PAUSE 0 BEFORE-HIDE, sets up a standard GUI default for automatically hiding windows and frames when the user selects some other object that is on top of them.

Now you get to the main block itself. The AppBuilder emphasizes this by giving the header name MAIN-BLOCK to the DO-END block, which is the heart of the procedure:

```
MAIN-BLOCK:
DO ON ERROR  UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
  ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:
    RUN enable_UI.
  IF NOT THIS-PROCEDURE:PERSISTENT THEN
    WAIT-FOR CLOSE OF THIS-PROCEDURE.
END.
```

The qualifiers on the DO statement aren't important for now. They simply assure that on an error, a cancel, or an escape request from the user, the block terminates. You'll learn more about undoing blocks in [Chapter 17, “Managing Transactions.”](#)

The first action the block takes is to run the internal procedure enable_UI. You've seen that enable_UI opens the queries, gets a **Customer**, displays it and its **Orders**, and enables all the fields. Remember that enable_UI doesn't have to create the window because one of the few executable statements in all the code you looked through before you got to the main block was the CREATE WINDOW statement. So that has already happened before this point in the code.

Don't worry about the meaning of THIS-PROCEDURE:PERSISTENT just yet. This condition isn't true for this window when you run it by itself, as you're doing. So Progress executes the statement WAIT-FOR CLOSE OF THIS-PROCEDURE.

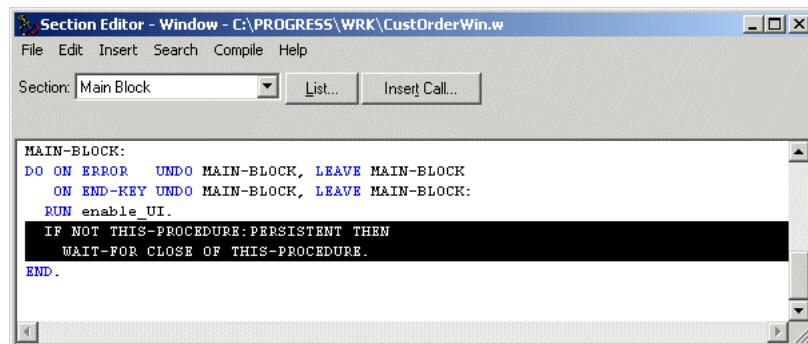
The WAIT-FOR statement is the crux of any event-driven application. A standard procedure in an older application, or any procedure that simply performs a task and then returns to its caller, proceeds through its list of statements and then ends. When Progress gets to the final executable statement in a procedure like that, it automatically destroys the procedure (giving back the memory its variables used) and returns to the caller.

But in our event-driven procedure, all that really has happened when this last statement is reached is to create a window, set up a few triggers, open two queries, display a record and a browse, and enable some fields. If the procedure were to terminate as soon as that was done, the user would have no chance to interact with the window.



To experiment and see what happens with and without the WAIT-FOR statement:

1. Highlight the two-line statement with the WAIT-FOR keyword in it:



The screenshot shows the Section Editor window with the title "Section Editor - Window - C:\PROGRESS\WRK\CustOrderWin.w". The menu bar includes File, Edit, Insert, Search, Compile, and Help. The toolbar has buttons for Section, Main Block, List..., and Insert Call... The main area shows the following code:

```

MAIN-BLOCK:
DO ON ERROR  UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
  ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:
    RUN enable_UI.
  IF NOT THIS-PROCEDURE:PERSISTENT THEN
    WAIT-FOR CLOSE OF THIS-PROCEDURE.
END.

```

The line "WAIT-FOR CLOSE OF THIS-PROCEDURE." is highlighted with a black rectangular selection.

2. From the AppBuilder menu, select **Edit→Format Selection→Comment**.

This option puts comment markers around the text you selected. The **Comment** and **Indent** options are a handy way to try things out with and without bits of code you're testing.

3. Choose the **Save** button or select **File→Save** from the menu to save **h-CustOrderWin1.w** back to the operating system.

Now comes the tricky part. If you simply run the window procedure now from the AppBuilder, it works perfectly well. That is, the window comes up and stays up until the user closes it. How come? The AppBuilder is smart enough to realize that whenever you run any procedure that has a window, it has to persist so that the user can interact with it. So it effectively adds a **WAIT-FOR** statement for you for testing purposes.



To see the actual effect of your change in a run-time environment::

1. To bring up a separate editor window, select **Tools→New Procedure Window** from the **AppBuilder** menu.
2. Type the statement: **RUN h-CustOrderWin1.w**.
3. Press **F2** to run this procedure.

The test window flashes almost imperceptibly and then disappears. All the initialization actions happen but then the procedure terminates. That's why you need the **WAIT-FOR**.



To correct this code:

1. Highlight the two lines you commented out and select **Edit→Format Selections→Uncomment**.
2. Choose the **Save** button to resave the procedure.
3. Select your Editor window and press **F2**.

Now the window comes up and stays up, waiting for the events that fire its trigger code, as you choose buttons, and then finally the close event that terminates the procedures and destroys the window.

This exercise illustrates why you have to tell Progress not to terminate the procedure until the user is done with it. This is what the **WAIT-FOR** statement does. And when you write a **WAIT-FOR** statement, you have to tell Progress to wait for some particular event, because otherwise the procedure would never end.

So the statement says to wait for the CLOSE event on the procedure. And when will this happen? It will be when the user closes the window, as you've seen. So at that point the main block ends, the CLOSE trigger fires, which executes the `disable_UI` code to clean up the resources the procedure and the window use, and everything goes away.

The internal procedures

Following the main block you find the code for the two internal procedures the AppBuilder generated, `disable_UI` and `enable_UI`. If you use the **Section Editor** to define internal procedures, the AppBuilder places them at the end of the source file and keeps them in alphabetical order for you. When the compiler encounters an internal procedure it acts much as it does when it finds a trigger block. It compiles the code and places it in a part of the r-code identified by the procedure name rather than by an event. When Progress encounters a RUN statement for the internal procedure as it executes, it transfers control to the statements for the internal procedure and then returns to the main procedure when it completes.

You can place internal procedure definitions anywhere within the source procedure file. By convention you should place them at the end, but this is strictly an organizational preference. If you use the AppBuilder and its **Section Editor** to create your procedures, which you should almost always do, then they will organize the code consistently for you.

Contrasting procedural and event-driven programs

Understanding the difference between procedural and event-driven programs is one of the most important requirements of learning how to write OpenEdge applications. The program you have written so far is a simple but good example of a procedural program. It executes basically from the top down. When it comes to a statement to RUN another procedure, such as the `calcDays` internal procedure, then it initiates that procedure, passes parameters into it, and when it gets to the end, returns any output parameters, and that's the end of it. There's no code placed off to the side for events initiated by the user. The programmer determines the procedure flow. A program doesn't prompt the user for input until it's ready to. The user has little ability to move around in the interface flexibly. This is typical of a character interface application. The simple diagram in [Figure 5–2](#) illustrates top-down or hierarchy programming.

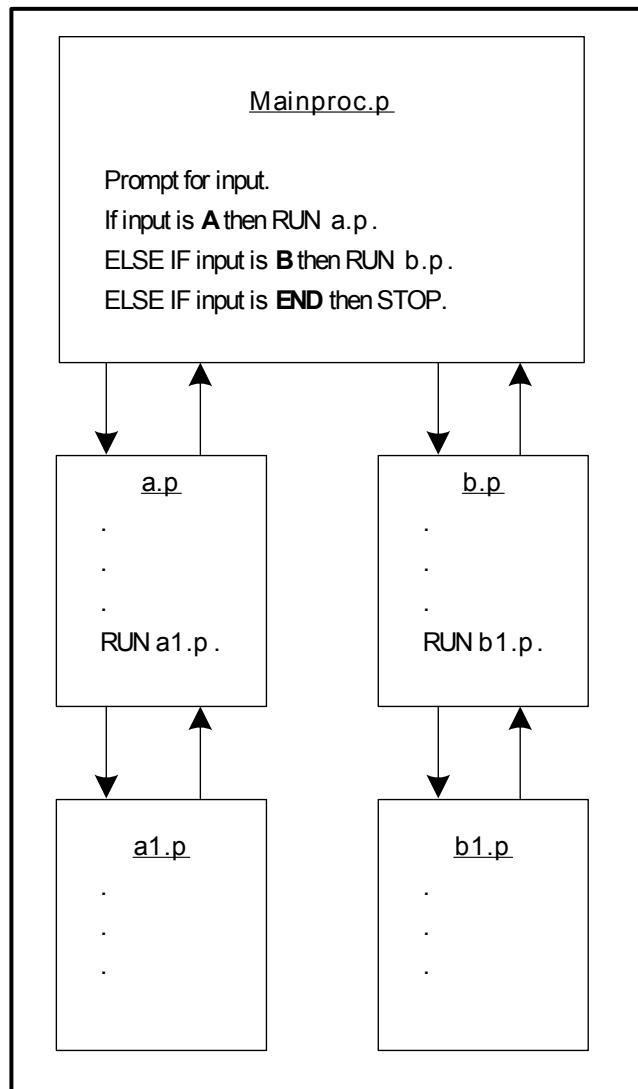


Figure 5–2: Procedural top-down application flow

When the procedure `mainproc.p` runs `a.p`, and then `a.p` runs `a1.p`, then `mainproc.p`, `a.p`, and `a1.p` are in memory. When `a1.p` returns, its context is removed and it goes out of scope. When `a.p` returns, its context is deleted. And only then can the code run `b.p`, which might run `b1.p`. The thread of execution goes right up and down through this hierarchy of called procedures.

By contrast, your sample procedure from this chapter is a simple but very good example of an event-driven procedure, as illustrated in [Figure 5–3](#). Most of the code the interpreter encounters as it initializes the procedure doesn't actually get executed. It just sets up bodies of code to run when the user requests it.

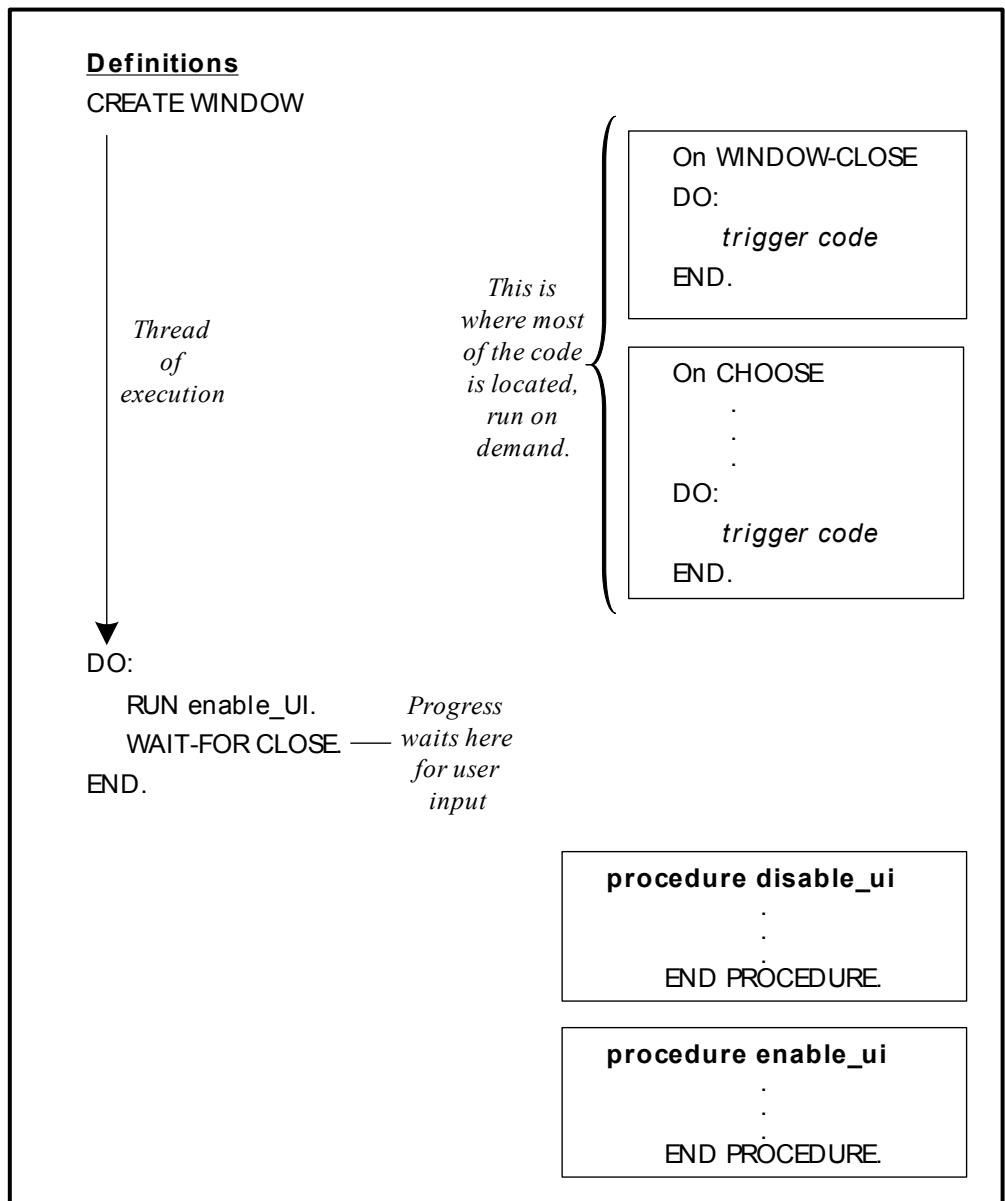


Figure 5–3: Event-driven application flow

Advantages of the AppBuilder file format

You have seen that the AppBuilder applies a special format to the procedures it generates, regardless of whether they represent a graphical user interface definition or just procedural logic for your application. When you open any procedure created with the AppBuilder, it parses this special format, which includes some specially formatted Progress comments that it alone is expected to read, and identifies all the different sections of the procedure, including object definitions and internal procedures.

Scanning through a long procedure in a listing, or in an editor window, or even in the Code Preview window, doesn't give you a very good overview of what the procedure does, how it's put together, and what its contents are. By contrast, when you use the **Section Editor**, you get a consistent predefined format to your procedures, with only the parts you're generally interested in shown to you for review or editing. You can get a proper listing of all the elements in the procedure, either all together by choosing the **List** button or grouped by **Section** type. You can also print individual sections from the **Section Editor**. You can print the entire file from the main menu.

The **Section Editor** provides shortcuts for inserting all manner of text into your procedure, under the **Insert** menu shown in Figure 5–4.

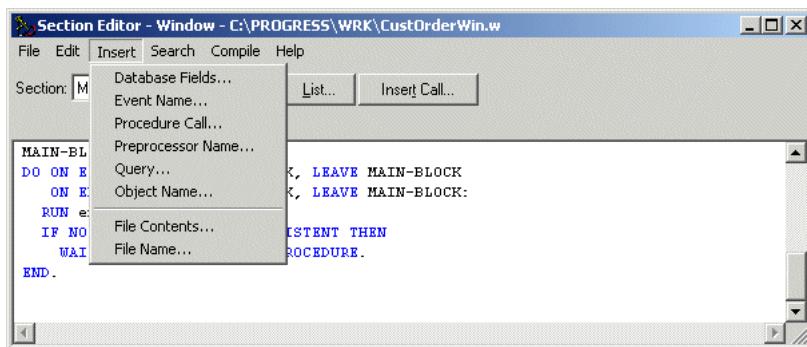


Figure 5–4: Section Editor Insert menu

The following options appear under the **Insert** menu:

- **Database Fields** — Shows you a list of all tables and fields in your connected databases so you can verify field names you need to insert into a procedure.
- **Event Name** — Brings up a list of all possible user interface events.
- **Procedure Call** — Shows you a list of all internal procedures so that you can insert a RUN statement for one of them. (Also shown as the **Insert Call** button in the window.)

- **Preprocessor Name** — Brings up a list of all the preprocessor definitions in the procedure.
- **Query** — Brings up the Query Builder.
- **Object Name** — Brings up a list of all fields, variables, and other objects in the procedure.
- **File Contents** — Lets you insert the contents of an operating system file.
- **File Name** — Lets you insert the name of a file (for example, in a RUN statement).

All these options are useful aids that can increase your productivity and accuracy as a programmer.

There's one more extremely important reason to use the AppBuilder whenever possible. In the example you've worked with in this chapter, the AppBuilder generates 4GL code and manages its format in a source file. The AppBuilder can do much more than that. In conjunction with the Progress Dynamics development framework and Progress SmartObjects, the AppBuilder can convert some kinds of procedures into data-driven application components, whose definitions are stored in a repository database and created from that data at run time, eliminating the need for procedural code altogether. This is part of a very advanced concept in OpenEdge development. Keep in mind that any procedure that is readable by the AppBuilder is in a better position to be converted to another useful form automatically, which won't be the case with hand-written procedures.

Looking ahead

Everything you've looked at in these first chapters looks like a pretty easy and powerful way to use the language and the tools to put applications together. And yet there's a lot more to it than this. In some ways you're still at the snowplow stage of learning principles that you'll later apply in very different ways. So what are the essential limitations of the kind of AppBuilder-generated procedure you built in this chapter?

Reusable components

For one thing, the sample procedure is not easily reusable. If you wanted to build a hundred table maintenance windows that all work much the same way (and that's a basic part of nearly any real business application), you'd have to set up an assembly line process to create them all. In the end you'd wind up with many procedures that create windows that all look and act more or less the same, but they would all be separate procedures requiring separate maintenance and testing. If you needed to make a change to your design, you'd need to make that change to all these separate procedures, and then retest and redeploy them. This would be a major headache and a source of unreliability in your application.

At the same time, you would undoubtedly want to extend the behavior of these windows far beyond what the simple **Customers and Orders** window does. You might want a real toolbar, for example, with standard icons and a menu. You might want the browse to perform a lot of additional tasks, such as sorting and column reordering and so forth. You might want other kinds of controls in the window, such as drop-down lists of valid values. If you coded each of these additional features for each window, you'd have a tremendous amount of work on your hands.

So OpenEdge provides a set of standard components that do many of these things for you, which you can use to build many different kinds of application windows, as well as the back-end business logic those windows talk to. You can get to know these *Progress SmartObjects™* in other OpenEdge documentation.

User interface independence

You might also like to make changes to the interface of your application, and even to change the client platform your application runs on, without having to rewrite your procedures. OpenEdge provides you with the tools to do this as well.

Distributed applications

Another limitation of your sample procedure is that it presumes that the client session running the interface has direct access to the database the application data is coming from. In a modern application, this is normally not the case. Whether you're running your application on the Web with a browser-based UI, or simply providing global access to your database server from client machines running all over the country or all over the world, you won't be able to provide all of your users with the same kind of direct database connection that you may have had with older host-based or client/server applications. Later chapters in this book introduce you to how to use the Progress 4GL to construct business logic procedures that can run close to your database server using the OpenEdge AppServer™ technology.

Dynamic programming

And perhaps most remarkable of all, in [Chapter 9, “Using Graphical Objects in Your Interface,”](#) you'll learn about 4GL constructs that help you build dynamic procedures that can handle whole classes of behavior, so that one procedure can replace dozens or hundreds of older procedures that all did a similar kind of job for you. In this way, you'll learn how to write more effective and certainly more flexible and maintainable applications by writing far less code than you used to.

So there is a lot of exciting territory ahead. But before you get into some of the more advanced topics, such as how to work with the OpenEdge AppServer and how to write dynamic procedures, there are a lot of important and very powerful basics still to cover. The next chapter, for instance, goes into more depth on some of the block-structured principles that make the Progress 4GL work and how you can use these blocks to retrieve and manage application data. Onward!

Procedure Blocks and Data Access

In the first three chapters of this book, you learn about the basic structure of the Progress 4GL and many of its language constructs. The next two chapters focus on the user interface of a GUI application as you learn to use the AppBuilder to build an application window, examining the language syntax that defines the elements in the window along the way. This chapter returns you to writing 4GL procedures on your own. These procedures contain some business logic that demonstrates in detail a few of the concepts touched on in the previous chapters.

The important concepts of this chapter are ones you've already seen in action and learned something about. Indeed, as discussed with the very simplest Progress 4GL procedure—`FOR EACH Customer: DISPLAY Customer.`—you can hardly write any 4GL code at all without defining blocks and using data access statements.

Nonetheless, this chapter goes into a lot more detail in these areas so that you have a more thorough understanding of these basic building blocks of Progress 4GL procedures. In this chapter, you'll learn:

- All the basic syntax for defining blocks of various kinds in the language, including some that iterate through a set of statements and some that just group them as a common action.
- About the scoping of blocks and of the variables and objects you define in them and how scoping affects the way your procedures behave.
- A variety of ways to access database data and integrate it into your procedures.

This chapter includes the following sections:

- [Blocks and block properties](#)
- [Procedure block scoping](#)
- [Language statements that define blocks](#)

Blocks and block properties

You saw several different kinds of blocks in the example procedures from the first two chapters. To review them:

- Every procedure itself constitutes a block, even just the simplest RUN statement executed from an editor window.
- Every call to another procedure, whether internal or external, starts another block. An external procedure can contain one or more internal procedures, each of which forms its own block.
- Progress 4GL statements such as FOR EACH and DO define the start of a new block.
- A trigger is a block of its own.

Procedure block scoping

Scope is the duration that a resource such as a variable or a button is available to an application. Blocks determine the scope of the resources defined within them.

This section describes some of the basic rules pertaining to procedures and scope. Variable and object definitions are always scoped to the procedure they are defined in. In this book, the word *object* refers to the various kinds of visual controls you can define, such as buttons and browses, as well as queries and other things you'll work with later.

You wrote some variable definitions in your very first procedure:

```
/* h-CustSample.p -- shows a few things about the Progress 4GL */

DEFINE VARIABLE cMonthList AS CHARACTER NO-UNDO
    INIT "JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC".
DEFINE VARIABLE iDays    AS INTEGER   NO-UNDO. /* used in calcDays proc */
```

Progress scopes those variables to the procedure. They are available everywhere within that main procedure block and every block it contains, including any internal procedures and triggers. For instance, you could write a line of code inside the calcDays internal procedure that is part of h-CustSample.p, and that code would compile and execute successfully. It would use the same copy of the variable that the enclosing procedure uses.

If you define variables or other objects within an internal procedure, then they are scoped to that internal procedure only and are not available elsewhere in the external procedure that contains it. You can use the same variable name both in an internal procedure and in its containing external procedure. You'll get a second variable with the same name but a distinct storage location in memory and therefore its own distinct value.

Here are a few simple examples to illustrate this point. In the first, the variable `cVar` is defined in the external procedure and therefore available, not only within it, but within the internal procedure `subproc` as well:

```
/* mainproc.p */
DEFINE VARIABLE cVar AS CHARACTER NO-UNDO.
cVar = "Mainproc". /* This is scoped to the whole external procedure. */

RUN subproc.

PROCEDURE subproc:
  DISPLAY cVar.
END PROCEDURE.
```

If you run this procedure, you see the value the variable was given in `mainproc` as displayed from the contained procedure `subproc`, as shown in [Figure 6–1](#).



Figure 6–1: Result of variable defined in main procedure only

By contrast, if you define cVar in the subprocedure as well, it can have its own value:

```
/* mainproc.p */
DEFINE VARIABLE cVar AS CHARACTER NO-UNDO.
cVar = "Mainproc". /* This is scoped to the whole external procedure. */

RUN subproc.
DISPLAY cVar.

PROCEDURE subproc:
  DEFINE VARIABLE cVar AS CHARACTER NO-UNDO.
  cVar = "Subproc".
END PROCEDURE.
```

Run this code and you get the same result you did before, as shown in [Figure 6–2](#).



Figure 6–2: Result of variable defined in both main and subprocedures

You assign a different value to the variable in the subprocedure, but because the subprocedure has its own copy of the variable, that value exists only within the subprocedure. Back in the main procedure, the value Mainproc is not overwritten even after the subprocedure call.

As a third example, if you define a new variable in the internal procedure, it won't be available in the main procedure at all:

```
/* mainproc.p */
DEFINE VARIABLE cVar AS CHARACTER NO-UNDO.
cVar = "Mainproc". /* This is scoped to the whole external procedure. */

RUN subproc.
DISPLAY cVar cSubVar.

PROCEDURE subproc:
  DEFINE VARIABLE cVar      AS CHARACTER NO-UNDO.
  DEFINE VARIABLE cSubVar AS CHARACTER NO-UNDO.
  cVar = "Subproc".
  cSubVar = "LocalVal".
END PROCEDURE.
```

Here cSubVar is defined only in the internal procedure, so when you try to display it from the main procedure block you get an error, as shown in [Figure 6–3](#).

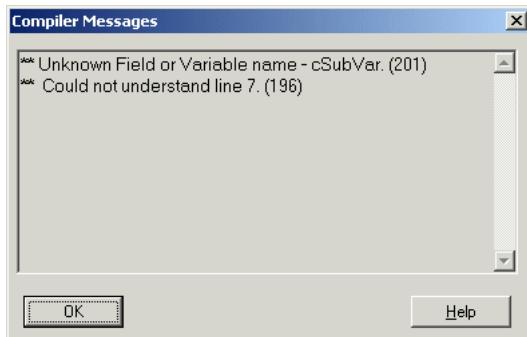


Figure 6–3: Result of variable defined in the subprocedure only

The main procedure block where the DISPLAY statement is located never heard of cSubVar, because it's defined in a procedure block inside of that. Even though it's defined within the same source procedure file, it's as separate from the main procedure block as it would be if it were in a completely separate external procedure file.

Language statements that define blocks

You've seen the `FOR EACH` block and a simple `DO` block. This section reviews all the statements that define blocks so that you can then study the differences between them and how each type of block is used.

DO blocks

A `DO` block is the most basic programming block in the 4GL. The keyword `DO` starts a block of statements without doing anything else with those statements except grouping them, unless you tell it to. You've already used the keyword `DO` as a block header in a couple of ways, including your trigger blocks, such as this trigger on the **Next** button in `h-CustOrderWin1.w`:

```
DO:  
    GET NEXT CustQuery.  
    IF AVAILABLE Customer THEN  
        DISPLAY Customer.CustNum Customer.Name Customer.Address Customer.City  
            Customer.State  
        WITH FRAME CustQuery IN WINDOW CustWin.  
        {&OPEN-BROWSERS-IN-QUERY-CustQuery}  
    END.
```

This block only assures that all the statements are executed together when the event occurs. If you take a closer look at this trigger, you can use another `DO` block inside it to correct a small error in the program logic.

**To see the error in the program logic:**

1. Run h-CustOrderWin1.w.
2. Choose the **Last** button to see the last **Customer** and its **Orders**.
3. Choose the **Next** button.

There is no next **Customer**, so the **Customer** record is not changed. The **IF AVAILABLE Customer** phrase in the **Next** button trigger assures that nothing happens if there is no **Customer** to display. However, the preprocessor **{&OPEN-BROWSERS-IN-QUERY-CustQuery}**, which opens the **Order** query, isn't affected by the **IF AVAILABLE Customer** check, because it's a separate statement. So the code opens the **Order** query even if there's no current **Customer**, and therefore displays nothing, as shown in [Figure 6–4](#).

The screenshot shows a Windows application window titled "Customers and Orders". At the top, there are four buttons: "First", "Next", "Prev", and "Last". Below these are four text input fields: "Cust Num: 66", "Name: First Down Football", "Address: 354 Edmonds Ave", and "City: Loudon State: NH". At the bottom is a table with five columns: "Order Num", "Ordered", "Promised", "Shipped", and "PO". The table has several empty rows, indicating no data is present.

Figure 6–4: Example of empty query result

If there's no **Customer** you shouldn't open the **Order** query, so you need to bring both statements into the code that is executed only when a **Customer** is available.



To correct the statement that opens the query:

1. Create another DO-END block in the trigger:

```

DO:
  GET NEXT CustQuery.
  IF AVAILABLE Customer THEN
    DO:
      DISPLAY Customer.CustNum Customer.Name Customer.Address Customer.City
          Customer.State
      WITH FRAME CustQuery IN WINDOW CustWin.
      {&OPEN-BROWSERS-IN-QUERY-CustQuery}
    END.    /* END DO IF AVAILABLE Customer */
  END.

```

Now the statement that opens the query won't execute either if there's no **Customer**. This is another illustration of how to use the DO block as a way to group multiple statements together, so that they all execute together or not at all. As this example illustrates, you can nest DO blocks as much as you wish.

When you enter this code in the **Section Editor**, the edit control recognizes the keyword DO followed by a colon and automatically adds the matching END statement. Just move this statement to where it belongs at the end of the new block. The editor generally matches the indentation of END statements correctly with the start of the block. Make sure you indent all the statements in the block properly so that someone reading your code can easily see how the logic is organized. It's also a good idea to get into the habit of always putting a comment with each END statement to clarify which block it's ending. When your code gets complex enough that a single set of nested blocks takes up more than a page, you'll be grateful you did this. It can prevent all sorts of logic errors.

2. Make this same DO-END correction to the trigger code for the **Prev** button.
3. Save the window as **h-CustOrderWin2.w**.

The DO block is considered the most basic kind of block because Progress doesn't provide any additional services to the block by default. There is no automatic looping within the block, no record reading, and no other special processing done for you behind the scenes. However, you can get a DO block to provide some of these services by adding keywords to the DO statement. The following section provides some examples.

Looping with a DO block

If you want to loop through a group of statements some specific number of times, use this form of the DO statement:

```
DO variable = expression1 TO expression2 [ BY constant ]:
```

The following example adds up the integers from one through five and displays the total:

```
DEFINE VARIABLE iCount AS INTEGER      NO-UNDO.  
DEFINE VARIABLE iTotals AS INTEGER     NO-UNDO.  
  
DO iCount = 1 TO 5:  
    iTotals = iTotals + iCount.  
END.  
DISPLAY iTotals.
```

Figure 6–5 shows the result.

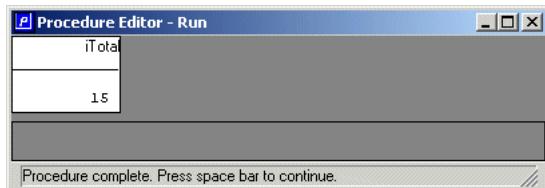


Figure 6–5: Result of looping with a DO block

The starting and ending values can be expressions and not just constants. You can use some value other than one to increment the starting value each time through the loop by using the BY phrase at the end. If the start and end values are variables or other expressions, Progress evaluates them just once, at the beginning of the first iteration of the block. If the values change inside the block, that doesn't change how many times the block iterates. For example, the following variation uses the variable that holds the total as the starting expression, after giving it an initial value of one using the INIT (or INITIAL) phrase at the end of the definition:

```
DEFINE VARIABLE iCount AS INTEGER      NO-UNDO.
DEFINE VARIABLE iTotals AS INTEGER     NO-UNDO INIT 1.

DO iCount = iTotals TO 5:
    iTotals = iTotals + iCount.
END.
DISPLAY iTotals.
```

When you run this procedure, the changes to the variable iTotals inside the loop don't affect how the loop executes. The final total is one greater than it was before, as shown in [Figure 6–6](#), only because the initial value of the variable is one instead of the default of zero.



Figure 6–6: Example of looping with a DO block with initial value set to 1

If you want to loop through a group of statements for as long as some logical condition is true, you can use this form of the DO block:

```
DO WHILE expression:
```

For the expression, you can use any combination of constants, operators, field names, and variable names that yield a logical (true/false) value. For example:

```
DEFINE VARIABLE iTotal AS INTEGER      NO-UNDO INIT 1.  
  
DO WHILE iTotal < 50:  
    iTotal = iTotal * 2.  
END.  
DISPLAY iTotal.
```

By its very nature, the DO WHILE statement must evaluate its expression each time through the loop. Otherwise, it would not be able to determine when the condition is no longer true. In this case the variable `iTotal`, which starts out at one and is doubled until the condition `iTotal` is less than 50, is no longer true. So what do you expect the final total to be? 32? Not for this code, as shown in [Figure 6–7](#).



Figure 6–7: Example DO WHILE loop result

The reason for this is that Progress evaluates the expression at the beginning of each iteration. As long as it's true at that time, the iteration proceeds to the end of the block. At the beginning of the final iteration, `iTotal` equals 32, so the condition is still true. During that iteration it is doubled one last time to 64. At the beginning of the next iteration the condition $64 < 50$ is no longer true, and the block terminates.

When you write a DO WHILE block, make sure that there is always a condition that terminates the block. If the condition is true forever, Progress goes into an infinite loop. If this should happen, press **CTRL-BREAK** on the keyboard to interrupt Progress so that you can go back and correct your mistake.

Using a DO block to scope records and frames

You'll learn more about record scoping in [Chapter 7, “Record Buffers and Record Scope.”](#) It's helpful to cover all the syntax forms that can affect it before discussing the meaning of record scope in different situations. For now, you can scope or associate records in one or more tables with a DO block by using the FOR phrase:

```
DO FOR record [, record] . . .
```

Each *record* names a record you want to work with in the block and scopes it to the block. References within that block to fields the record contains are automatically associated with that record.

You have already seen an example of frame scoping in the code in the `enable_UI` procedure of `h-CustOrderWin2.w` and in the button triggers you created based on that code:

```
DISPLAY Customer.CustNum Customer.Name Customer.Address Customer.City  
Customer.State  
WITH FRAME CustQuery IN WINDOW CustWin.
```

You can use the phrase WITH FRAME *frame-name* and, optionally, IN WINDOW *window-name* to identify which frame you're talking about when you display fields or take other actions on objects in frames. If you wish, you can scope all the statements in a DO block with a frame by appending the WITH FRAME phrase to the DO statement itself. For example, here's the `BtnNext` trigger block again with the frame qualifier moved to the DO statement:

```
DO:  
  GET NEXT CustQuery.  
  IF AVAILABLE Customer THEN  
    DO WITH FRAME CustQuery:  
      DISPLAY Customer.CustNum Customer.Name Customer.Address Customer.City  
Customer.State.  
      {&OPEN-BROWSERS-IN-QUERY-CustQuery}  
    END.  
  END.
```

Whether you name the frame in individual statements or in the block header, this makes sure that Progress understands that you want the fields displayed in the frame where you defined them. If you don't do this, then depending on the context, Progress might display the fields in a different frame.

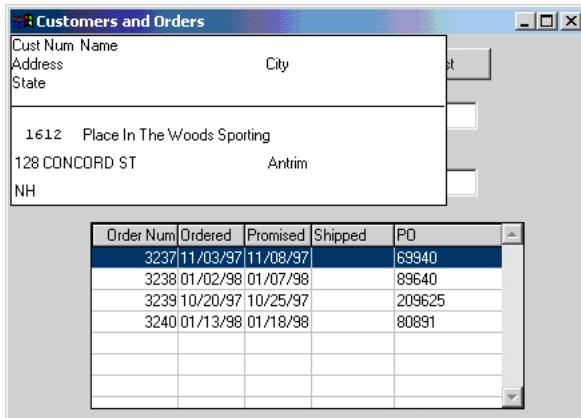


To see the result of not explicitly defining a frame scope:

1. Edit the WITH FRAME phrase out of the BtnNext trigger altogether, so it looks like this:

```
DO:
  GET NEXT CustQuery.
  IF AVAILABLE Customer THEN
    DO /* WITH FRAME CustQuery */:
      DISPLAY Customer.CustNum Customer.Name Customer.Address Customer.City
            Customer.State.
      {&OPEN-BROWSERS-IN-QUERY-CustQuery}
    END.
  END.
```

2. Run the window and choose the Next button:



What happened here? Since the DISPLAY statement wasn't qualified with any indication of where to display the fields, Progress created a brand new frame, laid out the fields in it, and displayed it on top of the other frame in your window. To keep this from happening, make sure that all statements that deal with frames are always clear about where actions should take place. The AppBuilder and the other tools take care of this for you most of the time, but when you add code of your own to procedures, you might need to qualify it with the frame name. Otherwise, Progress displays objects in the frame that is scoped to the nearest enclosing block that defines a frame. If there is no explicit frame definition, then you get the result you just saw here:

Progress displays the data in an unnamed default frame. Except in the simplest test procedures, like the procedures this book uses to demonstrate the looping syntax, you never want to use the default frame in your applications.

There are other phrases you can use to qualify a DO block, but they mostly deal with transactions and error handling, which you'll learn about in [Chapter 17, “Managing Transactions.”](#)

FOR blocks

You're already familiar with starting a block definition with the FOR keyword. You've seen the common FOR EACH *table-name* form, but there are a number of variations on the FOR statement. In contrast to the DO block, every FOR block provides all of the following services for you automatically:

- Loops automatically through all the records that satisfy the record set definition in the block.
- Reads the next record from the result set for you as it iterates.
- Scopes those records to the block.
- Scopes a frame to the block, and you can use the WITH FRAME phrase to specify that frame.
- Provides database update services within a transaction.

The FOR statement defines the set of records you want the block to iterate through. Typically you use the EACH keyword to specify this set:

```
FOR EACH Customer WHERE State = "NH":  
    DISPLAY CustNum Name.  
END.
```

When the block begins, Progress evaluates the expression and retrieves the first record that satisfies it. This record is scoped to the entire block. Each time the block iterates, Progress retrieves the next matching record and makes it available to the rest of the block. When the set of matching records is exhausted, Progress automatically terminates the block. You don't have to add any checks or special syntax to exit the block at this point.

Sorting records by using the BY phrase

As you've seen, you can sort the records by using the BY phrase. The default is ascending order, but you cannot use the keyword ASCENDING to indicate this. You'll get a syntax error, so just leave it out to get ascending order.

To sort in descending order, add the keyword DESCENDING to the BY phrase:

```
BY field [ DESCENDING ] . . .
```

To sort on multiple fields, you can repeat the BY phrase.

Joining tables using multiple FOR phrases

You can use multiple record phrases to join records from more than one table:

```
FOR EACH Customer WHERE State = "NH",
  EACH Order OF Customer WHERE ShipDate NE ? :
    DISPLAY Customer.Custnum Name OrderNum ShipDate.
END.
```

Figure 6–8 shows the result.



The screenshot shows a Windows application window titled "Procedure Editor - Run". Inside, there is a table with two columns: "Cust Num" and "Name" on the left, and "Order Num" and "Shipped" on the right. The data consists of 16 rows, each representing a customer and their corresponding order number and ship date. The table has a header row and 15 data rows. At the bottom of the window, a message says "Procedure complete. Press space bar to continue.".

| Cust Num | Name | Order Num | Shipped |
|----------|------------------------------|-----------|------------|
| 66 | First Down Football | 3 | 09/28/1997 |
| 66 | First Down Football | 66 | 10/22/1997 |
| 66 | First Down Football | 80 | 10/31/1997 |
| 1258 | Joe Jone's Ski Shop & Sports | 1979 | 10/05/1997 |
| 1594 | Franconia Sport Shop | 3170 | 02/23/1998 |
| 1594 | Franconia Sport Shop | 3174 | 01/28/1998 |
| 1595 | Tonto Sports | 3177 | 11/30/1997 |
| 1599 | True Sport Inc | 3190 | 10/27/1997 |
| 1599 | True Sport Inc | 3191 | 10/16/1997 |
| 1601 | Covered Bridge Sport Shop | 3196 | 01/18/1998 |
| 1604 | Ducret's Sporting Goods | 3207 | 09/29/1997 |
| 1606 | Olympia Sports Ctr | 3214 | 12/19/1997 |
| 1607 | Ben's Ski & Sports Ctr | 3220 | 01/25/1998 |
| 1608 | Sport-about Charlie | 3226 | 09/12/1997 |
| 1609 | Twin River Sports Shop | 3229 | 01/22/1998 |
| 1611 | Marty's Sport It | 3234 | 10/18/1997 |

Figure 6–8: Joining records from more than one table

There are several things to note about this example:

- Progress retrieves and joins the tables in the order you specify them in, in effect following your instructions from left to right. In this example, it starts through the set of all **Customers** where the **State** field = “NH”. For the first record, it defines a set of **Orders** with the same **CustNum** value (represented by the OF syntax in this case). For each matching pair, it establishes that **Customer** record and its **Order** record and makes them available to all the rest of the statements in the block. Because there are typically multiple **Orders** for a **Customer**, the result is a one-to-many join, where the same **Customer** remains current for multiple iterations through the block, one for each of its **Orders**, as the output in Figure 6–8 shows.
- If you change the sequence of the tables in the statement, Progress might retrieve a very different set of records. For example, using the syntax FOR EACH Order WHERE ShipDate NE ?, EACH Customer OF Order, you would get a list of every **Order** in the **Order** table with a **ShipDate**, plus its (one) matching **Customer** record. Because there’s just one **Customer** for each **Order**, this would result in a one-to-one join with no repeated records.

- The default join you get is called an *inner join*. In matching up **Customers** to their **Orders**, Progress skips any **Customer** that has no **Orders** with a **ShipDate** because there is no matching pair of records. The alternative to this type of join, called an *outer join*, doesn't skip those **Customers** with no **Orders** but instead supplies unknown values from a dummy **Order** when no **Order** satisfies the criteria. Progress has an OUTER-JOIN keyword, but you can use it only when you define queries of the kind you've seen in DEFINE QUERY statements, not in a FOR EACH block. To get the same effect using FOR EACH blocks, you can nest multiple blocks, one to retrieve and display the **Customer** and another to retrieve and display its **Orders**. You did this back in the sample procedure in Chapter 2, "Using Basic 4GL Constructs." Generally this is more effective than constructing a query that might involve a one-to-many relationship anyway, because it avoids having duplicated data from the first table in the join.
- You can add a WHERE clause and/or a BY clause to each record phrase if you wish. You should always move each WHERE clause up as close to the front of the statement as possible to minimize the number of records retrieved. For example, the statement FOR EACH Customer, EACH Order OF Customer WHERE State = "NH" AND ShipDate NE ? would yield the same result but retrieve many more records in the process. It would go through the set of all **Customers**, retrieve each **Order** for each **Customer**, and then determine whether the **State** was "NH" and the **ShipDate** was not unknown. This code is very inefficient. The way the 4GL handles data retrieval is different from SQL, where the table selection is done at the beginning of a SELECT statement and the WHERE clause is after the list of tables. The SQL form depends on the presence of an optimizer that turns the statement into the most efficient retrieval possible. The advantage of the Progress form is that you have greater control over exactly how the data is retrieved. But with this control comes the responsibility to construct your FOR statements intelligently.
- Because two records, **Customer** and **Order**, are scoped to the FOR block, you might need to qualify field names that appear in both of them. If you just write DISPLAY CustNum you get a syntax error when you try to run the procedure, as shown in Figure 6–9.

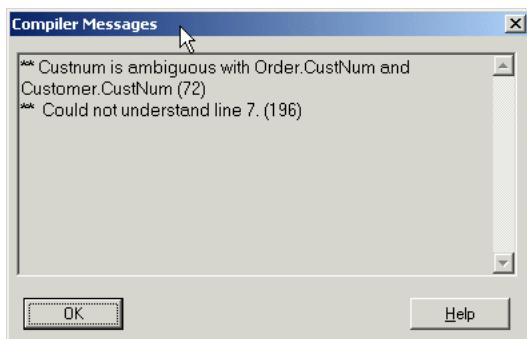


Figure 6–9: Syntax error message

Alternatives to the EACH keyword

Sometimes you just want a single record from a table. In that case, you can use the FIRST or LAST keyword in place of EACH, or possibly use no qualifier at all. For example, if you want to retrieve **Orders** and their **Customers** instead of the other way around, you can leave out the keyword EACH in the **Customer** phrase, because each **Order** has only one **Customer**:

```
FOR EACH Order WHERE ShipDate NE ?, Customer OF Order:  
    DISPLAY OrderNum ShipDate Name.  
END.
```

When you use this form, make sure that there is never more than one record satisfying the join. Otherwise, you get a run-time error telling you that there is no unique match.

If you'd like to see just the first **Order** for each **Customer** in New Hampshire, you can use the FIRST qualifier to accomplish that:

```
FOR EACH Customer WHERE State = "NH", FIRST Order OF Customer:  
    DISPLAY Customer.CustNum NAME OrderNum OrderDate.  
END.
```

Be careful, though. This form might not always yield the result you expect, because you have to consider just what *is* the first **Order** of a **Customer**? Progress uses an index of the **Order** table to traverse the rows.

Using indexes to relate and sort data

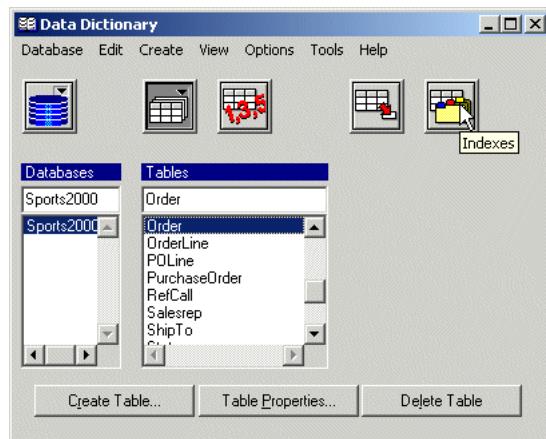
A *database index* allows the database manager to retrieve records quickly by looking up only the values of one or more key fields stored in separate database blocks from the records themselves, which then point to the location where the records are stored.

And what are the indexes of the **Order** table?

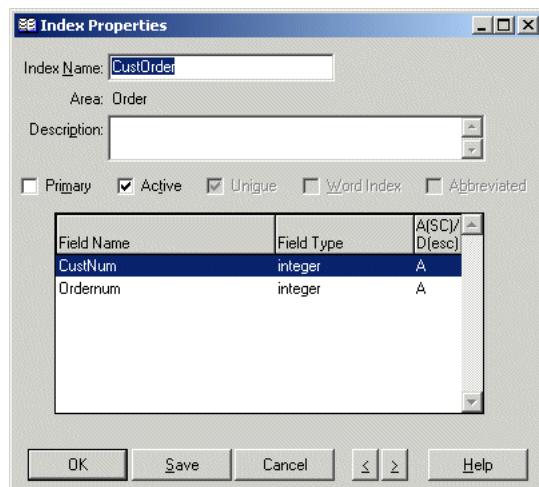


To get the answer to this question, take another look inside the Data Dictionary:

1. From the AppBuilder menu, select **Tools→Data Dictionary**.
2. Select the **Order** table from the list of tables, then choose the **Indexes** button:



3. Choose the **Index Properties** button. The **Index Properties** dialog box appears and shows the properties of the first index, **CustOrder**:



This is the index Progress uses to retrieve the **Orders**, because its first component is the **CustNum** field, and that is the field it has to match against the **CustNum** from the **Customer** table. Since the other component in the index is the **OrderNum** field, this index sorts records by **OrderNum** within **CustNum** so your request for the FIRST **Order** returns the record with the lowest **Order** number.

4. Exit the **Data Dictionary** before you continue. Otherwise, Progress won't let you run any procedures because it has a database transaction open and ready to save any changes you might make in the Data Dictionary.

Figure 6–10 shows the beginning of the display from the block FOR EACH Customer WHERE State = "NH", FIRST Order OF Customer.

| Cust Num | Name | Order Num | Ordered |
|----------|------------------------------|-----------|----------|
| 66 | First Down Football | 3 | 09/23/97 |
| 1257 | Center Harbor Sport Shop | 1972 | 01/21/98 |
| 1258 | Joe Jone's Ski Shop & Sports | 1977 | 11/04/97 |

Figure 6–10: Lowest Order number for each Customer

As expected, you see the **Order** with the lowest **Order** number for each **Customer**. If what you want is the earliest **Order** date, this output might not give you the information you are looking for.

Adding a **BY** phrase to the statement doesn't help because Progress retrieves the records before applying the sort. So if you want the **Order** with the earliest **Order** date, it won't work to do this:

```
FOR EACH Customer WHERE State = "NH",
  FIRST Order OF Customer BY OrderDate:
    DISPLAY Customer.CustNum NAME OrderNum OrderDate.
END.
```

This code retrieves the same **Orders** as before, but then sorts the whole result set by the **OrderDate** field, as shown in Figure 6–11.



The screenshot shows a table titled "Procedure Editor - Run" with three columns: "Cust Num", "Name", and "Order Num Ordered". The data is sorted by "Order Date". A circled cell highlights the row for Everett Sports Ctr (Customer 1605) with Order Num 3210 and Order Date 09/08/97.

| Cust Num | Name | Order Num Ordered |
|----------|--------------------------|-------------------|
| 1605 | Everett Sports Ctr | 3210 09/08/97 |
| 66 | First Down Football | 3 09/23/97 |
| 1613 | Mvp Sports | 3241 09/26/97 |
| 1611 | Marty's Sport It | 3234 10/13/97 |
| 1614 | Klaus' Ski & Sport | 3243 10/14/97 |
| 1600 | Waite Sports Specialists | 3193 10/22/97 |

Figure 6–11: Orders sorted by OrderDate

Using the USE-INDEX phrase to force a retrieval order

If you look at all the indexes for the **Order** table in the Data Dictionary, you can see that there is also an index called **OrderDate** that uses the **Order** field. You can select the index to use when the default choice is not the one you want. Progress does this by adding a **USE-INDEX** phrase to the record phrase. This form of the **FOR EACH** statement is guaranteed to return the earliest **OrderDate**, even if it's not the lowest **OrderNum**:

```
FOR EACH Customer WHERE State = "NH",
  FIRST Order OF Customer USE-INDEX OrderDate:
    DISPLAY Customer.CustNum NAME OrderNum OrderDate.
END.
```

The result in Figure 6–12 shows that there is indeed an earlier **Order** for the first of your **Customers** that doesn't have the lowest **OrderNum**.



The screenshot shows a table titled "Procedure Editor - Run" with three columns: "Cust Num", "Name", and "Order Num Ordered". The data is sorted by "Order Date". A circled cell highlights the row for First Down Football (Customer 66) with Order Num 199 and Order Date 09/11/97.

| Cust Num | Name | Order Num Ordered |
|----------|------------------------------|-------------------|
| 66 | First Down Football | 199 09/11/97 |
| 1257 | Center Harbor Sport Shop | 1975 09/09/97 |
| 1258 | Joe Jone's Ski Shop & Sports | 1979 09/30/97 |
| 1594 | Franconia Sport Shop | 3171 10/04/97 |

Figure 6–12: Earliest Customer Order

Using the LEAVE statement to leave a block

Use the USE-INDEX phrase only when necessary. Progress is extremely effective at choosing the right index, or combination of multiple indexes, to optimize your data retrieval. In fact, there's an alternative even in the present example that yields the same result without requiring you to know the names and fields in the **Order** table's indexes. Take a look at this procedure:

```
FOR EACH Customer WHERE State = "NH" WITH FRAME f:  
    DISPLAY Customer.CustNum Name.  
    FOR EACH Order OF Customer BY OrderDate:  
        DISPLAY OrderNum OrderDate WITH FRAME f.  
        LEAVE.  
    END. /* END FOR EACH Order */  
END. /* END FOR EACH Customer */
```

This code uses nested blocks to retrieve the **Customers** and **Orders** separately. These nested blocks allow you to sort the **Orders** for a single **Customer** BY **OrderDate**. You have to define the set of all the **Customer's Orders** using the FOR EACH phrase so that the BY phrase has the effect of sorting them by **OrderDate**. But you really only want to see the first one. To do this, you use another one-word 4GL statement: LEAVE. The LEAVE statement does exactly what you would expect it to: It leaves the block (specifically the innermost iterating block to the LEAVE statement) after displaying fields from the first of the **Customer's Orders**. It does not execute any more statements that might be in the block nor does it loop through any more records that are in its result set. Instead, it moves back to the outer block to retrieve the next **Customer**.

Because the LEAVE statement looks for an iterating block to leave, it always leaves a FOR block. It leaves a DO block only if the DO statement has a qualifier, such as WHILE, that causes it to iterate. If there is no iterating block, Progress leaves the entire procedure.

Using block headers to identify blocks

If it isn't clear what block the LEAVE statement applies to, or if you want it to apply to some other enclosing block, you can give a block a name followed by a colon and then specifically leave that block. This variant of the procedure has the same effect as the first one:

```
FOR EACH Customer WHERE State = "NH" WITH FRAME f:
    DISPLAY Customer.CustNum NAME.
    OrderBlock:
        FOR EACH Order OF Customer BY OrderDate:
            DISPLAY OrderNum OrderDate WITH FRAME f.
            LEAVE OrderBlock.
        END. /* END FOR EACH Order */
    END. /* END FOR EACH Customer */
```

Just to see the effect of specifying a different block, you can try this variant:

```
CustBlock:
FOR EACH Customer WHERE State = "NH" WITH FRAME f:
    DISPLAY Customer.CustNum NAME.
    OrderBlock:
        FOR EACH Order OF Customer BY OrderDate:
            DISPLAY OrderNum OrderDate WITH FRAME f.
            LEAVE CustBlock.
        END. /* END FOR EACH Order */
    END. /* END FOR EACH Customer */
```

If you run this code, Progress leaves the outer FOR EACH Customer block after retrieving the first Order for the first Customer because of the change to the LEAVE statement, as shown in [Figure 6–13](#).

| Cust Num | Name | Order Num | Ordered |
|----------|---------------------|-----------|----------|
| 66 | First Down Football | 199 | 09/11/97 |

Procedure complete. Press space bar to continue.

Figure 6–13: Specifying a different block

Using NEXT, STOP, and QUIT to change block behavior

There's another one-word statement that works much like LEAVE and that is NEXT. As you might expect, this statement skips any remaining statements in the block and proceeds to the next iteration of the block. You can qualify it with a block name the same way you do with LEAVE.

There are two more such statements that have increasingly more drastic consequences: STOP and QUIT.

STOP terminates the current procedure, backs out any active transactions, and returns to the Progress session's startup procedure or to the Editor. You can intercept a STOP action by including the ON STOP phrase on a block header, which defines an action to take other than the default when the STOP condition occurs.

QUIT exits from Progress altogether in a run-time environment and returns to the operating system. If you're running in a development environment, it has a similar effect to STOP and returns to the Editor or to the Desktop. There is also an ON QUIT phrase to intercept the QUIT condition in a block header and define an action to take other than quitting the session.

Qualifying a FOR statement with a frame reference

This most recent example also has an explicit frame reference in it:

```
FOR EACH Customer WHERE State = "NH" WITH FRAME f:
    DISPLAY Customer.CustNum Name.
    FOR EACH Order OF Customer BY OrderDate:
        DISPLAY OrderNum OrderDate WITH FRAME f.
        LEAVE.
    END. /* END FOR EACH Order */
END. /* END FOR EACH Customer */
```

Why is this necessary? A FOR EACH block scopes a frame to the block. By default, this is an unnamed frame. Without the specific frame reference, you get two nested frames, one for the **Customer** and one for its **Orders**. You saw this already in the sample procedure in Chapter 2, “Using Basic 4GL Constructs.”

In this case, that isn't what you want. Because there's only one **Order** of interest for each **Customer**, you want to display all the fields together in the **Customer** frame. To get this effect, you have to override the default behavior and tell Progress to use the frame from the **Customer** block to display the **Order** fields. That is what these two references to **WITH FRAME f** do for you. Progress just keeps making room for new fields in the frame as it encounters them (unless you tell it exactly where to put each field, which is the norm in your GUI applications that use the AppBuilder to lay things out).

REPEAT blocks

There's a third kind of iterating block that is in between DO and FOR in its effects, the REPEAT block. It supports just about all the same syntax as a FOR block. You can add a FOR clause for one or more tables to it. You can use a WHILE phrase or the *expression TO expression* phrase. You can scope a frame to it.

A block that begins with the REPEAT keyword shares these default characteristics with a FOR block:

- It is an iterating block.
- It scopes a frame to the block.
- It scopes records referenced in the REPEAT statement to the block.
- It provides transaction processing if you update records within the block.

By contrast, it shares this important property with a DO block: It does not automatically read records as it iterates.

So what is a REPEAT block for? It is useful in cases where you need to process a set of records within a block but you need to navigate through the records yourself, rather than simply proceeding to the next record automatically on each iteration. The sample procedure starting in the “[Index cursors](#)” section on page 6–30 shows you an example of where to use the REPEAT block.

One of the common ways to use a REPEAT block in older character applications is to repeat a data entry action until the user hits the ESCAPE key to end. Here's a very simple but powerful example:

```
REPEAT:  
  INSERT customer EXCEPT comments WITH 2 COLUMNS.  
END.
```

You haven't seen the `INSERT` statement before and you won't see much of it again, even though it's one of the most powerful statements in the language. [Figure 6–14](#) shows what you get from that one simple statement:

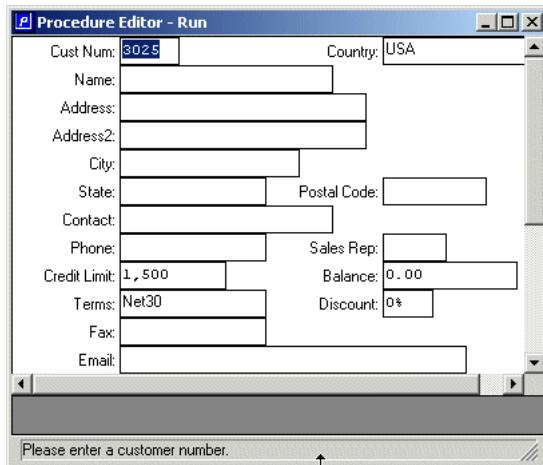


Figure 6–14: Results of using the `INSERT` statement

It's a complete data entry screen, complete with initial values for fields that have one. The field help displays at the bottom of the window. Inside a `REPEAT` loop, this lets you create one new **Customer** record after another until you're done and you press **ESCAPE**.

Why won't you see the `INSERT` statement again? Because `INSERT` is one of those statements that mixes up all aspects of Progress, from the user interface to saving the data off into the database. And in a modern GUI application, you need to separate out all those things into separate parts of your application. You'll look at database updates in [Chapter 16, “Updating Your Database and Writing Triggers,”](#) so there's no need to go into this example any further beyond understanding what the `REPEAT` block does around the statement.

Using the `PRESELECT` keyword to get data in advance

One typical use of the `REPEAT` block that is still valuable is when you use it with a construct called a `PRESELECT`. To understand this usage, you need to think a little about what happens when an iterating block like a `FOR EACH` block begins. Progress evaluates the record retrieval the `FOR EACH` statement defines. It then goes out to the database and retrieves the first record of the set of related records that satisfies the statement and makes the record available to the block. When the block iterates, Progress goes and gets the next record. As long as it's possible to identify what the first and the next records are by using one or more indexes, Progress doesn't bother reading all the records in advance. It just goes out and gets them when the block needs them.

If you specify a BY clause that requires a search of the database that an index can't satisfy, Progress has no choice but to retrieve all the records in advance and build up a list in sort order. But this is not ordinarily the case. It's much more efficient simply to get the records when you need them.

Sometimes, though, you need to force Progress to get all the records that satisfy the FOR EACH statement in advance, even when the sort order itself doesn't require it. For example, if it's possible that you will modify an indexed field in some of the records in such a way that they would appear again later in the retrieval process, you need to make sure that the set of records you're working with is predetermined. The PRESELECT keyword gives you this. It tells Progress to build up a list of pointers to all the records that satisfy the selection criteria before it starts iterating through the block. This assures you that each record is accessed only once.

In summary, a REPEAT block does everything a FOR block does, but it does not automatically advance to the next record as it iterates. You should use a REPEAT block in cases where you want to control the record navigation yourself, typically using the FIND statement described in the next section. It provides you with record, frame, and transaction scoping.

Because it provides all these services, a REPEAT block is relatively expensive compared to a DO block. Use the simpler DO block instead of a REPEAT block, unless you need the record-oriented services provided by the REPEAT block.

Data access without looping – the FIND statement

In addition to all of these ways to retrieve and iterate through a set of related records, Progress has a very powerful way to retrieve single records without needing a query or result set definition of any kind. This is the FIND statement.

The FIND statement uses this basic syntax:

```
FIND [ FIRST | NEXT | PREV | LAST ] record [ WHERE . . . ]
      [ USE-INDEX index-name ]
```

Using the FIND statement to fetch a single record from the database is pretty straightforward. This statement reads the first **Customer** and makes it available to the procedure:

```
FIND FIRST Customer.
```

This statement fetches the first **Customer** in New Hampshire:

```
FIND FIRST Customer WHERE State = "NH".
```

It gets more interesting when you FIND the NEXT record or the PREV record. This should immediately lead you to the question: NEXT or PREV relative to what? Even the FIND FIRST statement has to pick a sequence of **Customers** in which one of them is first. Although it might seem intuitively obvious that **Customer 1** is the first **Customer**, given that the **Customers** have an integer key identifier, this is the record you get back only because the **CustNum** index is the primary index for the table (you could verify this by looking in the Data Dictionary). Without any other instructions to go on, and with no WHERE clause to make it use another index, the FIND statement uses the primary index. You can use the USE-INDEX syntax to force Progress to use a particular index.

If you include a WHERE clause, Progress chooses one or more indexes to optimize locating the record. This might have very counter-intuitive results. For example, here's a simple procedure with a FIND statement:

```
FIND FIRST Customer.  
DISPLAY CustNum Name Country.
```

Figure 6–15 shows the expected result.

| Procedure Editor - Run | |
|--|------------|
| Cust Num | Name |
| 1 | Lift Tours |
| Procedure complete. Press space bar to continue. | |

Figure 6–15: Result of a simple FIND procedure

You can see that **Customer 1** is in the USA. Here's a variation of the procedure:

```
FIND FIRST Customer WHERE Country = "USA".  
DISPLAY CustNum Name Country.
```

Figure 6–16 shows the not-so-expected result.

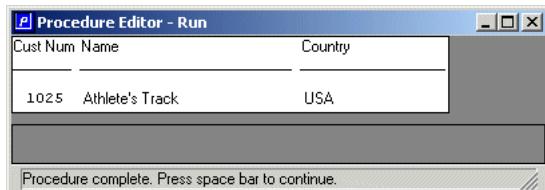


Figure 6–16: Result of variation on the simple FIND procedure

What happened here? If **Customer 1** is the first **Customer**, and **Customer 1** is in the USA, then why isn't it the first **Customer** in the USA? Progress uses an index in the **Country** field to locate the first **Customer** in the USA, because that's the most efficient way to find it. That index, called the **CountryPost** index, has the **PostalCode** as its secondary field. If you rerun this procedure again and ask to see the **PostalCode** field instead of the **Name** field, you'll see why it came up first using that index, as shown in Figure 6–17.



Figure 6–17: Result of the simple FIND procedure using PostalCode

The **PostalCode** is blank for this **Customer**, so it sorts first. Even if there is no other field in the index at all, that would only mean that the order of **Customers** within that index for a given country value would be undetermined. Only if the **CustNum** field is the next index component could you be sure that **Customer 1** would come back as the first **Customer** in the USA.

These examples show that you must be careful when using any of the positional keywords (FIRST, NEXT, PREV, and LAST) in a FIND statement to make sure you know how the table is navigated.

Index cursors

To understand better how Progress navigates through a set of data, you need to understand the concept of index cursors. When you retrieve a record from the database using any of the statements you've seen in this chapter, Progress keeps track of the current record position using an *index cursor*—a pointer to the record, using the location in the database indexes of the key value used for retrieval.

When you execute the statement `FIND FIRST Customer`, for example, Progress sets a pointer to the record for **Customer 1** within the **CustNum** index. If you execute the statement `FIND FIRST Customer WHERE Country = "USA"`, Progress points to **Customer 1025** through the **CountryPost** index.

When you execute another FIND statement on the same table using one of the directional keywords, Progress can go off in any direction from the current index cursor location, depending on the nature of the statement. By default, it reverts to the primary index. Here's an example that extends the previous one slightly:

```

FIND FIRST Customer WHERE Country = "USA".
DISPLAY CustNum NAME Country.
REPEAT:
  FIND NEXT Customer NO-ERROR.
  IF AVAILABLE Customer THEN
    DISPLAY CustNum Name FORMAT "x(20)" Country PostalCode.
  ELSE LEAVE.
END.

```

Using the FIND statement in a REPEAT block

Notice the use of the REPEAT block to cycle through the remaining **Customers**. Within that block, you must write a FIND statement to get the next **Customer** because the REPEAT block itself, unlike the FOR EACH block, does not do the navigation for you. Also, the REPEAT block does not automatically terminate when the end of the **Customers** is reached, so you need to program the block with these three actions:

1. You must do the FIND with the NO-ERROR qualifier at the end of the statement. This suppresses the error message that you would ordinarily get when there is no next **Customer**.
2. You must use the AVAILABLE keyword to check for the presence of a **Customer** and display fields only if it evaluates to true.
3. You must write an ELSE statement to match the IF-THEN statement, to leave the block when there is no **Customer** available. Otherwise, your block goes into an infinite loop when it reaches the end of the **Customer** records. And notice that this truly is a separate statement. The IF-THEN statement ends with a period and the ELSE keyword begins a statement of its own.

All of these are actions that the FOR EACH block does for you as it reads through the set of **Customers**. In the REPEAT block, though, where you're doing your own navigation, you need to do these things yourself.

Remember also that the REPEAT block scopes the statements inside the block to its own frame, unless you tell it otherwise. Therefore, you get one frame for the FIRST **Customer** and a new frame for all the **Customer** records retrieved within the REPEAT block.

The keyword AVAILABLE is a Progress 4GL built-in function, so its one argument properly belongs in parentheses, as in IF AVAILABLE (Customer). However, to promote the readability of the 4GL statement, the syntax also accepts the form as if it were a phrase without the parentheses, as in IF AVAILABLE Customer. This alternative is not generally available with other built-in functions.

Finally, the FORMAT "X(20)" phrase reduces the display size of the **Name** field from its default (defined in the Data Dictionary) of 30 characters, to make room for the **PostalCode** field.

Switching indexes between FIND statements

So what **Customer** do you expect to see as the next **Customer** after retrieving the first **Customer** using the **CountryPost** index (because of the WHERE clause)? If you remember that the default is always to revert to the primary index, then the result shown in [Figure 6–18](#) should be clear.



Figure 6–18: Result of using the primary index

Looking at the sequence of records displayed in the frame for the REPEAT block, it's clear that Progress is using the primary index (the **CustNum** index) to navigate through the records. This is unaffected by the fact that the initial FIND was done using the **CountryPost** index, because of its WHERE clause.

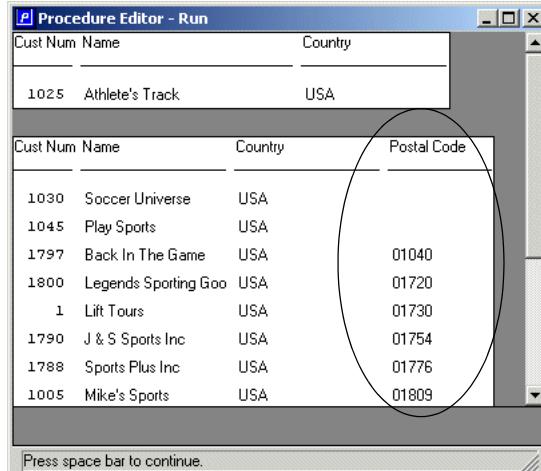
What if you want to continue retrieving only **Customers** in the USA? In this case, you need to repeat the WHERE clause in the FIND statement in the REPEAT block:

```

FIND FIRST Customer WHERE Country = "USA".
DISPLAY CustNum NAME Country.
REPEAT:
    FIND NEXT Customer WHERE Country = "USA" NO-ERROR.
    IF AVAILABLE Customer THEN
        DISPLAY CustNum NAME FORMAT "x(20)" Country PostalCode.
    ELSE LEAVE.
END.

```

Each FIND statement is independent of any other FIND statement, even if it refers to the same table, so the WHERE clause does not carry over automatically. If you do this, then Progress continues to use the **CountryPost** index for the retrieval, as the output in Figure 6–19 shows.



The screenshot shows a window titled "Procedure Editor - Run". Inside, there is a table with two columns: "Cust Num" and "Name". Below this, another table has three columns: "Cust Num", "Name", and "Country". A third table below it has four columns: "Cust Num", "Name", "Country", and "Postal Code". The "Postal Code" column is circled with a black oval. The data in the tables is as follows:

| Cust Num | Name | Country |
|----------|-----------------|---------|
| 1025 | Athlete's Track | USA |

| Cust Num | Name | Country |
|----------|----------------------|---------|
| 1030 | Soccer Universe | USA |
| 1045 | Play Sports | USA |
| 1797 | Back In The Game | USA |
| 1800 | Legends Sporting Goo | USA |
| 1 | Lift Tours | USA |
| 1790 | J & S Sports Inc | USA |
| 1788 | Sports Plus Inc | USA |
| 1005 | Mike's Sports | USA |

| Cust Num | Name | Country | Postal Code |
|----------|----------------------|---------|-------------|
| 1030 | Soccer Universe | USA | 01040 |
| 1045 | Play Sports | USA | 01720 |
| 1797 | Back In The Game | USA | 01730 |
| 1800 | Legends Sporting Goo | USA | 01754 |
| 1 | Lift Tours | USA | 01776 |
| 1790 | J & S Sports Inc | USA | 01809 |
| 1788 | Sports Plus Inc | USA | |
| 1005 | Mike's Sports | USA | |

Figure 6–19: Result of using the CountryPost index for record retrieval

Because the **PostalCode** is the second field in the index used, the remaining records come out in **PostalCode** order.

Using a USE-INDEX phrase to force index selection

You can also force a retrieval sequence with the USE-INDEX phrase. For instance, if you want to find the next set of **Customers** based on the **Customer** name, you can use the **Name** index, which contains just that one field:

```
FIND FIRST Customer WHERE Country = "USA".
DISPLAY CustNum NAME Country.
REPEAT:
  FIND NEXT Customer WHERE Country = "USA" USE-INDEX NAME NO-ERROR.
  IF AVAILABLE Customer THEN
    DISPLAY CustNum NAME FORMAT "x(20)" Country PostalCode.
  ELSE LEAVE.
END.
```

The output shown in [Figure 6–20](#) confirms that Progress is walking through the records in **Name** order, starting with the name of the first **Customer** in the USA.



A screenshot of a Progress 4GL application window titled 'P'. The window displays a list of customers from the USA. The columns are 'Cust Num', 'Name', 'Country', and 'Postal Code'. The data is as follows:

| Cust Num | Name | Country | Postal Code |
|----------|----------------------|---------|-------------|
| 1025 | Athlete's Track | USA | |
| 1849 | Athletic Annex Runni | USA | 46260 |
| 1228 | Athletic Attic | USA | 29406 |
| 1391 | Athletic Attic | USA | 37421 |
| 1764 | Athletic Attic | USA | 04210 |
| 1956 | Athletic Attic | USA | 32034 |
| 1488 | Athletic Department | USA | 97005 |
| 1411 | Athletic Edge | USA | 57106 |
| 1313 | Athletic Exchange | USA | 30062 |
| 1950 | Athletic Footwear Du | USA | 33323 |

At the bottom of the window, there is a message: 'Press space bar to continue.'

Figure 6–20: Result of forcing index selection

This technique can be very valuable in expressing your business logic in your procedures. You might need to identify a record based on one characteristic and then retrieve all other records (or perhaps just one additional record) based on some other characteristic of the record you first retrieved. This is one of the most powerful ways in which Progress lets you define your business logic without the overhead and cumbersome syntax required to deal with all data access in terms of sets.

Doing a unique FIND to retrieve a single record

Very often you just need to retrieve a single record using selection criteria that identify it uniquely. In this case, you can use a FIND statement with no directional qualifier. For example, you can identify a **Customer** by its **Customer** number. This is a unique value, so you can use the following FIND statement:

```
FIND Customer WHERE CustNum = 1025.
DISPLAY CustNum NAME Country.
```

Figure 6–21 shows the result.

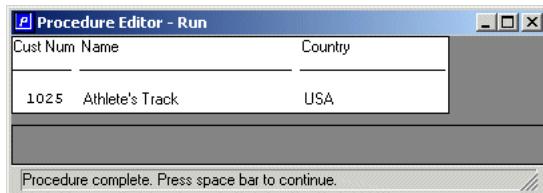


Figure 6–21: Result of unique FIND

You need to be sure when you do this that only one record satisfies the selection criteria. Otherwise, you get an error at run time.

There's also a shorthand for this FIND statement:

```
FIND Customer 1025.
```

You can use this shorthand form if the primary index is a unique index (with no duplication of values), the primary index contains just a single field, and you want to retrieve a record using just that field. You can only use this form when all these conditions are true, so it's not likely to be one you use frequently. Also, this shorthand form makes it harder to determine your criteria. It can break due to changes to the data definitions (for example, if someone went in and added another field to the **CustNum** index), so it's better to be more specific and use a **WHERE** clause to identify the record.

Using the CAN-FIND function

Often you need to verify the existence of a record without retrieving it for display or update. For example, your logic might need to identify each **Customer** that has at least one **Order**, but you might not care about retrieving any actual **Orders**. To do this, you can use an alternative to the FIND statement that is more efficient because it only checks index entries wherever possible to determine whether a record exists, without going to the extra work of retrieving the record itself. This alternative is the CAN-FIND built-in function. CAN-FIND takes a single parameter, which can be any record selection phrase. The CAN-FIND function returns true or false depending on whether the record selection phrase identifies exactly one record in the database.

For example, imagine that you want to identify all **Customers** that placed **Orders** as early as 1997. You don't need to retrieve or display the **Orders** themselves, you just need to know which **Customers** satisfy this selection criterion. Here's a simple procedure that does this:

```
FOR EACH Customer WHERE Country = "USA":  
    IF CAN-FIND (FIRST Order OF Customer WHERE OrderDate < 1/1/98)  
        THEN DISPLAY CustNum Name.  
    ELSE DISPLAY CustNum "No 1997 Orders" @ Name.  
END.
```

This procedure uses a little display trick you haven't seen before. If the **Customer** has any **Orders** for 1997, then the procedure displays the **Customer** name. Otherwise, it displays the text phrase **No 1997 Orders**. If you include that literal value in the DISPLAY statement, it displays it in its own column as if it were a field or a variable. To display it in place of the **Name** field, use the at-sign symbol (@). [Figure 6–22](#) shows the result.

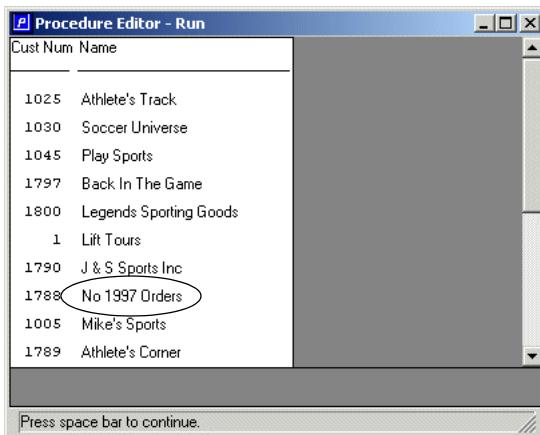


Figure 6–22: Result of CAN-FIND function procedure

The CAN-FIND function takes the argument FIRST Order OF Customer WHERE OrderData < 1/1/98. Why is the FIRST keyword necessary? The CAN-FIND function returns true only if exactly one record satisfies the selection criteria. If there's more than one match, then it returns false—without error—just as it would if there was no match at all. For example, if you remove the FIRST keyword from the example procedure and change the literal text to be **No unique 1997 Order**, and rerun it, then you see that most **Customers** have more than one **Order** placed in 1997:

```
FOR EACH Customer WHERE Country = "USA":
    IF CAN-FIND (Order OF Customer WHERE OrderDate < 1/1/98)
        THEN DISPLAY CustNum Name.
    ELSE DISPLAY CustNum "No unique 1997 Order" @ Name.
END.
```

After you page through the results, you see just a few records that don't satisfy the criteria, as shown in Figure 6–23.

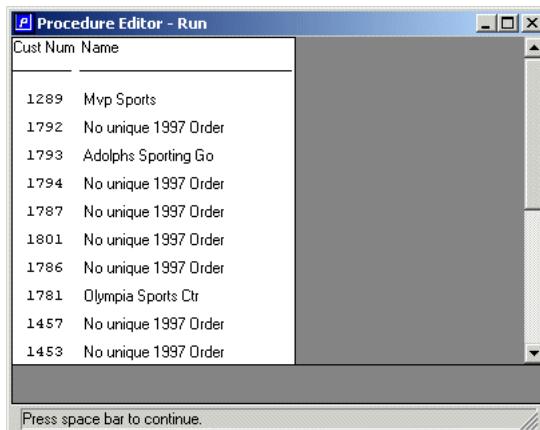


Figure 6–23: Result of CAN-FIND function procedure without FIRST keyword

Because you don't get an error if there's more than one match, it's especially important to remember to define your selection criteria so that they identify exactly one record when you want the function to return true.

The CAN-FIND function is more efficient than the FIND statement because it does not actually retrieve the database record. If the selection criteria can be satisfied just by looking at values in an index, then it doesn't look at the field values in the database at all. However, this means that the record referenced in the CAN-FIND statement is not available to your procedure. For example, this variation on the example tries to display the **OrderDate** from the **Order** record as well as the **Customer** fields:

```
FOR EACH Customer WHERE Country = "USA":
  IF CAN-FIND (FIRST Order OF Customer WHERE OrderDate < 1/1/98)
    THEN DISPLAY CustNum Name OrderDate.
  ELSE DISPLAY CustNum "No 1997 Orders" @ Name.
END.
```

This results in the error shown in [Figure 6–24](#), because the **Order** record is not available following the CAN-FIND reference to it.

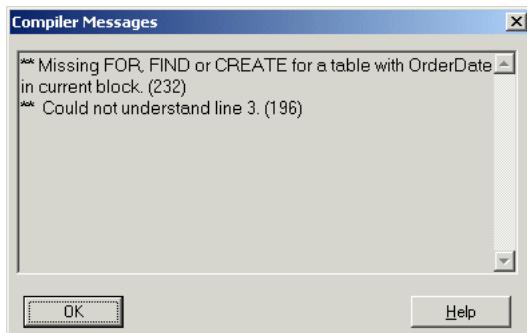


Figure 6–24: CAN-FIND error message

If you need the **Order** record itself then you must use a form that returns it to you:

```
FOR EACH Customer WHERE Country = "USA":
  FIND FIRST Order OF Customer WHERE OrderDate < 1/1/98 NO-ERROR.
  IF AVAILABLE Order THEN
    DISPLAY Customer.CustNum NAME OrderDate.
  ELSE DISPLAY "No 1997 Orders" @ NAME.
END.
```

When you run this code, you see the **OrderDate** as well as the **Customer** fields except in those cases where there is no **Order** from 1997, as shown in [Figure 6–25](#).

| Cust Num | Name | Ordered |
|----------|------------------------|----------|
| 1025 | Athlete's Track | 09/09/97 |
| 1030 | Soccer Universe | 11/23/97 |
| 1045 | Play Sports | 12/09/97 |
| 1797 | Back In The Game | 11/26/97 |
| 1800 | Legends Sporting Goods | 12/05/97 |
| 1 | Lift Tours | 09/27/97 |
| 1790 | J & S Sports Inc | 09/12/97 |
| | No 1997 Orders | |
| 1005 | Mike's Sports | 12/03/97 |
| 1789 | Athlete's Corner | 12/17/97 |

Figure 6–25: FIND FIRST Order result

The samples so far have shown the CAN-FIND function in an IF-THEN statement. You can also use it anywhere where a logical (true/false) expression is valid in a WHERE clause, such as this:

```
FOR EACH Customer WHERE Country = "USA" AND
    CAN-FIND (FIRST Order OF Customer WHERE OrderDate < 1/1/98):
        DISPLAY Customer.CustNum NAME.
END.
```

The next chapter continues the discussion on building complex procedures, with details on record buffers and record scope.

Record Buffers and Record Scope

This chapter continues the discussion on how to construct complex Progress 4GL procedures. It describes in detail the following concepts that were touched on in previous chapters:

- What record buffers are and how they manage data for you.
- More about how to use some of the string manipulation functions in Progress to manage lists of items for you.

You'll integrate these concepts with your GUI application to show the results of the calculations in your window for **Customers** and **Orders**. You'll also go back to your test window from [Chapter 2, “Using Basic 4GL Constructs,”](#) and define another user interface event for it to bring the data back to the window.

This chapter includes the following sections:

- [Record buffers](#)
- [Record scope](#)
- [Adding procedures to the test window](#)

Record buffers

This section discusses more precisely what record buffers do for you.

Whenever you reference a database table in a procedure and Progress makes a record from that table available for your use, you are using a record buffer. Progress defines a record buffer for your procedure for each table you reference in a FIND statement, a FOR EACH block, a REPEAT FOR block, or a DO FOR block. The record buffer, by default, has the same name as the database table. This is why, when you use these default record buffers, you can think in terms of accessing database records directly because the name of the buffer is the name of the table the record comes from. Think of the record buffer as a temporary storage area in memory where Progress manages records as they pass between the database and the statements in your procedures.

You can also define your own record buffers explicitly, though, using this syntax:

```
DEFINE BUFFER <buffer-name> FOR <table-name>.
```

There are many places in complex business logic where you need to have two or more different records from the same table available to your code at the same time, for comparison purposes. This is when you might use multiple different buffers with their own names. Here's one fairly simple example. In the following procedure, which could be used as part of a cleanup effort for the **Customer** table, you need to see if there are any pairs of **Customers** in the same city in the US with zip codes that don't match.

Here's the code that gives you these records:

```
DEFINE BUFFER Customer FOR Customer.  
DEFINE BUFFER OtherCust FOR Customer.  
  
FOR EACH Customer WHERE Country = "USA":  
    FIND FIRST OtherCust  
        WHERE Customer.State = OtherCust.State AND  
              Customer.City = OtherCust.City AND  
              SUBSTR (Customer.PostalCode, 1,3) NE  
                    SUBSTR (OtherCust.PostalCode, 1,3) AND  
              Customer.CustNum < OtherCust.CustNum NO-ERROR.  
    IF AVAILABLE OtherCust THEN  
        DISPLAY Customer.CustNum  
              Customer.City FORMAT "x(12)"  
              Customer.State FORMAT "xx"  
              Customer.PostalCode  
              OtherCust.CustNum  
              OtherCust.PostalCode.  
    END.
```

Take a look through this procedure. First, there is a pair of buffer definitions for the **Customer** table, one called **Customer** and one called **OtherCust**. The first definition, `DEFINE BUFFER Customer FOR Customer`, might seem superfluous because you get a buffer definition automatically when you reference the table name in your procedure. However, there are reasons why it can be a good idea to make all of your buffer definitions explicit like this. First, if you have two explicit buffer definitions up front, it makes it clearer that the purpose of this procedure is to compare pairs of **Customer** records. You might want to use alternative names for both buffers, such as **FirstCust** and **OtherCust**, to make it clear what your procedure is doing. This procedure uses an explicitly defined buffer with the same name as the table just to show that you can do this.

In addition, defining buffers that are explicitly scoped to the current procedure can reduce the chance that your code somehow inherits a buffer definition from another procedure in the calling stack. The defaults that the 4GL provides can be useful, but in serious business logic being explicit about all your definitions can save you from unexpected errors when the defaults don't work as expected.

Next the code starts through the set of all **Customers** in the USA. For each of those **Customers**, it tries to find another **Customer** with the same **City** and **State** values:

```
FOR EACH Customer WHERE Country = "USA":  
    FIND FIRST OtherCust  
        WHERE Customer.State = OtherCust.State AND  
            Customer.City = OtherCust.City . . .
```

Because you need to compare one **Customer** with the other, you can't simply refer to both of them using the name **Customer**. This is the purpose of the second buffer definition. Because the code is dealing with two different buffers that contain all the same field names, you need to qualify every single field reference to identify which of the two records you're referring to.

The next part of the **WHERE** clause compares the two zip codes, which are stored in the **PostalCode** field:

```
    AND SUBSTR (Customer.PostalCode, 1,3) NE  
        SUBSTR (OtherCust.PostalCode, 1,3) . . .
```

This procedure assumes that the last two digits of a zip code can be different within a given city, but that the first three digits are always the same. Because the **PostalCode** field is used for codes outside the US, which are sometimes alphanumeric, it is a character field, so the **SUBSTR** function extracts the first three characters of each of the two codes and compares them. If they are not equal, then the condition is satisfied.

The last bit of the **WHERE** clause needs some special explanation:

```
    .  
    .  
    .  
    AND Customer.CustNum < OtherCust.CustNum NO-ERROR.
```

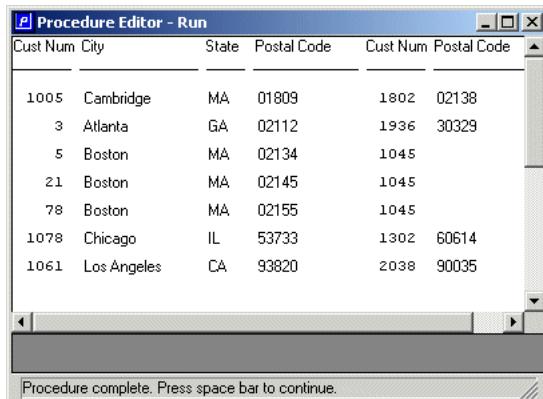
As the code walks through all the **Customers**, it finds a record using the **Customer** buffer and another record using the **OtherCust** buffer that satisfy the criteria. But later it also finds the same pair of **Customers** in the opposite order. So to avoid returning each pair of **Customers** twice, the code returns only the pair where the first **CustNum** is less than the second.

The FIND of the second **Customer** with a zip code that doesn't match the first is done with the NO-ERROR qualifier, and then the DISPLAY is done only if that record is AVAILABLE:

```
IF AVAILABLE OtherCust THEN
    DISPLAY Customer.CustNum
        Customer.City FORMAT "x(12)"
        Customer.State FORMAT "xx"
        Customer.PostalCode
        OtherCust.CustNum
        OtherCust.PostalCode.
```

In the DISPLAY statement you must qualify all the field names with the buffer name to tell Progress which one you want to see. In the case of the **City** and **State** it doesn't matter, of course, because they're the same, but you still have to choose one to display.

Figure 7–1 shows what you get when you run the procedure.



The screenshot shows a Windows application window titled "Procedure Editor - Run". Inside the window, there is a table with data. The columns are labeled "Cust Num", "City", "State", "Postal Code", "Cust Num", and "Postal Code". The data rows are as follows:

| Cust Num | City | State | Postal Code | Cust Num | Postal Code |
|----------|-------------|-------|-------------|----------|-------------|
| 1005 | Cambridge | MA | 01809 | 1802 | 02138 |
| 3 | Atlanta | GA | 02112 | 1936 | 30329 |
| 5 | Boston | MA | 02134 | | 1045 |
| 21 | Boston | MA | 02145 | | 1045 |
| 78 | Boston | MA | 02155 | | 1045 |
| 1078 | Chicago | IL | 53733 | 1302 | 60614 |
| 1061 | Los Angeles | CA | 93820 | 2038 | 90035 |

At the bottom of the window, a message says "Procedure complete. Press space bar to continue."

Figure 7–1: Comparing zip codes

You'll notice that the procedure takes a few seconds to run to completion. This is because the **City** field and the **State** field aren't indexed at all. For each of the over 1000 **Customers** in the USA, the procedure must do a FIND with a WHERE clause against all of the other **Customers** using these nonindexed fields. The **PostalCode** comparison doesn't help cut down the search either, because that's a nonequality match and the **PostalCode** is only a secondary component of an index. The code must work its way through all the **Customers** with higher **Customer** numbers looking for the first one that satisfies the selection. The fact that the OpenEdge database can do these many thousands of searches in just a few seconds is very impressive. There are various ways to make this search more efficient but they involve language constructs you haven't been introduced to yet, so this simple procedure serves for now.

Record scope

All the elements in a Progress 4GL procedure have a scope. That is, Progress defines the portion of the application in which you can refer to the element. The buffers Progress uses for database records are no exception. In this section, you'll look at how record scope affects statements that read database records. In [Chapter 16, “Updating Your Database and Writing Triggers,”](#) you learn about how record scope affects the way you save changes back to the database within a transaction. The update-related actions include determining when in a procedure a record gets written back to the database and when it clears a record from a buffer and reads in another one. Remember that a reference to a database record is always a reference to a buffer where that record is held in memory for you, and all buffers are treated the same, whether you define them explicitly or they are provided for you by default.

There are some rules Progress uses to define just how record scope is determined. The rules might seem a bit complex at first, but they are just the result of applying some common-sense principles to the way a procedure is organized. To understand the rules, you first need to learn a few terms.

If you reference a buffer in the header of a REPEAT FOR or DO FOR block, this is called a *strong-scoped reference*. Any reference to the buffer within the block is a strong-scoped reference. What does this mean? The term *strong-scoped* means that you have made a very explicit reference to the scope of the buffer by naming it in the block header. You have told Progress that the block applies to that buffer and is being used to manage that buffer. By providing you with a buffer scoped to that block, Progress is really just following your instructions.

On the other hand, if you reference a buffer in the header of a FOR EACH or PRESELECT EACH block, this is called a *weak-scoped reference*. Any reference to the buffer within the block is a weak-scoped reference.

Why this difference? Keep in mind that a REPEAT block or a DO block does not automatically iterate through a set of records. You can execute many kinds of statements within these blocks, and if you want to retrieve a record in one of them, you have to use a FIND statement to do it. This is why naming the buffer in the block header is called strong scoping.

By contrast, a FOR EACH block or a PRESELECT EACH block must name the buffers it uses, because the block automatically iterates through the set of records the block header defines. For this reason, because you really don't have any choice except to name the buffer in the block header, Progress treats this as a weak reference. Progress recognizes that the buffer is used in that block, but it doesn't treat it as though it can only be used within that block. You'll see how the difference affects your procedures in the next section.

The third type of buffer reference is called a *free reference*. Any other reference to a buffer other than the kinds already described is a free reference. Generally, this means references in FIND statements. These are called free references because they aren't tied to a particular block of code. They just occur in a single statement in your procedure.

The following sections describe the rules that determine how Progress treats record buffers that are used in different kinds of buffer references.

Record Buffer Rule 1: Each strong-scoped or weak-scoped reference to a buffer is self-contained.

You can combine multiple such blocks in a procedure, and each one scopes the buffer to its own block. This rule holds as long as no other reference forces the buffer to be scoped to a higher level outside these blocks. Here's an example:

```
FOR EACH Customer BY creditLimit DESCENDING:
    DISPLAY "Highest:" CustNum NAME CreditLimit
        WITH 1 DOWN.
    LEAVE.
END.

FOR EACH Customer WHERE state = "NH" BY CreditLimit DESCENDING:
    DISPLAY CustNum NAME CreditLimit.
END.
```

This code has two FOR EACH blocks each with a weak-scoped reference to the **Customer** buffer. This is perfectly valid. First, Progress scopes the **Customer** buffer to the first FOR EACH block. When that block terminates, Progress scopes the buffer to the second FOR EACH block.

The first block identifies the **Customer** with the highest **CreditLimit**. To do this, it sets up a FOR EACH block to cycle through all the **Customers**. The statement sorts the **Customers** by their **CreditLimit** in descending order. After it reads and displays the first of these records, the one with the highest **CreditLimit**, the LEAVE statement forces the block to terminate.

The qualifier WITH 1 DOWN on the DISPLAY statement for the first block tells Progress that it only needs to define a frame with space for one **Customer**. Otherwise it would allocate the entire display space. The literal Highest: makes it clear in the output what you're looking at.

The second block then independently displays all the **Customers** in the state of New Hampshire in order by their **CreditLimit**, with the highest value first. Because these two blocks occur in sequence, Progress can scope the **Customer** buffer to each one in turn and reuse the same **Customer** buffer in memory, without any conflict. That's why this form is perfectly valid.

Figure 7–2 shows what you see when you run the procedure.

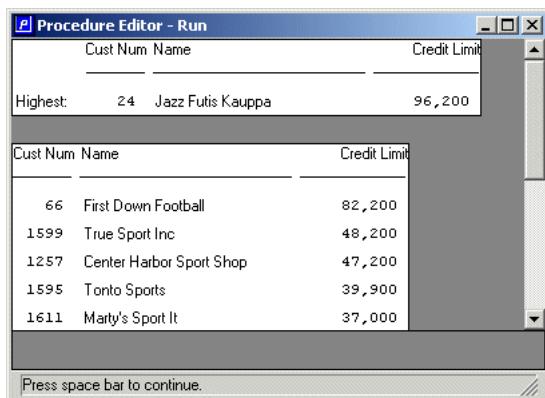


Figure 7–2: Result of record buffer Rule 1 example

Generating a procedure listing file

To verify how Progress is scoping the record buffers, you can generate a listing file that contains various information about how Progress processes the procedure when you compile or run it.



To do this, use the LISTING option on the COMPILE statement:

1. **Save** the procedure so that it has a name you can reference: `testscope.p`.
2. **Open** another procedure window and enter this statement:

```
COMPILE testscope.p LISTING testscope.lis.
```

3. Press **F2** to run the COMPILE statement.

4. Select **File→Open** to open `testscope.lis`. Here is the code you should see:

| {} Line Blk | | | | | |
|---------------|------|---|------|------|-------|
| -- ----- | | | | | |
| Line | Blk. | Type | Tran | Blk. | Label |
| 1 | 1 | FOR EACH Customer BY creditLimit DESCENDING: | | | |
| 2 | 1 | DISPLAY "Highest:" CustNum NAME CreditLimit | | | |
| 3 | 1 | WITH 1 DOWN. | | | |
| 4 | 1 | LEAVE. | | | |
| 5 | | END. | | | |
| 6 | | | | | |
| 7 | | FOR EACH Customer WHERE state = "NH" BY CreditLimit DESCENDING: | | | |
| 8 | 1 | DISPLAY CustNum NAME CreditLimit. | | | |
| 9 | | END. | | | |
| 10 | | | | | |
| <hr/> | | | | | |
| File Name | | | | | |
| .\testscope.p | 0 | Procedure | No | | |
| .\testscope.p | 1 | For | No | | |
| | | Buffers: sports2000.Customer | | | |
| | | Frames: Unnamed | | | |
| .\testscope.p | 7 | For | No | | |
| | | Buffers: sports2000.Customer | | | |
| | | Frames: Unnamed | | | |

This listing file tells you that line 1 of the procedure starts a FOR block and that this block does not start a transaction (again, more on that in [Chapter 16, “Updating Your Database and Writing Triggers”](#)). The next line tells you what you need to know about scoping. The line that reads `Buffers: sports2000.Customer` tells you that the **Customer** buffer is scoped to this FOR block and that it used an unnamed frame that is also scoped to that block. Next you see that another FOR block begins at line 7. The **Customer** buffer is also (independently) scoped to that block and it has its own unnamed frame.

You could construct similar examples using any combination of strong- and weak-scoped buffer references. For example, here's a variation on the test procedure that uses a DO FOR block with a strong scope to the **Customer** buffer:

```
DO FOR Customer:  
    FIND FIRST Customer WHERE CreditLimit > 60000.  
    DISPLAY CustNum NAME CreditLimit.  
END.  
  
FOR EACH Customer WHERE state = "NH" BY CreditLimit DESCENDING:  
    DISPLAY CustNum NAME CreditLimit.  
END.
```

This procedure scopes the **Customer** buffer to each block in turn, just as the first example does.

Record Buffer Rule 2: You cannot nest two weak-scoped references to the same buffer.

For example, here's a procedure that violates this rule:

```
DEFINE VARIABLE dLimit AS DECIMAL    NO-UNDO.  
FOR EACH Customer WHERE state = "NH" BY CreditLimit DESCENDING:  
    DISPLAY CustNum NAME CreditLimit.  
    dLimit = Customer.CreditLimit.  
    FOR EACH Customer WHERE CreditLimit > dLimit:  
        DISPLAY CustNum NAME CreditLimit.  
    END.  
END.
```

If you try to run this procedure, you get the error shown in [Figure 7–3](#) that tells you that your buffer references are invalid.

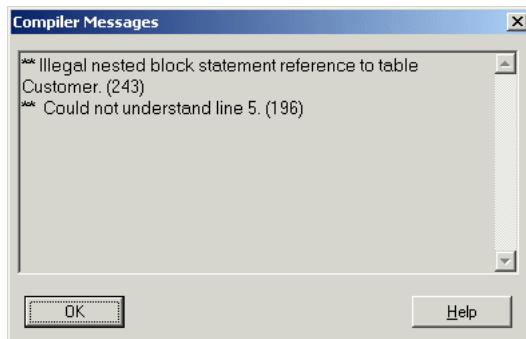


Figure 7–3: Invalid buffer references error message

When you think about it, this is perfectly sensible and necessary. Picture this situation:

Progress is using the **Customer** buffer for the current **Customer** record in the outer FOR EACH block. The first time through the block, it contains the New Hampshire **Customer** with the highest **CreditLimit**. Now suddenly Progress gets a request to use that same buffer to start another FOR EACH block, while it's still in the middle of processing the outer one. This could not possibly work. If Progress replaced the New Hampshire **Customer** with whatever **Customer** was the first one to satisfy the selection of **Customers** with higher **CreditLimits** and then you had another reference to the first Customer later on in the outer block (which would be perfectly valid), that **Customer** record would no longer be available because Progress would have used the same buffer for the inner FOR EACH block. Because this can't be made to work with both blocks sharing the same buffer at the same time, this construct is invalid.

Record Buffer Rule 3: A weak-scope block cannot contain any free references to the same buffer.

This rule makes sense for the same reasons as the second rule. Consider this example:

```
DEFINE VARIABLE dLimit AS DECIMAL      NO-UNDO.
FOR EACH Customer WHERE state = "NH" BY CreditLimit DESCENDING:
  DISPLAY CustNum NAME CreditLimit.
  dLimit = Customer.CreditLimit.
  FIND FIRST Customer WHERE CreditLimit > dLimit.
  DISPLAY CustNum NAME CreditLimit.
END.
```

While Progress is processing the FOR EACH block, it gets a request to use the same buffer to find a completely unrelated record. This fails with a similar error, as shown in [Figure 7–4](#).

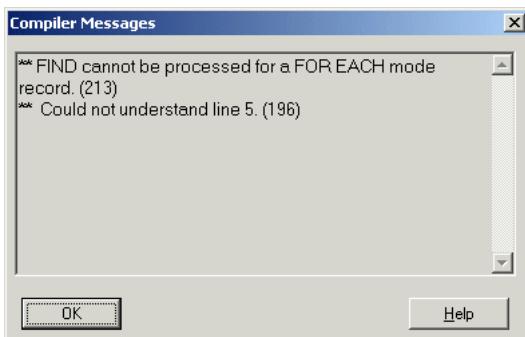


Figure 7–4: FOR EACH processing error message

Record Buffer Rule 4: If you have a free reference to a buffer, Progress tries to scope that buffer to the nearest enclosing block with record scoping properties (that is, a FOR EACH block, a DO FOR block, or a REPEAT block). If no block within the procedure has record scoping properties, then Progress scopes the record to the entire procedure.

This rule also makes good sense when you think about it. The FIND statements are called free references because they don't define a scope for the buffer, they just reference it. Therefore, Progress has to identify some scope for the record beyond the FIND statement. When a block has record scoping properties, it is a block Progress might try to scope a record to, when the record is referenced inside the block.

Here's another variation on the `testscope.p` procedure that demonstrates this rule:

```
DEFINE VARIABLE dLimit AS DECIMAL      NO-UNDO INIT 0.

FOR EACH Customer WHERE State = "NH" BY CreditLimit DESCENDING:
  IF dLimit = 0 THEN
    dLimit = Customer.CreditLimit.
  DISPLAY CustNum NAME CreditLimit.
END.

FIND FIRST Customer WHERE CreditLimit > dLimit.
DISPLAY CustNum NAME CreditLimit.
```

This procedure is perfectly valid. The first time through the FOR EACH loop, the procedure saves off the **CreditLimit** for use later in the procedure. Because the **dLimit** variable is initialized to zero, checking for **dLimit = 0** tells you whether it's already been set. When you run it, you see all the New Hampshire **Customer** records followed by the first **Customer** with a **CreditLimit** higher than the highest value for New Hampshire **Customers**. Because there's no conflict with two blocks trying to use the same buffer at the same time, it compiles and runs successfully.

But the rule that Progress raises the scope in this situation is a critically important one. In complex procedures, the combination of buffer references you use might force Progress to scope a record buffer higher in the procedure than you expect. Though this normally does not have a visible effect when you're just reading records, when you get to the discussion of transactions this rule becomes much more important. If you generate another listing file for this procedure, you see the effect of the FIND statement:

```
{} Line Blk
-----
1   DEFINE VARIABLE dLimit AS DECIMAL      NO-UNDO INIT 0.
2
3   1   FOR EACH Customer WHERE State = "NH" BY CreditLimit DESCENDING:
4   1       IF dLimit = 0 THEN
5   1           dLimit = Customer.CreditLimit.
6   1           DISPLAY CustNum NAME CreditLimit.
7       END.
8
9   FIND FIRST Customer WHERE CreditLimit > dLimit.
10  DISPLAY CustNum NAME CreditLimit.
11

File Name          Line    Blk. Type     Tran      Blk. Label
-----
.\testscope.p      0       Procedure  No

Buffers: sports2000.Customer
Frames:        Unnamed

.\testscope.p      3       For        No
Frames:        Unnamed
```

This tells you that the **Customer** buffer is scoped at line 0, that is, to the procedure itself. There's no reference to the **Customer** buffer in the information for the FOR block at line 3 because Progress has already scoped the buffer higher than that block.

Next is the rule concerning combining FIND statements with strong-scoped, rather than weak-scoped references.

Record Buffer Rule 5: If you have a strong-scoped reference to a buffer, you cannot have a free reference that raises the scope to any containing block.

This rule also makes perfect sense. The whole point of using a strong-scoping form, such as a DO FOR block, is to force the buffer scope to that block and nowhere else. If Progress encounters some other statement (such as a FIND statement) outside the strong-scoped block that forces it to try to scope the buffer higher than the strong scope, it cannot do this because this violates the strong-scoped reference. Here's an example:

```
DEFINE VARIABLE dLimit AS DECIMAL NO-UNDO.

DO FOR Customer:
    FIND FIRST customer WHERE state = "MA".
    DISPLAY CustNum NAME CreditLimit.
    dLimit = Customer.CreditLimit.
END.

FIND FIRST Customer WHERE Customer.CreditLimit > dLimit.
DISPLAY CustNum NAME CreditLimit.
```

If you try to run this procedure you get the error shown in Figure 7–5.

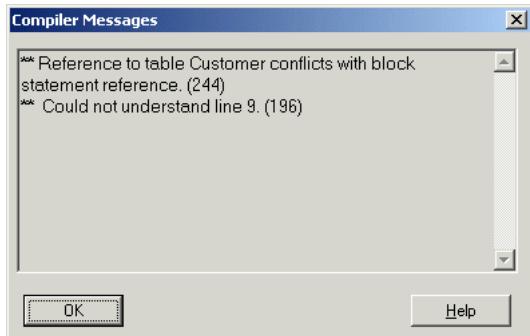


Figure 7–5: Conflicting table reference error message

Remember this distinction between Rule 1 and Rule 5. Rule 1 says that strong- and weak-scoped references in separate blocks are self-contained, so it is legal to have multiple blocks in a procedure that scope the same buffer to the block. Rule 5 tells you that it is not legal to have some other reference to the buffer that would force the scope to be higher than any of the strong-scoped references to it.

Here are a few more small examples that illustrate how these rules interact:

```
DEFINE VARIABLE iNum AS INTEGER      NO-UNDO INIT 0.

DO FOR Customer:
    FOR EACH Customer WHERE CreditLimit > 80000
        BY CreditLimit DESCENDING:
            DISPLAY CustNum NAME CreditLimit.
            IF iNum = 0 THEN iNum = Customer.CustNum.
    END.
    FIND Customer WHERE CustNum = iNum.
    DISPLAY NAME FORMAT "x(18)"
        City FORMAT "x(12)"
        State FORMAT "x(12)"
        Country FORMAT "x(12)".
END.
```

This procedure displays all the **Customers** with **CreditLimits** over 80000, saving off the **Customer** number of the highest one. [Figure 7–6](#) shows the first result.

| Procedure Editor - Run | | |
|------------------------|----------------------|--------------|
| Cust Num | Name | Credit Limit |
| 24 | Jazz Futis Kauppa | 96,200 |
| 1064 | Hook, Line & Sinker | 89,000 |
| 1075 | Tim's Sporting Goods | 83,500 |
| 66 | First Down Football | 82,200 |

Figure 7–6: Customers with CreditLimits over 80000 – first result

It then finds that **Customer** again with the highest **CreditLimit** and redisplays it with some more fields, as shown in [Figure 7–7](#).

| Procedure Editor - Run | | | |
|------------------------|------|-----------|---------|
| Name | City | State | Country |
| Jazz Futis Kauppa | Pori | Satakunta | Finland |

Figure 7–7: Customers with CreditLimits over 80000 – next result

This example illustrates that it is valid to have a weak-scoped block enclosed in a strong-scoped block. Progress raises the scope of the **Customer** buffer to the outer DO FOR block. This allows you to reference the buffer elsewhere in the DO FOR block, such as the FIND statement. The FIND statement raises the scope of the buffer to the DO FOR block, the nearest containing block with block-scoping properties.

Here's another example that illustrates raising buffer scope:

```
REPEAT:  
    FIND NEXT Customer USE-INDEX NAME.  
    IF NAME < "D" THEN NEXT.  
    ELSE LEAVE.  
END.  
  
DISPLAY CustNum NAME.
```

As it processes the procedure, Progress encounters the FIND statement and tentatively scopes the **Customer** buffer to the REPEAT block. The REPEAT block by itself does not force a buffer scope without a FOR phrase attached to it but it does have the record-scoping property, so it is the nearest containing block for the FIND statement. This block cycles through **Customers in Name** order and leaves the block when it gets to the first one starting with **D**. But after that block ends, Progress finds a free reference to the **Customer** buffer in the DISPLAY statement. This forces Progress to raise the scope of the buffer outside the REPEAT block. Since there is no available enclosing block to scope the buffer to, Progress scopes it to the procedure. Thus, the **Customer** buffer from the REPEAT block is available after that block ends to display fields from the record, as shown in [Figure 7–8](#).



Figure 7–8: Raising buffer scope example result

This next procedure has two free references, each within its own REPEAT block:

```

REPEAT:
  FIND NEXT Customer USE-INDEX NAME.
  IF NAME BEGINS "D" THEN DO:
    DISPLAY CustNum NAME WITH FRAME D.
    LEAVE.
  END.
END.

REPEAT:
  FIND NEXT Customer USE-INDEX NAME.
  IF NAME BEGINS "E" THEN DO:
    DISPLAY CustNum NAME WITH FRAME E.
    LEAVE.
  END.
END.

```

As before, Progress initially scopes the buffer to the first REPEAT block. But on encountering another FIND statement within another REPEAT block, Progress must raise the scope to the entire procedure. The first block cycles through **Customers** until it finds and displays the first one whose name begins with **D**, and then leaves the block. Because the buffer is scoped to the entire procedure, the FIND statement inside the second REPEAT block starts up where the first one ended, and continues reading **Customers** until it gets to the first one beginning with **E**. [Figure 7–9](#) shows the result.



Figure 7–9: Raising buffer scope example 2 result

This is a very important aspect of buffer scoping. Not only are both blocks using the same buffer, they are also using the same index cursor on that buffer. This is different from the earlier examples where multiple strong- or weak-scoped blocks scope the buffer independently. In these cases, each block uses a separate index cursor, so a second DO FOR or FOR EACH starts fresh back at the beginning of the record set. The difference is that the FIND statements inside these REPEAT blocks are free references, so they force Progress to go up to an enclosing block that encompasses all the free references.

Adding procedures to the test window

This section wraps up this chapter by having you write a couple of new procedures for the test window you built in [Chapter 2, “Using Basic 4GL Constructs.”](#) These procedures do various calculations involving blocks of code that illustrate the principles of block structure and scoping you’ve learned in this chapter. The procedures pass calculated values back to the window for display.

Defining h-OrderCalcs.p to calculate totals

This section describes how to create a sample subprocessure, called `h-ordercalcs.p`, to calculate the totals for the price and extended price for all **Order** lines.



To create the first subprocessure:

1. Open a **New Procedure Window**.
2. Enter this 4GL code:

```
/* h-OrderCalcs.p */

DEFINE INPUT  PARAMETER piOrderNum      AS INTEGER      NO-UNDO.
DEFINE OUTPUT PARAMETER pdOrderPrice    AS DECIMAL     NO-UNDO.
DEFINE OUTPUT PARAMETER pdOrderTotal   AS DECIMAL     NO-UNDO.

FIND Order WHERE Order.OrderNum = piOrderNum.

/* Add up the total price and the extended price for all order lines. */
FOR EACH OrderLine OF Order:
  ASSIGN pdOrderTotal = pdOrderTotal + OrderLine.ExtendedPrice
  pdOrderPrice = pdOrderPrice + OrderLine.Price * OrderLine.Qty.
END.
```

3. Save this code as h-OrderCalcs.p.

The h-CustOrderWin2.w test window will call this procedure. This procedure has an INPUT parameter, which is an **Order** number, and it passes back two OUTPUT parameters. It uses a naming convention that begins each parameter with the letter p to help identify them throughout the procedure.

The code finds the **Order** record for the **Order** number passed in, and then in a FOR EACH block, it reads each **OrderLine** for the **Order** and adds up two sets of values. The first value, stored in pdOrderTotal, is the total of all the **ExtendedPrice** values for the **OrderLines**. The **ExtendedPrice** is the price after the **Customer's** discount has been factored in. The second value, pdOrderPrice, is simply the total of the raw price for the **OrderLine** before the discount, which is the **Price** field times the **Qty** (quantity) field.

After the FOR EACH block, the procedure ends and returns the output parameters to the caller.

Notice that the FIND statement can be a unique find (without a qualifier such as FIRST) because there can be only one **Order** record with a given **OrderNum** value.

Next there's a new statement type in the FOR EACH block: the ASSIGN statement. Any time you are assigning more than one value at a time, as the code is here, it is more efficient to use this statement, which can contain any number of assignments, each using the equal sign, and a single period at the end.

Checking the syntax of a procedure

If you try to run the procedure directly by pressing F2, you get the error message shown in Figure 7–10.

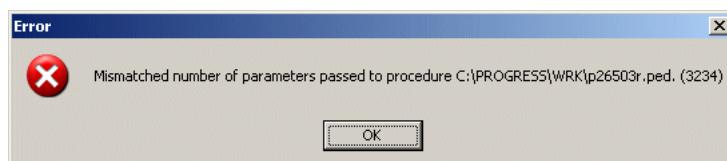


Figure 7–10: Mismatched parameters error message

This error message appears because the procedure expects parameters and you're not passing it any when you just run it from the editor. However, you can check whether your syntax is valid by pressing SHIFT-F2 or selecting **Compile**→**Check Syntax** from the menu. The information box shown in [Figure 7–11](#) appears.



Figure 7–11: Correct syntax information box

After you verify that the syntax is correct, continue on to write the code that calls the procedure.

Adding fill-ins to the window for the calculations

In this section you add fill-in fields to your sample window for **Order** calculations.



To write the code that calls h-OrderCalcs.p:

1. Open h-CustOrderWin2.w in the AppBuilder.
2. Stretch the window somewhat at the bottom to make room for some more fields.
3. To place fill-in fields on your window to hold the totals the procedure passes back, choose the **Fill-In** icon on the **Palette**:



4. Click under the browse in your design window to drop the fill-in field onto the window.
5. Repeat Steps 3 and 4 twice more to get a total of three new fill-in fields.

Alternatively, you can click twice on the **Fill-In** icon to lock it in selected mode, and then click three times on the window to get the three fill-ins. The icon shows a lock symbol to show the mode it's in:



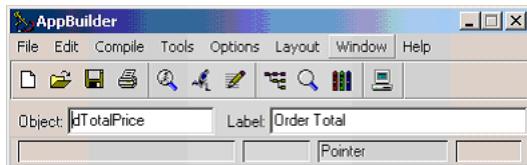
If you lock the **Fill-In** icon, deselect it when you're done adding the three fields by clicking the pointer icon on the **Palette**.

6. In the AppBuilder main window, give the three fields these names and labels, respectively:
 - **dTotalPrice** and **Order Total**.
 - **dTotalExt** and **Total Ext Price**.
 - **dAvgDisc** and **Average Discount**.

Changing field properties in the property sheet

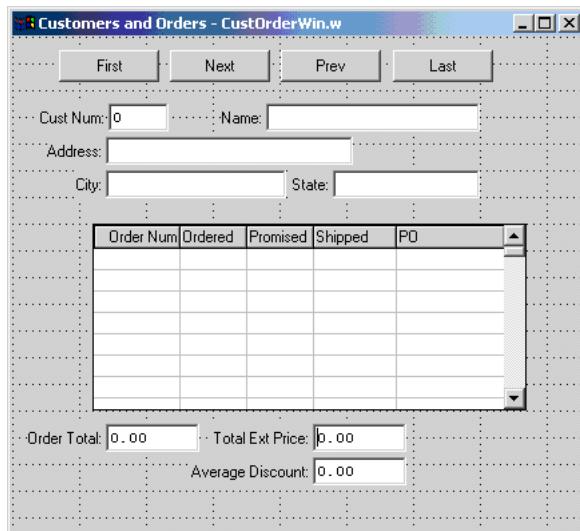
To edit the properties of the fill-in fields:

1. Double-click on a fill-in field to bring up its property sheet.
2. In the **Define As** drop-down list, change the data type to **Decimal**.
3. Check the toggle boxes to make the field **Enabled** but **Read-Only**. These options provide a shadow box around each value but prevent the user form modifying the field value.
4. Change the field **Width** to **16**:



5. Choose **OK**.
6. Repeat Steps 1 through 5 for the other two fill-in fields.

The design window should look roughly like this when you're done:



Writing a VALUE-CHANGED trigger for the browse

Now you need to write the code to run the `h-OrderCalcs.p` procedure. When does this need to happen? Whenever a new row is selected in the list of **Orders** for a **Customer**. This is called the **VALUE-CHANGED** event for the browse.



To write the trigger for the browse:

1. Click on the browse to select it, then choose the **Edit Code** icon in the toolbar. The **Section Editor** appears.

The default code block to define for a browse is the **VALUE-CHANGED** trigger, so this comes up automatically.

2. Fill in the empty DO-END block with this code:

```

DO:
  RUN h-OrderCalcs.p
    (INPUT Order.OrderNum,
     OUTPUT dTotalPrice, OUTPUT dTotalExt).
  dAvgDisc = (dTotalPrice - dTotalExt) / dTotalPrice.
  DISPLAY dTotalPrice dTotalExt dAvgDisc
    WITH FRAME CustQuery.
END.

```

This code runs the procedure, passes in the current **Order** number, and gets back the two values that map to the fill-in fields you defined. These are really just variables, but the AppBuilder generates definitions with a special phrase added to turn the variable into a value that can be displayed in the window as a proper GUI control. This is the **VIEW-AS** phrase and here is one of the three variable definitions the AppBuilder generates:

```
DEFINE VARIABLE dTotalPrice AS DECIMAL FORMAT "->>9.99":U INITIAL 0
  LABEL "Order Total"
  VIEW-AS FILL-IN
  SIZE 14 BY 1 NO-UNDO.
```

The **SIZE width BY height** phrase defines the display size of the fill-in field. A **DEFINE VARIABLE** statement can specify all the different kinds of ways a single field value can be displayed, including toggle boxes for logical values, selection lists, and editor controls, using the **VIEW-AS** phrase. You'll learn a lot more about this phrase in [Chapter 8, “Defining Graphical Objects.”](#)

After the **RUN** statement, the code does a calculation of its own to determine the average discount and stores that value in the third fill-in field:

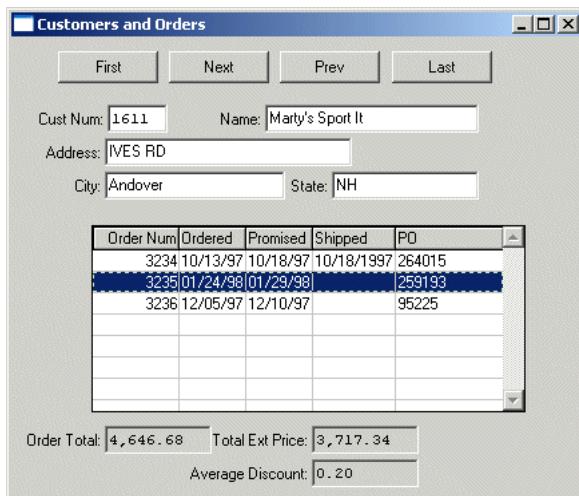
```
dAvgDisc = (dTotalPrice - dTotalExt) / dTotalPrice.
```

Finally, the code displays the three fields in the window's frame:

```
DISPLAY dTotalPrice dTotalExt dAvgDisc
  WITH FRAME CustQuery.
```

As you can see, this **DISPLAY** statement looks exactly like the **DISPLAY** statements you've been using in the little examples earlier in the chapter. But because the fields are given the specific display type of **FILL-IN**, they show up as GUI controls rather than just in the report-like format you get by default.

- To see the effects of your changes, **Run** the procedure. When you first run it, the fields come up with zeroes in them. But when you click on another row of the browse, then the calculations appear correctly:



The **VALUE-CHANGED** event happens only when you actually select a row in the browse, not when it is first populated with data. To correct this, you need to apply that event when the window is first initialized and also whenever the user selects a different **Customer**.

Adding APPLY statements to the procedure

This section provides an example of how you can use the **APPLY** statement to cause an event.



To add an **APPLY** statement to your test window procedure:

1. Add this code to the main block of the window procedure:

```

MAIN-BLOCK:
DO ON ERROR    UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
    ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:
        RUN enable_UI.
        APPLY "VALUE-CHANGED" TO OrderBrowse.
            IF NOT THIS-PROCEDURE:PERSISTENT THEN
                WAIT-FOR CLOSE OF THIS-PROCEDURE.
END.

```

The **APPLY** statement causes an event to happen programmatically, just as the user action of clicking on a browse row would cause the event. Now the event is fired when the window first comes up.

2. Add the same **APPLY** statement to each of the four trigger blocks for the **First**, **Prev**, **Next**, and **Last** buttons, after the **Order** query is opened. For example:

```

DO:
    GET NEXT CustQuery.
    IF AVAILABLE Customer THEN
        DO:
            DISPLAY Customer.CustNum Customer.Name Customer.Address Customer.City
                Customer.State
                    WITH FRAME CustQuery IN WINDOW CustWin.
                    {&OPEN-BROWSERS-IN-QUERY-CustQuery}
                    APPLY "VALUE-CHANGED" TO OrderBrowse.
        END.
    END.

```

Now the event is fired each time there's a different **Customer**, when the **Order** query is reopened.

It would be helpful to have this behavior become a part of every navigation button automatically so that you didn't have to enter this line of code all over the place. This is a small part of what a standard framework like Progress Dynamics does for you, and you'll learn more about that later. For now your modified window procedure should work properly in all the places where the VALUE-CHANGED event must occur.

Writing the BinCheck procedure to check inventory

In this section, you add another procedure call that illustrates some of the block types you studied in this chapter and some of the list handling functions summarized in [Chapter 1, “Introducing the Progress 4GL.”](#) The procedure looks at the **Warehouse** and **Bin** tables to see which **Warehouses** can and cannot supply the **Items** for a given **Order**.



To modify the h-OrderCalcs.p procedure, add the new statements in bold type:

```

DEFINE INPUT  PARAMETER piOrderNum          AS INTEGER      NO-UNDO.
DEFINE OUTPUT PARAMETER pdOrderPrice        AS DECIMAL     NO-UNDO.
DEFINE OUTPUT PARAMETER pdOrderTotal        AS DECIMAL     NO-UNDO.
DEFINE OUTPUT PARAMETER pcWarehouseList    AS CHARACTER    NO-UNDO.
DEFINE OUTPUT PARAMETER pcBestWarehouse   AS CHARACTER    NO-UNDO.

DEFINE VARIABLE cItemList      AS CHARACTER    NO-UNDO.

FIND order WHERE Order.orderNum = piOrderNum NO-ERROR.

FOR EACH OrderLine OF Order:
  ASSIGN pdOrderTotal = pdOrderTotal + OrderLine.ExtendedPrice
      pdOrderPrice = pdOrderPrice + OrderLine.Price * OrderLine.Qty
      cItemList = cItemList +
      (IF cItemList = "" THEN "" ELSE ",") +
      STRING(ItemNum).
END.

RUN h-BinCheck.p (INPUT cItemList, OUTPUT pcWarehouseList, OUTPUT
pcBestWarehouse).

```

The additional code defines two new OUTPUT parameters and a new variable.

Then, as part of the ASSIGN statement, it constructs a list of **Item** numbers for the **OrderLines** of the **Order**. To make a list, it uses a CHARACTER variable **cItemList**. The assignment statement effectively means:

1. Take the current value of the **cItemList** variable (which is initially blank).
2. If it's blank, then append a blank value to it (this is just a no-op condition for the IF-THEN-ELSE statement, which requires both a THEN phrase and an ELSE phrase). Otherwise, if there's already something in the list, append a comma to it to separate the Items.
3. Use the STRING built-in function to convert the integer **ItemNum** to a CHARACTER value and append it to the variable. (There are other built-in functions like this one named DECIMAL, INTEGER, DATE, and LOGICAL to convert character strings to those other data types as well.)

At the end of the FOR EACH block, **cItemList** holds a comma-separated list of all the **Items** for the current **Order**.

Finally, the procedure runs another procedure, **h-BinCheck.p**, which you'll write next.



To write the **h-BinCheck.p** procedure:

1. Save this modified version of **h-OrderCalcs.p**.
2. Open a **New Procedure Window** and start to write **h-BinCheck.p** with the following code. Add each new group of statements to the procedure as they are discussed:

```
/* h-BinCheck.p */

DEFINE INPUT  PARAMETER pcItemList          AS CHARACTER      NO-UNDO.
DEFINE OUTPUT PARAMETER pcWarehouseList     AS CHARACTER      NO-UNDO.
DEFINE OUTPUT PARAMETER pcBestWarehouse    AS CHARACTER      NO-UNDO.

DEFINE VARIABLE iEntry                      AS INTEGER        NO-UNDO.
DEFINE VARIABLE iItemNum                   AS INTEGER        NO-UNDO.
DEFINE VARIABLE iWQty                      AS INTEGER        NO-UNDO.
DEFINE VARIABLE iWHNum                     AS INTEGER        NO-UNDO.
DEFINE VARIABLE iBestWH                    AS INTEGER        NO-UNDO.
DEFINE VARIABLE cBestList                  AS CHARACTER      NO-UNDO.
```

This procedure takes the list of item numbers as an INPUT parameter and returns two CHARACTER parameters. The various variables are used throughout the procedure. Remember that you can use the editor shortcuts (IPC, OPC, DVI, and DVC) to generate most of the DEFINE PARAMETER and VARIABLE statements for you.

Using list and string functions to manage a list of values

The h-BinCheck.p procedure needs to make a list of how many **Items** are supplied by each **Warehouse**. There are various ways to code this, but to illustrate some more of the string manipulation functions you were introduced to in [Chapter 2, “Using Basic 4GL Constructs,”](#) you’ll build this as a character string.



To update h-BinCheck.p to make a list of the number of Items supplied by each Warehouse:

1. Add placeholders for the count of **Items** in each **Warehouse**. The following code forms a list with as many zeroes as there are **Warehouses**. The zero values are later incremented to count **Items** supplied by each **Warehouse**:

```
FOR EACH Warehouse:  
    cBestList = cBestList + "0,".  
END.  
cBestList = RIGHT-TRIM(cBestList, ",").
```

Note: The RIGHT-TRIM function removes the final comma from the list, rather than the IF-THEN-ELSE statement in the assignment that created the item list in OrderProcs.p. These are just different ways of doing the same thing. The RIGHT-TRIM function is a bit more efficient.

2. To loop through the list of **Items**, add a DO block with the NUM-ENTRIES function:

```
DO iEntry = 1 TO NUM-ENTRIES(pcItemList):
```

NUM-ENTRIES counts the entries in a list using a comma as the delimiter between entries by default. If you need to use a delimiter other than a comma, the delimiter can be an optional second argument to the function.

3. Add a statement that embeds two built-in functions into one statement:

```
iItemNum = INTEGER(ENTRY(iEntry, pcItemList)).
```

The ENTRY function extracts entry number **iEntry** from **pcItemList**. It returns this to the **INTEGER** function, which converts the value back to an integer. So now you've restored the **Item** number to its original form.

4. Add a block of code that operates on this **Item** number. The **Bin** table represents bins or containers in each **Warehouse** that are used to store the various **Items**. It has both an **ItemNum** field to point to the **Item** record, and a **WarehouseNum** field to point to the **Warehouse** where the **Bin** is located. If the **Qty** (quantity) field for a **Bin** record is 0, then the **Warehouse** that **Bin** is in cannot supply that part. The code builds up this list of **Warehouse** names. The **LOOKUP** function looks for a string in a list. If it finds it, it returns the position of the entry in the list. Otherwise, it returns 0 if the entry is not in the list. Here the **LOOKUP** function is used to make sure that a **Warehouse** name is added to the list once only if it's not already there:

```
FOR EACH Bin WHERE Bin.ItemNum = iItemNum:
  IF Bin.Qty = 0 THEN
    DO:
      FIND Warehouse WHERE Warehouse.WarehouseNum = Bin.WarehouseNum.
      IF LOOKUP(WarehouseName, pcWarehouseList) = 0 THEN
        pcWarehouseList = pcWarehouseList +
          (IF pcWarehouseList = "" THEN "" ELSE ",") + WarehouseName.
    END.
  END.
```

5. Still within the DO block that iterates on each item, add code that initializes two variables to zero using a single **ASSIGN** statement:

```
ASSIGN iWQty = 0 iWNum = 0.
```

These variables hold the quantity of each item at a **Warehouse** and the **Warehouse** number.

**To use the REPEAT PRESELECT block to pre-fetch records:**

1. Add a REPEAT block that preselects each **Bin** that holds the current **Item**, along with the **Warehouse** where the **Bin** is located, filtering these to include only **Warehouses** in the USA. The records are sorted in descending order of their quantity. This identifies which **Warehouse** has the largest quantity of the **Item** in inventory. Remember that the PRESELECT phrase forces Progress to retrieve all the matching records before beginning to execute the statements in the block:

```
REPEAT PRESELECT EACH Bin WHERE Bin.ItemNum = iItemNum,  
    FIRST Warehouse WHERE Warehouse.WarehouseNum = Bin.WarehouseNum AND  
    Warehouse.Country = "USA" BY Bin.Qty DESCENDING:
```

2. Add the code that finds the next **Warehouse** record in this preselected list. The first time through the REPEAT block, the FIND NEXT statement finds the first record:

```
FIND NEXT Warehouse.
```

Why does the statement name the **Warehouse** buffer and not the **Bin**? The rule is that whenever you are doing a FIND on a PRESELECT result set that involves a join, you must name the last table in the join. This makes sense, because if it is a one-to-many join, the record in the last (rightmost) table in the join is the only one to change on every iteration. The first table in the join might be the same for a number of records in the second table.

Remember also that the REPEAT block does not automatically iterate for you, even if you preselect the records. You have to use a FIND statement to move from record to record.

3. Add the following statements to determine whether the **Warehouse** with the highest inventory for the **Item** has a quantity at least 100 greater than the next best **Warehouse**. If so, it retrieves the entry in the list of best **Warehouses** that the code initialized with zeroes at the start of the procedure, increments it, and puts it back in the list, doing the necessary conversions to and from the **INTEGER** data type:

```

IF iWHDty NE 0 AND iWHDty - Bin.Qty > 100 THEN
    DO:
        ASSIGN
            iBestWH = INTEGER(ENTRY(iWHDnum, cBestList))
            iBestWH = iBestWH + 1
            ENTRY(iWHDnum, cBestList) = STRING(iBestWH).
    END.
    ELSE IF iWHDty NE 0 THEN
        LEAVE.
    ASSIGN iWHDty = Bin.Qty
        iWHDnum = Warehouse.WarehouseNum.

```

4. Terminate the REPEAT block and the DO block for each item:

```
END.          /* END REPEAT PRESELECT EACH Bin... */
END.          /* END DO iEntry... */
```

Using multiple weak-scoped references in a single block

If you take a look at the entire D0 block, you can inspect the buffer scoping:

```
DO iEntry . . .:  
    FOR EACH Bin . . .:  
        .  
        .  
        .  
    END.  
    .  
    .  
    .  
    REPEAT PRESELECT EACH Bin. . .: /* Weak-scoped reference to Bin */  
        .  
        .  
        .  
    /* -- Bin scoped to this block too */  
/*/  
    .  
    .  
    END.  
END.
```

The DO block itself doesn't scope any records. The FOR EACH block and the REPEAT PRESELECT EACH block each scope the **Bin** record with a weak scope. This is okay, and the **Bin** buffer is scoped to each of these two blocks in turn.

The final block of code walks through the list of best **Warehouses** for this **Order**'s items. At this point the **cBestList** variable holds a list of numbers for each **Warehouse**. Each number is the count of **Items** where that **Warehouse** has an inventory at least 100 better than the next best **Warehouse**. This block checks whether there's a **Warehouse** that is the best for either all or all but one of the **Items**. If so, you find that **Warehouse** record and save off the **WarehouseName** to pass back. By now all the statements and functions in this block should be familiar to you.



To end the procedure, use the following code:

```
DO iEntry = 1 TO NUM-ENTRIES(cBestList):
  IF INTEGER(ENTRY(iEntry, cBestList)) >= (NUM-ENTRIES(pcItemList) - 1) THEN
    DO:
      FIND Warehouse WHERE Warehouse.WarehouseNum = iEntry.
      pcBestWarehouse = Warehouse.WarehouseName.
      LEAVE.
    END.
  END.
```

This procedure is a little complicated, but these examples show how the different block types interact and how to use some of the built-in functions listed in [Chapter 2, “Using Basic 4GL Constructs.”](#)

Examining the scope with weak and strong references

One final question before you move on: The **Bin** record buffer is scoped to the two blocks inside the main DO block, but at what level is the **Warehouse** record buffer scoped? Look back through the entire procedure to come up with an answer before looking at this excerpt from the listing file:

| File Name | Line | Blk. | Type | Tran | Blk. Label |
|----------------|------|------|-----------|----------------------|------------|
| .\h-BinCheck.p | 0 | | Procedure | No | |
| | | | Buffers: | sports2000.Warehouse | |
| .\h-BinCheck.p | 14 | | For | No | |
| .\h-BinCheck.p | 19 | | Do | No | |
| .\h-BinCheck.p | 21 | | For | No | |
| | | | Buffers: | sports2000.Bin | |
| .\h-BinCheck.p | 22 | | Do | No | |
| .\h-BinCheck.p | 32 | | Repeat | No | |
| | | | Buffers: | sports2000.Bin | |
| .\h-BinCheck.p | 37 | | Do | No | |
| .\h-BinCheck.p | 51 | | Do | No | |
| .\h-BinCheck.p | 52 | | Do | No | |

You see the two blocks where the Bin buffer is scoped. You also see that the **Warehouse** buffer is scoped to the entire procedure (line 0). Why is this?

There are several free references to the **Warehouse** buffer that aren't in blocks that provide record scoping. This includes, among others, the final DO block of the procedure. As a result, Progress raises the scope of the buffer all the way to the procedure itself because there's no other block to scope it to. In your sample procedure, which is only reading records from the database and not updating them, it doesn't make a lot of difference. If the procedure had a transaction that updated the **Warehouse** record, though, you might find that the record and the record lock on it are held much longer than you expected or wanted, resulting in record contention between different users accessing the table at the same time.

What could you do to avoid this? Define a strong scope for the **Warehouse** record wherever it's used.



To define a strong scope for the Warehouse record:

1. First make the first DO block around the FIND Warehouse statement scope the buffer to the block:

```
DO FOR Warehouse:  
  FIND Warehouse WHERE Warehouse.WarehouseNum = Bin.WarehouseNum.
```

2. Press **SHIFT-F2** to do a syntax check. What do you get?



Why did this error happen? You tried to force the scope of the buffer to this block, but the free reference in the DO block at the end of the procedure still forces the scope up to the top, and those two conflict.

3. Change the final DO block to place a strong scope there:

```
DO FOR Warehouse:  
  FIND Warehouse WHERE Warehouse.WarehouseNum = iEntry.
```

Now a syntax check succeeds. If you compile the procedure and get a listing file, you see that the **Warehouse** buffer is scoped all over the place:

| File Name | Line | Blk. | Type | Tran | Blk. Label |
|----------------|------|--------|-----------|-----------------------------|------------|
| .\h-BinCheck.p | 0 | | Procedure | No | |
| .\h-BinCheck.p | 14 | For | | No | |
| | | | Buffers: | sports2000.Warehouse | |
| .\h-BinCheck.p | 19 | Do | | No | |
| .\h-BinCheck.p | 21 | For | | No | |
| | | | Buffers: | sports2000.Bin | |
| .\h-BinCheck.p | 22 | Do | | No | |
| | | | Buffers: | sports2000.Warehouse | |
| .\h-BinCheck.p | 32 | Repeat | | No | |
| | | | Buffers: | sports2000.Warehouse | |
| | | | Buffers: | sports2000.Bin | |
| .\h-BinCheck.p | 37 | Do | | No | |
| .\h-BinCheck.p | 51 | Do | | No | |
| .\h-BinCheck.p | 52 | Do | | No | |
| | | | Buffers: | sports2000.Warehouse | |

Take a look at the scope for each of these blocks:

- The **Warehouse** buffer is now scoped to the very first FOR EACH Warehouse block, with its weak scope. Progress tried to do this before. Because the other references to **Warehouse** forced the scope up to the procedure level, this weak scope disappeared (that's why it's called weak).
- It's scoped to the DO FOR Warehouse block because you added that strong-scoped reference.
- It's also scoped to the REPEAT PRESELECT block. This is another weak scope that didn't hold when the buffer scope was forced to the top.
- It's scoped to the final DO FOR Warehouse block, with its strong-scoped reference you just added.

When you start writing serious procedures that update the database, you'll be a lot more successful if you keep your buffer scope small like this. You should get into the habit now.

Displaying the new fields in the window

You've finished with h-BinCheck.p and you already made the change to h-OrderCalcs.p to run it.

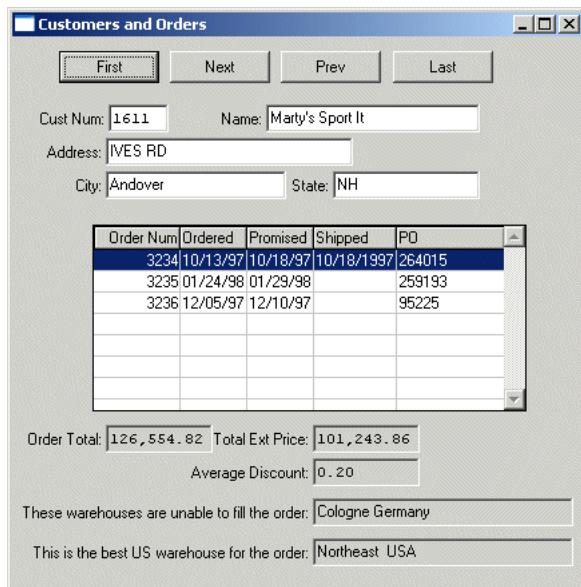


To change the window procedure to accept the new OUTPUT parameters and display them:

1. Add two more fill-in fields to the window and call them **cWorstWH** and **cBestWH**.
2. Give them labels that describe them, such as **These warehouses are unable to fill the order** and **This is the best US warehouse for the order**.
3. Go into their property sheets and make them **Enabled** but **Read-Only**, like the other fill-in fields. The default data type for fill-ins is CHARACTER, so you don't need to change that.
4. Change the RUN statement in the VALUE-CHANGED trigger to include the two new OUTPUT parameters to h-OrderCalcs.p, and to display them:

```
DO:  
    RUN h-OrderCalcs.p  
        (INPUT Order.OrderNum,  
         OUTPUT dTotalPrice, OUTPUT dTotalExt,  
         OUTPUT cWorstWH, OUTPUT cBestWH).  
    dAvgDisc = (dTotalPrice - dTotalExt) / dTotalPrice.  
    DISPLAY dTotalPrice dTotalExt dAvgDisc cWorstWH cBestWH  
        WITH FRAME CustQuery.  
END.
```

5. Run the window. You see values for the **Warehouses** that can't supply the **Items** for the **Order** at all and for the **Warehouse** that is the best source for most of the **Items**, if there is one:



The data in the Sports2000 database isn't too imaginative, so the Cologne **Warehouse** is almost always unable to supply **Items** and the Northeast USA **Warehouse** is almost always the best one. But the values are recalculated every time you select a different **Customer** or a different **Order**.

You've learned a great deal in the last two chapters about how to construct complex procedures in the 4GL. Using the principles of block types, data access statements, and record buffer scoping, you can write procedures that express your application's specialized business logic with a precision and conciseness not possible in any other programming language.

In the next chapter you go back to having some more fun. You'll look at some of the other graphical controls Progress supports and how to manipulate them to get the user interface you want.

Defining Graphical Objects

This chapter returns to the user interface of your application. In [Chapter 4, “Introducing the OpenEdge AppBuilder,”](#) you saw how to lay out fill-in fields and a browse control for data display. Now you’ll learn about other kinds of visual objects and their object definitions, attributes, methods, and events.

This chapter includes the following sections:

- [Types of objects](#)
- [Defining static objects](#)
- [Using and setting object attributes](#)
- [Invoking object methods](#)
- [Instantiating and realizing objects](#)
- [Using object events](#)

Types of objects

The fill-in field is the simplest representation of a single data field. Other ways to represent data fields include:

- **Editors** — For longer text strings.
- **Toggle boxes** — For logical values.
- **Selection lists** — For lists of valid values.
- **Combo boxes** — For a list display that disappears when you’re not using it.
- **Sliders** — For visual display of an integer value within a range.
- **Radio sets** — For presenting a choice among a small set of distinct values.

Among objects that don’t display data values, you have already worked with buttons. In addition, Progress supports rectangles for highlighting and grouping other objects on the screen, and images to display pictures and diagrams. Menus and menu items for a window are other types of visual objects. Windows and frames are objects that serve as containers for other objects within a user interface.

The browse is a major visual object with many capabilities. Because of its special use as a display device for an entire query, a detailed discussion of how to use and customize the browse is postponed until [Chapter 12, “Using the Browse Object,”](#) after you’ve learned more about how to define and use queries.

Various terms describe all these objects in general. The Progress language syntax often uses the term *widget*. In other places you see the word *control*. But there is such variety to the display devices Progress supports that this book refers to them all as *objects*. Sometimes this book refers to them as *basic objects* or *simple objects*, to differentiate them from SmartObjects™.

SmartObjects are procedure-based and have a great deal of additional standard behavior built into them via the 4GL procedures that support them. Progress Dynamics uses them to construct rich interfaces with very little developer coding. The basic objects Progress supports run the gamut from very simple objects (such as rectangles, which are purely decorative) to complex data controls (such as the browse).

For all their variety, all basic objects have the following in common:

- You can define them in a Progress 4GL procedure using forms of the `DEFINE` statement. These are called *static objects*, and it is these that you'll focus on in this chapter.
- You can also create them during program execution using the `CREATE` statement. These are called *dynamic objects*, and you'll learn much more about them in later chapters.
- They can have a *handle* that acts as a pointer to a control structure that describes the object. Since handles are used mostly with dynamic objects, you'll learn more about handles in the chapters on creating and using dynamic objects.
- They respond to various *events* that can come from user actions or can be applied to the object programmatically.
- They support blocks of 4GL code called *triggers* that Progress executes when an associated event occurs.
- They have various *methods* defined for them, which are procedural actions you can invoke in your programs to perform tasks related to the object.

In the next sections you'll learn about object definitions, attributes, and methods.

Defining static objects

You've already seen several uses of the `DEFINE` statement. You have defined program variables for your procedures with the `DEFINE VARIABLE` statement. You have also seen the AppBuilder-generated `DEFINE BUTTON` statements for the four buttons in the **Customers and Orders** window you built in [Chapter 4, “Introducing the OpenEdge AppBuilder.”](#)

These two forms are representative of two basic ways you can use the `DEFINE` statement, one to define a variable for a data value that is viewed in a particular way and the other to define an object that doesn't represent a data value. In this section, you'll look at the two forms in a general way. Then in the next chapter, you'll look at specific types of objects you define in each way.

Using the VIEW-AS phrase for data representation objects

Many visual objects represent a single data value. This value can be a field from a database table or it can be a program variable. In both these cases, the default visual representation of the field is normally a fill-in field.

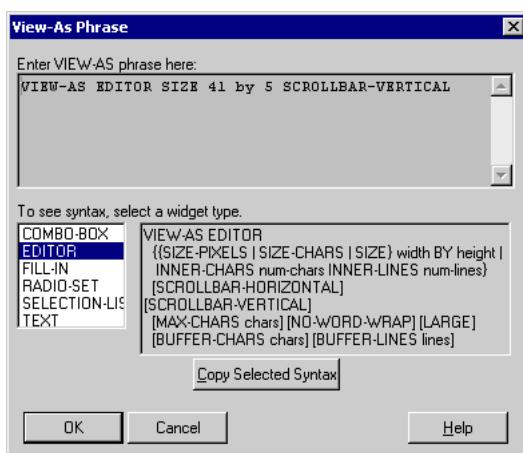
Why specify *normally*? You can define a visualization for a field when you create it as part of a database table definition in the Data Dictionary.



To see an example of how you define a field's visualization:

1. From the AppBuilder menu, select **Tools→Data Dictionary**.
2. From the **Sports2000** database, select the **Item** table. This table holds information about the different sports-related items that customers can order from your business.
3. Choose the **Fields** icon and select the **CatDescription** field from the **Fields** list. This field contains the full description of an **Item** in the catalog.
4. Choose the **Field Properties** button.
5. In the **Field Properties** dialog box, choose the **View-As** button.

A dialog box appears where you can define a default visualization for a field if you want it to be an object other than a fill-in. In this case, there is a definition for the field:



The **CatDescription** field is defined to be viewed as an editor object, with a size of 41 characters by 5 lines and a vertical scrollbar. Whenever you select the **CatDescription** field and drop it onto a frame in the AppBuilder, Progress automatically visualizes it as an editor of this description. If you define a variable to be **LIKE** the **CatDescription** field, it inherits these visual attributes along with the rest of the field description. You can change these attributes in your programs just as you can change the display for any other field, but the default is always to view it as an editor.

6. **Cancel** your way out of the dialog boxes and the Data Dictionary to return to the AppBuilder main window.

For all fields where there is no specific **VIEW-AS** definition in the Data Dictionary, the default visualization is a fill-in field. If you want another visualization of the field, you use a **VIEW-AS** phrase as part of the object definition. There are several variations on this.

In a **DEFINE VARIABLE** statement, you can append the **VIEW-AS** phrase to the definition:

```
DEFINE VARIABLE name AS datatype VIEW-AS display-type [options ].
```

The *options* are attributes for that visual type that you can choose, such as the **SIZE** of the editor and the **SCROLLBAR-VERTICAL** keyword.

If you're not defining a variable but simply placing a database field or other field into a frame, then you append the **VIEW-AS** phrase to the name of the field in the **DEFINE FRAME** statement, along with whatever options apply. Here's an example from the frame definition the AppBuilder generates if you drop the **Customer Comments** field onto a window and define it as an editor, as you'll do later in the next chapter:

```
DEFINE FRAME CustQuery
.
.
.
Customer.Comments AT ROW 5.29 COL 76 NO-LABEL
    VIEW-AS EDITOR SCROLLBAR-VERTICAL
    SIZE 36 BY 3.14
.
.
.
```

Defining objects that don't represent data values

Objects that don't represent single data values use a form of the DEFINE statement that names the object type directly. You have seen the DEFINE BUTTON, DEFINE BROWSE, and DEFINE FRAME statements already in the AppBuilder code for *CustOrders.w*. These are all examples of this form. Each statement type accepts the same kinds of options that the VIEW-AS phrase of the DEFINE VARIABLE statement does, with the options list specialized for each object type. You'll look at some of these object types in the next chapter.

Using and setting object attributes

Each different type of object has its own attributes, which represent some aspect or capability of the object that you can set and query. Attributes fall into several basic categories:

- **Geometry** — The size and location of the object in a frame or window.
- **Appearance** — The color and font, or whether the object has optional features such as a scrollbar, or whether it is visible or not, for example.
- **Data management** — The initial value and whether the value is undone as part of a transaction, for example.
- **Relationships to other objects** — The parent or sibling, for example.
- **Identifying characteristics of an object** — Its name, for example.

You can specify initial values for many object attributes when you define the object. The 4GL supports a large number of keywords that represent these various attributes. In some cases, a single keyword represents the attribute, such as SCROLLBAR-VERTICAL for an editor. In other cases, the attribute keyword takes one or more arguments, in the form of other values that follow the keyword in the DEFINE statement, such as ROW 5.

You can also query most attribute values from within the procedure that defines the object. To retrieve the value of an attribute, you use the form:

```
object-name:attribute-name [IN { FRAME | MENU | SUB-MENU } name ]
```

Do not use spaces on either side of the colon between the *object-name* and the *attribute-name*. Each attribute has an appropriate data type, such as DECIMAL for the ROW attribute or LOGICAL for the HIDDEN attribute. You can use an attribute value anywhere in an expression or assignment where you would use any other value. If the attribute reference is not unambiguous, you can provide context for it in the reference by qualifying it with the name of the frame, menu, or submenu it appears in. The default is the most recently defined container whose description includes the object.

For example, this attribute reference checks whether the First button is hidden:

```
IF BtnFirst:HIDDEN THEN . . .
```

Changing attribute values

You can also change many attribute values at run time, even those for static objects. You simply place the attribute reference on the left side of an assignment. The documentation of the individual attributes in the final volume of *OpenEdge Development: Progress 4GL Reference*, as well as the online help, tells you whether you can set them. Generally, at run time you cannot change attributes that are part of the definition of an object, such as its initial value or its display type. But many attributes can change during program execution, such as attributes that define whether an object is hidden or visible and whether it is enabled or disabled. Even some basic display attributes such as a field's font or a button's label can change at run time to give greater flexibility to your application's interface. If you try to set an attribute for an object that is not settable, you get a compile-time error telling you so.

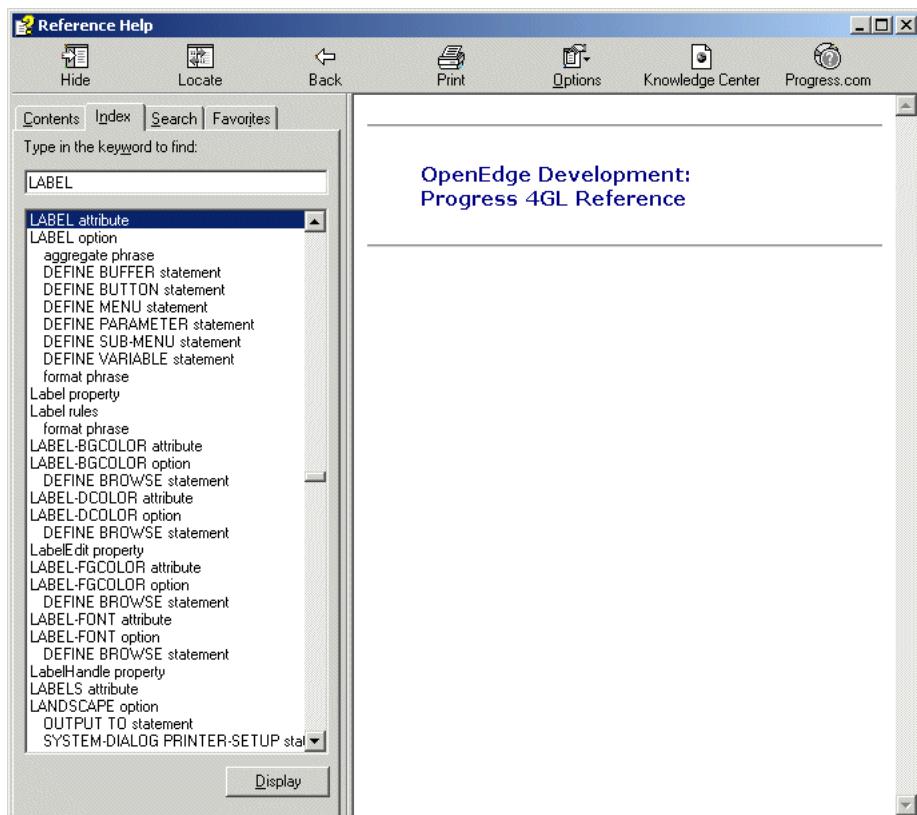


To get a full description of any attribute from online help, including what object types it applies to and whether it is readable, writable, or both:

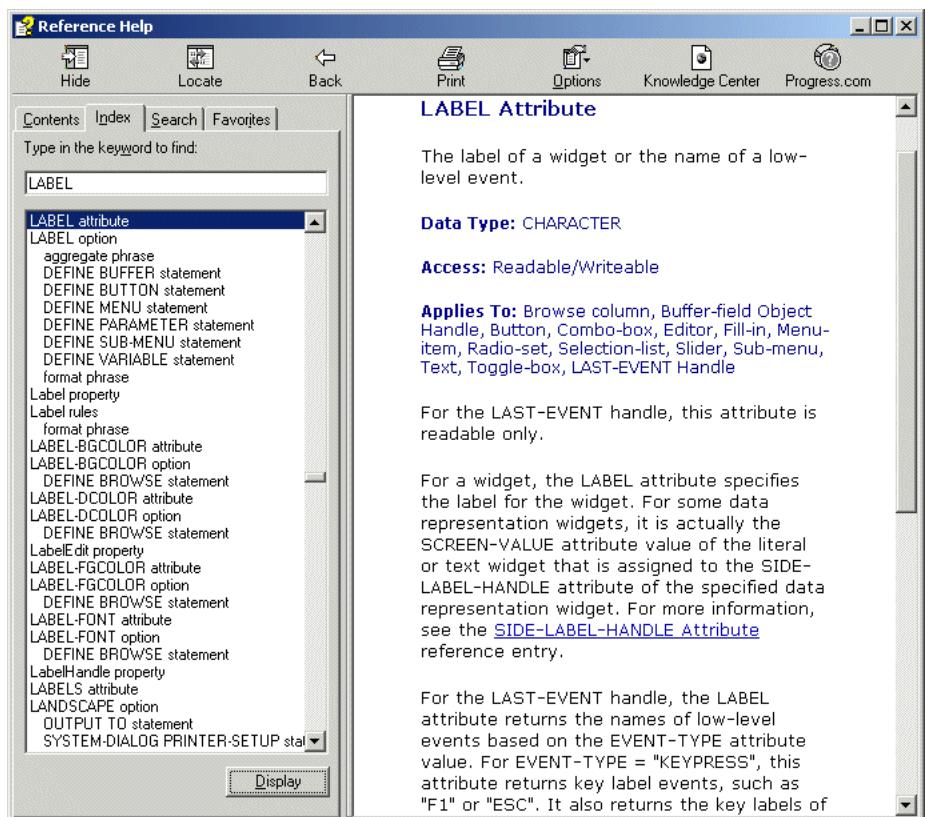
1. Highlight the attribute keyword in any Progress editor window:



2. Press F1. The online help for the selected entry appears:



3. When you choose the **DISPLAY** button, the help text appears:



Common attribute values for visual objects

There are literally hundreds of different attributes, many of which apply only to a single object type. But several basic ones that you will likely use most often apply to most or all visual objects. This section summarizes a few of these.

Geometry attributes

These attributes affect the size and location of the object in the frame or window:

- **ROW, COLUMN** — These DECIMAL attributes are the character positions of the object within its container. For objects in a frame, the values represent the position within the frame and not within the frame's window. The frame has its own position within its window, and the window has its own position within the display device. When you lay out objects in the AppBuilder, it generates code to position the objects using ROW and COLUMN coordinates. Alternately, you can use pixel values for positions. Generally, character positions are more portable and flexible if the font or display device changes.
- **X, Y** — These INTEGER attributes are equivalent to ROW and COLUMN but measured in pixels.
- **HEIGHT-CHARS, WIDTH-CHARS** — These DECIMAL attributes are the height and width of the object in character units.
- **HEIGHT-PIXELS, WIDTH-PIXELS** — These INTEGER attributes are the height and width of the object in pixels.

Appearance attributes

These attributes affect the appearance of the object:

- **HIDDEN** — If this LOGICAL attribute is true, then the object is hidden and does not appear even when its container is displayed. If it is false, then the object does appear when its container is displayed. If you set the HIDDEN attribute of a container object, such as a window or frame, to true, then the container and all the objects in it are hidden. If you set it to false for a container, then the container and any contained objects that aren't themselves hidden appear. This is an attribute you can set only at run time. The definition of an object cannot describe it as initially hidden.

- **VISIBLE** — This LOGICAL attribute is not simply the opposite of HIDDEN. Its relation to HIDDEN can be somewhat confusing to understand. Generally, you use it much less than the HIDDEN attribute. Setting the VISIBLE attribute of an object to true *forces* it to be viewed. For example, setting the VISIBLE attribute of a field-level object in a window to true forces the window to be displayed even if it was previously hidden. By contrast, setting the HIDDEN attribute to false doesn't force the container to be viewed. You can read all the details of the effects of the VISIBLE attribute in the online help.
- **SENSITIVE** — This LOGICAL attribute determines whether an object is enabled for input or not. Its use is parallel to the ENABLE verb. That is, executing an ENABLE statement for an object is the same as setting its SENSITIVE attribute to true. Similarly, executing a DISABLE statement for an object is the same as setting its SENSITIVE attribute to false. Like the HIDDEN attribute, you can set the SENSITIVE attribute only at run time. This means that if you check the **Enable** toggle box or the **Display** toggle box on or off in the AppBuilder property sheet for an object when you are building a screen, you do not change the DEFINE statement the AppBuilder generates for the object. Rather you change the ENABLE and DISPLAY statements the AppBuilder generates that execute when the window is initialized.
- **READ-ONLY** — This LOGICAL attribute applies to data-representation objects and prevents the user from modifying the field value. Sometimes you might want to combine setting the **Enable** toggle box in a field's property sheet with setting the READ-ONLY attribute to true. This gives a fill-in field some of the appearance of an enabled field (with its characteristic box outline, which can improve readability), but prevents the user from changing it. The **CustOrders** window uses this form for its **Customer** fill-ins.
- **HELP** — This is the help text to display when the object is selected.

- **TOOLTIP** — This is the text to display when the user hovers the mouse over the object. For data-representation objects, you can initialize the ToolTip text in the frame definition for the object, such as in this excerpt from the frame definition for the **CustOrders** window:

```
DEFINE FRAME CustQuery
.
.
.

Customer.City AT ROW 7.19 COL 13 COLON-ALIGNED
VIEW-AS FILL-IN
SIZE 27 BY 1 TOOLTIP "Enter the City"
.
```

- For other types of objects, you can specify the ToolTip text as part of the object definition, as in this button definition:

```
DEFINE BUTTON BtnFirst
LABEL "First"
SIZE 15 BY 1.14 TOOLTIP "Press this to see the first Customer.".
```

- **LABEL** — This CHARACTER attribute is the label of the field or button.
- **SELECTABLE** — You can set up most visual objects for direct manipulation. This means that the user can actually select, move, and resize the object at run time just as you can move and resize objects in a design window in the AppBuilder. The AppBuilder uses these attributes to provide you with the behavior you see in a design window, where you can drag objects around to where you want them. The **SELECTABLE** attribute is a LOGICAL value which, if true, allows the user to select the object by clicking on it with the mouse. It then sprouts the characteristic resize handles around the object border that let the user size or move it.
- **RESIZABLE** — You can set this LOGICAL attribute to true to let a user change the size of an object at run time. You must also set the **SELECTABLE** attribute to true to provide the resize handles.
- **MOVABLE** — You can set this LOGICAL attribute to true to let a user move an object at run time. You do not need to set the **SELECTABLE** attribute to make an object movable. The user can move the object without using its resize handles. However, it is considered a more standard user provision to set **SELECTABLE** along with **MOVABLE**.

Data management attributes

These attributes affect how to manage data associated with the object:

- **SCREEN-VALUE** — There is a special screen buffer that holds the displayed values of all data-representation objects in a frame. This buffer is separate from the underlying database record buffer for database fields and from the buffer where variable values are held. If a user enters a value into a field on the screen, that value is not assigned to the underlying record buffer until you execute an ASSIGN statement for the field. In the meantime, the input value is held only in the screen buffer. You can retrieve a value from the screen buffer using the SCREEN-VALUE attribute, which is a CHARACTER value representing the value as it appears on the screen, or as it would appear on the screen in a fill-in field. You can also set the SCREEN-VALUE attribute of an object to display a value without setting the underlying record value. It is important to remember that the value of the SCREEN-VALUE attribute is always the formatted value as it appears in the user interface. If this contains special format mask characters (such as commas and decimal points in a decimal value, for example), then any comparisons you do against this string must include those format characters. If this presents problems, you must assign the screen value to its underlying record buffer and reference the BUFFER-VALUE of the field to access it in its native data type and without format characters.
- **INITIAL** — This attribute holds the initial value of a data-representation object. You can set it only in the DEFINE statement.

Relationship attributes

These attributes affect how the object interacts with other objects:

- **FRAME-NAME** — This CHARACTER attribute holds the name of the container frame the object is in.
- **FRAME** — This HANDLE attribute holds the object handle of the container frame the object is in. In later chapters, you learn how to traverse from one handle to another to locate an object or to locate all objects that are in a container.
- **WINDOW** — This HANDLE attribute holds the object handle of the container window the object is in.

Identifying attributes

These attributes identify characteristics of the object:

- **NAME** — This CHARACTER attribute holds the name of the object.
- **PRIVATE-DATA** — This CHARACTER attribute lets you associate any free-form text with an object, which can help program logic that determine how to identify or treat the object at run time.

You can see most of the object attributes that you can set for an object type in the AppBuilder property sheet for the object, such as the one shown in [Figure 8–1](#) for the **Customer.Name** fill-in field in the **CustOrders** window.

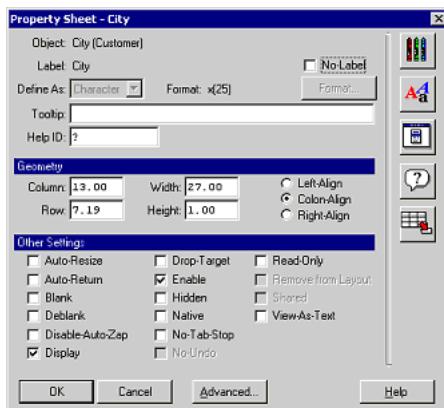


Figure 8–1: Property sheet for a fill-in field

The **Advanced** button takes you to the dialog box shown in [Figure 8–2](#). These are attributes that you might use less often, some of which are mentioned in this chapter.



Figure 8–2: Advanced Properties dialog box

Certain property sheet settings, such as **Display** and **Enable**, affect AppBuilder-generated executable statements and preprocessor values rather than the DEFINE statement for the object itself. You've seen the effect of the **Display** and **Enable** settings already in the code for the AppBuilder-generated enable_UI procedure:

```
IF AVAILABLE Customer THEN
    DISPLAY Customer.CustNum Customer.Name Customer.Address Customer.City
        Customer.State
    WITH FRAME CustQuery IN WINDOW CustWin.
ENABLE BtnFirst BtnNext BtnPrev BtnLast Customer.CustNum Customer.Name
    Customer.Address Customer.City Customer.State OrderBrowse
dTotalPrice
    dTotalExt dAvgDisc cWorstWH cBestWH
    WITH FRAME CustQuery IN WINDOW CustWin.
```

The AppBuilder also keeps the list of displayed and enabled objects in preprocessor values:

```
&Scoped-define FIELDS-IN-QUERY-CustQuery Customer.CustNum Customer.Name ~
Customer.Address Customer.City Customer.State
&Scoped-define ENABLED-FIELDS-IN-QUERY-CustQuery Customer.CustNum ~
Customer.Name Customer.Address Customer.City Customer.State
```

Other property sheet settings for attributes that you cannot set as part of the object definition are set in a series of ASSIGN statements the AppBuilder generates, such as these for the READ-ONLY, PRIVATE-DATA, SELECTABLE, MOVABLE, and RESIZABLE attributes in this example:

```
ASSIGN
  BtnFirst:PRIVATE-DATA IN FRAME CustQuery      =
    "Private data for the first button.".

ASSIGN
  cBestWH:READ-ONLY IN FRAME CustQuery          = TRUE.

ASSIGN
  cWorstWH:SELECTABLE IN FRAME CustQuery        = TRUE
  cWorstWH:MOVABLE IN FRAME CustQuery           = TRUE
  cWorstWH:READ-ONLY IN FRAME CustQuery         = TRUE
  cWorstWH:RESIZABLE IN FRAME CustQuery         = TRUE.
```

As you develop your own applications, you need to keep in mind which attributes you can define in a DEFINE statement, which in a frame definition, and which only in executable statements at run time.

Invoking object methods

In addition to attributes, some objects support *methods*. A method is an operation that performs a specific action related to an object. You can think of methods as being very much like the built-in functions the language supports. The methods are also identified by keywords that you use in 4GL syntax following an object reference, which can be the object name or handle followed by a colon, just as for attributes:

```
object-reference:method ( optional-arguments ) [ IN FRAME frame-name ]
```

Methods typically take one or more arguments, defined in a comma-separated list in parentheses following the method name.

For example, in the next chapter you'll learn how to view a text field as an editor, with multiple lines and scrollbars and so forth. There's an editor method, called READ-FILE, that you can use to open and read an operating system file into an editor, just as the Progress Procedure Editor does. READ-FILE takes a single argument, the name of the file to read. So this sample syntax reads a file into an editor called cEditor:

```
cEditor:READ-FILE('myTextFile.txt')
```

Methods always return a value, just as built-in functions do. Generally, that value is a LOGICAL indicating whether the operation succeeded or not (with a TRUE value indicating success). You can assign the return value to a variable or field in an assignment statement:

```
lSuccess = cEditor:READ-FILE('myTextFile.txt').
```

The initial letter l indicates that this is a logical variable.

You can also ignore the return value (as you can with any function) and simply treat the method reference as a statement of its own:

```
cEditor:READ-FILE('myTextFile.txt').
```

Some methods return more meaningful values that you would normally not ignore. Indeed, some methods exist solely to return a meaningful value. You can think of these methods as being similar to attributes. However, because an input parameter is required and attribute references cannot take parameters, you use a method instead to retrieve the return value. For example, the following code sample uses a Progress system-wide object called FONT-TABLE to calculate the width of a button label in the current font. It then uses this value to calculate the required width of a frame that has five buttons. Because the button label must be passed in to the operation, the syntax must be defined as a method (in this case called GET-TEXT-WIDTH-CHARS) rather than an attribute (which might have been called TEXT-WIDTH-CHARS):

```
DEFINE VARIABLE dWidth AS DECIMAL      NO-UNDO.  
  
/* You can specify the LABEL attribute for the button in its definition. */  
DEFINE BUTTON bChoose LABEL "Choose me".  
  
/* You must put the button in a frame because otherwise Progress does not  
compile the method reference. */  
DEFINE FRAME aFrame bChoose.  
  
/* Here you use the method return value in a large expression. */  
dWidth = 5 * (FONT-TABLE:GET-TEXT-WIDTH-CHARS(bChoose:LABEL) + 1) + 2.
```

As you can see from this example, you can reference the method within an expression anywhere another value could appear.

Instantiating and realizing objects

There are a couple of important principles that this chapter has been taking for granted in the discussion of objects. In this section, you look at exactly how objects are instantiated (or created) at run time, how they are realized (or viewed), and what their scope is.

Instantiating objects in a container

When you define a frame you give it a name. Progress creates only one instance of a named frame within a procedure, so that frame name identifies a unique object. Progress instantiates the frame when you first use it by executing an I/O-oriented statement involving the frame, such as a DISPLAY or VIEW statement. The same is true for dialog boxes (which are a type of frame), and for menus and their submenus.

Other objects, however, might not be uniquely identified by their names. When Progress encounters, for example, a `DEFINE BUTTON` statement, or a `DEFINE VARIABLE` statement with a `VIEW-AS` phrase that defines a particular type of visual object, it registers the description of the object but it does not actually create it. Progress can create the object only when you associate it with a container frame or window that has itself been instantiated. Then Progress can identify a unique instance of the object *in that container*, and create it as part of the container.

This means that you could create multiple instances of an object in different containers. For example, this code fragment describes a button and creates two instances of it in two different frames:

```
DEFINE BUTTON bBothFrames.  
ENABLE bBothFrames WITH FRAME F1.  
ENABLE bBothFrames WITH FRAME F2.
```

Qualifying object references to specify a unique identity

The two `ENABLE` statements for different frames are two distinct instances of a single button definition. Each one is said to have a *unique identity*, which is manifested in its object handle. Using either the handle or a frame qualifier that identifies which instance of the object you're referring to, you can set and retrieve attribute values for the object instances independently of one another.

Recall that Progress associates an object reference by default with the most recently defined container for that object. It's important to keep this in mind when you're writing your applications. As with all defaults, you are much better off being explicit about object references than taking your chances with how the defaults work in your case.

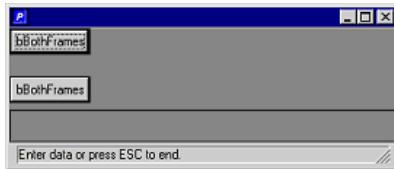
Here are a few examples to illustrate this point. If you run the code fragment above that defines and enables button `bBothFrames`, it terminates immediately because there is no statement to make it wait for user input.

**To test the effects of how Progress defines a unique identity for an object:**

1. Add a statement to make the procedure and its frame stay active while you observe the behavior of the two buttons:

```
DEFINE BUTTON bBothFrames.  
  
ENABLE bBothFrames WITH FRAME F1.  
ENABLE bBothFrames WITH FRAME F2.  
  
WAIT-FOR CLOSE OF THIS-PROCEDURE.
```

2. Run this code to see the two instances of the same button, each in its own frame:



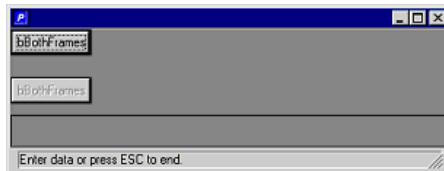
You can't really see the frames **F1** and **F2** because Progress has made them just big enough to hold the buttons. You can see that both buttons have the same default label (the button name), and they are both enabled.

3. Add an attribute reference to disable the button:

```
DEFINE BUTTON bBothFrames.  
  
ENABLE bBothFrames WITH FRAME F1.  
ENABLE bBothFrames WITH FRAME F2.  
  
bBothFrames:SENSITIVE = NO.  
  
WAIT-FOR CLOSE OF THIS-PROCEDURE.
```

The question is, just which instance of the button are you disabling?

4. Run the procedure again to see which button is disabled:

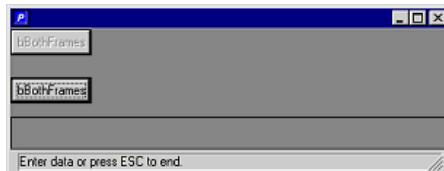


It's the second one, because that's the most recent reference to the button in a frame. To change that behavior (or to make it explicit without relying on the default), you can use the frame qualifier.

5. Add this frame phrase to the statement to identify the first frame:

```
DEFINE BUTTON bBothFrames.  
  
ENABLE bBothFrames WITH FRAME F1.  
ENABLE bBothFrames WITH FRAME F2.  
  
bBothFrames:SENSITIVE IN FRAME F1 = NO.  
  
WAIT-FOR CLOSE OF THIS-PROCEDURE.
```

6. Run the procedure again to see the result:



Now it is the first instance of the button that is disabled.

- To make sure it's clear which of these frames is which, you can put the frame name into the button. Make these changes to the procedure:

```
DEFINE BUTTON bBothFrames.

ENABLE bBothFrames WITH FRAME F1.
ENABLE bBothFrames WITH FRAME F2.

bBothFrames:SENSITIVE IN FRAME F1 = NO.
bBothFrames:LABEL IN FRAME F1 =
  "Frame " + bBothFrames:FRAME-NAME IN FRAME F1 .

bBothFrames:LABEL = "Frame " + bBothFrames:FRAME-NAME.
WAIT-FOR CLOSE OF THIS-PROCEDURE.
```

Take a look at the new code you added. The first new statement identifies the button instance in frame F1 explicitly: **bBothFrames:LABEL IN FRAME F1**. It sets the button label attribute to the text Frame plus the value of the FRAME-NAME attribute for the button. The second reference to the button in that statement likewise has to be qualified by the **IN FRAME** phrase to get the right one: **bBothFrames:FRAME-NAME IN FRAME F1**.

By contrast, the second new statement just takes the defaults: **bBothFrames:LABEL = "Frame " + bBothFrames:FRAME-NAME**. Because frame F2 is the most recently referenced frame for the button, the defaults use that frame. [Figure 8–3](#) shows the result when you rerun the procedure.

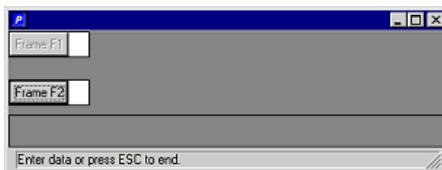


Figure 8–3: Frames for buttons

In [Figure 8–3](#) you can see the frames Progress created for the buttons. At first they were the same size as the buttons. Your new statements changed the label of each button, and Progress automatically reduced the button size accordingly. The remaining portion of the frame appears as a white space after each button.

The lesson of this little exercise is that you must always be aware that every use of an object in a different container is a distinct instance of the object, and you should always make your object references as explicit as needed to be sure that you do not simply get a default behavior that isn't what you want.

Realizing and derealizing objects

When you define an object, Progress creates an internal data structure associated with that object. Before the object can be displayed, the window system must also create a data structure for the object. When this second data structure has been created, then the object is *realized*.

You can modify some object attributes at any time. Others are initially modifiable, but become fixed once the object is realized. In addition, some attributes *must* be set before realization takes place. For this reason, it is important to have an understanding of when objects are realized.

In general, an object is realized when the application needs to make it visible on the screen. Therefore, field-level objects, buttons, and the like are typically realized when their containers are realized or made visible, and conversely, a container object is realized when a statement forces any of its contained objects to be realized or made visible.

In addition, an object is realized if a statement references any method of the object because methods operate on the realized instance object. Also, some attribute values, such as those that reference an object's size, can only be determined if the object has been realized, so a reference to any of those attributes forces the object to be realized if it is not already. Examples of this are the MODIFIED attribute of an editor, which is true if the text in the editor has been changed since it was initialized, and the MAX-HEIGHT or MAX-WIDTH attributes of a frame which cannot be calculated without realizing the frame.

A static object is *derealized*, or destroyed, when it goes out of scope. Generally, the scope of a defined object is the procedure in which it's defined. Field-level objects are derealized when their frame is derealized. In turn, a frame is derealized when its containing window is deleted.

Using object events

Graphical applications are often referred to as *event-driven* applications. Unlike the hierarchical, menu-driven applications typical in a character terminal environment, graphical applications put the user more in control of the sequence of events. Using the mouse, menus, and active controls (like buttons on the screen that can respond to user actions), the user can navigate through the application with much more flexibility than in most older applications.

But this flexibility does not happen automatically. You, the developer, must build it into the application by programming procedures or blocks of code that respond to user actions. These are called *triggers*. The construction of the user interface of an application around blocks of trigger code is the single most fundamental difference in the architecture of event-driven applications.

You've already seen some examples of the language constructs the Progress 4GL uses to establish triggers and respond to events when you looked through and extended the AppBuilder-generated code for the `Cust0rder`s window in [Chapter 2, “Using Basic 4GL Constructs.”](#) In this section, you'll look at those constructs in a little more detail.

User interface events

Each visual object type supports a set of user interface events. You can see a list of all these events in the AppBuilder if you go into the triggers section of the **Section Editor** for an object and then choose the **New** button. Each object type first supports a list of *common events*, such as the events for a button, shown in [Figure 8–4](#).

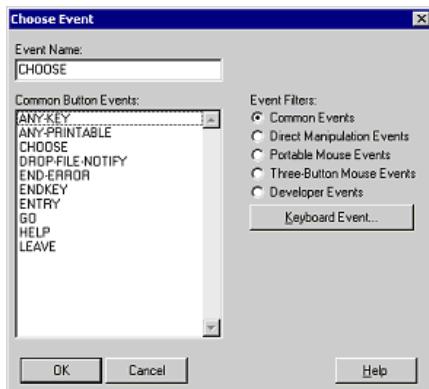


Figure 8–4: Common button events

They are called common events simply because they are the events most commonly associated with an object. Each object has its own set of these events but there is a lot of overlap. For example, any object that can be part of the tab order of a frame has the `ENTRY` event, which fires when the user tabs into the object, and the `LEAVE` event, which fires when the user tabs out of that object. Any object that can be the target of keystrokes (even an object like a button that doesn't use those keystrokes to set a data value) can use the `ANY-KEY` or `ANY-PRINTABLE` events to respond to them.

In some cases, the most common event is one that is distinctive for that object. For example, the whole purpose of a button is for someone to choose it and to have an action result. Therefore, the button supports the `CHOOSE` event, which is supported only by buttons and menu items. This is the default event that comes up in the **Section Editor** when you go into the triggers section for it. Data-representation objects, which can have actual values, support the `VALUE-CHANGED` event, which fires when the user enters a new value for the object.

To see a complete description of all the common events, see the [high-level widget events](#) topic in the online Help.

Objects support direct manipulation events, such as CHOOSE shown in [Figure 8–5](#), which fire when the user performs an action (typically using the mouse) that involves selecting an object and possibly moving or resizing it. Some of these events are the result of making the object SELECTABLE, RESIZABLE, or MOVABLE.

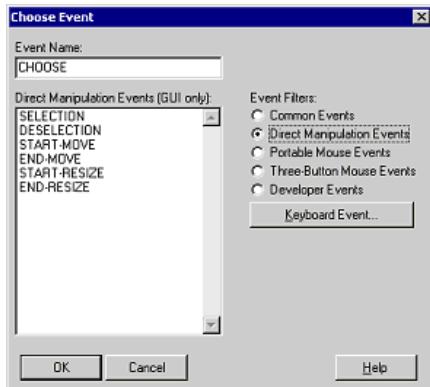


Figure 8–5: Direct manipulation events

There is a whole host of events associated with all of the possible ways the user can make a selection with either a standard two-button mouse or a three-button mouse, such as the *portable mouse events* shown in [Figure 8–6](#), that can map to either type of mouse.

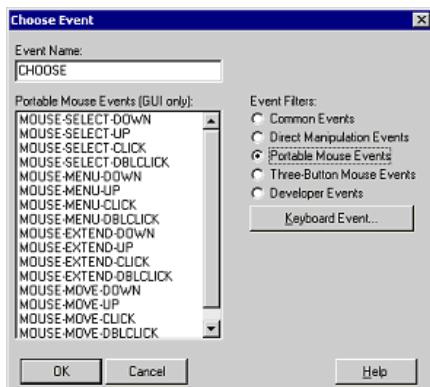


Figure 8–6: Portable mouse events

Progress also supports a set of ten miscellaneous events that have no standard meaning or action, but which are intended to let you associate a trigger with some event that only happens programmatically using the APPLY statement, and which has nothing to do with a particular user action. These are called the *developer events* and are numbered U1 through U10, as shown in Figure 8–7.

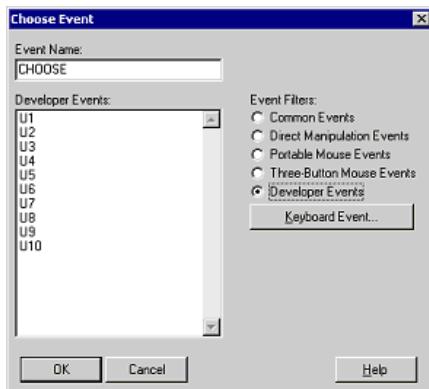


Figure 8–7: Developer events

These developer events have no built-in significance, but allow you to define a block of code to execute. For example, you can specify ON U1 OF an object and then programmatically APPLY “U1” to the object to execute the code.

Finally, you can associate a trigger with virtually any keyboard keystroke combination, by choosing the **Keyboard Event** button and then typing the keystroke combination, such as **CTRL-X** shown in Figure 8–8.



Figure 8–8: Keyboard Event dialog box

Defining triggers

The most basic way to define a trigger is to put the trigger definition directly into the object definition:

```
DEFINE BUTTON BtnQuit LABEL "Quit"
  TRIGGERS:
    ON CHOOSE
      QUIT.
  END TRIGGERS.
```

The TRIGGERS block in a DEFINE statement can contain one or more individual trigger definitions, each starting with an ON phrase naming the event followed by a single statement or a DO-END block of statements. This form is called a *definitional trigger*.

If you use the AppBuilder to create your application procedures, it always creates separate blocks of executable code that attach triggers to objects at run time. This form is called a *run-time trigger*. There is no inherent advantage to one form over the other. Either way, the code in the trigger block is compiled and turned into r-code along with the rest of the procedure. You should use the AppBuilder to organize your trigger blocks to provide a more readable structure to your procedures.

The form of the trigger block for a run-time trigger names both the event and the object it applies to:

```
ON event-name [ , . . . ] OF object-name [ , . . . ] statements
```

Either the *event-name* or the *object-name* (or both) can be a comma-separated list. The *statements* can be either a single statement or a DO-END block of statements.

Setting up triggers to be executed

A trigger is executed if the object it applies to is enabled (sensitive), and if the trigger is currently *active* and *in scope*.

The trigger is active if the statement that defines it has been executed. As discussed earlier in the book, Progress processes statements in a procedure in the order it encounters them in. Thus, the definition of an object must come before the block of code that defines a trigger for the object. Otherwise, Progress won't recognize the object reference. And the trigger definition must come before the user receives control and can perform the action that would fire the event and run the trigger.

The scope of a definitional trigger is the scope of the object it's defined for. The scope of a run-time trigger is the nearest containing procedure or trigger block where it is defined. So if a trigger definition is inside an internal procedure, then its scope is limited to that internal procedure. If it's outside of any internal procedure, its scope is the entire external procedure.

The AppBuilder places all trigger definitions toward the top of the procedure, following the Definitions section of the procedure but before the main block and all internal procedures. In this way the trigger blocks are scoped to the entire procedure and they are made active before any user actions that invoke them can occur.



To build a very simple example procedure to demonstrate some of the rules of run-time trigger processing in Progress:

1. Define a button and give it an initial label, then define an INTEGER variable as a counter:

```
DEFINE BUTTON bButton LABEL "Initial Label".  
DEFINE VARIABLE iCount AS INTEGER      NO-UNDO.
```

2. Add a statement to enable the button in a frame. As you learned earlier, this causes both the button and its frame to be instantiated and realized:

```
ENABLE bButton WITH FRAME fFrame.
```

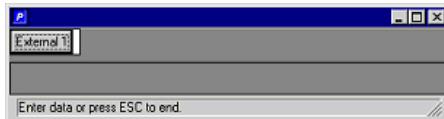
3. Define a run-time trigger for the button that changes its label so that you can see that the trigger fired:

```
ON CHOOSE OF bButton IN FRAME fFrame  
DO:  
    iCount = iCount + 1.  
    bButton:LABEL IN FRAME fFrame = "External " + STRING(iCount).  
END.
```

4. Create a WAIT-FOR statement that blocks the termination of the procedure and allows the user to choose the button:

```
WAIT-FOR CLOSE OF THIS-PROCEDURE.
```

5. Run the procedure. As you would expect, the button's label is changed when you choose the button:



The ON CHOOSE trigger you defined is scoped to the entire procedure and executes each time the button is pressed.

- To define another trigger for the button in an internal procedure, make these changes to the procedure:

```

DEFINE BUTTON bButton LABEL "Initial Label".
DEFINE VARIABLE iCount AS INTEGER      NO-UNDO.

ENABLE bButton WITH FRAME fFrame.

ON CHOOSE OF bButton IN FRAME fFrame
DO:
  iCount = iCount + 1.
  bButton:LABEL IN FRAME fFrame = "External " + STRING(iCount).
END.

RUN ChooseProc.

WAIT-FOR CLOSE OF THIS-PROCEDURE.

PROCEDURE ChooseProc.
  ON CHOOSE OF bButton IN FRAME fFrame
  DO:
    iCount = iCount + 1.
    bButton:LABEL In FRAME fFrame = "Internal " + STRING(iCount).
  END.
END.

```

Before the WAIT-FOR statement the code runs the internal procedure ChooseProc. This procedure defines a different trigger for the same button, which identifies the trigger with the label Internal1. The second trigger definition replaces the first one within the ChooseProc procedure.

Note that the scope of the button is the entire external procedure because it is defined at that level. The scope of the variable iCount is also the external procedure for the same reason. But what is the scope of the second trigger definition you just added?

It is scoped only to the internal procedure where it is defined. So what happens when you run this version of the procedure? Before you run it, walk through the code in your head to decide what happens.

The external procedure defines a trigger for the button. It then runs an internal procedure that defines a different trigger. That internal procedure then exits, without giving the user any chance to use the new trigger, and its trigger goes out of scope. So what happens when the user chooses the button?

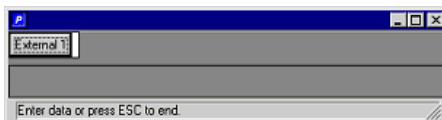


Figure 8–9: Result of trigger example procedure

Figure 8–9 shows that Progress reverts to the original trigger. This example shows you that Progress effectively keeps a stack of trigger definitions. If a later definition goes out of scope, Progress reverts to the trigger definition that is now back in scope (if there is one).

So how would you see the effect of the internal procedure trigger? You can place a WAIT-FOR statement inside the internal procedure to force Progress to wait until the user chooses the button.



To try this, add this statement to the end of the ChooseProc procedure:

```
WAIT-FOR CHOOSE OF bButton.
```

Note that you can wait for any event, not just the close of the procedure or its window. This statement waits until the user chooses the button exactly once. Then the WAIT-FOR is satisfied, the internal procedure exits, and the first trigger takes over.

- To see the result of each button press, run the procedure again. Figure 8–10 shows the result of the first four button presses.

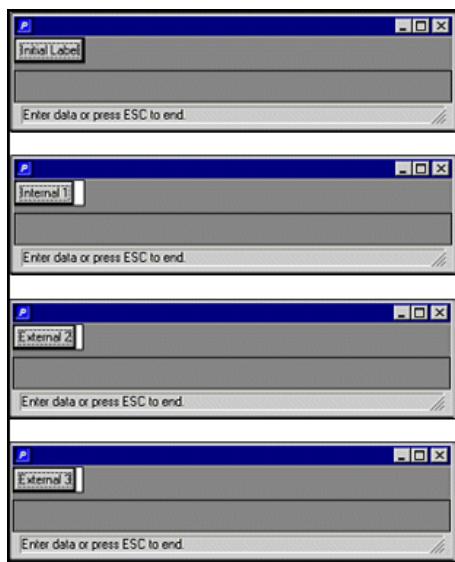


Figure 8–10: Results of running the ChooseProc procedure

Making a trigger persistent

Adding the second WAIT-FOR statement is very awkward, and it is pretty close to an absolute rule in Progress that your entire *application*, not just a single procedure, should have only a single WAIT-FOR statement. This rule is summarized in the saying: “One world, one WAIT-FOR.” Multiple WAIT-FOR statements can easily get tangled up in each other if they are not satisfied in the exact reverse order from the order they are defined in, and this can result in a WAIT-FOR that doesn’t terminate properly.

So how else do you establish a trigger inside an internal procedure or a trigger block, or for that matter inside another external procedure that you call, and have that trigger persist beyond the scope of its declaration?

Progress provides a special statement to let you do this:

```
PERSISTENT RUN procedure-name [ ( input-parameters ) ].
```

The persistent trigger definition can have only this one RUN statement (not a DO-END block with any other statements). You can pass optional INPUT parameters to the procedure you run but no OUTPUT or INPUT-OUTPUT parameters. If you pass parameters, the parameter values are evaluated *once* when the trigger is defined. They are *not* re-evaluated each time the trigger executes.

- To see how you write a persistent trigger and what its effects are, change the ChooseProc procedure and add the new procedure PersistProc, as follows:

```

PROCEDURE ChooseProc.
    ON CHOOSE OF bButton IN FRAME fFrame
        PERSISTENT RUN PersistProc.
    END PROCEDURE.

PROCEDURE PersistProc:
    iCount = iCount + 1.
    bButton:LABEL IN FRAME fFrame = "Internal " + STRING(iCount).
END PROCEDURE.

```

Now when you run the procedure and choose the button you get a very different result. Figure 8–11 shows the results of the first three button presses.

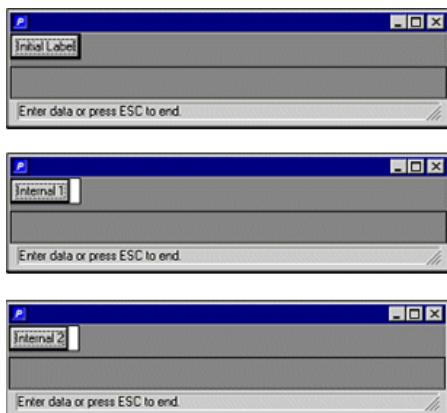


Figure 8–11: Results of running the PersistProc procedure

Once you have established the persistent trigger, it remains in effect as long as the object it's associated with exists, just as a definitional trigger would.

Using the REVERT statement to cancel a trigger

An ON statement can contain the single keyword REVERT to cancel an existing trigger definition before it goes out of scope:

```
ON events OF objects REVERT.
```

The REVERT statement cancels any run-time trigger for the event and object defined in the current procedure or trigger. Note that you cannot use REVERT to cancel a persistent trigger. Instead, you must run another persistent trigger procedure that either defines a new trigger or consists of just a RETURN statement.

Defining triggers to fire anywhere

You can use the ANYWHERE option of the ON statement to set up a trigger that applies to all objects in an application. Use the following syntax:

```
ON events ANYWHERE statement or code block.
```

Applying events in your application

You have already seen cases where a procedure causes an event to fire “artificially” by using the APPLY statement. In the CustOrders procedure, each button trigger has to APPLY the VALUE-CHANGED event to the **Order** browse to get it to run the internal procedure to display related data for the **Order**, such as this code for the **Next** button trigger:

```
DO:
  GET NEXT CustQuery.
  IF AVAILABLE Customer THEN
    DO:
      DISPLAY {&FIELDS-IN-QUERY-CustQuery}
        WITH FRAME CustQuery IN WINDOW CustWin.
      {&OPEN-BROWSERS-IN-QUERY-CustQuery}
      APPLY "VALUE-CHANGED" TO OrderBrowse.
    END.
  END.
```

The APPLY statement causes whatever trigger is currently active for the event and object to fire. Unlike the ON statement, you must place the name of the event in quotation marks in an APPLY statement, if it is a literal value. This makes it possible to APPLY the value of a character variable instead. In this way, you can define a CHARACTER variable, assign it a value of an event name during program execution, and then use that value in an APPLY statement, adding flexibility to your programming. For example:

```
DEFINE VARIABLE cEvent AS CHARACTER NO-UNDO.  
.  
. .  
cEvent = "VALUE-CHANGED".  
. .  
. .  
APPLY cEvent TO <some object>.
```

By contrast, Progress must know the event for an ON statement at compile time to prepare the trigger properly. For this reason the event name can't be a variable, so you can specify it with or without quotation marks in the ON statement.

As another example, here's a little procedure that transfers keystrokes from one fill-in field to another:

```
DEFINE VARIABLE cFillFrom AS CHARACTER NO-UNDO.  
DEFINE VARIABLE cFillTo AS CHARACTER NO-UNDO.  
  
ENABLE cFillFrom cFillTo WITH FRAME fFrame.  
  
ON ANY-PRINTABLE OF cFillFrom  
    APPLY LAST-KEY TO cFillTo.  
  
WAIT-FOR CLOSE OF THIS-PROCEDURE.
```

The LAST-KEY keyword is a built-in function that returns the value of the last keystroke the user pressed. Every time you enter a letter into the first fill-in field, cFillFrom, the ANY-PRINTABLE trigger fires which, as its name suggests, responds to any printable character typed on the keyboard, and this trigger retrieves that keystroke and applies it to the other fill-in field, cFillTo. So you can apply not just the standard events but even keyboard characters to an object like a fill-in field, if is enabled for input. Figure 8–12 shows the effect of typing **Text** into cFillFrom.



Figure 8–12: Result of typing into cFillFrom

Using NO-APPLY to suppress default processing for an event

When you type a letter such as **A** in a field, it naturally appears in the field on the screen. This is called the default processing for the event. If there were a trigger ON “A” OF cFillTo, then you could APPLY that event to the fill-in field, the trigger would fire, and the letter would appear. This is the normal result of applying an event: both the default processing and any trigger on the event occur. However, some event-object pairs do not get Progress default processing using the APPLY statement. For example, applying the CHOOSE event programmatically to a button executes the trigger on that button but does not give focus to the button in the way that choosing it would.

As another twist to this relationship between events and their actions, consider the action on the object that initiates the event, not the one that receives it and does its default processing for the event. Sometimes you want only the trigger action on the target object to occur and not the default processing for the object that initiated the event. In this case, you can use the special RETURN NO-APPLY statement at the end of the trigger definition to suppress the default processing on the object that initiated it.



To suppress the default processing for an event in your test procedure:

1. Add the RETURN NO-APPLY statement (along with the DO-END statements to turn the trigger into a block of code):

```

DEFINE VARIABLE cFillFrom AS CHARACTER NO-UNDO.
DEFINE VARIABLE cFillTo   AS CHARACTER NO-UNDO.

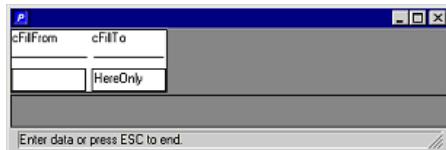
ENABLE cFillFrom cFillTo WITH FRAME fFrame.

ON ANY-PRINTABLE OF cFillFrom
DO:
  APPLY LAST-KEY TO cFillTo.
  RETURN NO-APPLY.
END.

WAIT-FOR CLOSE OF THIS-PROCEDURE.

```

2. Run the procedure and type some text into **cFillFrom**:



When you type, the keystrokes are applied to **cFillTo**, that is its default processing. But the RETURN NO-APPLY statement suppresses the default processing the **cFillFrom**, so it remains blank.

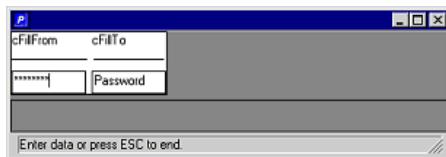
3. Add this statement to the trigger:

```

ON ANY-PRINTABLE OF cFillFrom
DO:
  APPLY LAST-KEY TO cFillTo.
  APPLY '*' TO cFillFrom.
  RETURN NO-APPLY.
END.

```

4. Run the procedure. You get asterisks in **cFillFrom** for each keystroke you type:



The trigger first applies the keystroke to the second fill-in, which causes the character to be displayed. It then applies the literal '*' to the first fill-in, causing it to be displayed there. Finally, it does a RETURN NO-APPLY to suppress the display of the character you actually typed into the first fill-in. You could use this sort of code for a password field, for example.

Applying a nonstandard event

With the APPLY statement, you can actually send any event to any object that has a trigger for that event. Progress executes the trigger even if there is no default handling for the event associated with that object type. Thus, you can use this feature to extend the repertoire of developer events (predefined events with the names U1 through U10) to include any event not normally associated with an object. For example, you could apply CHOOSE to a fill-in field, which does not normally support that event.

Using Graphical Objects in Your Interface

In this chapter, you'll learn how to use visual objects both for data display and for other UI support. As before, you'll let the AppBuilder do most of the work of laying out the objects and generating the 4GL code that defines them. You'll look at some of that code as examples of how the 4GL syntax works.

This chapter includes the following sections:

- [Using data representation objects](#)
- [Using other types of objects](#)
- [Using menus](#)

Using data representation objects

In this section, you make some changes and additions to the fields in the **Customers and Orders** window to learn about some of the alternative field visualizations and their attributes.

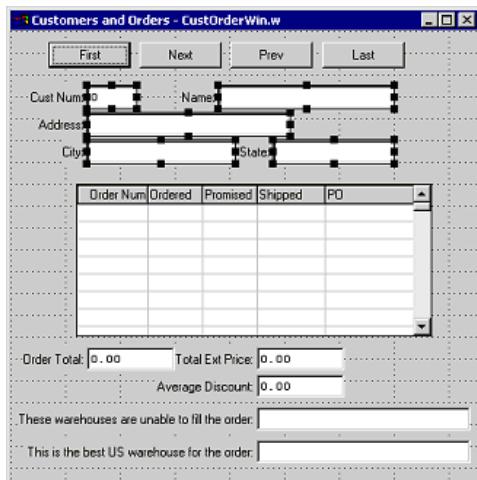
Modifying fill-in attributes in the Properties Window

First, you should disable the fill-in fields for the **Customer** table, because there's no way to actually save changes to them.



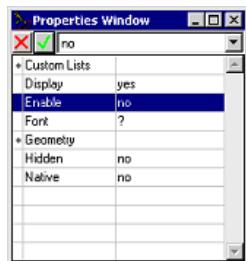
To make a change to more than one field at a time, use the special Properties Window in the AppBuilder:

1. In the AppBuilder, open the h-CustOrderWin2.w procedure.
2. Select all of the **Customer** fields, either by dragging a rectangle around them or by selecting them in turn using the **CTRL-Click** sequence:

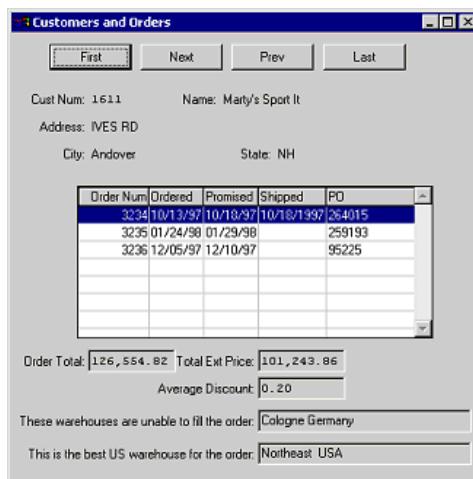


3. From the AppBuilder menu, select **Windows→Properties Window**.

- Double-click on the **Enable** line to change it from **Yes** to **No**:



- Choose the green check mark to register your change, and then close the **Properties Window**.
- Save the window as h-CustOrderWin3.w.
- Run the window to see the effect of your changes:



Because the fields are now disabled, they are no longer highlighted as they were when they were enabled. Note the difference between the appearance of these fields and the fill-in fields below the browse. When you defined the fields below the browse, you set the **Read-Only** attribute in the field property sheet and you turned **Enable** on. In this combination, the fields are not highlighted but their box outline remains.

The **Properties Window** shows all the properties of the objects you selected that you can change as a group. Because you selected more than one field, not all the field properties were displayed. For example, it is not possible to change the **Geometry** attributes of a whole set of fill-ins, because each setting must be different. If you were to select just a single field and then open the **Properties Window**, you would see more attributes that you could change there. Figure 9–1 shows an example using the **Customer.Name** field.

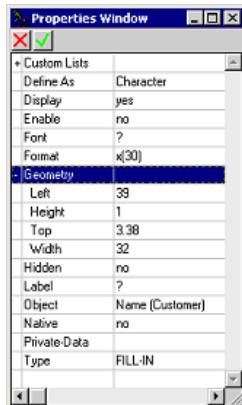


Figure 9–1: Properties Window

Note that among other attributes, the **Type** of the field (which is its visualization type) is available only when you select one field at a time. You'll use this option in the next section to change the display type of a field.

Changing a fill-in field to an editor

Now you'll add a field to the screen with a different visualization. The **Comments** field in the **Customer** table is a text field that can get fairly long, but its default display (as defined in the Data Dictionary) is just as a fill-in. You can add it to the screen and then change it to an editor, which is a more appropriate way to display the information.

Before you go any further, you should change the query on the **Customer** table so that it doesn't just show New Hampshire **Customers** sorted by the **City**. In the **sports2000** database, most of the fill-in field **Comments** values are toward the beginning of the **Customers**, so to see meaningful data in the field you need to look at **Customers** toward the beginning of the table.



To edit the query on the Customer table:

1. Double-click on the design window (outside of any field) to go into the property sheet for the frame.
2. Choose the **Query** button and edit the WHERE clause and the SORT phrase out of the query. You should know how to do this from using the **Query Builder** in Chapter 4, “Introducing the OpenEdge AppBuilder.”
3. Exit the **Query Builder** and extend the design window to the right by dragging the window edge with the mouse, to make room for new fields.



To change the fill-in field to an editor:

1. Choose the **DB Fields** button in the AppBuilder **Palette** window:

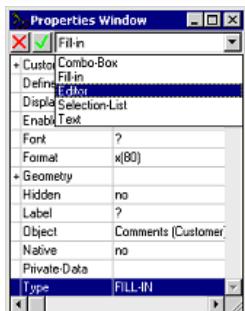


2. Click on an empty space on the right side of the design window.
3. Select **Customer** in the **Table Selector** dialog box, then choose **OK**.
4. Add the **Comments** field in the **Multi-Field Selector** dialog box that appears, and then choose **OK**.

The field now appears as a long fill-in in your design window. It's even possible you could get a warning message that the field is too long to allow the label to be displayed. That's not a problem, since an editor doesn't have a label anyway.

5. With the **Comments** field still selected in the design window, open the **Properties Window**.
6. Select the **Type** line in the display. This shows the current display type (Fill-In) in a combo-box at the top of the window.

7. Drop down this list and select **Editor**:



8. Choose the green check mark and close the window. Now your **Comments** field is an editor.

Using the Properties Window to change a display type

Normally, the **Properties Window** is an alternative way to set field attributes, instead of just using the AppBuilder property sheet dialog boxes that you've seen. The main advantage of using it is that you can set common field properties for multiple fields at a time, as you saw earlier.

But in this case, the only way to change the display type of a field after you've dropped it onto the design window is in the **Properties Window**. The AppBuilder property sheets for different display types are different, so you can't actually change the display type from the property sheet. But now that you've changed the display type in the **Properties Window**, you can set its editor-specific attributes.



To open the editor's property sheet and set attributes:

1. Double-click on the editor field to open the property sheet.
2. Check the toggle boxes for these attributes:
 - **Enable** — Even though you won't allow the user to change the value, checking on this toggle box makes the contrast between the displayed text and the background greater, so the text is more readable. This is an important consideration since, depending on the color combinations, it is possible wind up with a disabled editor with text that is gray on gray and therefore invisible!
 - **Read-Only** — This option keeps the value from being changed even though it is displayed as enabled.

- **Scrollbar-Vertical** — This option gives you a scrollbar on the right-hand side of the editor if the value gets too long to display. This is another reason to mark the editor as **Enabled**. If it is not, then the user can't use the scrollbar.
 - **Word-Wrap** — This option wraps whole words from one line to the next, rather than extending the text off the editor to the right. You should normally select this option if you use a vertical scrollbar.
3. Choose **OK** to exit the property sheet.
 4. Resize the editor as you like, to have multiple lines of display.
 5. **Run** the window and choose the **Next** button to see what happens.

The **Comments** field value is displayed properly for the first **Customer**, but it doesn't change when you choose the **Next** button. What went wrong? The AppBuilder regenerated the `enable_UI` procedure with its field list, to include the **Comments** field in its initial display. But the trigger blocks on the four buttons is code you copied from `enable_UI` earlier, so it didn't get changed automatically.

You don't want to have to keep editing this field list, so you're better off substituting the preprocessor value that the AppBuilder keeps up to date as the field list changes. This is the **DISPLAYED-FIELDS** preprocessor.

The next section describes how to use the **Section Editor** to go into each of the four button triggers, remove the field list from the **DISPLAY** statement, and substitute the preprocessor value for it.

Using the Section Editor's pop-up menu to insert a value

The **Section Editor** helps you insert a preprocessor value or other text into your procedures and triggers by giving you access to a number of useful lists of elements you might need in your code. These include database fields, event names, and more.

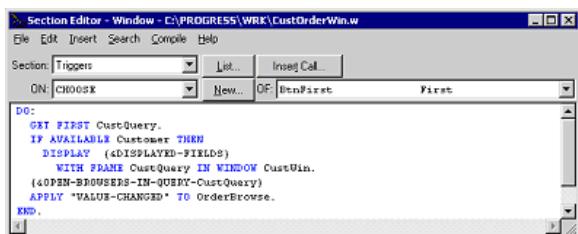
To use the Section Editor to edit the four button triggers in the sample procedure:

1. Select the first button in the design window, then choose the **Edit Code** icon from the AppBuilder toolbar to open the **Section Editor**.
2. In the **Section Editor** window, place the cursor after the **DISPLAY** statement, then right-click.

3. From the pop-up menu, select **Insert→Preprocessor Name**:

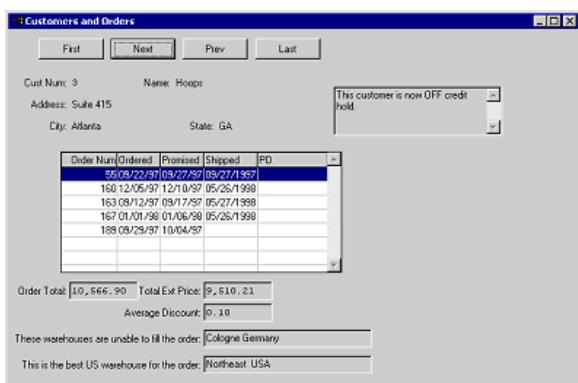


4. In the **Code References** dialog box, double-click on the **DISPLAYED-FIELDS** preprocessor to insert it into the code:



5. Repeat Steps 1 to 4 for the other three button triggers.

Now when you run the window you should see the proper **Comments** field for each **Customer**:



Looking at the code that defines the editor

Finally, look at the code the AppBuilder generated to define the **Comments** field as an editor. Because this is a database field, and not a variable, the definition is in the **DEFINE FRAME** statement. To see this, you'll need to go into the **Code Preview** window (press **F5**):

```
DEFINE FRAME CustQuery
.
.
.
Customer.Comments AT ROW 4.1 COL 76 NO-LABEL
    VIEW-AS EDITOR SCROLLBAR-VERTICAL
    SIZE 39 BY 2.71
.
.
```

Adding a toggle box to the window

There are no **LOGICAL** fields in the **Customer** and **Order** tables to display as toggle boxes, but you can add a variable to display as a toggle box instead.



To add a toggle box that lets the user change whether the fields with warehouse information are displayed at run time:

1. Select the **Toggle Box** icon from the **Palette**:



2. Click in the design window to the right of the **Total Ext** price field to drop the toggle box onto the window.
3. Change the **Object** name to **lShowWarehouse** (starting with the letter **l** because it's a representation of a **LOGICAL** variable) and change the **Label** to **Show Warehouse Info.**
4. Adjust the size of the toggle box to show the whole label.
5. Go into the property sheet for the toggle box, choose the **Advanced** button, and change its initial value from **no** to **yes**.

6. Select the **Edit Code** icon and define this VALUE-CHANGED trigger for the field:

```
DO:  
  ASSIGN lShowWarehouse.  
  IF lShowWarehouse THEN  
    ASSIGN cWorstWH:HIDDEN = NO  
    cBestWH:HIDDEN = NO.  
  ELSE  
    ASSIGN cWorstWH:HIDDEN = YES  
    cBestWH:HIDDEN = YES.  
END.
```

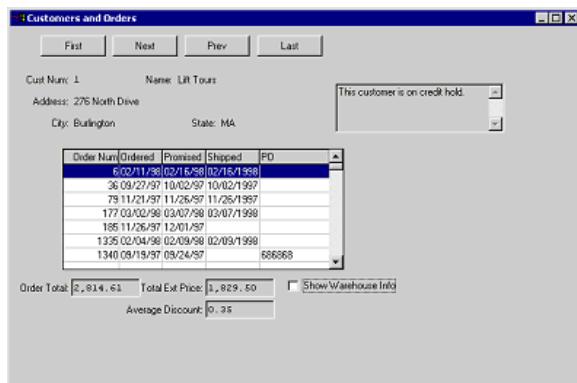
Remember that a field value entered on the screen does not get saved to the record buffer until it is assigned. Therefore, the trigger block has to start with an ASSIGN statement for the variable before you can refer to its value. Alternatively, you could query the SCREEN-VALUE attribute of the field without assigning it. When you reference the SCREEN-VALUE, however, the value is represented as a character string, even though it's not displayed that way, so you would have to explicitly compare the SCREEN-VALUE to YES or NO like this:

```
IF lShowWarehouse:SCREEN-VALUE = "YES" THEN . . .
```

By contrast, when you assign the value to its record buffer, you can refer to it as a LOGICAL so that it becomes in effect a one-field expression that evaluates to true or false:

```
IF lShowWarehouse THEN . . .
```

Your new trigger is setting the HIDDEN attribute of the two warehouse-related fields to true or false. When you run the procedure and click the toggle box, the fields appear and disappear:



Looking at the code to define a toggle box

This is the DEFINE VARIABLE statement the AppBuilder generated to define this toggle box (you can see this code from the **Code Preview** window):

```
DEFINE VARIABLE lShowWarehouse AS LOGICAL INITIAL yes
  LABEL "Show Warehouse Info"
  VIEW-AS TOGGLE-BOX
  SIZE 29 BY .81 NO-UNDO.
```

The INITIAL value, LABEL, VIEW-AS type, and SIZE are all defined here. As with other fields, the toggle box is enabled not by syntax in the DEFINE VARIABLE statement, but because it's added to the ENABLE statement in enable_UI.

Defining a slider

Next you'll add a slider object to the window. A slider visually represents an INTEGER value within some range. In this case, you'll just make the slider show the same value as the **Average Discount** field, but as a percentage between 0 and 50.



To define a slider:

1. Select the **Slider** icon from the **Palette**:

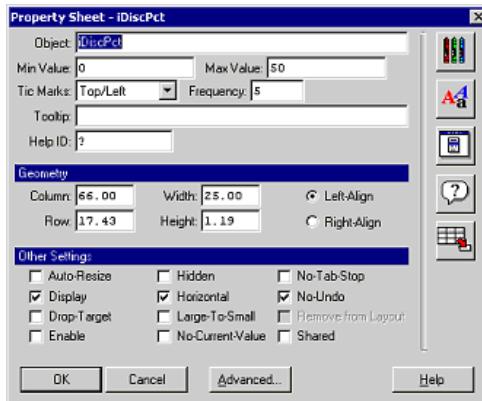


2. Drop it onto the design window to the right of the **Average Discount** field.
3. Change the **Object** name to **iDiscPct**. You'll see that the **Label** field is not enabled for the slider because a slider cannot have a label.
4. Go into the property sheet for the slider and change the attribute values, as described in [Table 9–1](#).

Table 9–1: **iDiscPct** attribute values

| Attribute | Change value to . . . | Description |
|------------|-----------------------|---|
| Max Value | 50 | Shows values in the range from 0 to 50. |
| Tic Marks | Top/Left | Shows tic marks on the top of the slider. |
| Frequency | 5 | Shows tic marks every 5 units. |
| Horizontal | true | Displays the slider horizontally instead of vertically. |
| Enable | false | Prevents the user from manipulating the slider object. |

When you are done, the property sheet should look like this:



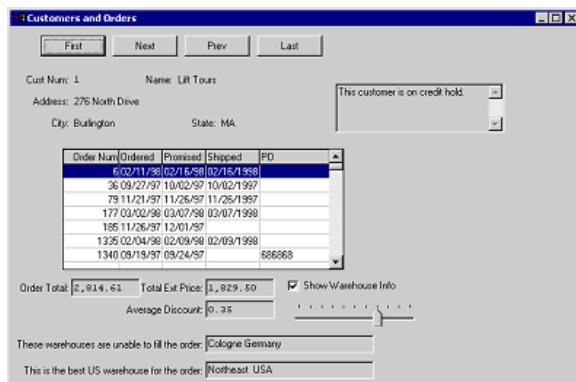
5. Choose **OK** after making these changes.
6. You want the slider value to display along with the other extra fields at the bottom of the window each time the user selects a browse row, so edit the VALUE-CHANGED trigger for the browse and replace the line that assigns the **dAvgDisc** field and the DISPLAY statement that follows it with these lines:

```

ASSIGN dAvgDisc = (dTotalPrice - dTotalExt) / dTotalPrice
    iDiscPct = dAvgDisc * 100.
DISPLAY dTotalPrice dTotalExt dAvgDisc iDiscPct cWorstWH cBestWH
    WITH FRAME CustQuery.

```

7. You can make the slider tall enough to show the percentage value above the slider itself, or shrink it down vertically so that it just shows the slider itself. It should show the same value as the **Average Discount** field it's next to:



Looking at the code for a slider

Here is the AppBuilder-generated code for the slider:

```
DEFINE VARIABLE iDiscPct AS INTEGER INITIAL 0
VIEW-AS SLIDER MIN-VALUE 0 MAX-VALUE 50 HORIZONTAL
TIC-MARKS TOP FREQUENCY 5
SIZE 25 BY 1.19 NO-UNDO.
```

Objects that display a list of choices for a data value

Several graphical objects are suitable to display using a limited set of possible values the user must choose from.

A *combo box* is a list of values that displays the current field value and drops down into view when the user chooses an arrow icon to the right of the current value. This makes it effectively a combination of a list box and a text box. The user can then select a different value from the list or, depending on how the combo box has been set up, enter a value not already on the list. A combo box is, therefore, an appropriate visualization when you don't want the list of choices displayed at all times and also when it is sometimes valid for the user to enter a value not on the choice list.

A *selection list* displays a list of values the user can choose from. Though the list is not contracted after the user chooses a value, you can provide a scrollbar to allow more possible values than can reasonably be displayed at one time. You can define a selection list to allow multiple selections, making it an appropriate visualization when the user must be able to pick more than one value or when the number of possible values is long enough that it requires a scrolling list.

If you need to provide a choice from a large or potentially large number of values, a *browse* can be an appropriate object to use as a selection list without real limits on its size.

To display a small number of choices in either a vertical or horizontal format, you can also use a *radio set* object.

Using a radio set to display a set of choices for a value

In this section, you'll look at the radio set as a representative example of the ways you can display a list of values. You can find out more about each object type's individual attributes in the online Help and *OpenEdge Development: Progress 4GL Reference*.

The term *radio set* derives from the preset radio station buttons found on most car radios. The radio set is an appropriate visualization when you want to make it clear that the user must choose one value at the expense of others, when making one choice automatically cancels any other choice. Even though making a single choice from a list naturally results in not choosing another value, the radio set is appropriate when the choices are somehow related, and when the user should consider the list and think about the effect of choosing one value over another. You also need to lay out a radio set carefully when you're designing a window, making room in either a vertical or a horizontal format for the radio buttons themselves and their labels. A radio set is appropriate when the number of values is fixed and not likely to change. So, for example, a list of possible **Sales Reps** wouldn't be an appropriate field to display as a radio set, because the number of values in the list could change over time and also because **Sales Rep** initials are just alternative data values without a relationship to one another, which would make it necessary to consider one choice versus another.



To add a radio set to your test window:

1. Choose the **Radio Set** icon from the AppBuilder **Palette**:



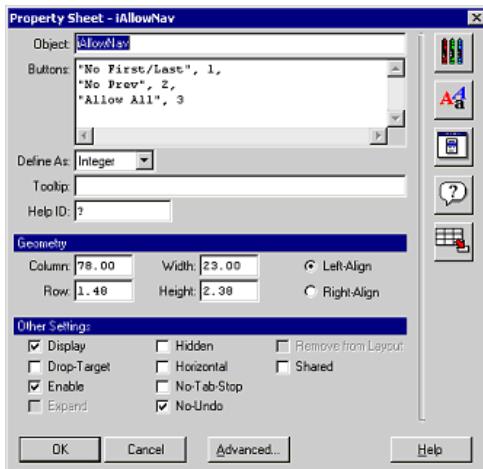
2. Drop it onto the design window to the right of the navigation buttons.

You're going to use the radio set to allow the user to selectively enable or disable different combinations of navigation buttons.

3. Double-click on the radio set to go into its property sheet. Name it **iAllowNav**.

When you define the list of choices for a radio set, you must specify both the radio button label to display and the associated value you want to assign to the field when that button is chosen. If the labels and values are the same, you have to specify each one as both the label and the value. Because you can assign any field value you want for a radio button choice, you can define a radio set as representing any of the standard data types CHARACTER, DECIMAL, INTEGER, DATE, and LOGICAL.

4. In the property sheet, select **INTEGER** as the data type and enter the button labels and values shown here:



5. Choose the **Advanced** button and set the initial value of the radio set to **3**.

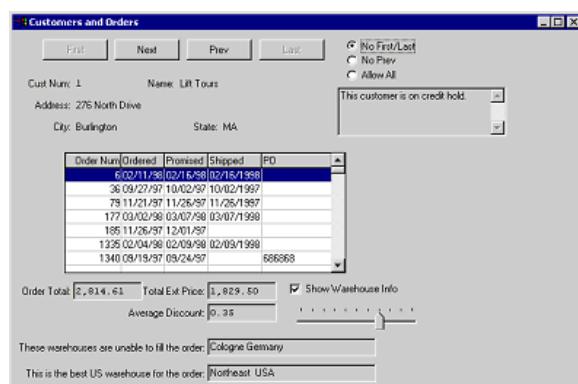
This makes the **Allow All** choice the initial value, which is appropriate because all the buttons are initially enabled.

Now you have to define the trigger code to enable and disable buttons when the user makes a choice.

6. Define this VALUE-CHANGED trigger for the radio set:

```
DO:
  ASSIGN iAllowNav.
  IF iAllowNav = 1 THEN
    ASSIGN BtnFirst:SENSITIVE = NO
    BtnLast:SENSITIVE = NO.
  ELSE IF iAllowNav = 2 THEN
    ASSIGN BtnPrev:SENSITIVE = NO.
  ELSE ASSIGN BtnFirst:SENSITIVE = YES
    BtnLast:SENSITIVE = YES
    BtnPrev:SENSITIVE = YES.
END.
```

7. Use the mouse to adjust the size of the radio set in the design window so that the labels all display correctly.
8. Run the window again and try out the different radio set choices:



This is the code the AppBuilder generated to display the radio set:

```
DEFINE VARIABLE iAllowNav AS INTEGER INITIAL 3
  VIEW-AS RADIO-SET VERTICAL
  RADIO-BUTTONS
    "No First/Last", 1,
    "No Prev", 2,
    "Allow All", 3
  SIZE 23 BY 2.38 NO-UNDO.
```

Using other types of objects

Some visual objects aren't intended to represent field values. These use DEFINE statements that name the object type directly rather than defining it with a VIEW-AS phrase on a DEFINE VARIABLE statement or a field reference in a frame definition. You've already worked with buttons as an example of this. A button has no value but simply exists to allow the user to trigger a CHOOSE event. Now you'll look at two other objects that don't represent data values: rectangles and images.

Using rectangles to organize objects

You might not be inclined to think of a visualization as primitive as a rectangle as being a real visual object at all, in the same sense that the other objects you've looked at are. But a rectangle has all the basic characteristics the other object types do:

- It has attributes you can set to change its appearance.
- It has a handle you can use to access the object and to set and query its attribute values.
- Though it would be unusual to do so, you can define triggers for a rectangle and apply events to it.



To add a rectangle to the sample window:

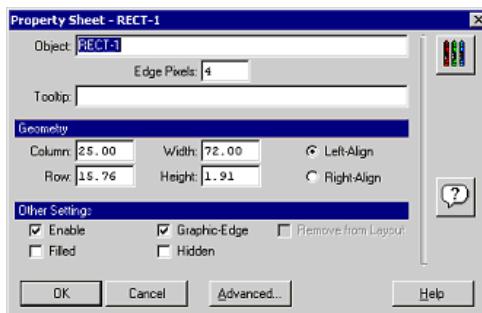
1. Select its icon from the Palette:



2. Use the mouse to drag a rectangular area around the **Average Discount** field and the corresponding slider value.
3. Double-click on the rectangle (that is, anywhere within the area enclosed by the rectangle that isn't within another object) to bring up its property sheet.

Because you're not going to define triggers or other code that references the rectangle, there's no need to change its default name.

4. If you want the line to appear thicker, change the **Edge Pixels** value from 2 to 4:



This is the code the AppBuilder generates for the rectangle:

```
DEFINE RECTANGLE RECT-2
EDGE-PIXELS 4 GRAPHIC-EDGE NO-FILL
SIZE 72 BY 1.91.
```

Using images to display pictures

An image object represents an area on a frame into which you can load an image file from disk. This can be an image in the bitmap (.bmp) or Jpeg (.jpg) format.



To add an image to the window:

1. Select the **Image** icon from the **Palette**:



2. Drop the image onto the design window to the right of the browse and shape it to be roughly the height of the browse.

The image area displays a default black-and-white image just so you can see how large it is, until you load a specific image into it.

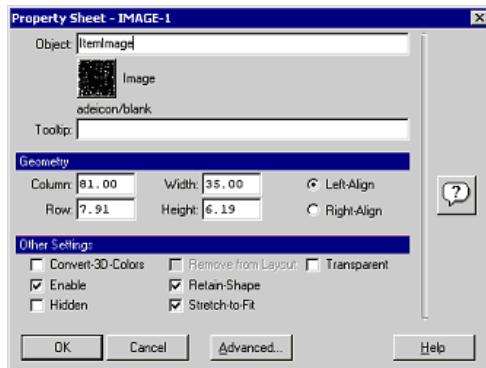
3. Double-click on the image to bring up its property sheet.
4. Name it **ItemImage**.

You're going to use the image to display a picture of one of the **Items** in the **Sports2000** database.

To show each image properly within whatever space you've allocated to it, you can select two attributes that will adjust it as needed.

5. Check on the **Stretch-to-Fit** toggle box. This option expands or contracts the image to fit the space.

6. Check on the **Retain-Shape** toggle box. This option retains the overall ratio of height to width of the image as its size is changed to fit the space:



Each **Order** displayed for a **Customer** can have one or many **Order Lines**. Each of these **Order Lines** is for a specific **Item** the **Customer** has ordered. Since there isn't room in this simple test window to show all the **Order Lines** for each **Order** along with the **Item** information, you'll just show the **Item** image for the first **Order Line** for the currently selected **Order**.

There's an image file in the `sports2000/images` directory for each **Item** in the item catalog of the **Sports2000** database. The name of the file is `cat` plus the item number as a five-digit number, with the `.jpg` extension. To load the image into the image object area in the window, you use the `LOAD-IMAGE` method.



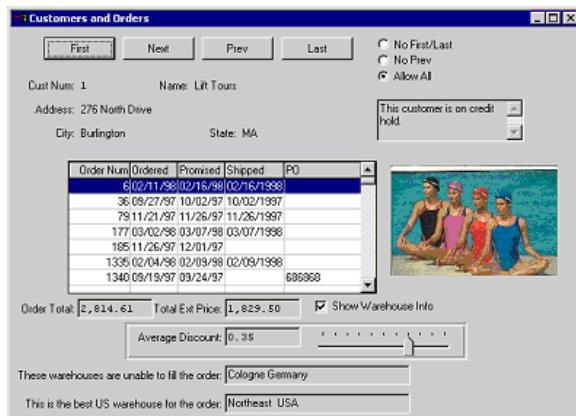
To find the right item to display and load its image:

- Add these three lines of code to the end of the VALUE-CHANGED trigger for the **Order** browse:

```
FIND FIRST Orderline OF Order.
FIND Item OF Orderline. /* No need for FIRST because there's just one
item. */
ItemImage:LOAD-IMAGE('src/sports2000/images/cat' +
STRING(Item.ItemNum, '99999') + '.jpg').
```

- Save and Run** the window once more.

Each time you select a different **Order**, either by selecting a row in the browse or switching to a different **Customer**, a new image should appear:



Adding text to the window

There is one case of something you can add to a window that *isn't* actually an object, and that is text.



To add text to your sample window:

1. Select the **Text** icon from the **Palette** and click above the editor object to add a text field as a label for it:



Note that the **Object** field in the AppBuilder window is not enabled. This is because the name the AppBuilder assigns to the text is not meaningful; it is simply TEXT- plus an integer ID. The name is not actually referenced within the code.

2. Enter a **Text** value of **Customer comments:** for the text:



Note that you have to include the colon if you want one because Progress does not automatically recognize this as a field label and does not add the colon for you.

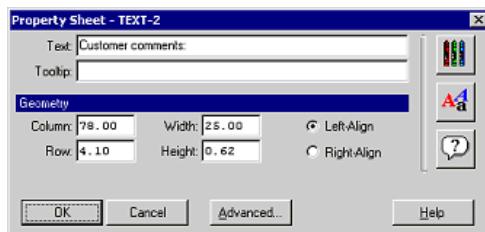
3. Stretch the text field in the design window to hold the text you entered and position it above the editor.

This is the code the AppBuilder generates for the text field:

```
DEFINE FRAME CustQuery
.
.
.
cBestWH AT ROW 19.33 COL 44 COLON-ALIGNED
ItemImage AT ROW 7.91 COL 81
RECT-1 AT ROW 15.76 COL 25
"Customer comments:" VIEW-AS TEXT
    SIZE 25 BY .62 AT ROW 4.1 COL 78
WITH 1 DOWN NO-BOX KEEP-TAB-ORDER OVERLAY
    SIDE-LABELS NO-UNDERLINE THREE-D
    AT COL 1 ROW 1
    SIZE 119.8 BY 19.62.
```

It is simply a literal value inserted into the frame definition. It has no name, and it is not created as an object with a handle or any attributes.

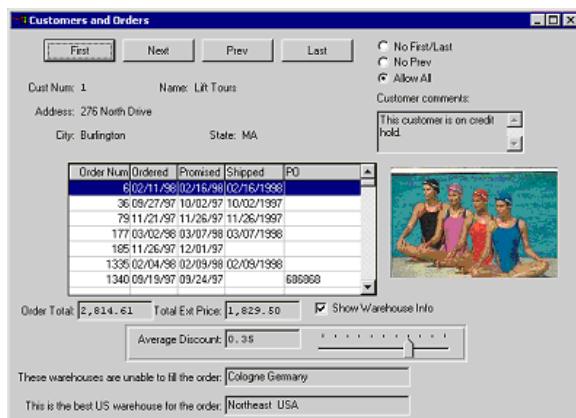
4. You can change the font and color of the text in its limited property sheet if you wish:



However, if you wanted to be able to manipulate the text in some way at run time, for example to change the value displayed when some event happens, you cannot do this because you have no access to the text at run time. If you want to access a text string as an object in this way, then you need to define it as a fill-in field rather than as text, and then set the VIEW-AS-TEXT attribute in its property sheet. The visual appearance will be the same as a simple text string, but the text is an object with a handle and attributes you can set and query at run time.



To see the final effects of all your changes, **Run** the window one more time:



Using menus

Progress supports both menu bars and pop-up menus. A *menu bar* is a horizontal bar displayed at the top of a window. The menu bar can contain submenus you can drop down to select individual items. A *pop-up menu* is a list of menu items associated with an individual visual object in a window. Normally, you invoke a pop-up menu by clicking the right mouse key on the object it's associated with.

Menu bars

A Progress window can have one and only one menu bar. The label for each of its menu items and its submenus appears in a line at the top of the window. When the user selects a submenu label, a pull-down menu appears showing the submenu's items. You can nest submenus within other submenus, in which case the nested submenu is shown to the right of its pull-down menu, as shown in [Figure 9–2](#).

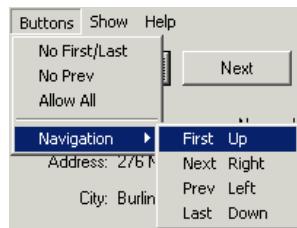


Figure 9–2: Example submenu

Menus, submenus, and their items form a *menu hierarchy*. Menus are the parents of their menu items and submenus. Each child submenu can in turn be the parent of menu items and/or other submenus. Your application can define multiple levels of submenus for a window.

Each menu has an `OWNER` attribute that identifies its owning object. For a menu bar, this is the menu's window. For a pop-up menu, it is the object the pop-up menu is defined for.

You can use the `PARENT`, `FIRST-CHILD`, and `NEXT-SIBLING` attributes of menus and menu items to traverse the menu hierarchy. There's an example later in this section that shows how to do this.

Defining a menu bar

You define a menu bar from the bottom up, by defining first the submenus that make up a menu and then the menu itself. In this way, the submenus are already defined when you reference them in the menu bar definition. This is the general syntax for defining a submenu:

```
DEFINE SUB-MENU submenu-name
{ LIKE other-menu | menu-element-descriptor ... }
```

Every submenu has a name like any other object, which can be used to identify it.

There are other keywords you can use in menu, submenu, and menu item definitions to define colors, fonts, and other details. Check the online Help for details on these attributes.

If two or more menus share the same elements, you can define one to be LIKE the other. This would more likely be the case for a pop-up menu rather than a menu bar, if multiple visual objects share many of the same pop-up menu options.

Otherwise, a submenu definition has one or more menu element descriptors. Each of these can be one of the following:

```
{ RULE
SKIP
SUB-MENU submenu-name [ DISABLED ] [ LABEL label ] menu-item-phrase
}
```

A RULE inserts a line in the submenu to separate groups of related items. A SKIP simply leaves an empty line, which can serve the same purpose.

If one submenu contains another, you nest one within the other. You must define the innermost submenu first, so that as each submenu is referenced within another submenu or within the main menu bar itself, its name and definition are already identified.

A nested submenu can be initially disabled. Submenus and menu items have a SENSITIVE attribute just like other objects, which you can set to true or false at run time to enable or disable the submenu or menu item.

If you specify a LABEL, then that string appears as a header for the submenu. Otherwise, the submenu name is used as the default label.

Each *menu-item-phrase* identifies a single item that the user can choose from the menu, and has this general syntax:

```
MENU-ITEM menu-item-name
  [ ACCELERATOR keylabel ]
  [ DISABLED ]
  [ LABEL label ]
  [ READ-ONLY ]
  [ TOGGLE-BOX ]
{ [ trigger-phrase ] }
```

Each menu item is an object in its own right, with its own name. The ACCELERATOR provides a shortcut key for selecting the menu item. Like a submenu, it can be initially disabled and can have its own label.

If a menu item is marked READ-ONLY, then it appears in the menu but cannot be chosen. You could use such a menu item as a kind of title for a group of selectable menu items that appear after it.

A menu item designated as a TOGGLE-BOX represents a logical value. Each time you select it its logical true/false value is reversed and a check box appears or disappears in front of the label to indicate this. The CHECKED attribute tells you at run time whether the item is currently checked or not.

You can define a CHOOSE event trigger phrase in place of a menu item or you can define an ON CHOOSE block in the procedure following the menu definition.

Once you've defined all the submenus your menu bar needs, you define the menu bar itself:

```
DEFINE MENU menu-name MENUBAR
  [ { LIKE other-menu } | menu-element-descriptor ... ]
```

Use the MENUBAR keyword to identify that this is a menu bar for a window and not a pop-up menu. The menu element descriptors for the menu itself can be submenus and menu items, but not rules or skips.

Assigning a menu bar to a window

To assign a menu bar to a window, you set the MENUBAR attribute of the window to the menu bar's handle:

```
window-name-or-handle:MENUBAR = MENU menu-name:HANDLE.
```

Typically, you have to identify a menu name by preceding it with the MENU keyword as in this example. You identify a submenu using the MENU or SUB-MENU keyword. You identify a menu item by preceding it with the MENU-ITEM keyword.

Defining a menu bar for the sample window

To see how menus work, you can add one to the h-CustOrderWin3.w window. As it does with other objects, the AppBuilder can define most of the statements you need for your menu, using a special menu builder tool. Once you've completed the menu, you can examine the syntax to confirm the exact 4GL statements needed to do the job.



To define a menu bar for the sample window:

1. Open h-CustOrderWin3.w and save a new version as h-CustOrderMenu.w.
2. Choose the **Object Properties** button  for the window.

Remember that if the window's frame or some other object is selected, you can use the key **CTRL-Click** sequence to make the window the currently selected object in the AppBuilder.

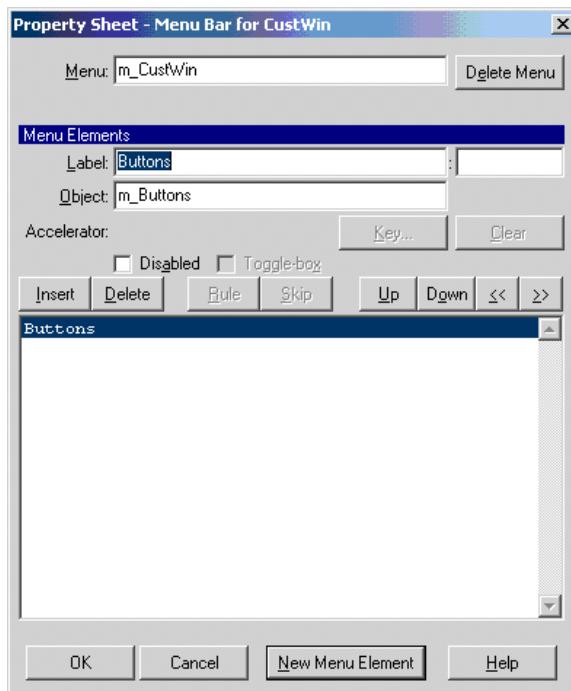


3. In the **Object Properties** dialog box, choose the **Menu Bar** button .
4. In the menu bar's property sheet, call the **Menu m_Custwin**.

You'll add the following submenus to the menu:

- One named **Buttons** that duplicates what the radio set and the **First**, **Next**, **Prev**, and **Last** buttons do.
- One named **Show** that duplicates the **Show Warehouse Info** toggle box.
- One named **Help** that you can use to display information about the window.

5. Define the first of these submenus by entering a **Label** of **Buttons**. When you tab out of that field, a default **Object** name of **m_Butons** is provided for you:



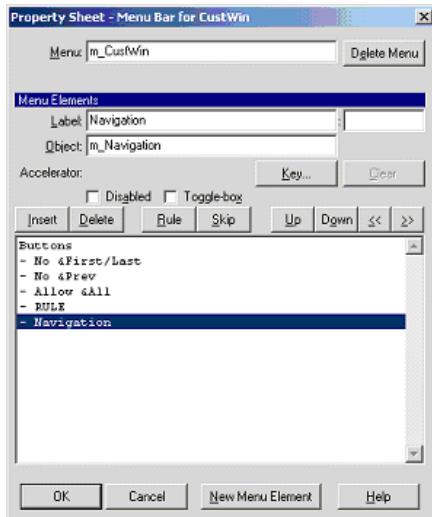
You can define all the levels of submenu and menu items together in this property sheet. To indicate the hierarchy, use the << and >> buttons to indent one item relative to the one above it.

6. Choose the **Insert** button to add three new menu items in the **Buttons** submenu.
7. To indicate that they are menu items of the **Buttons** submenu, choose the >> button to indent the first of the items. The others indent automatically.
8. Give the three items labels of **No &First**, **Last**, **No &Prev**, and **Allow &All**, respectively. Accept the default object names provided for you.

The ampersand in each label represents a menu mnemonic. The mnemonics let the user press a sequence of keys, each beginning with the **ALT** key, to select any menu item from the hierarchy. The ampersand precedes the mnemonic for each of these menu items. By default the top-level submenus are given mnemonics of their first letters, so when you're done the key sequence **ALT-B**, **ALT-F** selects the **No First/Last** menu item. When you press the **ALT** key the mnemonics are underlined to let you see what key sequences are available to you. (Showing the mnemonic underscores *after* the user presses **ALT** is now the Microsoft standard and OpenEdge follows it.)

If you define a menu mnemonic key combination that duplicates an accelerator, the accelerator takes precedence.

9. Choose the **Rule** button to insert a rule following these three items.
10. Choose the **Insert** button and define an item with a label of **Navigation**:



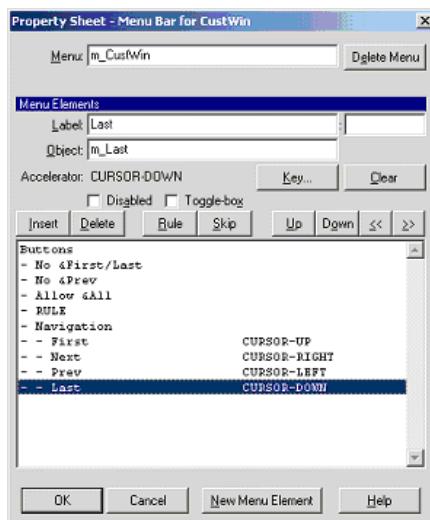
This new item is a submenu. The menu bar property sheet is flexible enough to allow you to define the entire hierarchy from the top down. It determines that this is a submenu when you add further nested items to it. If you need to, you can choose the **Up** and **Down** buttons to rearrange the order of items in the menu.

11. Choose **Insert** followed by the **>>** button to add an item nested under **Navigation**, with a label of **First**.
12. Choose the **Key** button to enter the dialog box that lets you define an accelerator key sequence for the menu item.

13. Press the up arrow on the keyword to register that as the accelerator key for the **First** button. Its keyboard name is displayed as CURSOR-UP:



14. Choose **OK** to accept this.
15. Define three more menu items at this level with labels of **Next**, **Prev**, and **Last**. Give them the accelerator values right arrow (CURSOR-RIGHT), left arrow (CURSOR-LEFT), and down arrow (CURSOR-DOWN), respectively:



The double indentation identifies **Navigation** as a new submenu with four items as menu items under it.



To provide trigger blocks to execute when these items are selected:

1. Go into the **Section Editor** to define a CHOOSE trigger for menu item **m_No_FirstLast**.
2. Get to this trigger block in one of two ways:
 - Go into the **Section Editor**, select the **Triggers** section and then the menu item from the list of objects on the right.
 - Select the menu item in the design window, and then choose the **Edit Code** button.

As you can see, the menu is active and its menu items are selectable even in design mode.

3. To duplicate the effect of the radio set button with the same label, copy its ASSIGN statement into the new trigger:

```
DO:
  ASSIGN BtnFirst:SENSITIVE IN FRAME CustQuery = NO
        BtnLast:SENSITIVE IN FRAME CustQuery = NO.
END.
```

4. Make one change to the statement to qualify each button with the name of the frame it's in.

Why is this necessary here and not in the trigger block for the radio set? Both the buttons and the radio set are in the frame called **CustQuery**, so Progress identifies the frame for the buttons by default. The menu, however, is *not* part of the frame, but rather is owned by the window. For this reason Progress requires that you identify the frame where the buttons are located.

This is the CHOOSE trigger for the **m_No_Prev** item:

```
DO:
  BtnPrev:SENSITIVE IN FRAME CustQuery = NO.
END.
```

The next code block is the trigger for the **m_Allow_All** item.

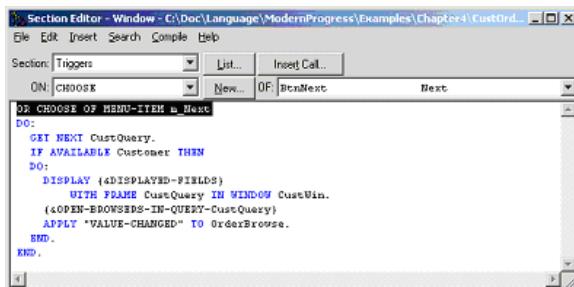
5. So that you don't have to type **IN FRAME CustQuery** three times, make it the default frame for an entire DO block:

```
DO:
  DO WITH FRAME CustQuery:
    ASSIGN BtnFirst:SENSITIVE = YES
    BtnLast:SENSITIVE = YES
    BtnPrev:SENSITIVE = YES.
  END.
END.
```

6. Define trigger actions for the four **Navigation** menu items. These triggers duplicate the actions of the CHOOSE triggers for the four buttons with the same labels. Often you will want some of your menu items to duplicate the action of toolbar buttons. In general, it is considered appropriate user interface design to provide a menu item for every toolbar button, such that the toolbar buttons duplicate the most frequently used menu items. One way to do this without duplicating code is to apply the CHOOSE event to the button from the menu item, as in this trigger block for the `m_First` menu item:

```
DO:
  APPLY "CHOOSE" TO btnFirst IN FRAME CustQuery.
END.
```

A cleaner way to do this, however, is to avoid defining a CHOOSE trigger for the menu item at all. Instead, overload the trigger definition for the button by including the menu item name in the ON phrase, as in this modified trigger block for `btnNext`:



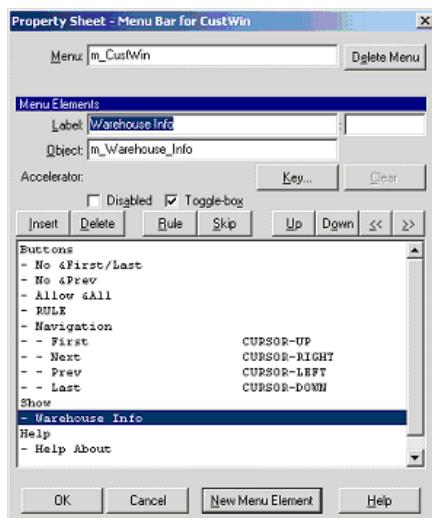
7. Make similar changes to the `btnPrev` and `btnLast` triggers to enable their actions for the same menu items as well.
8. Run the window and try out all the buttons and accelerators, including the nested items under the **Navigation** submenu:



► **To complete your menu bar:**

1. Go back into its property sheet. You can do this by selecting any item in the menu in the design window and clicking the **Object Properties** button.
2. Add a top-level submenu called **Show**, with a menu item of **Warehouse Info**. Check on the **Toggle-box** toggle box to make this a menu item that represents a logical value.
3. Add a final top-level submenu called **Help**, with a menu item called **Help About**.

The property sheet should look like this when you're done:



The **Warehouse Info** item is selected to show that the toggle box is checked.

4. Define a trigger for the **Warehouse Info** item:

```
DO:  
  DO WITH FRAME CustQuery:  
    IF SELF:CHECKED THEN  
      ASSIGN lShowWarehouse:SCREEN-VALUE = "YES"  
      cWorstWH:HIDDEN = NO  
      cBestWH:HIDDEN = NO.  
    ELSE  
      ASSIGN lShowWarehouse:SCREEN-VALUE = "NO"  
      cWorstWH:HIDDEN = YES  
      cBestWH:HIDDEN = YES.  
    END.  
  END.
```

Because it is a toggle box item, it has a VALUE-CHANGED trigger rather than a CHOOSE trigger.

This duplicates the code for the **Warehouse Info** toggle box in the window, but also sets the screen value of that toggle so that the two are kept in sync. Remember that the SELF keyword within a trigger block always refers to the object handle that invoked the trigger. The CHECKED attribute tells you whether the menu item is currently checked or not. Each time you select this menu item, the check box alternately appears and disappears, and the value of CHECKED reverses between true and false.

As with earlier triggers, you have to enclose all of this in a DO WITH FRAME CustQuery block to identify all these objects.

Using the MENU-DROP event

Just as this trigger block changes the displayed value for the toggle box called **lShowWarehouse** when the corresponding menu item is selected, make sure that the initial value of the menu item matches the current value of the toggle box when you drop down the **Show** menu. There is a MENU-DROP event for submenus (and for pop-up menus) to let you intercept this event and execute whatever code you need before the menu items under the submenu or the pop-up menu are displayed. In this case, you use the event to initialize the CHECKED attribute of the **m_Warehouse_Info** item to match the toggle box. In other cases, you could use this event to disable menu items that aren't currently applicable or even to add additional items to the menu dynamically. You'll learn how to create dynamic menus and to make changes to menus dynamically in [Chapter 18, “Using Dynamic Graphical Objects.”](#)

This is the MENU-DROP event for the `m_Show` submenu:

```
DO:
  MENU-ITEM m_Warehouse_Info:CHECKED IN MENU m_CustWin =
    LOGICAL (1ShowWarehouse:SCREEN-VALUE IN FRAME CustQuery).
END.
```

All the required qualifiers make this statement a bit more complex than it otherwise might be:

- You have to identify `m_Warehouse_Info` as a MENU-ITEM.
- You have to qualify the item as being IN MENU `m_CustWin`.
- You have to identify the `1ShowWarehouse` toggle box as being IN FRAME `CustQuery`.

Using OWNER, PARENT, FIRST-CHILD, and NEXT-SIBLING

In this section, you complete the menu item for the sample window.

- To define a CHOOSE trigger for the **Help About** item:

```
DO:
  DEFINE VARIABLE cChildren AS CHARACTER NO-UNDO.
  DEFINE VARIABLE hChild   AS HANDLE   NO-UNDO.
  hChild = SUB-MENU m_Navigation:FIRST-CHILD IN MENU m_CustWin.
  DO WHILE VALID-HANDLE(hChild):
    cChildren = cChildren + hChild:LABEL + ",".
    hChild = hChild:NEXT-SIBLING.
  END.

  MESSAGE "Here's some information about this menu:" SKIP
    "The menu's owner is " MENU m_CustWin:OWNER:TITLE SKIP
    "This item's label is "
      MENU-ITEM m_Help_About:LABEL IN MENU m_CustWin SKIP
    "This item's parent object is "
      MENU-ITEM m_Help_About:PARENT:LABEL IN MENU m_CustWin SKIP
    "The Navigation submenu's children are " cChildren.
END.
```

This displays some information about the menu to show you this information:

- The menu itself has the window as an OWNER, not a PARENT.
- Each menu item has attributes, such as its LABEL, that you can query and set as you can for other objects.

- Each menu item and submenu has a PARENT attribute that leads you up through the menu hierarchy.
- Each parent object has a FIRST-CHILD attribute. Using that and the NEXT-SIBLING attribute of each child, you can identify all the objects at any level in the menu tree.

Figure 9–3 shows what you see when you run the window and select **Help→Help About**.



Figure 9–3: Sample menu information message

Looking at the code

Now look at the statements the AppBuilder generated when you built the menu. You can also see this code in the **Code Preview** window.

First the code defines each submenu:

```
/* Menu Definitions */  
DEFINE SUB-MENU m_Navigation  
    MENU-ITEM m_First      LABEL "First"          ACCELERATOR "CURSOR-UP"  
    MENU-ITEM m_Next       LABEL "Next"           ACCELERATOR "CURSOR-RIGHT"  
    MENU-ITEM m_Prev       LABEL "Prev"            ACCELERATOR "CURSOR-LEFT"  
    MENU-ITEM m_Last       LABEL "Last"            ACCELERATOR "CURSOR-DOWN".  
  
DEFINE SUB-MENU m.Buttons  
    MENU-ITEM m_No_FirstLast LABEL "No &First/Last"  
    MENU-ITEM m_No_Prev     LABEL "No &Prev"  
    MENU-ITEM m_Allow_All   LABEL "Allow &All"  
    RULE  
    SUB-MENU   m_Navigation  LABEL "Navigation" .  
  
DEFINE SUB-MENU m_Show  
    MENU-ITEM m_Warehouse_Info LABEL "Warehouse Info" TOGGLE-BOX.  
  
DEFINE SUB-MENU m_Help  
    MENU-ITEM m_Help_About   LABEL "Help About" .
```

Once all the submenus are defined, the code defines the menu bar itself:

```
DEFINE MENU m_CustWin MENUBAR
    SUB-MENU  m.Buttons      LABEL "Buttons"
    SUB-MENU  m.Show         LABEL "Show"
    SUB-MENU  m.Help          LABEL "Help" .
```

After the window is defined, the menu bar is attached to the window:

```
ASSIGN {&WINDOW-NAME}:MENUBAR = MENU m_CustWin:HANDLE.
```

Defining a pop-up menu

Nearly any visual object, with a few exceptions such as images and rectangles, can have a pop-up menu that you activate with the right mouse button. The pop-up menu uses the same syntax as the menu bar, except that you omit the keyword MENUBAR:

```
DEFINE MENU menu-name
[ { LIKE other-menu } | menu-element-descriptor . . . ].
```

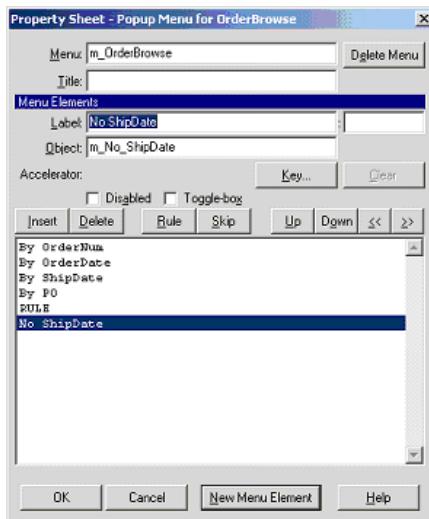
Pop-up menus can contain menu items, submenus, rules, and skips.



To define a pop-up menu in the sample window:

1. Go into the object's property sheet and select the **Pop-Up Menu** button . This brings up the same menu property sheet as before.
2. Define a pop-up menu for the **Order** browse and call it **m_OrderBrowse**.
3. Define menu items called **By OrderNum**, **By OrderDate**, **By ShipDate**, and **By PO**.
4. Following these items, add a RULE.

5. Add one more item called **No ShipDate**:



6. Define a CHOOSE trigger for each item.

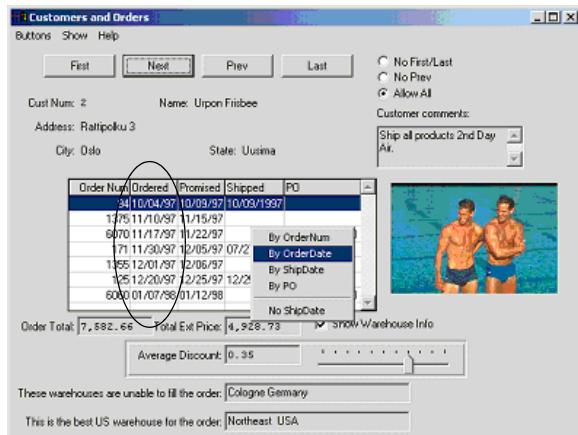
For the first four menu items, you want to sort the **Orders** by the appropriate field by reopening the **Order** query with a different BY clause. This is the trigger for **By_OrderNum**:

```
DO:
  OPEN QUERY OrderBrowse FOR EACH Order OF Customer
    BY Order.OrderNum.
END.
```

7. Define similar triggers for **BY OrderDate**, **BY PO**, and **BY ShipDate**.
8. For the final item, you want to filter the query to show only those **Orders** with no **ShipDate**:

```
DO:
  OPEN QUERY OrderBrowse FOR EACH Order OF Customer
    WHERE Order.ShipDate = ?.
END.
```

9. To try out your new pop-up menu, **Run** the window and then right-click on the browse to bring up the pop-up:



Character-mode considerations

Here is some information that will be useful to you if you need to run your application in character-mode:

In a character interface, use the ENTER-MENUBAR key function (usually mapped to **F3**) to give focus to the menu bar. You can then use the left and right arrow keys to position to a submenu label within the menu. Press **RETURN** to pull down the menu and the up and down arrow keys to position to a menu item. Press **RETURN** again to choose the menu item.

The choice of key labels for accelerators is limited by the keys available on your keyboard. For example, some terminals have no **F2** key, but might have a **PF2** key or require a **CTRL** key combination to emulate common **F2** key functionality.

For portability, you can use the **KBLABEL** function to specify the accelerator key label, as shown in this example:

```
MENU-ITEM m_Help LABEL "Help Menu" ACCELERATOR KBLABEL("HELP")
```

This function lets you use a portable Progress key function, such as **HELP**, to identify a valid key label for the current keyboard or terminal type. The key label is one that is associated with the key function for the terminal type in the current environment. On Windows, the current environment might reside in the registry or in an initialization file.

However, note that the current environment can (and often does) define more than one key label for a given Progress key function. The KBLABEL function returns the first such definition specified for the terminal type. For example, some terminal types define **F2** and also **ESC+?** as the HELP key function. In this case, using KBLABEL("HELP") changes **F2** to a menu item accelerator (losing its HELP key function), but leaves **ESC-?** as the one remaining HELP key function.

There are two types of invalid menu accelerators:

- Those specified by using a key label that is unsupported by the terminal.
- Those specified with supported key labels, that when invoked, generate unrecognized keyboard codes.

If you specify an unsupported key label, such as **ALT** in **ALT-F9**, Progress substitutes a supported key label in its place. Thus, Progress might substitute **ESC-F9** for **ALT-F9** as the accelerator definition.

However, an accelerator like **ESC-F9** might not work at run time. In this case, **ESC-F9** generates unrecognizable keyboard code sequences that fail to fire the appropriate Progress event. The terminal might also indicate the failure with a beep or other warning signal.

When in doubt, specify the menu accelerator using the terminal-portable KBLABEL function.

Conclusion

You have learned how to add many different kinds of visual objects to your application windows in this chapter.

In the next chapter, you'll return to the nonvisual aspects of the language, and you'll learn more about how to define and use queries. This will prepare you for doing more with the browse object, which uses a query as its source for the result set it displays.

10

Using Queries

The programming syntax you have learned so far uses blocks of 4GL code to define and iterate through a set of records. Progress defines an alternative to this form of data access called a *query*. You've seen queries in action already, in the chapter where you looked at the syntax the AppBuilder generates to define a default result set for a window and to display its records in a browse. This chapter discusses why these different forms exist and how you can use queries in ways that are distinct from how you would use a result set in a FOR EACH block.

This chapter includes the following sections:

- [Why you use queries in your application](#)
- [Defining and using queries](#)

Why you use queries in your application

The first question to answer about queries is why Progress has them at all. For many years, the Progress 4GL did not support queries, and developers wrote very powerful and complete applications without them. So, before you go into the details of how to define and use queries, you'll learn the differences between queries and the block-oriented data access language you've used so far.

Queries versus block-oriented data access

The first and most obvious characteristic of queries is precisely that they are not block-oriented. The language statements you've used in the last few chapters are all tied to blocks of 4GL code. You define a result set inside a block beginning with DO, FOR, or REPEAT. The result set is generally scoped to the block where it is defined. The term *result set* denotes the set of rows that satisfy a query.

You learned which of the block types iterate through the result set automatically and which require you to explicitly find the next record.

Even the FIND statement itself, although it does not define a block, is subject to the same rules of record scoping. You've learned how the presence of record-oriented blocks like a FOR EACH block and FIND statements together define the scope of a record buffer within a procedure.

These are among the most powerful features in the Progress 4GL. They help give it its unique flexibility and strength for defining complex business logic in a way that ties data access statements closely to the logic that uses the data.

However, there are times when you don't want your data access tied to the nested blocks of 4GL logic in your procedures. Earlier chapters briefly discussed the notion of event-driven applications. The examples you built in the chapter on the AppBuilder are a small but representative example of this. Think about the procedure `h-CustOrderWin1.w`, for instance. When you run this, a **Customer** and its **Orders** are displayed for you, as shown in Figure 10-1.

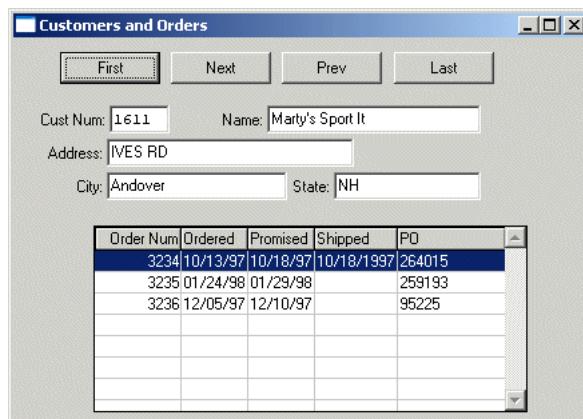


Figure 10–1: Customers and Orders sample window

Iterating through the data, for example through all the **Customers** in New Hampshire, isn't done inside a block of 4GL logic. It's done entirely under user control. Defining the **Customer** result set and its **Order** result set using queries is essential to this. When the user chooses the **Next** button, the code retrieves and displays the next record in the result set:

```
DO:  
  GET NEXT CustQuery.  
  IF AVAILABLE Customer THEN  
    DISPLAY Customer.CustNum Customer.Name Customer.Address Customer.City  
          Customer.State  
    WITH FRAME CustQuery IN WINDOW CustWin.  
    {&OPEN-BROWSERS-IN-QUERY-CustQuery}  
  END.
```

This happens independently of the block of code where the query is defined, or where it is opened. This gives you great flexibility as a developer to let user interface events, or other programmatic events, determine the flow of the application and the processing of data. By contrast, the examples you built in Chapter 7, “Record Buffers and Record Scope,” were chunks of logic that executed independently of the user interface, such as in the procedure `h-BinCheck.p`, which does some calculations and returns the results to another procedure.

This is the essence of the difference between queries and block-oriented data access. You use queries when you need to keep the data access separate from the structure of the procedure, and instead control it by events. You use block-oriented data access when you need to define business logic for data within a defined scope.

Thus queries give your data access language these important characteristics:

- **Scope independence** — You can refer to the records in the query anywhere in your procedure.
- **Block independence** — You aren't required to do all your data access within a given block.
- **Record retrieval independence** — You can move through the result set under complete control of either program logic or user events.
- **Repositioning flexibility** — You can position to any record in the result set at any time.

Using queries to share data between procedures

A query is also a true object in Progress. It has a definition and a name, and you can use the name to access it anywhere in your procedure. You have already learned a little about handles, which give you a reference to an object that you can pass from procedure to procedure. Using a query's handle, you can access the query and its result set from anywhere in your application session. This gives you the ability to modularize your application in ways that can't be done with block-oriented result sets, which are *not* named objects and which have no meaning or visibility outside their defined scope. You'll learn a lot more about how to use a query's handle to access its data in later chapters that discuss dynamic data retrieval language.

Using queries to populate a browse

Another thing you observed in the chapters on the AppBuilder and the code it generates is that a query is the basis for the data definition for a browse object. You can't populate a browse with the data from a `FOR EACH` block, only from a query. The browse gives you a viewport into the query. The repositioning you can do by selecting a record in the browse or scrolling through its contents reflects the repositioning that you can do programmatically through the query. In fact, selecting a row in a browse automatically repositions the query to that row, making it the current row in the buffers the query uses.

The chapters on graphical objects didn't cover the browse control for this very reason: you need to know more about defining and using queries before you can use the browse. You'll look at some of the many capabilities of the browse after you know how to define the queries they use.

Defining and using queries

There is a `DEFINE` statement for a query just as there is for other Progress objects. This is the general syntax:

```
DEFINE QUERY query-name FOR buffer [ , . . . ] [ SCROLLING ].
```

The statement gives the query a name and, in turn, names the buffer or buffers that the query uses. In the simplest case, the buffers are database table names. But you can also use other buffers you've defined as alternatives to table names or, as you'll learn later in this chapter, buffers for temporary tables.

If you want to reposition within the result set without using the `GET FIRST`, `NEXT`, `PREV`, and `LAST` statements, you need to define the query as `SCROLLING`. You'll learn later in this section how to reposition within the result set of a scrolling query. You must also define a query that is going to be associated with a browse as `SCROLLING`. There is a slight performance cost to using the `SCROLLING` option, so you should leave it off if you are not using the capabilities it enables.

You don't actually specify the exact data selection statement for the buffers and the tables they represent until you open the query. At that time, you can describe the joins between tables and any other parts of a `WHERE` clause that filter the data from the tables. As with other `DEFINE` statements, nothing actually happens when Progress encounters the `DEFINE QUERY` statement. No data is retrieved. Progress simply registers the query name and sets up storage and a handle for the query itself as an object.

OPEN and CLOSE QUERY statements

To get a query to retrieve data, you need to open it. When you open it, you specify the name of the query and a FOR EACH statement that references the buffers you named in the query definition, in the same order. If the query is already open, Progress closes the current open query and then reopens it. This is the general syntax:

```
OPEN QUERY query-name [ FOR | PRESELECT ] EACH record-phrase [ , . . . ]
[ BY phrase ].
```

The syntax of the *record-phrase* is generally the same as the syntax for FOR EACH statements. If you use the PRESELECT EACH phrase instead of the FOR EACH phrase, then all the records that satisfy the query are selected and their row identifiers pre-cached, just as for a PRESELECT phrase in an ordinary data retrieval block. However, there are a few special cases for the record phrase in a query:

- The first record phrase must specify EACH, and not FIRST, because the query is intended to retrieve a set of records. It is, however, valid to specify a WHERE clause in the *record-phrase* for the table that resulted in only a single record being selected, so a query can certainly have only one record in its result set. The *record-phrase* for any other buffers in the query can use the FIRST keyword instead of EACH if that is appropriate.
- You cannot use the CAN-FIND keyword in a query definition. Doing so results in a compile-time error.
- Queries support the use of an outer join between tables, using the OUTER-JOIN keyword, as explained below. FOR EACH statements outside of a query do not support the use of OUTER-JOIN.

Using an outer join in a query

An outer join between tables is a join that does not discard records in the first table that have no corresponding record in the second table. For example, consider this query definition:

```
DEFINE QUERY CustOrd FOR Customer, Order.
OPEN QUERY CustOrd FOR EACH Customer, EACH Order OF Customer.
```

As Progress retrieves records to satisfy this query, it first retrieves a **Customer** record and then the first **Order** record with the same **CustNum** field. When you do a NEXT operation on the query, Progress locates the next **Order** for that **Customer** (if there is one), and replaces the contents of the **Order** buffer with the new **Order**. If there are no more **Orders** for the **Customer**, then Progress retrieves the next **Customer** and its first **Order**.

The question is: What happens to a **Customer** that has no **Orders** at all? The **Customer** does not appear in the result set for the query. The same is true for a FOR EACH block with the same record phrase. This is simply because the record phrase asks for **Customers** and the **Orders** that match them, and if there is no matching **Order**, then the **Customer** by itself does not satisfy the record phrase.

In many cases this is not the behavior you want. You want to see the **Customer** data regardless of whether it has any **Orders** or not. In this case, you can include the OUTER-JOIN keyword in the OPEN QUERY statement:

```
DEFINE QUERY CustOrd FOR Customer, Order.  
OPEN QUERY CustOrd FOR EACH Customer, EACH Order OF Customer OUTER-JOIN.
```

Now Progress retrieves **Customers** even if they have no **Orders**. When the **Customer** has no **Orders**, the values for all fields in the **Order** buffer have the unknown value.

Sorting the query results

You can specify a BY phrase on your OPEN QUERY statement just as you can in a FOR EACH block. In this case, Progress either uses an index to satisfy the sort order if possible or, if no index can allow Progress to retrieve the data in the proper order, preselects and sorts all the query results before any data is made available to the application.

GET statements

You use a form of the GET statement to change the current position within the record set that the OPEN QUERY statement defines. You've already seen these statements on the button triggers in the h-CustOrderWin1.w procedure you built in [Chapter 4, “Introducing the OpenEdge AppBuilder.”](#) This is the syntax for the GET statement:

```
GET [ FIRST | NEXT | PREV | LAST | CURRENT ] query-name.
```

The query must be open before you can use a GET statement. If the query involves a join, Progress populates all the buffers used in the query on each GET statement. As noted earlier, a record can remain current through multiple GET statements if a second table in the query has multiple records matching the record for the first table. This would be the case for a **Customer** and its **Orders**, for example. A series of GET NEXT statements leaves the same **Customer** record in its buffer as long as there is another matching **Order** record to read into the **Order** buffer.

When you first open a query, it is positioned in effect before the first record in the result set. Either a GET FIRST or a GET NEXT statement positions to the first record in the result set.

If you execute a GET NEXT statement when the query is already positioned on the last record, then the query is positioned effectively beyond the end of the result set. A GET PREV statement then repositions to the last record in the result set. Likewise, a GET PREV statement executed when already on the first record results in the query being positioned before the first record, and a GET FIRST or GET NEXT statement repositions to the first record. When the query is repositioned off the beginning or off the end of the result set, no error results. You can use the AVAILABLE function that you're already familiar with to check whether you have positioned beyond the result set. For example, this code opens a query and cycles through all the records in its result set, simply counting them as it goes:

```
DEFINE QUERY CustOrd FOR Customer, Order.  
DEFINE VARIABLE iCount AS INTEGER      NO-UNDO.  
  
OPEN QUERY CustOrd FOR EACH Customer, EACH Order OF Customer.  
GET FIRST CustOrd.  
DO WHILE AVAILABLE(Customer):  
    iCount = iCount + 1.  
    GET NEXT CustOrd.  
END.  
DISPLAY iCount.
```

Figure 10–2 shows the result.



Figure 10–2: Result of GET statement example

The GET FIRST statement is needed to make a record available before the IF AVAILABLE statement is first encountered. Otherwise, the AVAILABLE test would fail before the code ever entered the loop.

Note also that the AVAILABLE function must take a buffer name as its argument, not the name of the query. If the query involves an outer join, then you should be careful about which buffer you use in the AVAILABLE function. If you name a buffer that could be empty because of an outer join (such as an empty **Order** buffer for a **Customer** with no **Orders**), then your loop could terminate prematurely. On the other hand, you might want your application logic to test specifically for the presence of one buffer or another in order to take special action when one of the buffers has no record.

Using the QUERY-OFF-END function

There is a built-in Progress function that you can use for the same purpose as the AVAILABLE statement:

```
QUERY-OFF-END ( query-name ).
```

QUERY-OFF-END is a logical function that returns `true` if the query is positioned either before the first result set row or after the last row, and `false` if it is positioned directly on any row in the result set. The `query-name` parameter must be either a quoted literal string with the name of the query or a variable name that has been set to the name of the query. In this way, you can use the statement programmatically to test potentially multiple different active queries in your procedure. For example, here is the same procedure used above, this time with the QUERY-OFF-END function in place of AVAILABLE:

```
DEFINE QUERY CustOrd FOR Customer, Order.  
DEFINE VARIABLE iCount AS INTEGER      NO-UNDO.  
  
OPEN QUERY CustOrd FOR EACH Customer, EACH Order OF Customer.  
GET FIRST CustOrd.  
DO WHILE NOT QUERY-OFF-END('CustOrd'):  
    iCount = iCount + 1.  
    GET NEXT CustOrd.  
END.  
DISPLAY iCount.
```

The difference between QUERY-OFF-END and AVAILABLE is simply that AVAILABLE requires a buffer name as a parameter, whereas QUERY-OFF-END requires a query name. If you use the AVAILABLE function with the name of the first buffer in the query, it is equivalent to using QUERY-OFF-END with the query name. Just for stylistic reasons, it is more appropriate to use the QUERY-OFF-END function in most cases, since it is the position of the query and not the presence of a record in a particular buffer that you're really interested in. By contrast, if you really want to test for the presence of a record, especially when your query does an outer join that might not always retrieve a record into every buffer, then use the AVAILABLE function.

Closing a query

When you are done with a query, you should close it using this statement:

```
CLOSE QUERY query-name.
```

An OPEN QUERY statement automatically closes a query if it was previously open. For this reason, it isn't essential to execute a CLOSE QUERY statement just before reopening a query. However, you should explicitly close a query when you are done with it and you are not immediately reopening it. This frees the system resources used by the query. After you close the query you cannot reference it again (with a GET statement, for instance). However, if there are records still in the buffer or buffers used by the query, they are still available after the query is closed unless your application has specifically released them.

Determining the current number of rows in a query

You can use the NUM-RESULTS function to determine how many rows there are in the current results list:

```
NUM-RESULTS ( query-name )
```

This INTEGER function returns the number of rows currently in the query's results list. As with the QUERY-OFF-END function, the *query-name* is an expression, which can be either the quoted name of the query or a variable containing the name.

The phrase “currently in the query's results list” requires some explanation. The *results list* is a list of the row identifiers of all rows that satisfy the query, and that have already been retrieved.

Progress normally builds up the results list as you walk through the query using the GET statement. Therefore, when you first open a query with a FOR EACH clause in the OPEN QUERY statement, the results list is empty and the NUM-RESULTS function returns zero.

As you move through the query using the GET NEXT statement, Progress adds each new row's identifier to the results list and increments the value returned by NUM-RESULTS. For example, this example retrieves all the **Customers** in the state of Louisiana using a query. For each row, it displays the **Customer Number**, **Name**, and the value of NUM-RESULTS:

```
DEFINE QUERY CustQuery FOR Customer.  
  
OPEN QUERY CustQuery FOR EACH Customer WHERE State = "LA".  
GET FIRST CustQuery.  
DO WHILE NOT QUERY-OFF-END("CustQuery"):  
    DISPLAY Customer.CustNum Customer.NAME  
        NUM-RESULTS("CustQuery") LABEL "Rows"  
        WITH FRAME CustFrame 15 DOWN.  
    GET NEXT CustQuery.  
    DOWN WITH FRAME CustFrame.  
END.
```

When you run the procedure, you see the value of NUM-RESULTS change as each new row is retrieved, as shown in [Figure 10–3](#).

The screenshot shows a Windows application window titled "Procedure Editor - Run". The main area is a table with two columns: "Cust Num" and "Name". The "Cust Num" column contains values from 1291 to 1815, and the "Name" column lists various sporting goods companies. To the right of the table, there is a vertical column labeled "Rows" with values 1 through 13, corresponding to each row in the table. At the bottom of the window, a message says "Procedure complete. Press space bar to continue.".

| Cust Num | Name | Rows |
|----------|------------------------------|------|
| 1291 | Aaa Sporting Goods Inc | 1 |
| 1292 | Southern Sporting Goods | 2 |
| 1798 | Bell's Sporting Goods | 3 |
| 1799 | Chaney's Not Just Soccer | 4 |
| 1803 | Mickey's Sporting Goods | 5 |
| 1804 | Club Soccer | 6 |
| 1805 | New Orleans Sports | 7 |
| 1806 | Sports Avenue | 8 |
| 1807 | Cenla Sports Inc | 9 |
| 1808 | Champs Sports | 10 |
| 1809 | Songy's Sporting Goods Inc | 11 |
| 1810 | Bill's Guns & Sporting Goods | 12 |
| 1815 | Dickie's Sportsman's Ctr | 13 |

Figure 10–3: Result of NUM-RESULTS example

Using a DOWN frame and the DOWN WITH statement

As a small digression, it is necessary to explain a couple of statements in this example that you haven't seen before. They illustrate one of the key characteristics about queries: there's no built-in block scoping or iteration in a query. First, here's the new phrase on the DISPLAY statement:

```
WITH FRAME CustFrame 15 DOWN.
```

You learned a little about down frames in [Chapter 2, “Using Basic 4GL Constructs,”](#) and [Chapter 5, “Examining the Code the AppBuilder Generates.”](#) A *down frame* is a frame that can display more than one row of data, each showing the same fields for a different record in a report-like format. In the examples you wrote in earlier chapters, you didn't have to specify the DOWN phrase to indicate how many rows the frame should have. Progress gave you a down frame with a default number of rows automatically.

Why doesn't it do that in this case? Because a query is not associated with a particular block, and doesn't have any automatic iteration, Progress doesn't know how the data is going to be displayed. So by default, it just gives you a one down frame that displays a single record.

The second new piece of syntax is this statement at the end of the block:

DOWN WITH FRAME CustFrame.

No, this is not a political protest slogan! Rather, it tells Progress to display a row in the frame **CustFrame**, and then to position down a row in the frame before displaying the next row. If you don't use this statement, then even if you define the frame to be 15 DOWN, all the rows are displayed on top of each other on the first row of the frame. Once again, this is because Progress doesn't know how you're going to display the data. It does not associate iteration through a result set with your DO block automatically, as it would with a FOR EACH block. Therefore, you have to tell it what to do.

Retrieving query results in advance

The value of NUM-RESULTS does not always increment as you execute GET NEXT statements and operate on each row, however. There are various factors that force Progress to retrieve all the results in advance of presenting you with any data.

One of these is under your direct control: the PRESELECT option on the OPEN QUERY statement. When you use a PRESELECT EACH rather than a FOR EACH statement to define the data selection, you are telling Progress to retrieve all the records that satisfy the query in advance and to save off their record identifiers in temporary storage. Then Progress again retrieves the records using their identifiers as you need them. As discussed in “[OPEN and CLOSE QUERY statements](#)” section on page 10–6, you typically use the PRESELECT option to make sure that the set of records is not disturbed by changes that you make as you work your way through the list, such as changing a key value in such a way as to change a record’s position in the list.



To see visible evidence of the effect of the PRESELECT keyword in your OPEN QUERY statement:

1. Change the OPEN QUERY statement in the sample procedure:

```
OPEN QUERY CustQuery PRESELECT EACH Customer WHERE State = "LA".
```

2. Run the procedure again to see the different value of NUM-RESULTS:

The screenshot shows a Windows-style application window titled "Procedure Editor - Run". Inside, there is a table with two columns: "Cust Num" and "Name". The "Cust Num" column contains values from 1291 to 1815. The "Name" column lists various sporting goods companies. To the right of the table, the word "Rows" is followed by the number 13, indicating the total number of records. At the bottom of the window, a message reads "Procedure complete. Press space bar to continue.".

| Cust Num | Name | Rows |
|----------|------------------------------|------|
| 1291 | Aaa Sporting Goods Inc | 13 |
| 1292 | Southern Sporting Goods | 13 |
| 1798 | Bell's Sporting Goods | 13 |
| 1799 | Chaney's Not Just Soccer | 13 |
| 1803 | Mickey's Sporting Goods | 13 |
| 1804 | Club Soccer | 13 |
| 1805 | New Orleans Sports | 13 |
| 1806 | Sports Avenue | 13 |
| 1807 | Cenla Sports Inc | 13 |
| 1808 | Champs Sports | 13 |
| 1809 | Songy's Sporting Goods Inc | 13 |
| 1810 | Bill's Guns & Sporting Goods | 13 |
| 1815 | Dickie's Sportsman's Ctr | 13 |

All the records are pre-retrieved. Therefore, the value of NUM-RESULTS is the same no matter what record you are positioned to. This means that you could use the PRESELECT option to display, or otherwise make use of, the total number of records in the results list before displaying or processing all the data.

Another factor that can force Progress to pre-retrieve all the data is a sort that cannot be satisfied using an index.

**To see an example of this:**

1. Change the OPEN QUERY statement back to use a FOR EACH block and then try sorting the data in the query by the **Customer Name**:

```
OPEN QUERY CustQuery FOR EACH Customer WHERE State = "LA" BY Name.
```

2. Run the query:

The screenshot shows a Windows application window titled "Procedure Editor - Run". Inside, there is a table with two columns: "Cust Num" and "Name". The "Cust Num" column contains customer numbers from 1291 to 1806. The "Name" column contains their respective company names. To the right of the table, there is a vertical column labeled "Rows" with values 1 through 13. At the bottom of the window, a message reads "Procedure complete. Press space bar to continue.".

| Cust Num | Name | Rows |
|----------|------------------------------|------|
| 1291 | Aaa Sporting Goods Inc | 1 |
| 1798 | Bell's Sporting Goods | 2 |
| 1810 | Bill's Guns & Sporting Goods | 3 |
| 1807 | Cenla Sports Inc | 4 |
| 1808 | Champs Sports | 5 |
| 1799 | Chaney's Not Just Soccer | 6 |
| 1804 | Club Soccer | 7 |
| 1815 | Dickie's Sportsman's Ctr | 8 |
| 1803 | Mickey's Sporting Goods | 9 |
| 1805 | New Orleans Sports | 10 |
| 1809 | Songy's Sporting Goods Inc | 11 |
| 1292 | Southern Sporting Goods | 12 |
| 1806 | Sports Avenue | 13 |

The **Name** field is indexed, so Progress can satisfy the **BY** phrase and present the data in the sort order you want by using the index to traverse the database and retrieve the records.

3. By contrast, try sorting on the **City** field:

```
OPEN QUERY CustQuery FOR EACH Customer WHERE State = "LA" BY City.
```

4. Add the **City** field to the DISPLAY list and rerun the procedure to see the result:

| Cust Num | Name | City | Rows |
|----------|----------------------|------------|------|
| 1815 | Dickie's Sportsman's | Addis | 13 |
| 1807 | Cenla Sports Inc | Alexandria | 13 |
| 1808 | Champs Sports | Alexandria | 13 |
| 1809 | Songy's Sporting Goo | Houma | 13 |
| 1292 | Southern Sporting Go | Jefferson | 13 |
| 1810 | Bill's Guns & Sporti | Jonesville | 13 |
| 1803 | Mickey's Sporting Go | Kaplan | 13 |
| 1805 | New Orleans Sports | Kenner | 13 |
| 1806 | Sports Avenue | Kenner | 13 |
| 1804 | Club Soccer | Kenner | 13 |
| 1291 | Aaa Sporting Goods I | Kenner | 13 |
| 1799 | Chaney's Not Just So | Lafayette | 13 |
| 1798 | Bell's Sporting Good | Lafayette | 13 |

Procedure complete. Press space bar to continue.

There is no index on the **City** field, so Progress has to retrieve all 13 of the records for **Customers** in Louisiana in advance to sort them by the **City** field before presenting them to your procedure. Therefore, NUM-RESULTS is equal to the total number of records from the beginning, as soon as the query is opened.

Identifying the current row in the query

As you move through the results list, Progress keeps track of the current row number, that is, the sequence of the row in the results list. You can retrieve this value using the CURRENT-RESULT-ROW function:

`CURRENT-RESULT-ROW (query-name)`

The function returns an INTEGER value with the sequence of the current row. The *query-name* is an expression, either a quoted query name or a variable reference.

For CURRENT-RESULT-ROW to work properly, you must define the query to be SCROLLING. If you don't define the query as SCROLLING, the CURRENT-RESULT-ROW function returns a value, but that value is not reliable.

To use CURRENT-RESULT-ROW, make these changes to your sample procedure:

```
► DEFINE QUERY CustQuery FOR Customer SCROLLING.

OPEN QUERY CustQuery FOR EACH Customer WHERE State = "LA".
GET FIRST CustQuery.

DO WHILE NOT QUERY-OFF-END("CustQuery"):
    DISPLAY Customer.CustNum Customer.NAME
        NUM-RESULTS("CustQuery") LABEL "Rows"
        CURRENT-RESULT-ROW("CustQuery") LABEL "Row#"
    WITH FRAME CustFrame 15 DOWN.
    GET NEXT CustQuery.
    DOWN WITH FRAME CustFrame.
END.
```

When you run the procedure, you see that the value of CURRENT-RESULT-ROW keeps pace with NUM-RESULTS, as shown in [Figure 10–4](#).

The screenshot shows a software interface titled "Procedure Editor - Run". Inside, there is a table with four columns: "Cust Num", "Name", "Rows", and "Row#". The data consists of 13 rows, each containing a customer number, name, the value "1", and the current row number from 1 to 13 respectively. A message at the bottom of the window reads "Procedure complete. Press space bar to continue."

| Cust Num | Name | Rows | Row# |
|----------|------------------------------|------|------|
| 1291 | Aaa Sporting Goods Inc | 1 | 1 |
| 1292 | Southern Sporting Goods | 2 | 2 |
| 1798 | Bell's Sporting Goods | 3 | 3 |
| 1799 | Chaney's Not Just Soccer | 4 | 4 |
| 1803 | Mickey's Sporting Goods | 5 | 5 |
| 1804 | Club Soccer | 6 | 6 |
| 1805 | New Orleans Sports | 7 | 7 |
| 1806 | Sports Avenue | 8 | 8 |
| 1807 | Cenla Sports Inc | 9 | 9 |
| 1808 | Champs Sports | 10 | 10 |
| 1809 | Songy's Sporting Goods Inc | 11 | 11 |
| 1810 | Bill's Guns & Sporting Goods | 12 | 12 |
| 1815 | Dickie's Sportsman's Ctr | 13 | 13 |

Figure 10–4: Result of CURRENT-RESULT-ROW example

This is not always the case, of course. If you use the PRESELECT option or a nonindexed sort to retrieve the data, then NUM-RESULTS is always 13, as you have seen. But the value of CURRENT-RESULT-ROW changes from 1 to 13 just as it does above.

You can use the CURRENT-RESULT-ROW function to save off a pointer to reposition to a specific row. See the “[Using a RowID to identify a record](#)” section on page 10–20 for information on how to identify a record.

Here are a few special cases for CURRENT-RESULT-ROW:

- If the query is empty, the function returns the unknown value (?).
- If the query is explicitly positioned before the first row, for example by executing a GET FIRST followed by a GET PREV, then the function returns the value 1.
- If the query is explicitly positioned after the last row, for example by executing a GET LAST followed by a GET NEXT, then the function returns the value one more than the number of rows in the results list.

Using INDEXED-REPOSITION to improve query performance

If you anticipate jumping around in the result set using statements such as GET LAST, you should add another option to the end of your OPEN QUERY statement: the INDEXED-REPOSITION keyword. If you do this, your DEFINE QUERY statement must also specify the SCROLLING keyword.

If you don’t open the query with INDEXED-REPOSITION, then Progress retrieves all records in sequence in order to satisfy a request such as GET LAST. This can be very costly. If you do use INDEXED-REPOSITION, Progress uses indexes, if possible, to jump directly to a requested row, greatly improving performance in some cases. There are side effects to doing this, however, in terms of the integrity of the results list, as discussed next.

Factors that invalidate CURRENT-RESULT-ROW and NUM-RESULTS

Under some circumstances, when you open your query with the INDEXED-REPOSITION keyword, the value of CURRENT-RESULT-ROW or NUM-RESULTS becomes invalid. As explained earlier, the results list holds the row identifiers for those rows that satisfy the query and that have already been retrieved.

Thirteen rows satisfy the query for **Customers** in Louisiana, so the value of these two functions goes as high as 13 for that query. When you do a PRESELECT or a nonindexed sort, all the rows have already been retrieved before any data is presented to you, so NUM-RESULTS is 13 at the beginning of the DISPLAY loop. Normally, Progress adds the identifiers for all the rows it retrieves to the results list, but there are circumstances where this is not the case. If you execute a GET LAST statement on a query, and your OPEN QUERY statement does not use a PRESELECT or a sort that forces records to be pre-retrieved, Progress jumps directly to the last record using a database index, without cycling through all the records in between. In this case, it has no way of knowing how many records would have been retrieved between the first one and the last one, and it cannot maintain a contiguous results list of all rows that satisfy the query. For this reason, Progress flushes and reinitializes the results list when you jump forward or backward in the query. So after a GET LAST statement, NUM-RESULTS returns 1 (because the GET LAST statement has retrieved one row) and CURRENT-RESULT-ROW is unknown (because there is no way to know where that row would fit into the full results list).

Repositioning a query

Often you need to reposition a query other than to the first, last, next, or previous row. You might need to jump to a row based on data the user entered or return to a row that you previously saved off. Or you might want to jump forward or backward a specific number of rows to simulate paging through the query. You can do all these things with the REPOSITION statement, which has this syntax:

```
REPOSITION query-name
{ | TO ROW row-number
  | FORWARDS n
  | BACKWARDS n
  | TO ROWID buffer-1-rowid [, . . .] [ NO-ERROR ]
}
```

The *query-name* in this case is not an expression. It can only be an unquoted query name, not a variable.

If you specify the TO ROW option followed by an integer expression, the query repositions to that sequential position within the results list. If you have previously saved off that position using the CURRENT-RESULT-ROW function, you can use the value that function returned as the value in the TO ROW phrase to reposition to that row.

If you use the FORWARD or BACKWARD phrase, you can jump forward or backward any number of rows, specified by the *n* integer expression. You can use the FORWARD or BACKWARD keywords instead of FORWARDS or BACKWARDS.

The last of the REPOSITION options requires an explanation of a Progress data construct you haven't seen before.

Using a RowID to identify a record

Every record in every table of a database has a unique row identifier. (Technically, the row identifier is only unique within a single storage area of a database. Since an entire database table must be allocated to a single storage area, this effectively makes the identifier unique at least within that table. A discussion of database constructs such as storage areas is beyond the scope of this book.)

The identifier is called a *RowID*. There is both a ROWID data type that allows you to store a row identifier in a procedure variable and a ROWID function to return the identifier of a record from its record buffer.

Generally, you should consider a RowID to be a special data type without being concerned about its storage format. The RowID is (among other things) designed to be valid, not just for the OpenEdge database, but for all the different databases you can access from the 4GL using OpenEdge DataServers, which provide access from the 4GL to database types such as Oracle and Microsoft SQLServer.

In fact, you can't display a RowID directly in a Progress 4GL procedure. If you try to, you get an error. You can see a RowID by converting it to a CHARACTER type using the STRING function. For instance, here is a procedure that shows you the RowIDs of the rows that satisfy the sample query you've been working with:

```
DEFINE QUERY CustQuery FOR Customer SCROLLING.

OPEN QUERY CustQuery FOR EACH Customer WHERE State = "LA".
GET FIRST CustQuery.

DO WHILE NOT QUERY-OFF-END("CustQuery"):
    DISPLAY Customer.CustNum Customer.NAME
        CURRENT-RESULT-ROW("CustQuery") LABEL "Row#"
        STRING(ROWID(customer)) FORMAT "x(12)" LABEL "RowId"
        WITH FRAME CustFrame 15 DOWN.
    GET NEXT CustQuery.
    DOWN WITH FRAME CustFrame.
END.
```

Figure 10–5 shows the result.

The screenshot shows a Windows application window titled "Procedure Editor - Run". Inside, there is a table with three columns: "Cust Num", "Name", "Row# RowId". The data consists of 13 rows of customer information, each with a unique RowID value. The table has horizontal and vertical scroll bars. At the bottom of the window, a message reads "Procedure complete. Press space bar to continue."

| Cust Num | Name | Row# RowId |
|----------|------------------------------|---------------|
| 1291 | Aaa Sporting Goods Inc | 1 0x000001bb |
| 1292 | Southern Sporting Goods | 2 0x000001bc |
| 1798 | Bell's Sporting Goods | 3 0x000003f5 |
| 1799 | Chaney's Not Just Soccer | 4 0x000003f6 |
| 1803 | Mickey's Sporting Goods | 5 0x000003fa |
| 1804 | Club Soccer | 6 0x000003fb |
| 1805 | New Orleans Sports | 7 0x00000400 |
| 1806 | Sports Avenue | 8 0x00000401 |
| 1807 | Cenla Sports Inc | 9 0x00000402 |
| 1808 | Champs Sports | 10 0x00000403 |
| 1809 | Songy's Sporting Goods Inc | 11 0x00000404 |
| 1810 | Bill's Guns & Sporting Goods | 12 0x00000405 |
| 1815 | Dickie's Sportsman's Ctr | 13 0x0000040a |

Figure 10–5: Result of RowID example

The RowID is displayed as a hexadecimal value. The values you would see in your own copy of the Sports2000 database might be different from these, and certainly they would be different if you modified the data, dumped it, and reloaded it into the database, because the RowID reflects the actual storage location of the data for the record, and this is not in any way predictable or necessarily repeatable. You should never count on a RowID as a permanent identifier for a record. However, you can use a RowID if you need to relocate a record you have previously retrieved within a procedure and whose RowID you saved off. This is what the TO ROWID phrase in the REPOSITION statement is for.

Even in this case, you must be aware that in the event of a record deletion, it is possible that a new record could be created that has the same RowID as the record that was deleted. So, even within a single session a RowID is not an absolutely reliable pointer to a record. In addition, RowIDs are unique only within a single database storage area. Therefore, the same RowID might occur for records in different tables that happen to be in different storage areas.

With these conditions in mind, you can use the TO ROWID phrase to reposition to a record in a query. Note that the RowID is for a particular database record, not an entire query row, so you need to save off the RowID of the record buffer, not of the query name, to reuse it. And in the case of a query with a join between tables, you need to save the RowID of each record buffer in order to reposition to it later and restore each of the records in the join.

The NO-ERROR option in this phrase lets you suppress an error message if the RowID turns out to be invalid for any reason. You could then use the AVAILABLE function or the ERROR-STATUS handle (see [Chapter 17, “Managing Transactions”](#)) to determine whether the query was successfully repositioned.

There is another similar identifier in Progress called a RECID. This identifier was used in earlier versions of Progress to identify records in the same way as RowIDs do now. The RECID is essentially an integer identifier for a record, though it has its own data type. It is still supported but for several reasons (including, but not limited to, portability of code between database types that you can access with DataServers), it is strongly recommended that you use only the RowID form in new application code. The Progress 4GL continues to support RECsIDs mainly for backward compatibility with older applications that still use them.

Positioning details with the REPOSITION statement

So if you execute a REPOSITION statement that repositions TO ROW 5 or FORWARDS 10 or TO ROWID rRow, then your procedure is positioned to that row so that you can display it or otherwise use it, right? Well, not exactly. When you use the REPOSITION statement, Progress positions the query in between records, so that you then have to execute a GET NEXT or GET PREV statement to position on a row so that you can actually use it. Here are some of the specifics:

- When you execute a REPOSITION statement that uses the TO ROWID or TO ROW phrase, the query is positioned before the record requested. You then need to execute a GET NEXT statement to make the row you want available.
- When you execute a REPOSITION *query-name* BACKWARDS statement, the query is positioned between records, and you need to execute a GET NEXT statement to make the row you intend available. For example, if the query is positioned to row 6 of a query results list and you execute a REPOSITION *query-name* BACKWARDS 2, then the query is positioned effectively between results list rows 3 and 4, and a GET NEXT statement makes row 4 available. A GET PREV statement makes row 3 available.
- When you execute a REPOSITION statement with the FORWARDS phrase, you are actually positioned *beyond* the record you might expect. For example, if you are on row 6 and issue a statement with FORWARDS 2, then the query is positioned between row 8 and row 9, and you need to then execute a GET PREV statement to make row 8 available or a GET NEXT statement to make row 9 available.

Note as well that there is only one way for Progress to know which row is five rows ahead of the current row, or five rows behind it, or the fifth row of the result set, and that is to walk through the query row by row until it gets to the one you want. If the row in question has already been retrieved and is in the results list, then Progress can reposition to the row you want very quickly using the results list. Therefore, the REPOSITION statement with any of the options TO ROW, FORWARDS, or BACKWARDS maintains the integrity of the results list and the functions that use it. The REPOSITION TO ROWID statement also maintains the integrity of the list if the row you want has already been retrieved and the results list has not been flushed since you retrieved it, because Progress can scan the results list to locate the RowID.

Extending the sample window to use the queries

Now you can try out some of the things you've learned about queries.



To extend the sample application window to use some query statements:

1. Open the procedure h-CustOrderWin1.w.

This gives you the test procedure with **Customer** fields and the **Order** browse but without some of the other graphical objects you added later.

2. Extend the window somewhat to the right, then drop two buttons onto the window.
3. Name the first button **PosButton** and give it a label of **Save Position**.
4. Name the second button **ReposButton** and give it a label of **Restore Position**.
5. Open the **Section Editor** and select the **Definitions** section.
6. Define a variable to hold a value for the **Customer** query's CURRENT-RESULT-ROW:

```
/* Local Variable Definitions --- */  
DEFINE VARIABLE iQueryRow AS INTEGER      NO-UNDO.
```

Remember that the **Definitions** section is scoped to the entire procedure file, so anything you define here is available to any trigger or internal procedure inside it.

7. Go into the **Triggers** section for the PosButton and give it this CHOOSE trigger:

```
DO:
  iQueryRow = CURRENT-RESULT-ROW('CustQuery').
END.
```

8. Define this CHOOSE trigger for the ReposButton:

```
DO:
  REPOSITION CustQuery TO ROW iQueryRow.
  APPLY "CHOOSE" TO BtnNext.
END.
```

When you choose the **Save Position** button, your code saves off the current row number from the results list. You can then move around in the query. When you press the **Restore Position** button, the **Customer** query is repositioned to the row you saved, and the **Order** query is opened for that **Customer**.

Why did you need the `APPLY "CHOOSE" TO BtnNext` ? statement. Remember that when you use the `REPOSITION` statement Progress positions before the record you want, so you need to execute a `GET NEXT` to make it available. At the same time, in this case, your code needs to reopen the dependent **Order** query for the **Customer** as well. All this is done by the trigger code on the **Next** button.

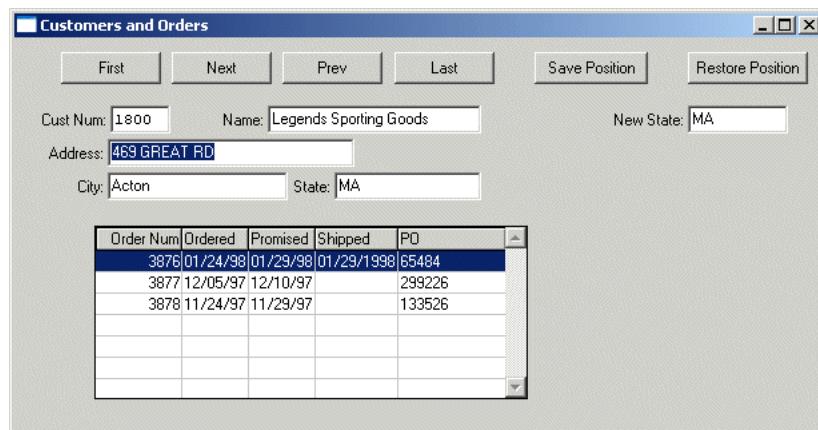
9. Run the procedure and test saving off and restoring the current row.
10. Add a fill-in field to give you a reason to test reopening the query with a different WHERE clause:
- Drop a fill-in on the window. Call it **cState** and give it a **label** of **New State**.
 - Write this LEAVE trigger for the fill-in field in the **Section Editor**:

```
DO:
  OPEN QUERY CustQuery FOR EACH Customer WHERE State =
    cState:SCREEN-VALUE.
  APPLY "CHOOSE" TO BtnFirst.
END.
```

Each time you enter a state abbreviation in the field and tab out of it, the trigger fires and the query is reopened with that new state. Remember that it isn't necessary to close a query explicitly if you are immediately going to reopen it, so a CLOSE QUERY *CustQuery* statement here is optional. Also remember that unless you specifically assign it using the ASSIGN statement, the value the user types into the fill-in exists only in the frame's screen buffer, so you can retrieve it using the SCREEN-VALUE attribute of the field.

Similarly for the CHOOSE trigger that repositions to the previously saved row, you need to apply CHOOSE to the **First** button trigger to get the first row for the new query and reopen the **Order** query.

11. Run your procedure. You can enter a state name and tab out of the **New State** field and see the **Customers** in that state along with their **Orders**:



If you enter an invalid state name, the procedure doesn't give you any feedback to confirm this. In the next section, you make a few more changes to the procedure to check whether there were any results for the state that is entered. This makes use of the NUM-RESULTS function and also introduces you to another new and useful 4GL statement.

Using NUM-RESULTS to check the validity of the query

The query is sorted by the **City**, which is a nonindexed field. Therefore, the NUM-RESULTS function returns the total number of records that satisfy the query as soon as it is opened, because they all have to be retrieved before they can be sorted.

Next, you add another field to the screen to display that number, and then check the same value in the trigger for the New State field to make sure that the state entered was valid.



To add another field to the screen:

1. Drop another fill-in field onto the window. Name it **iMatches** and give it a **Label** of **Number of Matches**.

To display initial values for the **New State** and **Number of Matches** fields, you'll put some code into the **Main Block** of the procedure, to be executed right after the query is initially opened.

2. In the **Section Editor**, select the **Main Block** and add this code after the `RUN enable_UI` statement:

```

DO ON ERROR    UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
    ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:
        RUN enable_UI.
ASSIGN cState = Customer.State
        iMatches = NUM-RESULTS("CustQuery").
DISPLAY cState iMatches WITH FRAME CustQuery.
END.

```

This displays the initial value of the **Customer State** (“NH”) along with the number of **Customers** in New Hampshire. Remember that, in this case, *CustQuery* is both the name of the query and also the name of the frame the AppBuilder created for you.

3. Edit the **LEAVE** trigger for the **New State** field again. Add a statement before the **OPEN QUERY** statement to save off the current **State** in the **cState** field, and a statement following the **OPEN QUERY** statement to retrieve the **NUM-RESULTS** value for the new query:

```

cState = Customer.State.
OPEN QUERY CustQuery FOR EACH Customer WHERE Customer.State =
    cState:SCREEN-VALUE BY Customer.City.
    /* Check whether the State was valid or not. */
    iMatches = NUM-RESULTS("CustQuery").

```

Remember that the value you assign to **cState** is not overwritten by the **SCREEN-VALUE** that you type into the field on the screen, so you can open the query based on the **SCREEN-VALUE** and still keep the old value around if you need it later (which you will).

Using the MESSAGE statement

Next you need to display a warning message if there are no results, which means that no **Customer** records match the state value that was entered. Progress has a MESSAGE statement to display a message to the screen or to a message area at the bottom of the window if there is one. A MESSAGE statement can contain one or more character expressions (quoted literals or CHARACTER variables) that make up the message. If there are multiple expressions in the statement, Progress simply concatenates them all, with a single space between them. You can also insert the SKIP keyword anywhere in the message to skip a line, or SKIP(*n*) to skip *n* lines.

If you want the message to appear in its own alert box, you can include the phrase VIEW-AS ALERT-BOX in the statement. This is the default if the message is displayed from a GUI window. If you want to give the message alert box a special format, you can include one of the QUESTION, INFORMATION, ERROR, or WARNING keywords after the VIEW-AS ALERT-BOX phrase. The MESSAGE statement has other features, including the ability to display different sets of buttons, beyond just an **OK** button to acknowledge the message. It can also capture an answer from the message and store it in a variable. You can see all the syntax details in *OpenEdge Development: Progress 4GL Reference* or in the online help topic for the MESSAGE statement.



To display a warning message that tells you that there are no matching Customers:

1. In the **New State** trigger, add this code:

```
IF iMatches = 0 THEN
DO:
  MESSAGE "There are no Customers that match the State"
  cState:SCREEN-VALUE "." SKIP
  "Restoring the previous State."
  VIEW-AS ALERT-BOX WARNING.
```

2. Reopen the query using the value of the **State** field that you saved off in the **cState** variable and redisplay the previous valid **New State** value:

```
/* Reopen the query with the previous state. */
OPEN QUERY CustQuery FOR EACH Customer WHERE Customer.State =
cState BY Customer.City.
/* Display the previous valid state as well. */
DISPLAY cState WITH FRAME CustQuery.
END.
```

This ends the DO block that is executed if NUM-RESULTS is zero.

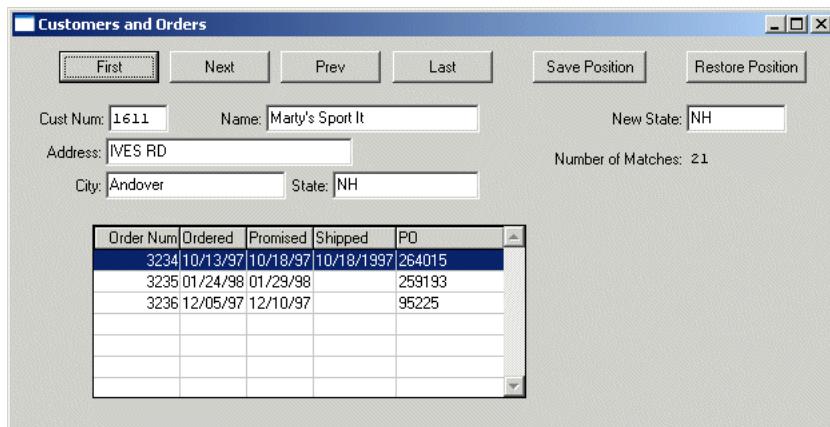
3. To display the number of matching **Customers** if NUM-RESULTS is not zero, add the following code:

```
ELSE DISPLAY iMatches WITH FRAME CustQuery.
```

4. Add the following code so that, regardless of whether the **New State** was valid, the statement executes APPLY "CHOOSE" to the **First** button to do a GET FIRST and opens the **Order** query:

```
APPLY "CHOOSE" TO BtnFirst.
```

5. Save the procedure as h-CustOrderWin4.w.
6. Run the procedure to see the number of matching records for any **State** you enter:



7. To see your warning message, enter an invalid **State** value, such as QQ:



Summary

This ends the discussion of using queries in your application. In this chapter, you've learned how to define and use queries and what the differences are between queries and other Progress 4GL data access statements.

The next chapter discusses how to define and use temporary tables. At the end of the next chapter, you'll be using queries and temp-tables together.

Defining and Using Temp-tables

Another important Progress language construct is the temporary table (temp-table). A *temp-table* gives you nearly all the features of a database table, and you can use a temp-table in your procedures almost anywhere you could reference a database table. However, temp-tables are not persistent. They aren't stored anywhere permanently. And they are also private to your own Progress session, the data you define in them can't be seen by any other users. You can define sets of data using temp-tables that don't correspond to tables and fields stored in your application database, which gives you great flexibility in how you use them. You can also use a temp-table to pass a whole set of data from one procedure to another, or even from one Progress session to another. Together, queries and temp-tables provide much of the basis for how data is passed from one application module to another. They are essential to creating distributed applications, with data and business logic on a server machine and many independent client sessions running the user interface of the application. By the time you complete this chapter, you'll understand how to use these features in your applications.

This chapter includes the following sections:

- [Using temporary tables in your application](#)
- [Using include files to duplicate code](#)
- [Adding an Order Line browse to the Customer window](#)

Using temporary tables in your application

Temporary tables, or *temp-tables*, are effectively database tables that Progress stores in a temporary database. You use them when you need temporary storage for multiple rows of data during a session, and when you need to pass data between OpenEdge sessions on different machines, such as between a server and a client using the OpenEdge AppServer. Temp-tables exist only for the duration of the procedure that defines them or, at most, for the duration of a OpenEdge session. A temp-table is private, visible only to the session that creates it or receives it as a parameter passed from another session. But because temp-tables use the same support code that actual database tables use, you can take advantage of almost all database features that don't require persistence of data and multi-user access to data, for example defining indexes for fields in the temp-table.

Temp-tables are a wonderfully useful construct in the Progress 4GL, and you can apply them for a many different kinds of programming problems. Fundamentally, you can think of them as providing two basic capabilities:

1. Temp-tables allow you to define a table within a session that does not map to any single database table. A temp-table can be *based on* a database table, but you can then add fields that represent calculations or fields from other tables or any other type of data source. Or you can define a temp-table that is not related in any way to any database table, that is used for example to store calculations requiring a two-dimensional table of data that is then used in a report or in some other way.
2. Temp-tables allow you to pass data between OpenEdge sessions. When you begin to build distributed applications, which you can do with your own AppServer statements in the 4GL or with the Progress Dynamics framework, you allow your application to be divided between procedures that are on the same machine with your application database and procedures that execute on a client machine that produces the user interface and interacts with the user. You use the OpenEdge AppServer™ product to communicate between client OpenEdge sessions and server sessions. When a client procedure needs to get data from the server, it runs a 4GL procedure on the server that returns data as an OUTPUT parameter, or that accepts updates from the client as an INPUT parameter. You define these parameters as temp-tables that you pass between the sessions. You cannot pass record buffers as parameters between sessions, so whether you need to pass a single row or many rows together, you do this in the form of a temp-table as a parameter.

Because the client session is sending and receiving only temp-tables and not dealing directly with database records, your client sessions can run without a database connection of any kind, which makes it easier to build flexible and efficient distributed applications. Temp-tables are therefore a fundamental part of any distributed application. Writing your own 4GL statements to use the OpenEdge AppServer is beyond the scope of this book, but in this chapter you'll write a procedure that passes a temp-table as a parameter to a user interface program, so that you can begin to get a feel for what it means to design your procedures for this kind of separation of user interface from database access and business logic.

Progress work-tables

Before you proceed, you should know that there is a variation on the temp-table supported in Progress called a *work-table*. You use a `DEFINE WORK-TABLE` statement to define one. This is an older feature that predates temp-tables and lacks some of their features. Most important, a work-table is memory-bound, and you must make sure your session has enough memory to hold all of the records you create in it. In addition, you cannot create indexes on work-tables. You also cannot pass a work-table as a parameter between procedures. Although you could use a work-table for an operation that needed to define a small set of temporary records that don't need a lot of memory and don't need an index, it is better to think of work-tables as being fully superseded by temp-tables. You should use the newer temp-tables for all of your temporary data storage needs.

The temporary database for temp-tables

The temporary database for temp-tables is largely invisible to you. You can design and use temp-tables as if they were entirely memory resident, even though Progress automatically flushes temp-table records to disk when this is necessary. The temp-table database is stored in whatever you have designated as your *temporary directory*. By default, this is your working directory, but you can change this by specifying a temporary directory with the `-T` startup option to your OpenEdge session. If you need to adjust the amount of buffer space that Progress uses in memory before it writes temp-table to disk, you can use the `-Bt` startup option to set this size. You can learn more about these and other startup options in [OpenEdge Deployment: Startup Command and Parameter Reference](#).

Defining a temp-table

You define a temp-table using the DEFINE TEMP-TABLE statement. There are two basic ways to define the fields and indexes in the table. You can make the temp-table LIKE some single database table (or even like another temp-table that you've already defined), which gives it all the fields and indexes from the other table, or you can define fields and indexes individually. You can also do both, so that in a single statement you can define a temp-table to be LIKE another table but also to have additional fields and indexes. Here is the basic syntax for the statement:

```
DEFINE TEMP-TABLE temp-table-name
  [ LIKE table-name [ USE-INDEX index-name [ AS PRIMARY ] ] . . .
  [ FIELD field-name { AS data-type | LIKE field-name } [ field-options ]
  ]
  .
  .
  [ INDEX index-name [ IS [ UNIQUE ] [ PRIMARY ] ]
  { index-field [ ASCENDING | DESCENDING ] } . . .
  ]
```

You can see a description of the entire statement in the online help or in *OpenEdge Development: Progress 4GL Reference*.

Defining fields for the temp-table

If you use the LIKE option on your temp-table definition, the temp-table inherits all the field definitions for all the fields in the other table it is defined to be LIKE. The definitions include all of these attributes:

- The field name.
- The field's data type and, for an array field, its extent.
- The field's initial value.
- The field's label and column-label.
- The field's format and, for a DECIMAL field, the number of decimals.

- The field's help text, if any.
- The field's font and color.

Whether you use the **LIKE** option to base your temp-table on another table or not, you can also use the **FIELD** phrase to define one or more fields for the temp-table. If you use the **LIKE** option, then any fields you define with the **FIELD** phrase are additional fields beyond the fields inherited from the other table. If you don't use the **LIKE** option, then these are the only fields in the temp-table.

For fields you define individually with the **FIELD** phrase, you must specify at least the field name and data type, or a **LIKE** phrase to define the field to be **LIKE** a field from another table. If you use the **LIKE** phrase on an individual field, your field inherits all of its attributes by default. You can override any of the field attributes of an inherited field except for the name, data type, and array extent, by using the appropriate keywords such as **LABEL**, **FORMAT**, and **INITIAL**. You can find a complete description of these keywords under the Field Options topic in the online help. You can also define the same attributes for fields that aren't inherited from another table using the same keywords on the **FIELD** phrase.

You can use the **FIELD** phrase for one of three purposes:

- If you want to base your temp-table on another table, but you don't want all of its fields or you want to change some of the field attributes, including even the field names, then you can name the individual fields using the **FIELD** phrase rather than making the whole table **LIKE** the other table. You then specify each field to be **LIKE** a field from the other table, possibly with a different name or modified attributes. If you only need to change certain display attributes of some of the fields, but otherwise want to use all the fields from the other table, you can use the **LIKE** phrase on the temp-table definition to base it on the other table. You then modify the field attributes in the definition of the browse or frame fields where they are displayed. This makes your temp-table definition simpler than if you explicitly named each field just to change a few attributes of some of the fields.
- If you want to base your temp-table on another table, but you want some additional fields from one or more other tables as well, then you can define the temp-table to be **LIKE** the other table that it uses all the fields from, and then add **FIELD** phrases for each field that comes from another related table. You can use the **LIKE** keyword on these additional fields to inherit their database attributes as well.
- If you need fields in your table that are not based at all on specific database fields, then you define them with the **FIELD** phrase. Again, you can do this whether the basic temp-table definition is **LIKE** another table or not.

You can define temp-tables with all of the Progress data types that you can use for database fields (CHARACTER, DATE, DECIMAL, INTEGER, LOGICAL, RECID, and RAW). In addition, you can define a temp-table field to be of type ROWID, which is something you can't do with fields in database tables. You could use such a field to store the RowID of a record you read from a database, in order to use the RowID to relocate the record later in the procedure.

Defining indexes for the temp-table

If you use the `LIKE other-table` option for the temp-table, your temp-table is based on the other table you name. In this case, the following index rules apply:

- If you don't specify any index information in the `DEFINE TEMP-TABLE` statement, the temp-table inherits all the index layouts defined for the other table.
- If you specify one or more `USE-INDEX` options, the temp-table inherits only the indexes you name. If one of those indexes is the primary index of the other table, it becomes the primary index of the temp-table, unless you explicitly specify `AS PRIMARY` for another you index you define.
- If you specify one or more `INDEX` options in the definition, then the temporary table inherits only those indexes from the other table that you have named in a `USE-INDEX` phrase.

The primary index is the one Progress uses by default to access and order records. Progress determines the primary index in this way:

- If you specify `AS PRIMARY` in a `USE-INDEX` phrase for an index inherited from the other table, then that is the primary index.
- If you specify `IS PRIMARY` on an `INDEX` definition specific to the temp-table, then that becomes the primary index.
- If you inherit the primary index from the other table, then that becomes the primary index for the temp-table.
- The first index you specify in the temp-table definition, if any, becomes the primary index.
- If you don't specify any index information at all, then Progress creates a default primary index that sorts the records in the order in which they're created.

If you do not use the `LIKE` option to use another table as the basis for your temp-table, then your temp-table only has indexes if you define them with the `INDEX` phrase in the definition.

You should use the same considerations you would use for a database table in determining what indexes, if any, to define or inherit for a temp-table. On the one hand, there is a small cost to creating index entries on update that could become measurable if you have multiple indexes with multiple fields, or if you are creating large numbers of records in a performance-intensive situation where many records are created between user actions. You shouldn't create or use indexes your procedure won't need.

On the other hand, if you know that you need to access the records in the temp-table based on specific field values, then you should create or inherit the right indexes to let Progress locate the records you need without having to read through the whole temp-table to find them. The OpenEdge database is tremendously efficient and you might find that it can locate specific records in your table based on nonindexed field values so quickly that you hardly notice the lack of an index for those fields. However, it is a good idea to define one or more indexes if your temp-table is going to have more than a handful of records and if you know which fields will be used to locate, filter, or sort records in the table. If your code doesn't need to access the records in the temp-table in *any* order other than the order in which they were created, then you don't need an index other than the default index Progress gives you automatically.

Temp-table scope

Generally speaking, the *scope* of a temp-table is the procedure in which it's defined. In fact, you can *only* define a temp-table at the level of the whole procedure, not within an internal procedure. When the procedure terminates, any temp-tables defined in it are emptied and deleted. Likewise, a temp-table is *visible* only within the procedure that defines it. If that procedure passes the temp-table to another procedure, the other procedure has to have its own definition of the temp-table, and it obtains a copy of the temp-table when it receives it as an INPUT parameter. (It is possible to send a temp-table to another procedure that has no prior definition of the temp-table, but instead receives the definition along with the temp-table data, but this involves the use of dynamic temp-tables, which you'll learn about in [Chapter 20, “Creating and Using Dynamic Temp-tables and Browses.”](#))

There are keywords to define a temp-table to be SHARED between procedures, and also to make the scope and visibility of the temp-table GLOBAL to the entire session. [Chapter 13, “Advanced Use of Procedures in Progress,”](#) discusses this since you need to think about the relationship between different procedures in an OpenEdge application to understand properly when you should and shouldn't use shared or global objects such as temp-tables.

You can also pass the handle to a temp-table from one procedure to another within a session, to avoid actually copying the temp-table between procedures. [Chapter 20, “Creating and Using Dynamic Temp-tables and Browses,”](#) discusses this, along with other uses of handles. For the purposes of this introduction to temp-tables, think of them as being statically defined and scoped to a single procedure.

Temp-table buffers

When you reference a database table in a Progress procedure, Progress provides you with a default buffer with the same name as the table. In this way, when you write a statement, such as `FIND FIRST Customer`, you are actually referring to a buffer with the name **Customer**. You can, of course, define additional buffers explicitly that have different names from the database table.

The same is true of temp-tables. When you define a temp-table, Progress provides you with a default buffer of the same name. Just as for database tables, you can define additional buffers with other names if you like. When you refer to the temp-table name in a statement such as `FIND FIRST ttCust`, you are referring to the default buffer for the temp-table just as you would be for a database table.

There is a temp-table attribute, `DEFAULT-BUFFER-HANDLE`, that returns the handle of the default buffer.

Using a temp-table to summarize data

The beginning of this chapter notes two basic purposes for temp-tables: first, to let you define a table unlike any single database table for data summary or other uses; and second, to let you pass a set of data as a parameter between procedures. In this section, you'll work through an example of the first kind. You'll write a procedure that defines a temp-table and uses it to total invoice amounts for each **Customer**, and at the same time to count the number of **Invoices** for each **Customer** and identify which one has the highest amount. The finished procedure is saved as `h-InvSummary.p`.

In addition to the **Customer** table you're familiar with, the example uses the **Invoice** table in the **Sports2000** database. The **Invoice** table holds information for each **Invoice** a **Customer** has been sent for each **Order**. It has a join to the **Customer** table and to the **Order** table, along with the date and amount of the **Invoice** and other information.

First is the statement to define the temp-table itself:

```
/* Procedure h-InvSummary.p -- uses a temp-table to build a summary report
of invoices by customer. */

DEFINE TEMP-TABLE ttInvoice
  FIELD iCustNum    LIKE Invoice.CustNum  LABEL "Cust#"  FORMAT "ZZ9"
  FIELD cCustName   LIKE Customer.NAME   FORMAT "X(20)"
  FIELD iNumInvs    AS INTEGER        LABEL "# Inv's"   FORMAT "Z9"
  FIELD dInvTotal   AS DECIMAL       LABEL "Inv Total" FORMAT ">>,>>9.99"
  FIELD dMaxAmount  AS DECIMAL       LABEL "Max Amount" FORMAT ">>,>>9.99"
  FIELD iInvNum     LIKE Invoice.InvoiceNum LABEL "Inv#"  FORMAT "ZZ9"
INDEX idxCustNum IS PRIMARY iCustNum
INDEX idxInvTotal dInvTotal.
```

The procedure creates one record in the temp-table for each **Customer**, summarizing its **Invoices**. As you can see, the temp-table has these fields:

- A **Customer Number** field derived from that field in the **Invoice** table.
- A **Customer Name** field derived from that field in the **Customer** table. Later you'll use the **Customer Number** to retrieve the **Customer** record so that you can add the name to the invoice information.
- A count of the number of **Invoices** for the **Customer**.
- A total of the **Invoices** for the **Customer**.
- The amount of the largest **Invoice** for the **Customer**.
- The number of the **Invoice** with the largest amount for the **Customer**.

The field definitions define or override the field label and default format in some cases, using phrases attached to the FIELD definition. By default, the right-justified label for a numeric field extends somewhat to the right of the data, which in the case of the **InvTotal** and **MaxAmount** fields doesn't look quite right, so the extra spaces in their labels correct that.

The temp-table also has two indexes. The first orders the records by **Customer Number**. This index is useful because the code finds records based on that value to accumulate the **Invoice** total and other values. This is the primary index for the temp-table, so if you display or otherwise iterate through the temp-table records without any other specific sort, they appear in **Customer Number** order.

The second index is by the **Invoice Total**. This index is useful because the procedure uses it as the sort order for the final display of all the records.

The first executable code begins a FOR EACH block that joins each **Invoice** record to its **Customer** record. The OF phrase uses the **CustNum** field that the two tables have in common to join them. The FIND statement checks to see whether there is already a temp-table record for the **Customer**. If there isn't, it uses the CREATE statement to create one. You'll learn a lot more about the CREATE statement in Chapter 16, “Updating Your Database and Writing Triggers.” For now you just need to know that this statement creates a new record either in a database table or, as you see here, in a temp-table. That new record holds the initial values of the fields in the table until you set them to other values.

After the new record is created, the code sets the key value (the **iCustNum** field) and saves off the **Customer Name** from that table. The ASSIGN statement lets you make multiple field assignments at once and is more efficient than a series of statements that do one field assignment each:

```
/* Retrieve each invoice along with its Customer record, to get the Name. */
FOR EACH Invoice, Customer OF Invoice:
  FIND FIRST ttInvoice WHERE ttInvoice.iCustNum = Invoice.CustNum NO-ERROR.
  /* If there isn't already a temp-table record for the Customer,
  create it and save the Customer # and Name. */
  IF NOT AVAILABLE ttInvoice THEN
    DO:
      CREATE ttInvoice.
      ASSIGN ttInvoice.iCustNum = Invoice.CustNum
                     ttInvoice.cCustName = Customer.NAME.
    END.
```

Next, the code compares the **Amount** of the current **Invoice** with the **dMaxAmount** field in the temp-table record (which is initially **0** for each newly created record). If the current **Amount** is larger than that value, it's replaced with the new **Amount** and the **Invoice** number is saved off in the **iInvNum** field. In this way, the temp-table records wind up holding the highest **Invoice Amount** for the **Customer** after it has cycled through all the **Invoices**:

```
/* Save off the Invoice amount if it's a new high for this Customer. */
IF Invoice.Amount > dMaxAmount THEN
  ASSIGN dMaxAmount = Invoice.Amount
     iInvNum = Invoice.InvoiceNum.
```

Still in the FOR EACH loop, the code next increments the **Invoice** total for the **Customer** and the count of the number of **Invoices** for the **Customer**:

```
/* Increment the Invoice total and Invoice count for the Customer. */
ASSIGN ttInvoice.dInvTotal = ttInvoice.dInvTotal +
                           Invoice.Amount
ttInvoice.iNumInvs = ttInvoice.iNumInvs + 1.
END.      /* END FOR EACH Invoice & Customer */
```

Now the procedure has finished cycling through all of the **Invoices**, and it can take the summary data in the temp-table and display it, in this case with the **Customer** with the highest **Invoice Total** first:

```
/* Now display the results in descending order by invoice total. */
FOR EACH ttInvoice BY dInvTotal DESC:
  DISPLAY iCustNum cCustName iNumInvs dInvTotal iInvNum dMaxAmount.
END.
```

Figure 11–1 shows the first page of the output report you should see when you run the procedure.

| Cust# | Name | # Inv's | Inv Total | Inv# | Max Amount |
|-------|----------------------|---------|-----------|------|------------|
| 66 | First Down Football | 3 | 68,279.67 | 66 | 36,587.69 |
| 83 | Fallen Arch Running | 4 | 65,179.49 | 46 | 29,762.58 |
| 27 | Bumm Bumm Tennis | 5 | 45,317.29 | 116 | 19,075.61 |
| 1 | Lift Tours | 4 | 41,635.50 | 79 | 34,707.84 |
| 40 | Heikin Kuntosali | 4 | 40,875.47 | 64 | 26,912.81 |
| 65 | Lagt Kot Ligger | 1 | 40,439.43 | 114 | 40,439.43 |
| 34 | Quick Toss Lacrosse | 4 | 36,238.48 | 62 | 17,502.95 |
| 46 | Olympique marseilles | 5 | 33,281.60 | 118 | 22,266.26 |
| 30 | Fast Flipper Pinball | 4 | 28,288.32 | 18 | 14,530.46 |
| 82 | Second Skin Scuba | 3 | 28,113.96 | 144 | 23,947.60 |
| 75 | Hard Knocks Skating | 3 | 27,812.78 | 138 | 14,780.25 |
| 19 | Buffalo Shuffleboard | 2 | 27,657.58 | 141 | 25,236.78 |
| 9 | Pihtiputaan Pyora | 5 | 25,792.08 | 108 | 10,866.06 |
| 28 | Luistin ja Pyora Oy | 4 | 21,865.26 | 102 | 9,721.35 |
| 24 | Jazz Fulus Kauppa | 2 | 20,806.15 | 134 | 15,147.81 |
| 72 | Birdy's Badminton | 2 | 19,929.86 | 5 | 15,126.78 |
| 43 | Sub Par Golf | 4 | 19,675.60 | 100 | 5,597.71 |
| 14 | Paris St Germain | 2 | 19,427.80 | 20 | 19,108.20 |
| 26 | Bulls Eye Sports | 3 | 18,810.17 | 98 | 9,911.25 |
| 51 | Butternut Squash Inc | 3 | 18,564.52 | 83 | 11,975.88 |

Figure 11–1: First page result of h-InvSummary.p

Using the temp-table made it easy to accumulate different kinds of summary data and to combine information from different database tables together in a single report. You could easily display this data in different ways, for example sorted by different fields, without having to again retrieve it from the database.

Using a temp-table as a parameter

The second major use for temp-tables is to let you pass a set of data from one procedure to another as a parameter. In particular, this is useful when you need to pass a set of one or more records from one OpenEdge session to another. This book doesn't cover the sending of data between an AppServer session and a client session, but the next test procedure simulates this by building a temp-table in a procedure that has no user interface, and then passing the temp-table to a separate procedure that manages the UI. The UI procedure is the same **Customers and Orders** window you've been using. You'll add a second browse to it to display **Order Lines** for the currently selected **Order**.

Temp-table parameter syntax

The syntax you use to pass a temp-table as a parameter is special in order to identify the temp-table to Progress. In the RUN statement from the calling procedure, this is the parameter syntax:

```
[ INPUT ] TABLE temp-table-name
| ]{ INPUT-OUTPUT | OUTPUT } TABLE temp-table-name [ APPEND ]
```

If you're passing a temp-table as a parameter to another procedure, you must add the TABLE keyword before the temp-table name. As with other parameter types, the default direction of the parameter is INPUT.

If you use the APPEND option for an OUPUT or INPUT-OUTPUT temp-table, then the records passed back from the called procedure are appended to the end of the data already in the temp-table. Otherwise, the new data replaces whatever the contents of the temp-table were at the time of the call.

In the called procedure, you must define temp-table parameters in this way for an INPUT or INPUT-OUTPUT table:

```
DEFINE [ INPUT ] | INPUT-OUTPUT
PARAMETER TABLE FOR temp-table-name [ APPEND ].
```

Once again, INPUT is the default. If you use the APPEND option for an INPUT or INPUT-OUTPUT temp-table parameter to a called procedure, then the records passed in from the calling procedure are appended to the end of the data already in the local temp-table. Otherwise, the new data replaces whatever the contents of the temp-table were at the time of the call.

For an OUTPUT temp-table parameter returned from a called procedure, use this syntax:

```
DEFINE OUTPUT PARAMETER TABLE FOR temp-table-name.
```

An INPUT parameter moves data from the calling procedure to the called procedure at the time of the RUN statement. An OUTPUT parameter moves data from the called procedure to the calling procedure when the called procedure terminates and returns to its caller. An INPUT-OUTPUT parameter moves data from the calling procedure to the called procedure at the time of the RUN, and then back to the calling procedure when the called procedure ends.

You must define the temp-table in both procedures. The temp-table definitions must match with respect to the number of fields and the data type of each field (including the array extent if any). The field data types make up what is called the signature of the table.

Other attributes of the tables can be different. The field names do not have to match. The two tables do not need to have matching indexes, because Progress dynamically builds the appropriate indexes for the table when it is instantiated either as an INPUT parameter in the called procedure or as an OUTPUT parameter in the calling procedure. Other details, such as field labels and formats, also do not have to match.

Note: It is extremely important to keep in mind that when you pass a temp-table as a parameter from one procedure to another, whether those procedures are in the same OpenEdge session or not, Progress copies the temp-table to make its data available to the called procedure. This can be very expensive if the temp-table contains a large number of records. If you are making a local procedure call, you should simple pass the handle of the temp-table instead of the table itself whenever possible. This avoids copying the table and can provide a significant performance advantage.

Defining a procedure to return Order Lines

This section discusses a procedure that retrieves the **Order Lines** from the database. This sample procedure is called `h-fetch0lines.p`. It takes the current **Order Number** as an INPUT parameter. It then defines a temp-table with the fields from the **OrderLine** table plus the **Item Name** and the total **Weight** for the quantity ordered. Finally, it defines an OUTPUT parameter to return the set of **Order Lines** as a temp-table to the caller:

```
/*
 * h-fetch0lines.p -- retrieve all orderlines for an Order
 * and return them in a temp-table. */
DEFINE TEMP-TABLE tt0line LIKE OrderLine
  FIELD ItemName LIKE ITEM.ItemName
  FIELD TotalWeight AS DECIMAL LABEL "Tot.Wgt." .

DEFINE INPUT  PARAMETER piOrderNum AS INTEGER      NO-UNDO.
DEFINE OUTPUT PARAMETER TABLE FOR tt0line.
```

Notice that you must define the temp-table before you use it in a parameter definition.

Using the INPUT parameter that passed in the **Order Number**, the code retrieves all the **Order Lines** for that **Order**. For each one, it creates a temp-table record.

The procedure then needs to move all the **OrderLine** fields from the database record into the temp-table. In addition, it must copy the **Item Name** from the **Item** table and a calculation of the total weight for the **Order Line** based on the weight of the item and the quantity ordered.

You could do this with a long ASSIGN statement that names and assigns each field, but there's a shorter way to do this, using the BUFFER-COPY statement, as in this example:

```
FOR EACH OrderLine WHERE Orderline.OrderNum = piOrderNum,
  ITEM OF OrderLine:
    CREATE tt0line.
    BUFFER-COPY OrderLine TO tt0line
      ASSIGN tt0line.ItemName = Item.ItemName
          tt0line.TotalWeight = OrderLine.Qty * Item.Weight.
  END.
```

Using BUFFER-COPY to assign multiple fields

The BUFFER-COPY statement, as the name implies, copies like-named fields from one record buffer to another. You can extend the statement to either include or exclude fields from the copy, and to do other assignments of fields with different names and assignments involving expressions. Here is the syntax of the BUFFER-COPY statement:

```
BUFFER-COPY source-buffer [ { EXCEPT | USING } field . . . ]  
TO target-buffer [ ASSIGN assign-expression ... ] [ NO-ERROR ].
```

By default, the BUFFER-COPY statement copies all like-named fields from the *source-buffer* to the *target-buffer*. As with any other assignment, if the data types and extents of the fields are not compatible, an error results. Any fields in the *source-buffer* that don't have a like-named field in the *target-buffer* are not copied. Likewise, any fields in the *target-buffer* that don't have a like-named field in the *source-buffer* are ignored. No error results from this mismatch of fields.

If you want to copy only certain fields from the *source-buffer*, you can do this in one of two ways:

- If you use the EXCEPT option followed by a list of field names, then those fields in the *source-buffer* are left out of the copy.
- If you use the USING option followed by a list of field names, then only the fields in the list are copied.

Thus, you can use either option depending on which one lets you specify a shorter list of fields for special handling. You might also want to consider future code maintenance in making this choice. If you use the EXCEPT option, then any fields that you might add in the future to the tables you're copying will automatically be picked up when you recompile the procedure. If you use the USING option they will not be. The option you choose depends on your reasons for including or excluding fields and on what you want to have happen if the buffer definitions change.

In addition to like-named fields, you can do any other kind of assignment from the *source-buffer* to the *target-buffer* using an ASSIGN statement. As with the ASSIGN statements you've seen in examples so far, this is basically a list of assignments with the *target* field on the left side of an equal sign and the *source* field on the right. You can also include a WHEN phrase in an assignment to define a logical expression that determines whether the assignment is done. For more information, see the section for the ASSIGN statement in the online help or in *OpenEdge Development: Progress 4GL Reference*.

As with other statements, the NO-ERROR keyword suppresses any error messages that might result from improper assignment and lets you query the ERROR-STATUS handle afterwards instead.

Using include files to duplicate code

Now it's time for another digression to introduce you to a powerful feature of the 4GL that will be helpful in completing the temp-table example. You might have noticed that to pass a temp-table from one procedure to another, you need to have an equivalent temp-table definition in both procedures. So the next thing you have to do in the **Customers and Orders** window is define the same temp-table as you just did in `h-fetch01ines.p` so you can use it as an INPUT parameter to the RUN statement to that procedure. Surely there must be a better way to duplicate code than retyping it or coping and pasting it?

Progress provides a mechanism for just this purpose, called an *include file*. An include file is a source procedure that is included in another procedure file to form a complete procedure that you can then compile. Using include files saves you from having to copy and paste code from one procedure to another when you need the same definition or other code in multiple places. More important, it can also save you from many maintenance headaches when you need to change the duplicated code. Rather than having to identify all the places where the code exists and change every one of them consistently, you just have to change the one include file and recompile the procedures that use it.

Having said this, it's important to make an observation about include files in an OpenEdge application. In later chapters, you learn how to use dynamic language constructs to allow the same procedure to work many different ways. For example, you can use them to manage a query on different tables or with different WHERE clauses that are defined dynamically at run time or to build a window that can display fields and a browse for many different tables, with all the objects defined dynamically, so their attributes can be changed at run time. This limits the need for include files in many cases.

In earlier applications, written before these dynamic features existed, Progress developers used include files to pass in, for example, the name of a table or a list of fields or a WHERE clause for a FOR EACH statement. In this way, the same procedure could be compiled many different ways to do similar jobs. With dynamic programming, you can often modify the same kinds of procedure attributes at run time, allowing a single compiled procedure to handle all the cases that you need, without creating many different .r files for all the different cases, or compiling

and running procedures on the fly during program execution. For that reason, once you've learned about dynamic programming, if you find yourself doing extraordinarily complex or clever things with include files in a new application, you should consider whether you could do the same job more easily and more effectively with dynamic language constructs that are part of the language for exactly this purpose.

With these disclaimers aside, it's still valuable to know about include files and when you might use them. If nothing else, you'll find many include file references in existing Progress code, including code generated by the AppBuilder.

The syntax for an include file is extremely simple. You merely place the include filename, enclosed in braces ({ }) into your procedure exactly where you want the code it contains to be inserted. This can be anywhere at all in your procedure. An include file can contain a complete statement, a whole set of statements, a partial statement, a single keyword or name, or even a fraction of a name, depending on what purpose it serves. The only limitation is that Progress always inserts a single space after (but not before) the contents of the include file when it pulls it in during the compilation. This means you could append a partial name or keyword directly to the end of a code fragment in the main procedure, but not to the beginning without a space being inserted.

By convention, include files end in the `.i` extension, but this is not actually required. If Progress needs a pathname relative to the Propath to locate your include file, then you need to include the pathname within the braces just as you would specify a relative pathname as part of the filename in a RUN statement.

In the simplest case, an include file just inserts the contents of the file into the procedure that includes it when the main procedure is compiled. You can also define arguments to include files. To add an argument to an include file, place the string `{1}` into the include file where you want the argument inserted. Then, in each include file reference, add the value of the argument after the name of the include file, separated by a space. For example, in the next section you'll replace the four button triggers for the **First**, **Next**, **Prev**, and **Last** buttons with an include file. The code for all of those buttons is the same except for the one keyword in the GET statement FIRST, NEXT, PREV, and LAST. You can write all four triggers with one include file by adding an argument to the include file. If you call it `h-ButtonTrig1.i`, then the first line of code in `h-ButtonTrig1.i` is `GET {1} CustQuery`. And the trigger for the **First** button becomes `{h-ButtonTrig.i FIRST}`.

For multiple arguments to an include file, just mark them with placeholders `{1}`, `{2}`, and so on. You can also create named arguments that let you determine the arguments you pass into the include file when it becomes part of the compilation. For a complete description of include files and their arguments, see the Include File Reference topic in the online help.

Adding an Order Line browse to the Customer window

You've completed the h-fetch0lines.p procedure. When it ends, it returns a temp-table with the **OrderLines** for the **Order** in it, along with the name and weight total from the **Item** table.



To use an include file to put the same code in multiple procedures:

1. Open h-fetch0lines.p and copy the temp-table definition.
2. Paste it into a new procedure window and save the procedure as h-tt0line.i:

```
/* h-tt0line.i -- include file to define tt0line temp-table. */
DEFINE TEMP-TABLE tt0line LIKE OrderLine
  FIELD ItemName LIKE ITEM.ItemName
  FIELD TotalWeight AS DECIMAL LABEL "Tot.Wgt." .
```

3. Remove the code from h-fetch0lines.p (or just comment it out to remind you of what the include files replaces), and put in a reference to h-tt0line.i in its place:

```
/* h-fetch0lines.p -- retrieve all orderlines for an Order
and return them in a temp-table. */
{h-tt0line.i}
/* DEFINE TEMP-TABLE tt0line LIKE OrderLine
  FIELD ItemName LIKE ITEM.ItemName
  FIELD TotalWeight AS DECIMAL LABEL "Tot.Wgt." . */

DEFINE INPUT  PARAMETER piOrderNum AS INTEGER      NO-UNDO.
DEFINE OUTPUT PARAMETER TABLE FOR tt0line.

FOR EACH OrderLine WHERE Orderline.OrderNum = piOrderNum,
  ITEM OF OrderLine:
  CREATE tt0line.
  BUFFER-COPY OrderLine TO tt0line
    ASSIGN tt0line.ItemName = ITEM.ItemName
          tt0line.TotalWeight = OrderLine.Qty * ITEM.Weight.
END.
```

Remember that part of the purpose of this exercise is to give you a taste of what it is like to write separate Progress procedures for user interfaces and for data access and update code, so that you can ultimately distribute those procedures between client and server machines in a true enterprise application. In this simple example, both procedures run as part of the same session, but they could run more or less exactly the same in different OpenEdge sessions on completely different machines.



To add code to the `CustOrderWin` procedure to run `h-fetch0lines.p` and display the `OrderLines` in a browse of their own:

1. Open `h-CustOrderWin4.w` in the AppBuilder.
2. In the **Definitions** section, add a reference to `h-tt0line.i`:

```
{h-tt0line.i}
/* DEFINE TEMP-TABLE tt0line LIKE Orderline
   FIELD ItemName LIKE ITEM.ItemName
   FIELD TotalWeight AS DECIMAL LABEL "Tot.Wgt." . */
```

3. Add a query definition for a query on this temp-table:

```
DEFINE QUERY OlineQuery FOR tt0line.
```

When you first built this window, you let the AppBuilder generate the browse definition for the **Order** browse for you. This time you'll write it yourself. First of all, this is to let you say you wrote one browse definition in your life, even if you never write another one. And you might never have to, given that the AppBuilder can generate it for you, and that a standard framework like Progress Dynamics can generate the browses for your application dynamically.

But there is another reason for you to write the browse definition: the AppBuilder is not great at generating queries and browses based on temp-tables, simply because there is no database schema where you can look up the definitions. You can define them using a special design-time database called `temp-db`, but because you're here to learn the language, it's better just to write the definition yourself.

4. Put this browse definition for the temp-table after the other elements in the **Definitions** section. Select these fields from the **ttOline** table, including the **Item Name** and **Weight** from the **Item** table:

```
DEFINE BROWSE OlineBrowse
  QUERY OlineQuery NO-LOCK DISPLAY
    ttOline.Ordernum
    ttOline.LineNum LABEL "Line"
    ttOline.ItemNum LABEL "Item" FORMAT "ZZZ9"
    ttOline.ItemName FORMAT "x(20)"
    ttOline.TotalWeight
    ttOline.Price FORMAT "ZZ,ZZ9.99"
    ttOline.Qty
    ttOline.Discount
    ttOline.ExtendedPrice LABEL "Ext.Price" FORMAT "ZZZ,ZZ9.99"
  WITH NO-ROW-MARKERS SEPARATORS 7 DOWN ROW-HEIGHT-CHARS .57.
```

Remember that your BUFFER-COPY from the **OrderLine** table gives you all the fields from that table in your temp-table, but you don't have to use all of them in the browse.

When the AppBuilder generates a browse, it always gives it a specific size based on how you lay it out (generating the phrase SIZE x BY y). In this case, you define a number of rows to display instead with the 7 DOWN phrase and leave out the size altogether. That way, Progress displays it tall enough to show seven rows of data and wide enough to display all the fields you've selected.

Now you need to run the procedure that fetches the **Order Lines** each time a row is selected in the **Order** browse. The event that is fired when a row is selected is called VALUE-CHANGED.

5. In the **Section Editor**, define this trigger on VALUE-CHANGED of the **OrderBrowse**:

```
DO:
  RUN h-fetchOlines.p (INPUT Order.OrderNum, OUTPUT TABLE ttOline).
  OPEN QUERY OlineQuery FOR EACH ttOline.
  DISPLAY OlineBrowse WITH FRAME CustQuery.
  ENABLE OlineBrowse WITH FRAME CustQuery.
END.
```

The trigger code runs the procedure, passing in the current **Order** number from the **Order** buffer and getting a temp-table back. Then it opens the **OlineQuery**. Because the **DEFINE BROWSE** statement associates the query with the new browse, the rows from the temp-table are automatically displayed in the browse. The **DISPLAY** statement displays the browse itself in the same frame with everything else. Without more specific instructions on where to place it, Progress places it to the right of the last object that's already defined for the frame, which is the **Order** browse. You could also use **ROW** and **COLUMN** attributes on the **DISPLAY** statement to display the new browse somewhere else. Finally, enabling the browse lets the user select rows in it and scroll in it if it has too many rows to display at once. This code does not enable the individual cells in the browse for update.

The **VALUE-CHANGED** trigger fires whenever the user selects a different row in the browse. But there are two other times when a row is selected. The first is when the window first comes up. The **Order** browse then holds the **Orders** for the first **Customer**, and the first of those **Orders** is selected implicitly.

6. Make sure the **VALUE-CHANGED** trigger fires at that time, in the Main Block of the window procedure, by adding a line to **APPLY** the **VALUE-CHANGED** trigger on startup:

```
MAIN-BLOCK:  
DO ON ERROR    UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK  
    ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:  
        RUN enable_UI.  
        ASSIGN cState = Customer.State  
            iMatches = NUM-RESULTS("CustQuery").  
        DISPLAY cState iMatches WITH FRAME CustQuery.  
        APPLY "VALUE-CHANGED" TO OrderBrowse.  
END.
```

The second place where an **Order** is implicitly selected is whenever the user presses one of the buttons that selects a new **Customer**. This reopens the **Order** query, and you want the **OrderLines** for the first **Order** for that **Customer** to be retrieved.

Because there are four buttons with almost exactly the same code on them, this is another good place to use an include file. As mentioned earlier, the one keyword that is different in these triggers is whether it is a **GET FIRST**, **NEXT**, **PREV**, or **LAST**. So you make that keyword an argument to the include file.

7. Create an include file for the four buttons:
 - a. Copy the code from the **BtnFirst** CHOOSE trigger and paste it into a new procedure window.
 - b. Replace the FIRST keyword with the include file argument marker {1}.
 - c. Add the same statement as you did in the **Main Block** to APPLY the VALUE-CHANGED event.
 - d. Save this code as h-ButtonTrig1.i:

```
/* h-ButtonTrig1.i -- include file for the First/Next/Prev/Last buttons
   in h-CustOrderWin4.w. */
GET {1} CustQuery.
IF AVAILABLE Customer THEN
  DISPLAY Customer.CustNum Customer.Name Customer.Address Customer.City
        Customer.State
  WITH FRAME CustQuery IN WINDOW CustWin.
{&OPEN-BROWSERS-IN-QUERY-CustQuery}
APPLY "VALUE-CHANGED" TO OrderBrowse.
```

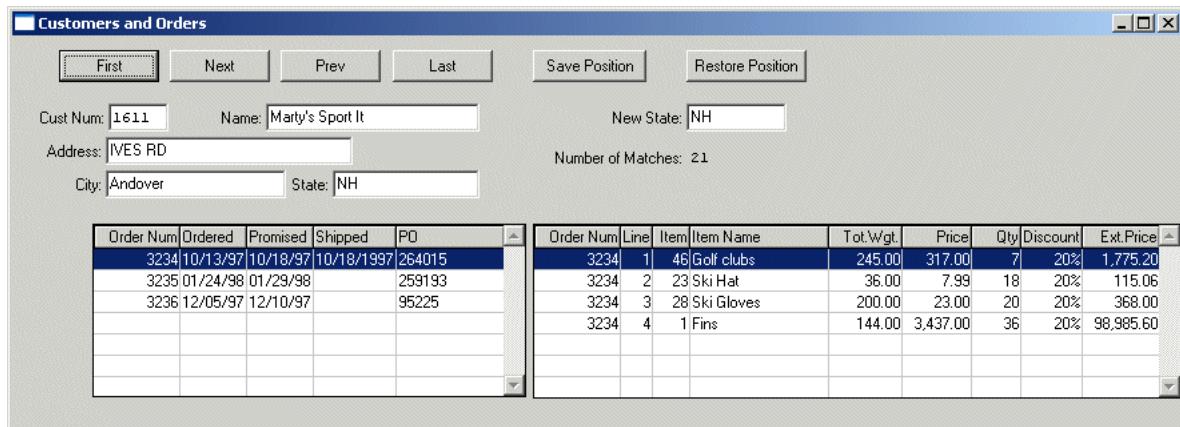
8. In each of the four CHOOSE triggers, replace the trigger code with a reference to the h-ButtonTrig.i include file, passing in the appropriate GET keyword, as in this **BtnFirst** trigger:

```
{ h-ButtonTrig1.i FIRST }
```

Now your procedure is ready to retrieve and display **OrderLines**.

9. Widen the window and its frame substantially by dragging the right edge out further to the right, to make room for the **OrderLine** browse.

10. Run the window to see the effect of your changes:



Summary

In this chapter you've learned:

- How to define temp-tables that create result sets that don't correspond to a single database table, how to pass them as parameters, and how to display their data.
 - How these temp-table parameters can help you separate out the data access and business logic of your application from the user interface procedures, in preparation for building distributed enterprise applications with the Progress 4GL.

In the next chapter, you'll learn more about Progress procedures and how to use them. There's a lot more to using procedures than just the RUN statement, and you'll see how the right procedure architecture can form the basis for a truly modern, object-oriented, event-driven application.

Using the Browse Object

This chapter describes how to design and interact with static browse objects and the queries they represent. In [Chapter 20, “Creating and Using Dynamic Temp-tables and Browses,”](#) you’ll learn how to create a dynamic browse at run time.

Like most other Progress graphical objects, the browse is also supported on character terminals, though with somewhat reduced functionality. Although this book concentrates on graphical interfaces, this chapter includes notes, along with a section at the end, that describe some of the differences between using the browse in a Windows GUI and in character mode.

A *browse* is a visual representation of a query. You already know from [Chapter 10, “Using Queries,”](#) that one of the principal uses of a query in Progress is to define a set of rows that you can navigate visually using a browse. The browse displays data from the query in rows and columns. A row represents the data of a single row in the query’s result list. A column represents the value of a particular field for each row. A row and column intersection, called a cell, represents the value of the field (column name) in that particular row. A user can scroll up and down the rows, and left and right through the columns.

You’ve already learned how to use the AppBuilder to generate a browse definition for you, and also how to define a browse on your own. In this chapter, you’ll learn about some of the many features that make the browse the most flexible and powerful visual object the 4GL supports. Even an entire chapter isn’t sufficient to cover every aspect of using the browse. For a description of all browse functionality, including attributes and methods not covered here, see the **DEFINE BROWSE Statement** reference entry and the **Browse Widget** reference entry, in [OpenEdge Development: Progress 4GL Reference](#) or the online help.

This chapter includes the following sections:

- Defining a query for a browse
- Defining a browse
- Programming with the browse
- Resizable browse objects
- Using browse objects in character interfaces

Defining a query for a browse

By default, a query against database tables uses SHARE-LOCK, so that other users can see the same records but not update them. A query is not scrollable by default. That is, the query does not have the ability to move backward as well as forward in the query. When you associate a browse with a query (by way of the DEFINE BROWSE statement), Progress automatically changes the query to use the NO-LOCK and SCROLLABLE options. You can also associate a query with a browse at run time. In this case, you should make the query SCROLLABLE when you define it. Since it's important for you to begin to separate user interface procedures from database access procedures even in these first exercises, this chapter extends the temp-table-based **OrderLine** browse from [Chapter 11, “Defining and Using Temp-tables,”](#) so that you think in terms of browsing temp-tables in a client procedure that might be in a separate session from where the database is located. In this case, locking the temp-table records is not an issue since those records are always used only within your local session.

Once you define the query, define the browse, and open the query, the browse and the query become tightly bound. The currently selected row and the result list cursor are in sync and remain so. When the user manipulates the browse, the user is also performing the same manipulation on the cursor of the result list. Many programmatic actions performed on the result list or the browse automatically update the other, although this is not universally true. You could say that learning all the subtleties of the browse involves learning what occurs by default, what you have to manage, and what behaviors you can override.

As a rule, a query associated with a browse should be used exclusively by that browse. While you can use GET statements on the query to manipulate the query's cursor, the result list, and the associated buffers, you run the risk of putting the browse out of sync with the query. If you do mix browse widgets and GET statements with the same query, you must use the REPOSITION statement to manually keep the browse in sync with the query.

Planning for the size of the result set

The number of records in your result list can have a serious impact on the performance of your browse, especially when you consider that you typically have to load those records into a temp-table and ship them across a network connection before they are seen in the browse. You might want to optimize your query definition to take advantage of features that work well with small and large sets of data, and design your user interface to encourage or require users to provide selection criteria to reduce the number of records that need to go into the data they're browsing.

Defining a browse

This section describes some of the major functionality and style choices available when you define a browse, and the issues involved with those choices.

You can browse records by defining a browse for the query and opening the query. If you do not specifically enable the browse columns, the result is a read-only browse. Once the user finds and selects a record, your application can use the selected record, which the associated query puts in one or more associated buffers. This is the general syntax for a browse definition:

```
DEFINE BROWSE browse-name QUERY query-name
  [ SHARE-LOCK | EXCLUSIVE-LOCK | NO-LOCK ]
  DISPLAY { column-list | record [ EXCEPT field ... ] }
  [ browse-enable-phrase ] browse-options-phrase .
```

As with all objects you define, you first give the browse a name. Next, you associate it with a query that you have previously defined. You can later associate the browse with a different query at run time by setting its QUERY attribute, but the query it's defined for is the source of initial information about the buffers and fields you use to define columns in the browse.

Once again, NO-LOCK is the only reasonable locking option even for a browse on a database table and the only possible one for a browse on a temp-table, since temp-table don't support record locking. It's also the default, so you can leave out the NO-LOCK keyword altogether.

The DISPLAY phrase tells Progress which fields from the buffers in the query to display as columns in the browse. You can specify either an explicit space-delimited *column-list* or a list of one or more record buffers from the query. In the latter case, you can use the EXCEPT phrase to remove one or more fields from each *record* when it is displayed. The default is not to display any fields at all, so you must always include a DISPLAY phrase in your browse definition unless you want to define all the columns at run time, as you will learn to do in [Chapter 20, “Creating and Using Dynamic Temp-tables and Browses.”](#)

If you just specify the DISPLAY list, the browse is read-only. None of its cells are enabled for input. The *browse-enable-phrase* lets you enable one or more cells for input:

```
ENABLE { column . . . | ALL [ EXCEPT column . . . ] }
```

Each column in the **ENABLE** phrase must be a column in the **DISPLAY** list. If you want to enable all or almost all the columns, you can use the **ALL** keyword optionally followed by an **EXCEPT** list.

There are many options you can specify in the *browse-options-phrase* to customize the appearance and behavior of your browse. The phrase begins with the **WITH** keyword. Here are some of the more important options:

- **rows DOWN [WIDTH *width*]** — You can specify how many rows to display in the viewport of the browse and, optionally, the width of the browse in characters. If the result of the query contains more rows than can be displayed at once, then the browse has a vertical scrollbar to let the user see them. If you use the **DOWN** phrase but don't specify a **WIDTH**, Progress allocates enough horizontal space to display all the browse columns. Otherwise, you can use a horizontal scrollbar to let the user scroll left and right through the columns.
- **{ SIZE | SIZE-PIXELS } *width* BY *height*** — As an alternative, you can define the size of the browse in both dimensions. This is the outer size of the browse including its border. When you use this option, Progress determines how many rows can be displayed at once, based on the *height*. This might result in a partial row being displayed at the bottom of the browse.

Note: You must specify either a **DOWN** phrase or a **SIZE** phrase in the browse options.

- **MULTIPLE | SINGLE** — You can let the user select only a single row at a time or multiple rows. The default is **SINGLE**.
- **SEPARATORS | NO-SEPARATORS** — *Separators* are vertical and horizontal lines between columns and rows. The default is **NO-SEPARATORS**, but you will probably find that your browses look better with **SEPARATORS**.
- **NO-ROW-MARKERS** — By default, an updateable browse displays row markers, which let the user select currently displayed rows in an updateable browse widget without selecting any particular cell to update. This option prevents row markers from being displayed.
- **NO-LABELS** — This option suppresses the display of column labels for the columns.
- **TITLE *string*** — You can optionally display a title bar across the top of the browse.

- **NO-ASSIGN** — If this option is not specified, data entered into an updateable browse is assigned on any action that results in a ROW-LEAVE event. The NO-ASSIGN option is intended for use with user-defined triggers on the ROW-LEAVE event. Essentially, when you specify this option, all data assignments by way of the updateable browse are up to you, the 4GL programmer. If you are defining a browse on a temp-table, then it is likely that the default support for automatic assigns back to the temp-table is appropriate because there are no transaction semantics to worry about. If you were to define an updateable browse directly against a database table, it is unlikely that the default assignment would be appropriate in any but the most trivial situations, so you would specify NO-ASSIGN to take control of the update process yourself.
- **NO-SCROLLBAR-VERTICAL | SCROLLBAR-VERTICAL** — By default, a browse gets a vertical scrollbar, and this scrollbar is enabled whenever the size of the browse is not sufficient to display all the rows in the result set. You can suppress this default with the NO-SCROLLBAR-VERTICAL keyword. If you don't have a vertical scrollbar, then the user must use the up and down arrow keys or other keys to navigate the browse. A browse always gets a horizontal scrollbar if the width of the browse is not sufficient to display all the columns.
- **ROW-HEIGHT-CHARS | ROW-HEIGHT-PIXELS** — (GUI only) By default, Progress assigns a row height appropriate for the font size used in the browse. You can change this default by specifying the row height in either characters or pixels.
- **FIT-LAST-COLUMN** — (GUI only) This option allows the browse to display so that there is no empty space to the right and no horizontal scroll bar, by widening or shrinking the last browse column's width. When this attribute is specified, and the last browse column can be fully or partially displayed in the browse's viewport, then the last browse column's width is adjusted so that it fits within the viewport with no empty space to its right and no horizontal scroll bar. If the last browse column is fully contained in the viewport with empty space to its right, it grows so that its right edge is adjacent to the vertical scroll bar. If the last browse column extends outside the viewport, it shrinks so its right edge is adjacent to the vertical scroll bar and the horizontal scroll bar is not needed.

Changing the test window for the OrderLine browse

You can try out some of the browse options using the browse you defined for a temp-table based on the **OrderLine** table in Chapter 11, ‘Defining and Using Temp-tables.’



To change the test window:

1. Open the *CustBrowseWin4.w* procedure and save it as *CustBrowseWin5.w*.
2. Remove the **Save Position** and **Restore Position** buttons, and the **New State** and **Number of Matches** fill-ins.
3. Remove these lines from the **Main Block**:

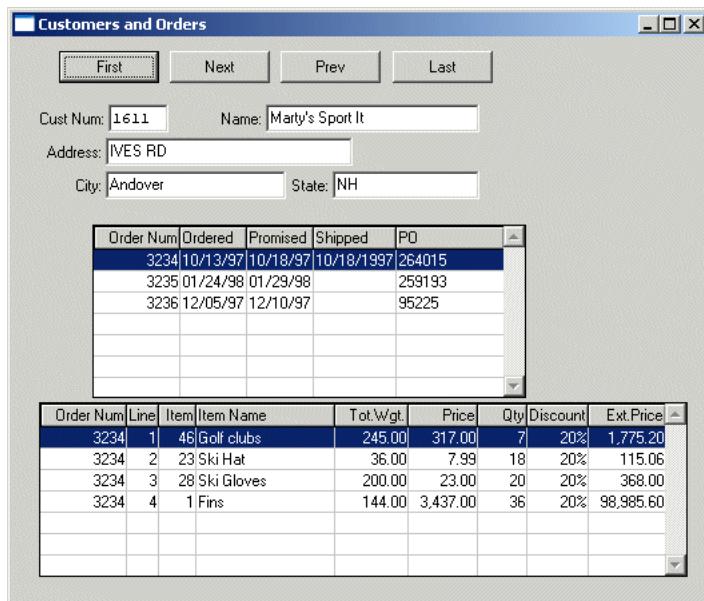
```
ASSIGN cState = Customer.State
      iMatches = NUM-RESULTSC("CustQuery").
DISPLAY cState iMatches WITH FRAME CustQuery.
```

The default position for the **OrderLine** browse was to the right of the **Order** browse.

4. To keep the window from being so wide, change its position to be explicitly below the **OrderBrowse** in the **VALUE-CHANGED** trigger block for the **OrderBrowse**:

```
DO:
  RUN h-fetchOlines.p (INPUT Order.OrderNum, OUTPUT TABLE ttOline).
  OPEN QUERY OlineQuery FOR EACH ttOline.
  DISPLAY OlineBrowse AT ROW 14 COLUMN 5 WITH FRAME CustQuery.
  ENABLE OlineBrowse WITH FRAME CustQuery.
END.
```

5. Resize the window so that it is somewhat narrower but taller than it was before. Experiment with it so that it is large enough to display the **OrderLine** browse when you run it:



6. Now that you better understand the complete syntax for defining a browse, take another look at the browse definition you created for **Order Lines** in the **Definitions** section of your window procedure:

```
DEFINE BROWSE OlineBrowse
  QUERY OlineQuery NO-LOCK DISPLAY
    ttoline.Ordernum
    ttoline.LineNum LABEL "Line"
    ttoline.ItemNum LABEL "Item" FORMAT "ZZZ"
    ttoline.ItemName FORMAT "x(20)"
    ttoline.TotalWeight
    ttoline.Price FORMAT "ZZ,ZZ9.99"
    ttoline.Qty
    ttoline.Discount
    ttoline.ExtendedPrice LABEL "Ext.Price" FORMAT "ZZZ,ZZ9.99"
  WITH NO-ROW-MARKERS SEPARATORS 7 DOWN ROW-HEIGHT-CHARS .57.
```

The definition has the necessary browse name and query name, and the optional NO-LOCK keyword, which could have been left off because it's the default. This is followed by the list of temp-table fields to display as browse columns. A browse column definition has a long list of available display options, including the LABEL and FORMAT that you specified for some of them. Because there is no ENABLE phrase, the browse is read-only.

The browse options phrase specifies:

- **NO-ROW-MARKERS** — Provides the default when there are no enabled columns.
- **SEPARATORS** — Provides the lines between columns and rows.
- **7 DOWN** — Provides the height of the browse in terms of rows displayed (because there is no WIDTH phrase all columns are displayed in full).
- **ROW-HEIGHT-CHARS** — Specifies the precise height of each row. The value **57** is the same value Progress would provide for the default font if you left this option out.

You can experiment with changing some of these options to see how they affect the appearance of the browse.

Enabling columns in the browser

In this section, you experiment with browse columns.

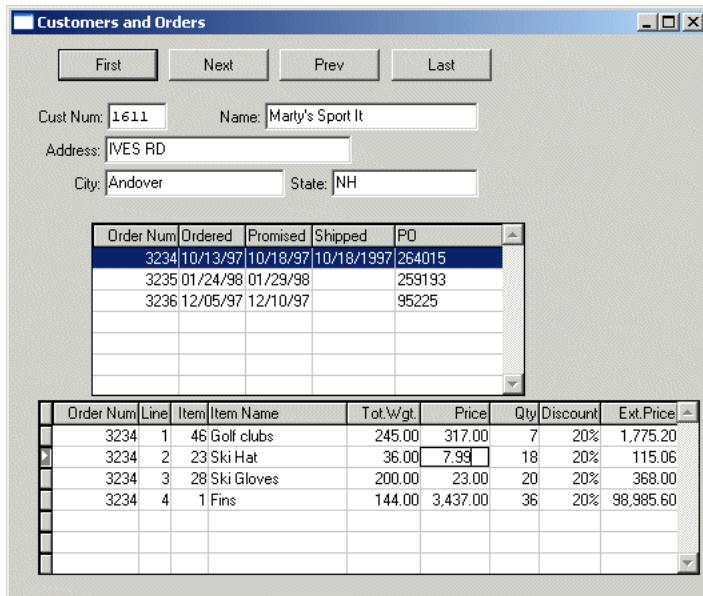


To enable some of the columns in the OrderLine browse:

1. Add this ENABLE phrase to the browse definition:

```
DEFINE BROWSE OlineBrowse
  QUERY OlineQuery NO-LOCK DISPLAY
    tt0line.Ordernum
    .
    .
    .
  ENABLE tt0line.Price tt0line.Qty tt0line.Discount
  WITH SEPARATORS 7 DOWN ROW-HEIGHT-CHARS .57.
```

2. You can also remove the NO-ROW-MARKERS keyword to see the effect of row markers in your updateable browse:



Now that the browse is updateable there is a row marker at the beginning of each row, if you don't specify the NO-ROW-MARKERS keyword, to indicate which row is selected. This is useful because the way a row is highlighted is very different depending on whether the user happens to click in an enabled or read-only column:

- If the user selects a row by clicking on a column that isn't enabled, the entire row is highlighted to indicate the selection (as in the first row of the **Order** browse above), just as if the browse had no enabled columns.
- If the user selects a row by clicking on an enabled column, then that cell is highlighted and a cursor appears in it to let the user change its value, as in the selected row of the **OrderLine** browse above.

The user should always click on the row marker to select a row to avoid inadvertently selecting an enabled column when this is not the user's intent.

If the user selects the **Price** column for an **OrderLine**, or the **Qty** or **Discount**, which are all enabled, the browse cell for that column behaves in much the same manner as a fill-in field. Tabbing or clicking moves the user from one updateable cell to the next. If the user changes the value, then that value is assigned when the user leaves the row. But assigned *where*? Because this is a browse of a temp-table, the value is assigned only back to the row in the temp-table. If the user browses an actual database table, then the update goes back to the database unless you included the NO-ASSIGN keyword in your browse definition. You'll learn more about updating records in [Chapter 16, “Updating Your Database and Writing Triggers,”](#) so you won't do any more with enabled columns for now.

Note: In character interfaces, a key sequence (**ESC-R**) toggles between two modes, to select rows or edit cells. By default, the selected row is the highlighted row that you change with CURSOR-DOWN and CURSOR-UP. Character row markers appear as an asterisk (*). Also, the EDITOR-TAB and BACK-TAB key functions (usually **CTRL-G** and **CTRL-U**) tab from cell to cell.

One very important fact to remember is that you cannot *enable*, or turn on, a read-only browse after you have defined it. If you expect to need any of the capabilities of the updateable browser, enable the appropriate fields in the definition and then use the READ-ONLY logical attribute to temporarily turn them off at run time.

Defining a single- or multiple-select browse

A browse can support single and multiple selections of rows. By default, a browse is single-select. The **MULTIPLE** option sets up a multiple-select browse. Multiple-selection lets you select any combination of rows from the browse, which you can then access using the appropriate browse attributes and methods.

In a multiple-select browse, the user first clicks in a row to select it. To select additional rows without deselecting the first, the user holds down the **CTRL** key and clicks in each row.

Alternatively, the user can click in one row, hold down the mouse key, and drag the cursor through multiple rows to select them all. Clicking on a selected row while pressing **CTRL** deselects it. Clicking on a row without pressing **CTRL** selects it and deselects all other rows. In a multiple-select updateable browse, the user must click on one row marker, hold down the mouse button and drag down through the row markers to select multiple contiguous rows.

Note that an updateable browse can only have one selected row when it is in edit mode. So, if the user selects five rows and then an updateable cell, the four rows that do not contain the active cell are deselected.

Note: For a multiple-select browse in character interfaces, selection of each row is switched by the space bar. The close angle bracket (>) indicates each selected row. The same technique applies to both read-only and updateable multiple-select browses.

There are several attributes and methods you can use in conjunction with selected rows in a single-selection or multiple-selection browse. You can learn about these in *OpenEdge Development: Progress 4GL Reference*.

Next, you modify your test procedure to use the **NUM-SELECTED-ROWS** attribute and the **FETCH-SELECTED-ROW** method to gather information about a set of selected rows in a multiple-selection browse



To modify your test procedure:

1. Add a button named **BtnTotal** with the label **Total** to your window.
2. Add a fill-in called **dTotal** to the window with no label.
3. In its property sheet, change its data type to **Decimal**.

4. Define this trigger block for the CHOOSE event for **BtnTotal**:

```

DO:
  DEFINE VARIABLE iRow      AS INTEGER      NO-UNDO.
  DEFINE VARIABLE hBrowse  AS HANDLE       NO-UNDO.

  hBrowse = BROWSE OlineBrowse:HANDLE.

  DO iRow = 1 TO hBrowse:NUM-SELECTED-ROWS:
    hBrowse:FETCH-SELECTED-ROW(iRow).
    dTotal = dTotal + tt0line.ExtendedPrice.
  END.
  DISPLAY dtotal WITH FRAME CustQuery.
END.

```

The NUM-SELECTED-ROWS attribute returns the number of rows the user has selected in the browse. You can then use the FETCH-SELECTED-ROW method, which takes the sequence within the list of selected rows as an argument, to position the query to each selected row in turn, and add up the **ExtendedPrice** for each of those rows.

One question you might have is why is it necessary to define a handle variable to hold the handle of the browse, rather than just referring to it in each statement as BROWSE OlineBrowse? The browse is not part of the initial frame definition for the window's frame. It is displayed after the window is initialized. If you refer to BROWSE OlineBrowse in the statements with the NUM-SELECTED-ROWS attribute or the FETCH-SELECTED-ROW method, Progress insists that you qualify the reference with a frame name so that it can identify the browse. Normally, you would qualify each reference to make it explicit or else simply put the entire set of references inside a DO block that defines the context:

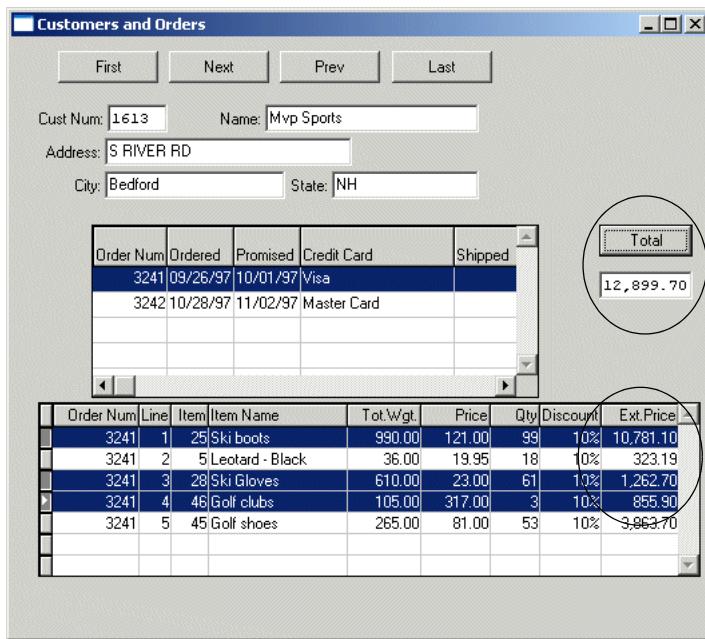
```

DO WITH FRAME CustQuery:
  DO iRow = 1 TO BROWSE OlineBrowse:NUM-SELECTED-ROWS:
    .
    .
    .

```

But because the browse is not part of the frame at compile time, Progress gives you an error message when you compile the procedure (or when you run it, which compiles it first) to the effect that it can't find **OlineBrowse** in frame **CustQuery**. To get around this, you need to fool Progress into accepting the reference at compile time. To do this, you first set the handle **hBrowse** variable to the **HANDLE** attribute of the browse, and then use the handle in place of the name. Progress accepts this because handle references are always difficult or impossible to check at compile time, so Progress has to assume that the reference will be valid when the code is executed, which it is.

- To see a total displayed, **Run** the window, select some **OrderLines**, and then choose the **Total** button:



Browse selection and query interaction

When a browse is initialized, the buffer for its query contains the first record in the query, since the user has not yet selected a record. To keep the browse in sync with the query, the buffer contains the first record in the browse viewport while there is no selected record. Whenever the user selects a row, that row becomes the current row in the query's buffer.

In a multiple-select browse, if you deselect the current row, the query is repositioned to the previously selected row. If no rows are selected, the query is repositioned to the first record in the viewport.

Table 12–1 summarizes how actions on the browse affect the associated query.

Table 12–1: Browse and query interaction

| Browse action | Effect on query |
|-------------------------------|---|
| Vertical keyboard navigation. | Moves the result list cursor for single-select browse widgets. |
| Select row. | Puts records for that row into the record buffers. |
| Deselect current row. | If other records are selected in the browse, put records for the most recently selected of those rows into the record buffers. Otherwise, repositions to the first row in the viewport. |
| Deselect noncurrent row. | None. |

Using the `GET` statement (such as `GET NEXT`) to navigate within the result list of the query has no effect on the browse. However, the `REPOSITION` statement does update the current position of the browse. If you use `GET` statements for a query on which a browse is defined, you should use the `REPOSITION` statement to keep the browse synchronized with the query. Also, when you open or reopen the query with the `OPEN QUERY` statement, the browse is automatically refreshed and positioned to the first record.

Using calculated columns

A browse can show calculated results involving other browse columns and other available data values as separate columns in the browse. You accomplish this by adding the expression for each calculated column to the `DEFINE BROWSE` statement. All valid browse format phrase options are legal extensions to the expression (for example, a `LABEL` option). When the browse is opened, the calculated columns are displayed for each row.

For an updateable browse, however, you need to refresh the calculation if one of the values in the expression changes. The browse displays the appropriate calculations only when the query refreshes the data. It is more appropriate to refresh the calculated data programmatically when the user finishes editing the affected row. To accomplish this, use a base field as a placeholder for the expression. You then can reference the calculation and refresh the result as needed.

To see an example, you can add a column to the **Order** browse in your test procedure to display the number of days between the **OrderDate** and the **PromiseDate**.



To add a column to the Order browse that displays the number of days between the OrderDate and the PromiseDate:

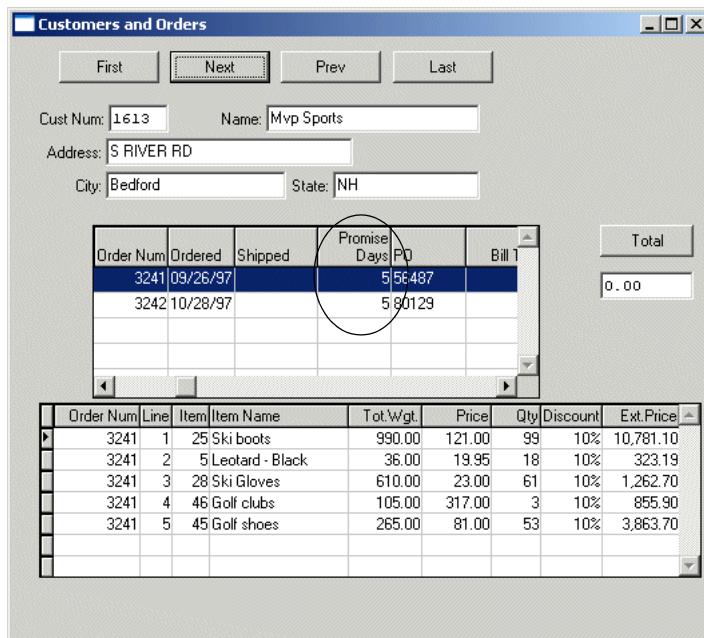
1. Open h-CustOrderWin4.w and save it as h-CustOrderWin5.w.
2. Define a variable in the **Definitions** section to act as the placeholder for the calculated field. Call this **Integer** variable **iPromiseDays**:

```
DEFINE VARIABLE iPromiseDays AS INTEGER      NO-UNDO.
```

3. Define the calculation as a column in the browse that is effectively displayed *at* the placeholder variable. You do this by including the expression for the calculation in the DISPLAY list followed by the at-sign (@) followed by the name of the placeholder variable that is used to store the value and represent its display format. Give the calculated field a COLUMN-LABEL of "Promise!Days" to see the effect of creating a stacked label:

```
DEFINE BROWSE OrderBrowse
  QUERY OrderBrowse NO-LOCK DISPLAY
    Order.Ordernum FORMAT "zzzzzzzz9":U WIDTH 10.2
    Order.OrderDate FORMAT "99/99/99":U
    Order.PromiseDate FORMAT "99/99/99":U
    Order.ShipDate FORMAT "99/99/9999":U
    Order.PromiseDate - Order.OrderDate @ iPromiseDays
      COLUMN-LABEL "Promise!Days"
    Order.PO FORMAT "x(20)":U WIDTH 17.2
  WITH NO-ROW-MARKERS SEPARATORS SIZE 65 BY 6.19 ROW-HEIGHT-CHARS .57
  EXPANDABLE.
```

4. Run h-CustOrderWin5.w. You see the calculation along with the other columns (it just happens that the number of days is always 5 in the test data in the Sports2000 database):



Note: When there are calculated fields in the DISPLAY phrase, you cannot use the ENABLE ALL option unless you use the EXCEPT option to exclude the calculated field, since calculated fields cannot be enabled.

Sizing a browse and browse columns

Normally, the browse calculates its size based on the column width of each field and the number of display rows requested with the DOWN option. If the resultant browse is too big for the frame, Progress displays an error message at compile time.

Use the DOWN option to specify the number of rows to display and the WIDTH option to specify the width of the browse. If the total column width is greater than the specified browse width, the browse displays a horizontal scrollbar to access the columns that don't display initially and you avoid the compilation error.

Note: In character interfaces, there is no horizontal scrollbar for a wide browse but the user can scroll the browse one column at a time, using CURSOR-LEFT and CURSOR-RIGHT.

The horizontal scrollbar can work in two different ways. By default, the browse scrolls whole columns in and out of the viewport. To change this behavior to pixel scrolling, specify the NO-COLUMN-SCROLLING option of the DEFINE BROWSE statement or set the COLUMN-SCROLLING attribute to **No** at run time.

You can also use the SIZE phrase to set an absolute outer size of the browse in pixel or character units, and Progress determines how many rows and columns to display. The AppBuilder always uses the SIZE phrase to specify the size of a browse that you lay out visually.

The browse also supports a WIDTH attribute for each column. Use this attribute to set the width of individual columns. When a column WIDTH sets the width of an updateable column smaller than the size specified by the FORMAT string, the browse cell scrolls to accommodate extra input up to the size specified by the FORMAT. You can take advantage of this behavior to allow the display of longer character fields. For example, you can use it where the WIDTH limits the display space the column takes up, but the user can scroll left and right through the entire text of the string up to the size of the FORMAT, using the left and right arrow keys.

The width of each column label is also a determining factor in the width of the displayed column. If the column label is wider than the data for that column, then the column is made wide enough to accommodate the label. To reduce this width, you can specify a shorter label in the browse definition (as you did in the **OrderLine** browse, for example). As you've seen, you can also create stacked labels by inserting an exclamation mark (!) when the label should be divided between two lines. For more details, see the “[Using stacked labels](#)” section on page 12–45.

Programming with the browse

The following sections discuss important programming techniques involving the browse, including browse events, managing rows in the browse, and manipulating the browse as a visual object.

Browse events

This section covers the essential events supported by the browse, including:

- **Basic events** — Occur when the user selects a row in the browse or scrolls through the data in the browse.
- **Row events** — Occur when the user enters or leaves a particular browse row or when the browse displays data values in a row.
- **Column events** — Occur when the user enters or leaves an enabled cell for a browse column.

Basic events

The basic browse events include:

- **VALUE-CHANGED** — Occurs each time the user selects or deselects a row. Note that the event name is slightly misleading in that it is not meant to imply that the value of the data in the row has been modified, only that a different row has been selected.
- **HOME** — Occurs when the user repositions the browse to the beginning of the query's result set by pressing the **HOME** key.
- **END** — Occurs when the user repositions the browse to the end of the query's result set by pressing the **END** key.
- **OFF-END and OFF-HOME** — Occur when the user uses the vertical scrollbar or arrow keys to scroll all the way to the end or the top of the browse.
- **DEFAULT-ACTION** — Occurs when the user presses **RETURN** or **ENTER** or when the user double-clicks a row. **DEFAULT-ACTION** also has the side effect of selecting the row that is currently highlighted.
- **SCROLL-NOTIFY** — Occurs when the user adjusts the scrollbar.

Typically, you use the VALUE-CHANGED and DEFAULT-ACTION events to link your browse to other parts of your application. Your current test window contains an example of the use of VALUE-CHANGED. Whenever the current row in the **Order** browse is changed, the VALUE-CHANGED event occurs and the trigger executes to retrieve its **OrderLines** and to display them in the second browse.

The DEFAULT-ACTION event occurs when the user explicitly selects a row not just by positioning to it, but by either double-clicking it or pressing **ENTER**.

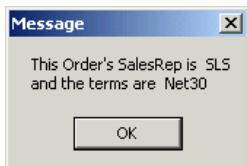


To define a DEFAULT-ACTION trigger for the Order browse:

1. In the **Section Editor**, select the **Triggers** section and the **OrderBrowse** object.
2. Choose the **New** button and select **DEFAULT-ACTION** from the list of browse events.
3. Enter this MESSAGE statement for the event:

```
DO:  
  MESSAGE "This Order's SalesRep is " Order.SalesRep SKIP  
          "and the terms are " Order.Terms VIEW-AS ALERT-BOX.  
END.
```

4. Run the window procedure and double-click on one of the **Order** rows to see the message:



The SCROLL-NOTIFY event is important if you plan to overlay other botches on browse cells, because it notifies you whenever the position of the rows of data within the browse viewport is changed by a scrolling action. SCROLL-NOTIFY gives you the ability to move the overlay widget to keep pace with the user's scrolling. See the “[Overlaying objects on browse cells](#)” section on page 12–39 for more information on this technique.

You could use the OFF-END or OFF-HOME event to retrieve more data to all the rows in the browse’s query, for example, when there are too many rows to retrieve at once. Note that if the user scrolls by clicking and holding the up or down arrows at the bottom and top of the vertical scrollbar, the OFF-HOME or OFF-END event does not occur until the user releases the mouse key.

Row events

The browse supports three row-specific events:

- **ROW-ENTRY** — Occurs when a user enters edit mode on a selected row. This occurs when the user clicks on an enabled cell within a row.
- **ROW-LEAVE** — Occurs when the user leaves edit mode. This occurs when the user leaves the enabled cells within a row, either by selecting a different row or selecting a nonenabled cell within the same row.
- **ROW-DISPLAY** — Occurs when a row becomes visible in the browse viewport.

You can find examples of how ROW-ENTRY and ROW-LEAVE work in the sections on updating and creating browse rows later in the “[Manipulating rows in the browse](#)” section on page 12–25.

The ROW-DISPLAY event lets you change the color and font attributes of a row or individual cells or to reference field values in the row. The event also lets you change the format of a browse-cell by changing the value of its FORMAT attribute. You can also use the ROW-DISPLAY event to change the SCREEN-VALUE of one or more cells within the row.

For example, you can create a ROW-DISPLAY event for the **OrderBrowse** in your test window with this code:

```
DO:  
  Order.PO:SCREEN-VALUE IN BROWSE orderBrowse =  
    "PO" + STRING (Order.OrderNum).  
  IF Order.ShipDate = ? THEN  
    ShipDate:BGCOLOR IN BROWSE OrderBrowse = 12.  
  END.
```

This changes the displayed value for the **PO** column and also checks the value of the **ShipDate** and changes its background color to signal that the **ShipDate** has not been entered, as shown in Figure 12–1.

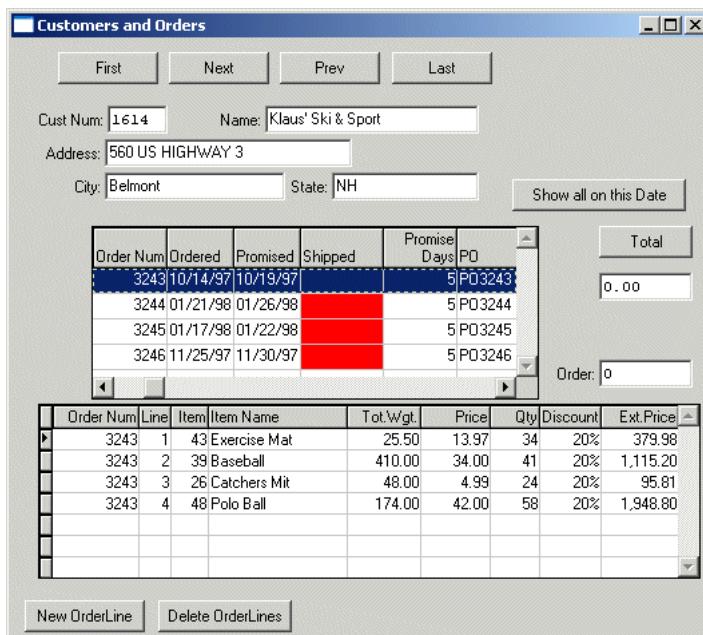


Figure 12–1: Result of ROW-DISPLAY event example

Notes: If you want to use this technique to change the colors of an array extent field, you must do so by directly referencing each member.

In character interfaces, the DCOLOR attribute specifies the color of an individual cell.

Column events

There are LEAVE and ENTRY events that reference the browse object itself. For example:

```
On ENTRY OF OrderBrowse
DO:
  .
  .
  .
END.
```

You can also write LEAVE and ENTRY triggers for individual columns. These triggers can refer to fields in the current row and can set and get attributes of the current cell by using either the name of the column in the browse or the SELF keyword. For example, you can enable the **PO** column in the **OrderBrowse** and then define this LEAVE trigger for it:

```
DO:
  IF Order.PO:SCREEN-VALUE IN BROWSE OrderBrowse BEGINS "X" then
    Order.PO:BGCOLOR IN BROWSE OrderBrowse = 12. /* RED */
  END.
```

Or more simply, you can use the SELF keyword to refer to the cell:

```
DO:
  IF SELF:SCREEN-VALUE BEGINS "X" then
    SELF:BGCOLOR = 12. /* RED */
  END.
```

Either way, you see the result shown in [Figure 12–2](#) when you run the procedure.

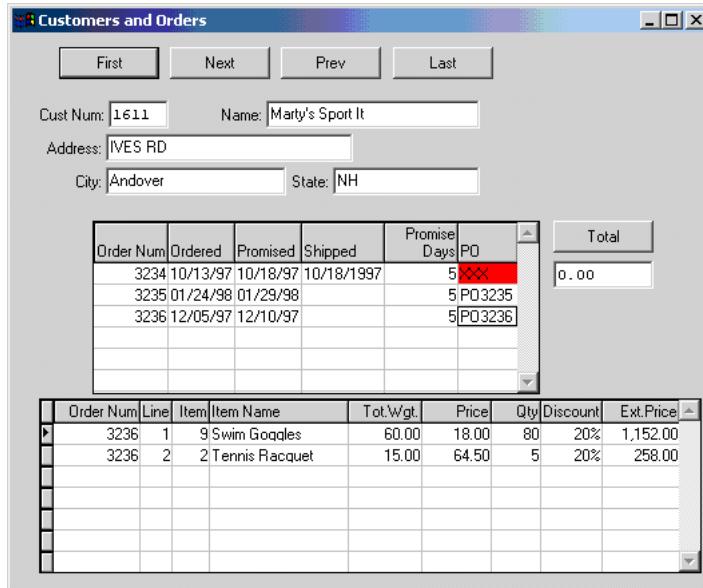


Figure 12–2: Result of column event example

Searching columns

There are also two events that let you interact with search mode. START-SEARCH occurs when the user chooses a column label. END-SEARCH occurs when the user enters edit mode on a cell or selects a row. You can apply both of these events to columns to force the start and end of search mode.

When a read-only browse has focus, typing any printable key (an alphanumeric character) forces Progress to reposition the browse to the first row where the first character of the first browse column matches the typed character. Thus, the read-only browse allows searching, but based only on the data in the first column. If the browse finds no match, it wraps to the top of the column and continues searching.

In an updateable browse, the user can select the column that is to be the basis of the character search. First, the user clicks a column label. The column label depresses. When the user types any printable key, Progress searches for the first row in that column that matches and focuses that row. Column searches also wrap to the top if necessary.

Note: A column search on a browse associated with a query with the INDEXED-REPOSITION option does not wrap to the top of the column if it cannot find a record to satisfy the search. This behavior is a side effect of the reposition optimization. To work around this, you can apply HOME to the query before starting the search.

There are two ways to extend this basic behavior.:

- You can configure an updateable browse to look like a read-only browse. This technique gives you selectable columns and searching on those columns even when you don't want to allow rows in the browse to be modified.
- You can use the START-SEARCH and END-SEARCH browse events to trap the beginning and end of search mode and write to your own search routines.

As noted, the column-searching feature is exclusive to the updateable browse. The read-only browse can do one-character deep searches on the initial column of the browse. You can work around this by defining your browse with one field enabled for input and the NO-ROW-MARKERS option specified. After the definition, disable the enabled column by setting the column's READ-ONLY attribute to false. This provides an updateable browse that looks like a read-only browse. The updateable browse retains the ability for the user to select individual columns and perform searches on them.

Note: The column searching capability is not available in character interfaces.

Manipulating rows in the browse

This section describes different ways you can change browse rows.

Refreshing browse rows

If you use a read-only browse and a user updates a record in some way outside the browse, you need a way to refresh the currently focused browse row with the new data. Use the following statement form at the end of the update code to refresh the browse:

```
DISPLAY column-name ... WITH BROWSE browse-name.
```

Repositioning focus

The browse and its query must remain in sync. At times, you might have to do some behind-the-scenes manipulation of the result set to keep them in sync. Generally speaking, whenever the user repositions the browse by selecting a new row, its query repositions automatically to that same row. Likewise, if the query is programmatically repositioned to a different row, the browse automatically coordinates with it and the new current row within the query becomes the currently selected row in the browse. However, if you simply FIND a row using the query's buffer, this does not reposition either the query or its browse. This is why you might need to resync the query in your code in order to properly refresh the browse. To do this, you can use the REPOSITION statement. The REPOSITION statement moves the database cursor to the specified position and adjusts the browse viewport to display the new row.

To avoid display flashing when doing programmatic repositions, you can set the REFRESHABLE browse attribute to FALSE, do the REPOSITION, and then set REFRESHABLE to TRUE. This suspends any redisplay of the browse until after the operation is complete.

In addition, the SET-REPOSITIONED-ROW() method gives you control over the position in the viewport where the browse displays the repositioned row. The method takes two arguments:

1. Its first Integer argument tells Progress which row (that is within the browse viewport) to position to. For example, if your browse displays seven rows at a time, you could use the SET-REPOSITIONED-ROW method to show a newly positioned row in the middle of the viewport by using an argument value of 4.
2. The second Character argument to the method can be ALWAYS or CONDITIONAL. If you specify ALWAYS, the browse is always adjusted to show the repositioned row in the specified position. If you specify CONDITIONAL, then the browse adjusts only if the repositioned row is not already in the viewport.

Note that normally you set SET-REPOSITIONED-ROW() once for the session for a browse to establish its behavior wherever it is used. You can also use the GET-REPOSITIONED-ROW() method to return as an Integer the current target viewport row for repositions.



To reposition the query and the browse along with it:

1. Define a new fill-in called **iOrder** of **INTEGER** data type.
2. Define this LEAVE trigger for the fill-in field:

```
DO:  
  BROWSE OrderBrowse:SET-REPOSITIONED-ROW(3, "CONDITIONAL").  
  ASSIGN iOrder.  
  FIND Order WHERE order.orderNum = iOrder NO-ERROR.  
  IF AVAILABLE (Order) THEN  
    REPOSITION OrderBrowse TO ROWID ROWID(Order) NO-ERROR.  
  END.
```

When you enter a value into the field and press **TAB**, the query and the browse are both repositioned to that row. If you enter a row that Progress can't find or that isn't an **Order** for the current **Customer**, Progress suppresses these errors and nothing changes in the browse. The SET-REPOSITIONED-ROW method makes the new row the third row in the viewport unless it's already displayed.

For a multiple-select browse, the concepts of focus and selection separate. Selection indicates that the user has selected a record. The user can select many records. The last selected record is the one in the record buffer. Focus indicates that the record has input focus. There can be only one focused row, and it might or might not be a selected row. In a single-select browse, selection and focus are one and the same.

Updating browse rows

By default, Progress handles the process of updating data in the records managed by an updateable browse's query. Because you should normally create applications that do not have a client-side dependency on a direct database connection, it is not advisable for you to define browses directly against database tables or to update database tables directly through a browse. Therefore, the default update behavior is described here to make you aware of what Progress does when you are browsing a database table, not to recommend that you take advantage of it. Some of these steps might not be completely clear to you until you read [Chapter 17, “Managing Transactions,”](#) but it gives you an overview of the steps.

Assuming that the browse starts with the record in NO-LOCK state, Progress follows these steps:

- On any user action that modifies data in an editable browse cell, Progress again gets the record with a SHARE-LOCK, which means that no other user can change the record. If the data has changed, Progress warns the user and redisplays the row with the new data.
- When the user leaves the row and has made changes to the row, Progress starts a transaction (or subtransaction) and gets the record from the database with EXCLUSIVE-LOCK NO-WAIT, which means that no other user can lock the record in any way. If no changes were made, Progress does not start a transaction.
- If the GET with EXCLUSIVE-LOCK is successful, Progress updates the record, disconnects it (removes the lock), ends the transaction, and downgrades the lock to its original status.
- If the GET with EXCLUSIVE-LOCK fails, Progress backs out the transaction, displays an error message, keeps the focus on the edited row, and retains the edited data.

You also have the option to disable this default behavior and programmatically commit the changes by way of a trigger on the ROW-LEAVE event. To do this, you must supply the NO-ASSIGN option in the DEFINE BROWSE statement.

If you define your browses against temp-tables, as you are doing in the **OlineBrowse** of the test procedure, then there are no record locks and no possibility of contention with other users for the records in the temp-table. Therefore, you can let Progress update the temp-table by defining the browse without the NO-ASSIGN keyword and then take care of updating the database from the temp-table yourself. You learn how to do this in [Chapter 20, “Creating and Using Dynamic Temp-tables and Browses.”](#)

Creating browse rows

In an updateable browser, you might want to allow the user to add new records to the underlying temp-table or database table. Programmatically, this requires three separate steps:

1. Create a blank line in the browse viewport with the `INSERT-ROW()` method and populate it with new data. `INSERT-ROW` takes a single optional argument, which is the string “BEFORE” or “AFTER”. This argument tells Progress whether to insert the new row before or after the currently selected row in the browse. The default is “BEFORE”. You can use the `INSERT-ROW()` browse method in an empty browse. It places a new row at the top of the viewport.
2. Use the `CREATE` statement and `ASSIGN` statement to update the database or the underlying temp-table.
3. Add a reference to the result list with the `CREATE-RESULT-LIST-ENTRY()` method. This is the list of record identifiers that Progress uses to keep track of the set of rows in the query. This step is only necessary if you do not plan to reopen the query after the update, because reopening the query completely refreshes the list. However, this method makes reopening the query unnecessary for most applications.

All three steps are required to create the record and keep the database, query, and browse in sync. Also, there are several possible side effects to allowing the user to add a record through an updateable browse. They include placing new records out of order and adding records that do not match the query. To eliminate these side effects, you can reopen the query after each new record is added.



To add a button to the test window that lets you add new OrderLines to the temp-table through the OlineBrowse:

1. In the **Definitions** section of the `h-CustOrderWin5.w` procedure, beneath the browse define for **OlineBrowse**, define a handle variable called `hBrowse`. You will use this variable to hold the handle of the **OrderLine** browse because it is needed in several places.
2. Define a new buffer for the **OrderLine** temp-table called `tt0line2`. You’ll use this buffer to hold onto the values in the current temp-table record as you create a new record, and to copy some of the values from the old record to the new one.

3. Assign the **OlineBrowse** handle to the new **hBrowse** variable:

```
DEFINE VARIABLE hBrowse AS HANDLE      NO-UNDO.  
DEFINE BUFFER   ttOline2 FOR ttOline.  
  
hBrowse = BROWSE OlineBrowse:HANDLE.
```

4. Drop a new button onto the window where it will not be overwritten by the **OrderLine** browse when it is displayed.
5. Name the new button **btnNew** and give it a label of **New OrderLine**.
6. Define a CHOOSE trigger for the button:

```
DO:  
  IF hBrowse:NUM-SELECTED-ROWS = 0 THEN  
    DO:  
      APPLY "END" TO hBrowse.  
      hBrowse:SELECT-FOCUSED-ROW().  
    END.  
    hBrowse:INSERT-ROW("AFTER").  
  END.
```

This trigger code uses the handle of the **OlineBrowse** that you just assigned up in the **Definitions** section and inserts a new row in the browse after the currently selected row. First, it checks to make sure that there is a selected row using the browse's **NUM-SELECTED-ROWS** attribute. If the value of this attribute is zero, then the user has not selected any row. In this case, the trigger positions to the end of the browse by applying the **END** event to it and then selects that row using the **SELECT-FOCUSED-ROW** method.

7. Back in the procedure's **Definitions** section, add this ROW-LEAVE trigger block for the **OlineBrowse** following the statement that assigns its handle to hBrowse:

```
ON "ROW-LEAVE" OF BROWSE OlineBrowse
DO:
  IF hBrowse:NEW-ROW THEN
    DO:
      FIND LAST tt0line2.
      CREATE tt0line.
      BUFFER-COPY tt0line2 TO tt0line
        ASSIGN tt0line.LineNum = tt0line2.LineNum + 1.
      ASSIGN INPUT BROWSE OlineBrowse tt0line.Qty
        tt0line.Price tt0line.Discount.
      DISPLAY tt0line.OrderNum tt0line.LineNum
        tt0line.ItemNum tt0line.ItemName
        WITH BROWSE OlineBrowse.
      hBrowse:CREATE-RESULT-LIST-ENTRY().
    END.
  END.
```

This code first checks the NEW-ROW attribute of the browse to see if the row the user left is one newly created by the **New OrderLine** button.

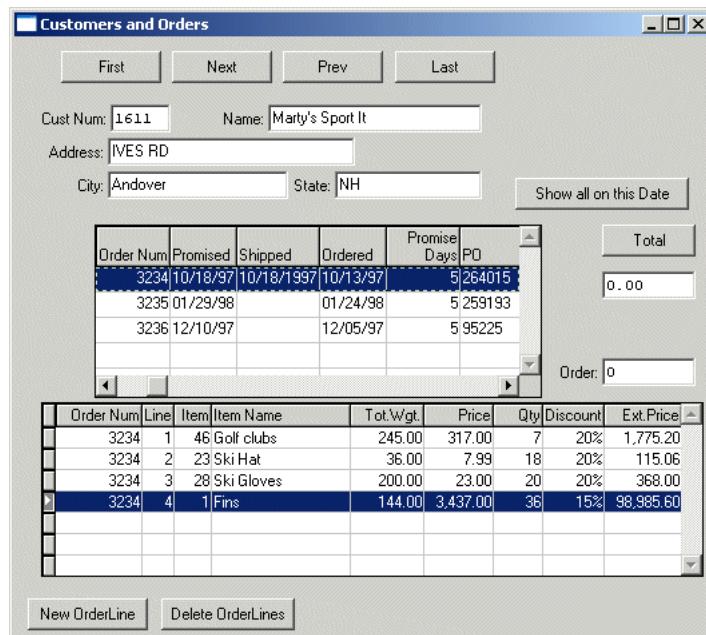
Because you're not prepared to do a full-fledged database update from the browse yet, the example simply creates a new row in the temp-table. To simplify the example, you start by copying the fields from the previous record into the new one. To do this, you FIND the LAST record in the temp-table using the second buffer tt0line2. This gives you the highest **LineNum** value so that you can increment it for the new record. Note that this means that other initial field values are also assigned from the last record regardless of where in the viewport the user inserted the new record.

You create a new temp-table record and BUFFER-COPY the last record into the new one, incrementing the **LineNum** field to give it a distinct value. This is important because the temp-table has the same unique index on the **OrderNum** and **LineNum** fields as the underlying **OrderLine** table in the database, so changing the **LineNum** has to happen in the BUFFER-COPY statement to avoid a unique index violation.

Next, the code assigns values to those columns that are enabled for input. The **INPUT** keyword on the **ASSIGN** statement tells Progress to take the values from the screen buffer for each column. Then the new values are displayed in the browse.

Finally, you create a result list entry for the new record. If you didn't create a result list entry the row would be lost if you were to scroll it out of the browse's viewport. Reopening the query would also rebuild the result list, and also assure that all rows are in the proper order, so you can use either technique for getting the browse and the query back in sync.

8. Run the window. Now you can add a record to the **OrderLine** browse:



Once again, you'll learn how to get new records into the database in [Chapter 16, “Updating Your Database and Writing Triggers.”](#)

Deleting browse rows

Deleting a record by way of a browse is a two-step process. First, you need to delete the record from the underlying temp-table or database table, and then you need to remove the record from the browse itself, along with the query's result list. If you are browsing a database table directly and the user indicates a deletion, you should again get the records by EXCLUSIVE-LOCK NO-WAIT and then use the DELETE statement to remove the records from the database. In the case of a temp-table, you can simply use the DELETE statement to remove the records.

Next, you use the `DELETE-SELECTED-ROWS()` method to delete one or more selected records from both the browse widget and the associated query result list.



To add a Delete button to the test window and use it to remove rows from the OlineBrowse and its temp-table:

1. Drop another button onto the window called **btnDelete** and give it a label of **Delete OrderLines**.
2. Write this CHOOSE trigger for the new button:

```
DO:  
  DEFINE VARIABLE iRow AS INTEGER      NO-UNDO.  
  DO iRow = 1 TO hBrowse:NUM-SELECTED-ROWS:  
    hBrowse:FETCH-SELECTED-ROW(iRow).  
    DELETE tt0line.  
  END.  
  hBrowse:DELETE-SELECTED-ROWS().  
END.
```

Because the **OlineBrowse** is a multiple-selection browse, you can use it to delete one or more rows at once. This code walks through the set of selected rows and retrieves each one in turn using the **FETCH-SELECTED-ROW** method. This method repositions the temp-table query to that row, so that it can be deleted. The code then uses the **DELETE-SELECTED-ROWS** method to delete all the rows from the browse itself, along with the query's result list entries for them.

If you run the test window, you can now delete one or more rows from the list of **OrderLines** for an **Order**. Remember that these records are deleted only from the temp-table. You would have to make a call to a procedure connected to the database to pass the list of rows to delete from the database table.

Manipulating the browse itself

There are various ways in which you can allow users to change the appearance of the browse at run time. In some cases, you can provide for these changes programmatically so that they are fully under your control. In other cases, you can just let users use the mouse to change the size and shape of the browse to their liking. This section describes some of those capabilities.

Setting the query attribute for the browse

When you define a browse, you must associate it with a previously defined query. This association allows Progress to identify the source for the browse columns and other information. However, at run time you can associate a browse with any query that supplies the columns the browse needs by setting the browse's QUERY attribute to the handle of the query. The query must supply the same columns as the one you defined the browse with.

When you set the QUERY attribute, Progress immediately refreshes the browse with the results of the new query. If the query is not open then Progress empties the browse.

You can use this capability to define a placeholder query for a browse simply to satisfy the definition, and then to associate the actual query with it at run time. Or you can use this to alternate between two or more queries to display different useful sets of data in the same browse.



To define a different query on the Order table to alternate with the display of Customers of the current Order:

(This alternative shows all the **Orders** in the database that have the same **OrderDate** as the currently displayed **Order**.)

1. In the **Definitions** section of h-CustOrderWin5.w, add these lines:

```
DEFINE BUFFER    bOrder2      FOR Order.
DEFINE QUERY     qOrderDate   FOR bOrder2 SCROLLING.
DEFINE VARIABLE  lOrderDate  AS LOGICAL      NO-UNDO.
```

Buffer **bOrder2** is a second buffer for the **Order** table and query **qOrderDate** is a query on this buffer. You'll use this to define a query for other **Orders** that have the same **OrderDate** as the currently displayed **Order**. The **lOrderDate** flag tells the program whether the user is currently seeing the results of the query on the **OrderDate** or not.

2. Add a button to the window named **btnOrderDate** with the label **Show all on this Date**.

3. Define this CHOOSE trigger for the button:

```

DO:
  IF NOT lOrderDate THEN
    DO:
      /* If the standard query is displayed, open the query for
      Orderdates and make that the browse's query.
      Hide the OrderLine browse while we're
      doing this, and adjust the button label accordingly. */
      OPEN QUERY qOrderDate FOR EACH bOrder2
        WHERE bOrder2.OrderDate = Order.OrderDate.
      ASSIGN BROWSE OrderBrowse:QUERY = QUERY qOrderDate:HANDLE
        /* Signal that we're showing the OrderDate query. */
        lOrderDate = TRUE
        hBrowse:HIDDEN = TRUE
        SELF:LABEL = "Show Customer's Orders".
    END.
    ELSE
      /* If we're showing the OrderDate query, switch back to the
      regular query for the current Customer's Orders. */
      ASSIGN BROWSE OrderBrowse:QUERY = QUERY OrderBrowse:HANDLE
        lOrderDate = FALSE
        hBrowse:HIDDEN = FALSE
        SELF:LABEL = "Show all on this Date".
  END.

```

If the **lOrderDate** flag is not set, then the user sees the default query of **Orders** for the current **Customer**. In this case the code:

- a. Opens the other query for all **Orders** matching the current **OrderDate**.
- b. Assigns this to be the query for the **OrderBrowse**.
- c. Reverses the value of the flag variable.
- d. Hides the **OrderLine** browse because it's not relevant to this display.
- e. Switches the label of the button to allow the user to change back to the original query.

If the flag *is* set, then the trigger reverts to the original query and label, resets the flag, and views the **OrderLine** browse.

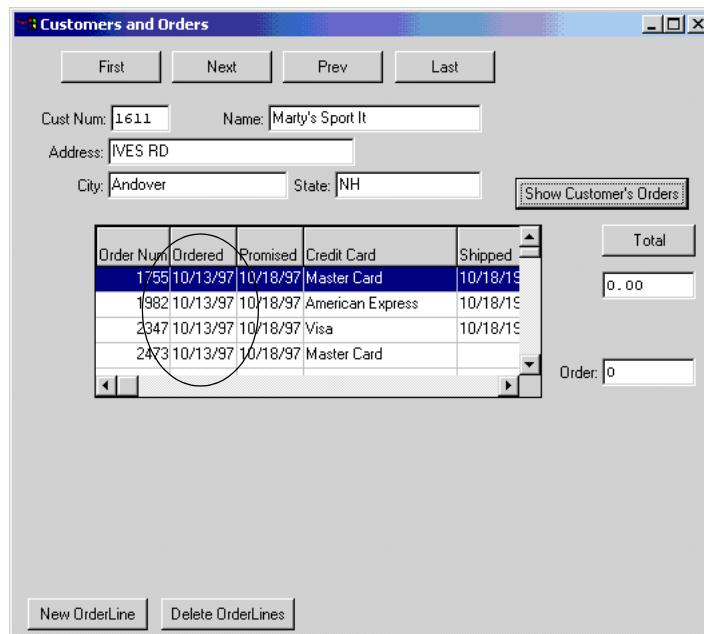
There's one more step you need to take, so that the procedure always reverts back to the original query if the user chooses one of the **First**, **Next**, **Last**, or **Prev** buttons.

4. Create a local copy of the h-ButtonTrig1.i include file, call it h-ButtonTrig2.i, and add these lines to it:

```
/* h-ButtonTrig2.i -- include file for the First/Next/Prev/Last buttons
in h-CustOrderWin5.w. */
GET {1} CustQuery.
IF AVAILABLE Customer THEN
  DISPLAY Customer.CustNum Customer.Name Customer.Address Customer.City
    Customer.State
  WITH FRAME CustQuery IN WINDOW CustWin.
IF !OrderDate THEN
  APPLY "CHOOSE" TO btnOrderDate.
{&OPEN-BROWSERS-IN-QUERY-CustQuery}
APPLY "VALUE-CHANGED" TO OrderBrowse.
```

This code runs the trigger block if the flag is true, so that it reverts to the Orders OF Customer query.

5. Run the procedure now and choose the **Show all on this Date** button. You can see the effect of switching queries for the browse:



Accessing the browse columns

In Chapter 20, “Creating and Using Dynamic Temp-tables and Browses,” you will learn how to create a dynamic browse. Even with a static browse it is often useful to get at some of its attributes through its handle. You have already seen how you can use the handle of a browse rather than the `BROWSE browse-name` static language construct to refer to a browse and access its attributes. The individual columns of the browse also have handles. You can get the handle of any browse column by walking through the list of browse columns from left to right. This section introduces you to this concept, and in later examples you’ll use this technique to act on the columns of a browse. These are the basic attributes you need to identify any column in a browse:

- **NUM-COLUMNS** — This browse attribute returns the number of columns in the browse. You can use this as a limit, for example, in a DO statement that looks at every column.
- **CURRENT-COLUMN** — This browse attribute returns the handle of the currently selected column in the browse, if the user has clicked on a column.
- **FIRST-COLUMN** — This browse attribute returns the handle of the first (leftmost) column in the browse. Use this attribute to get started walking through the columns.
- **NEXT-COLUMN** — This is an attribute of each browse column, not of the browse itself. After retrieving the handle of the first column using the FIRST-COLUMN attribute, you then retrieve the column’s NEXT-COLUMN attribute to walk through the columns.
- **PREV-COLUMN** — This column attribute returns the handle of the previous column, that is, the one to the current column’s left in the browse.

Browse columns have various useful attributes that you can get and, in some cases, set (such as its NAME or LABEL). You can learn about all of these attributes in the online help and in the third volume of *OpenEdge Development: Progress 4GL Reference*. Later examples, such as the one in the “Moving columns” section on page 12–38, show how to use a few of these.

Locking columns

You can use the NUM-LOCKED-COLUMNS attribute to prevent one or more browse columns from scrolling out of the browse viewport when the horizontal scrollbar is used. A nonhorizontal-scrolling column is referred to as a *locked column*.

Locked columns are always the leftmost columns in the browse. In other words, if you set NUM-LOCKED-COLUMNS to 2, the first two columns listed in the `DEFINE BROWSE` statement are locked. In the next example, the **Order Number** and **Order Date** never move out of the browse viewport, no matter which of the remaining fields the user accesses with the horizontal scrollbar.



To experiment with locking columns:

1. Double-click on the **OrderBrowse** to go into its property sheet.
2. Choose the **Fields** button to edit the field list.
3. Choose the **Add** button and add all the rest of the **Order** fields to the browse. Don't add any **Customer** fields, as they just repeat all the values from the **Customer** record you're already displaying above the browse.
4. Back in the property sheet, set **Locked Columns** to **2**.

The AppBuilder generates a statement from the **Locked Columns** setting to set the **NUM-LOCKED-COLUMNS** attribute at run time, because this attribute cannot be set using a keyword in the **DEFINE BROWSE** statement:

```
ASSIGN
    OrderBrowse:NUM-LOCKED-COLUMNS IN FRAME CustQuery      = 2.
```

5. Run the window again. You can scroll through all the rest of the **Order** columns, but the first two columns are always displayed:

| Order Num | Ordered | Carrier | Credit Card |
|-----------|----------|------------------|------------------|
| 3234 | 10/13/97 | Walkers Delivery | American Express |
| 3235 | 01/24/98 | Fedex Overnight | American Express |
| 3236 | 12/05/97 | Fedex Overnight | Visa |

| Order Num | Line | Item | Item Name | Tot Wgt | Price | Qty | Discount | Ext Price |
|-----------|------|------|------------|---------|----------|-----|----------|-----------|
| 3234 | 1 | 46 | Golf clubs | 245.00 | 317.00 | 7 | 20% | 1,775.20 |
| 3234 | 2 | 23 | Ski Hat | 36.00 | 7.99 | 18 | 20% | 115.06 |
| 3234 | 3 | 28 | Ski Gloves | 200.00 | 23.00 | 20 | 20% | 368.00 |
| 3234 | 4 | 1 | Fins | 144.00 | 3,437.00 | 36 | 20% | 98,985.60 |

Moving columns

You can use the MOVE-COLUMN() method to rearrange the columns within a browse. For example, rather than forcing the users to scroll horizontally to see additional columns, you might allow them to reorder the columns. MOVE-COLUMN takes two arguments:

- The integer sequence within the browse of the column to move, counting from left to right.
- The integer position to move the column to, again counting from left to right.

The following simple example shows how to use the MOVE-COLUMN method along with the START-SEARCH event and the column attributes introduced earlier.

The START-SEARCH event occurs when the user clicks on the column header for a browse with enabled columns. You can use this event to sort by column or for other purposes. In this case, you want to identify which column position was selected and move this column one position to the left.



To rearrange the columns you see first in the viewport without scrolling, define this START-SEARCH trigger block for the **OrderBrowse**:

```

DO:
  DEFINE VARIABLE hColumn AS HANDLE      NO-UNDO.
  DEFINE VARIABLE iColumn AS INTEGER     NO-UNDO.
  hColumn = SELF:FIRST-COLUMN.
  DO iColumn = 1 TO SELF:NUM-COLUMNS:
    IF hColumn = SELF:CURRENT-COLUMN THEN LEAVE.
    hColumn = hColumn:NEXT-COLUMN.
  END.
  IF iColumn NE 1 THEN
    SELF:MOVE-COLUMN(iColumn, iColumn - 1).
END.

```

Whenever you enter a trigger block, the SELF keyword evaluates to the handle of the object that initiated the event. In this case, this is the browse itself, not the browse column. The CURRENT-COLUMN attribute returns the handle of the column the user clicked on.

The code initializes the hColumn handle variable to the first column in the browse and then walks through all the columns, looking for the one with the same handle as the browse's CURRENT-COLUMN. This identifies the sequential position of the one selected. The MOVE-COLUMN method moves this column one position to the left, unless the user selected the first column.

You can also let the user simply drag columns left or right by setting the browse COLUMN-MOVABLE attribute to true or an individual column's MOVABLE attribute to true, as described in the “[Moving the browse](#)” section on page 12–48.

Overlaying objects on browse cells

One useful way to extend the behavior of an updateable browse in a Windows GUI is to overlay an enabled browse cell with another object when the user enters the cell to edit it. For example, on ENTRY of a cell, you could display a combo box or a toggle box. The user selects an entry from the combo box, or checks the toggle box, and you use those values to update the SCREEN-VALUE attribute of the browse cell. The values get committed or undone along with the rest of the row.

There are two problems to overcome:

1. The first is in calculating the geometry to let you precisely position the overlay widget. You do this by adding the X and Y attributes of the browse cell to the X and Y attributes of the browse and assigning the result to the X and Y attributes of the overlay object.
2. You might need to move the overlay object if the user accesses the scrollbar or manipulates the browse in some other way. A trigger on the SCROLL-NOTIFY event notifies you when the user scrolls, and you can update the X and Y positioning accordingly.

Note: It is not possible in every instance to capture every manipulation of a browse to reposition an overlaid object, especially if you have made browse columns resizable and so forth. Thus, this technique is not completely foolproof. However, it can be useful in many cases for extending the browse to have visual behavior that users can expect and that is not natively available with the browse.

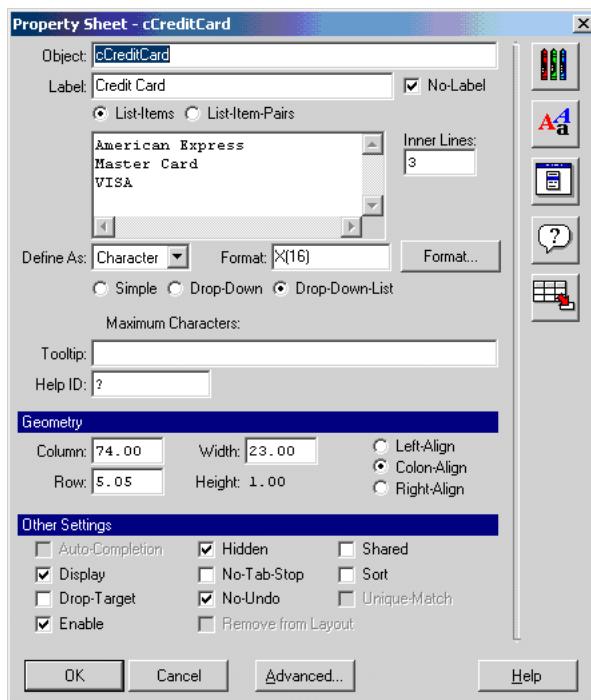


To overlay the CreditCard field in the Order browse with a combo box:

(Not only is this convenient, but it assures that the user selects only a valid choice for the field.)

1. Go into the **OrderBrowse** property sheet, click on the **Fields** button and move the **CreditCard** field up toward the head of the list, so that it is visible without horizontal scrolling.
2. Enable the **CreditCard** column.

3. Select the **Combo Box** icon  from the **AppBuilder palette** and drop it onto any empty spot on the window. It is initially hidden and then moved to the proper location when it's needed, so it doesn't matter where it starts out.
4. In its property sheet, give the new object a name of **cCreditCard**, check the **No-Label** toggle box, and provide the three choices shown for the **List-Items**:



5. Make the **Format X(16)** and check the **Hidden** toggle box from the set of **Other Settings** at the bottom.
6. Choose **OK** to accept these settings.

Now you have two objects that will interact: the browse column called **CreditCard** representing the **CreditCard** field in the **Order** table and the combo box called **cCreditCard** that you will overlay on it.

7. You might also have to add a line to the procedure's main block to make sure that the combo box is hidden when the window first comes up:

```

MAIN-BLOCK:
DO ON ERROR UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
  ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:
    RUN enable_UI.
cCreditCard:HIDDEN = TRUE.
  APPLY "VALUE-CHANGED" TO OrderBrowse.
END.

```

Now you need to define a series of trigger blocks to handle the following events on the browse and on the combo box:

- You want the combo box to appear in the right place when the user selects the **CreditCard** field.
- Then, when the user selects one of its values, you need to transfer that selection from the combo box to the browse cell.
- You also need to adjust the placement of the combo if the user scrolls the browse or moves the **CreditCard** column.



To create these trigger blocks:

1. Define an internal procedure called **placeCombo** to position and view the combo box on top of the selected browse cell:

```

PROCEDURE placeCombo:
  DEFINE VARIABLE hCredit AS HANDLE      NO-UNDO.

  hCredit = CreditCard:HANDLE IN BROWSE OrderBrowse.
  IF hCredit:X < 0 OR hCredit:Y < 0 THEN /* Column not on screen */
    cCreditCard:VISIBLE IN FRAME CustQuery = FALSE.
  ELSE DO:
    ASSIGN cCreditCard:X IN FRAME CustQuery = hCredit:X + OrderBrowse:X
          cCreditCard:Y IN FRAME CustQuery = hCredit:Y + OrderBrowse:Y
          cCreditCard:SCREEN-VALUE = CreditCard:SCREEN-VALUE
          cCreditCard:VISIBLE IN FRAME CustQuery = TRUE.
    cCreditCard:MOVE-TO-TOP().
  END.
END PROCEDURE.

```

This procedure gets the handle to the **CreditCard** cell in the browse and checks its X and Y coordinates (which are in pixels). If they are not both greater than zero, then the column is not currently displayed and no action is taken. This might happen in response to a browse event that has scrolled the contents of the browse, for instance. If the column has been scrolled out of the viewport, then the column can't be updated.

Otherwise, the code totals the X and Y positions of the browse itself and the **CreditCard** cell. This is because the X and Y coordinates of the browse are relative to its frame, and the X and Y coordinates of the cell are relative to the browse. Since the combo box coordinates are relative to the frame, you need to add the browse position and the cell position together to get the cell's position relative to the frame, which is where the combo box will go.

Next, the code initializes the **cCreditCard** combo box to the SCREEN-VALUE of the selected cell. Finally, the code makes the combo box visible and uses the MOVE-TO-TOP method to make sure that it's displayed on top of the browse and not hidden underneath it.

2. Define an ENTRY trigger for the **CreditCard** browse column to run the procedure when the user clicks in the column:

```
DO:  
  RUN placeCombo.  
END.
```

3. Define a LEAVE trigger for the **CreditCard** column:

```
DO:  
  cCreditCard:VISIBLE IN FRAME CustQuery = FALSE.  
END.
```

You want this trigger to execute when the user really leaves this column, for example, by selecting another column or another row. The overlay combo has finished its usefulness at that point and you should hide it so that the simple value in the cell itself is shown. However, there is one wrinkle to this.

As soon as the user clicks in the combo box to change its value, this action leaves the browse cell and then enters the combo box. The LEAVE trigger you just defined therefore hides the combo box, which is not at all what you want! So you have to define an ENTRY trigger for the combo box to make sure it is visible whenever it is being used.

4. Define an ENTRY trigger for the **cCreditCard** combo box to make sure it displays whenever it is selected:

```
DO:  
  SELF:VISIBLE = TRUE.  
END.
```

After the user has selected a value from the combo box, you need to hide the combo box, apply the new value to the browse cell, and move focus back into the browse cell.

5. To do this, write this VALUE-CHANGED trigger for **cCreditCard**:

```
DO:  
  cCreditCard:VISIBLE IN FRAME CustQuery = FALSE.  
  CreditCard:SCREEN-VALUE IN BROWSE OrderBrowse =  
    cCreditCard:SCREEN-VALUE.  
  APPLY "ENTRY" TO CreditCard IN BROWSE OrderBrowse.  
END.
```

Now your changes basically work, but the placement of the combo box is messed up if the user scrolls the browse when the combo box is visible.

6. Write this SCROLL-NOTIFY trigger for the **OrderBrowse** to reposition the overlay combo box properly:

```
DO:  
  RUN placeCombo.  
END.
```

Because you enabled column moving, you need to do the same thing on the START-SEARCH event so that in case the user moves the **CreditCard** column, the combo box moves with it.

7. Add this line to the START-SEARCH trigger for the browse:

```
DO:  
  DEFINE VARIABLE hColumn AS HANDLE      NO-UNDO.  
  DEFINE VARIABLE iColumn AS INTEGER      NO-UNDO.  
  hColumn = SELF:FIRST-COLUMN.  
  DO iColumn = 1 TO SELF:NUM-COLUMNS:  
    IF hColumn = SELF:CURRENT-COLUMN THEN LEAVE.  
    hColumn = hColumn:NEXT-COLUMN.  
  END.  
  IF iColumn NE 1 THEN  
    SELF:MOVE-COLUMN(iColumn, iColumn - 1).  
  RUN placeCombo.  
END.
```



To test your changes:

1. Run the window and select a **Credit Card** cell. The combo box overlays the cell:

| Order Num | Ordered | Promised | Credit Card | Shipped |
|-----------|----------|----------|------------------|----------|
| 3234 | 10/13/97 | 10/18/97 | Master Card | 10/18/98 |
| 3235 | 01/24/98 | 01/29/98 | American Express | |
| 3236 | 12/05/97 | 12/10/97 | VISA | |

2. If necessary, edit your code to adjust the width of the **Credit Card** column and the row height of the browse so that the combo box looks as though it really fits into the cell.
3. Select the combo box, then select one of the valid choices:

| Order Num | Ordered | Promised | Credit Card | Shipped |
|-----------|----------|----------|---|----------|
| 3234 | 10/13/97 | 10/18/97 | Master Card | 10/18/98 |
| 3235 | 01/24/98 | 01/29/98 | American Express | |
| 3236 | 12/05/97 | 12/10/97 | Master Card American Express VISA | |

When you leave the field, this becomes the new value for the cell:

| Order Num | Ordered | Promised | Credit Card | Shipped |
|-----------|----------|----------|-------------|----------|
| 3234 | 10/13/97 | 10/18/97 | Master Card | 10/18/98 |
| 3235 | 01/24/98 | 01/29/98 | VISA | |
| 3236 | 12/05/97 | 12/10/97 | VISA | |

Browse style options

This section offers a few ideas for changing the look of your browse.

Using stacked labels

To use more than one line for your column labels, use the stacked label syntax. Here is an example:

```
DEFINE BROWSE CustBrowse QUERY CustQuery DISPLAY
  CustNum COLUMN-LABEL "Customer!Number"
  Name COLUMN-LABEL "Customer!Name" WITH 10 DOWN.
```

You must use the COLUMN-LABEL option instead of the LABEL option. The exclamation point character indicates the line breaks.

Justifying labels

Column labels in the browse are left-justified by default. You can use the C, L, and R options (Center, Left, Right) of the LABEL attribute to modify the justification of column labels:

```
DEFINE BROWSE CustBrowse QUERY CustQuery DISPLAY
  CustNum LABEL "Cust. No.":C
  Name WITH 10 DOWN.
```

Note the colon (:) syntax. To use this option, you must attach the justification option to the end of the LABEL option. So, even if you want to use the default labels, you need to re-enter them here in order to append the justification option.

Using color to distinguish updateable columns

You can make the updateable columns in your browse a different color. For example, you can make the read-only columns gray with black text and the updateable columns blue with yellow text. This code fragment assumes you defined variables to hold the standard color values:

```
ASSIGN CustNum:COLUMN-BGCOLOR IN BROWSE CustBrowse = gray-color  
CustNum:COLUMN-FGCOLOR IN BROWSE CustBrowse = black-color  
Name:COLUMN-BGCOLOR IN BROWSE CustBrowse = blue-color  
Name:COLUMN-FGCOLOR IN BROWSE CustBrowse = yellow-color.
```

Note: In character interfaces, the COLUMN-DCOLOR attribute specifies the column color.

Using color and font to distinguish cells

On top of your basic color scheme, you may want individual cells that have key values to display in a different color or font. For example, you might want to color overdue accounts in red. This kind of cell manipulation is only valid while the cell is in the viewport. For this reason, you need to use the special ROW-DISPLAY event to check each new row as it is scrolled into the viewport. See the “[Browse events](#)” section on page 12–19 for examples and implementation notes.

Establishing ToolTip information

The DEFINE BROWSE statement supports the TOOLTIP option. You can elect to specify a *ToolTip*, a brief text message string that automatically displays when the mouse pointer pauses over a browse widget for which a ToolTip value is defined. You can set a ToolTip value for a variety of field-level widgets. However, they are most commonly defined for button widgets.

Using a disabled updateable browse as a read-only browse

A browse with no enabled columns is considered a read-only browse. A read-only browse has certain limitations that you might want to circumvent by defining one or more enabled columns in the browse definition. You then disable those columns at run time by setting their READ-ONLY attribute to TRUE. Using an updateable browse that has had its enabled columns turned off by way of the READ-ONLY attribute provides these benefits:

- Column searching (Windows only).
- Selection behaviors closer to native Windows standards (Windows only).
- Row markers.
- Ability to enable the browse columns programmatically.

Resizable browse objects

In graphical interfaces, you (the programmer) and the user can:

- Resize the browse.
- Move the browse.
- Resize a column of the browse.
- Move a column of the browse.
- Change the row height of the browse.

Resizing the browse

To let the user resize a browse, you set the browse's RESIZABLE and SELECTABLE attributes to TRUE, as the following code fragment shows:

```
ASSIGN CustBrowse:RESIZABLE = TRUE  
      CustBrowse:SELECTABLE = TRUE.
```

To resize a browse through direct manipulation, the user clicks on the browse to display the resize handles, then drags a resize handle as desired.

To resize a browse programmatically, you set the browse's WIDTH-CHARS or WIDTH-PIXELS, HEIGHT-CHARS or HEIGHT-PIXELS, or DOWN attributes as desired.

The following code fragment programmatically resizes a browse to 50 characters wide by 40 characters high:

```
ASSIGN CustBrowse:WIDTH-CHARS = 50  
      CustBrowse:HEIGHT-CHARS = 40.
```

Moving the browse

To let the user move a browse, you set the browse's MOVABLE attribute to TRUE, as the following code fragment shows:

```
CustBrowse:MOVABLE = TRUE.
```

To move a browse at run time, the user drags the browse as desired. To move a browse programmatically, you set the browse's X and Y attributes as desired.

The following code fragment programmatically moves a browse to the point (50,50) (in pixels) relative to the parent frame:

```
ASSIGN CustBrowse:X = 50  
      CustBrowse:Y = 50.
```

Resizing the browse column

To let the user resize a browse column, use one of the following techniques:

- To let the user resize any column of a browse, you set the browse's COLUMN-RESIZABLE attribute to TRUE.
- To let the user resize a single column of a browse, you set the column's RESIZABLE attribute to TRUE.
- To resize a column through direct manipulation, the user drags a column separator horizontally.
- To resize a column programmatically, you set the column's WIDTH-CHARS or WIDTH-PIXELS attribute.

Moving the browse column

To let the user move the columns of a browse, use one of the following techniques:

- To let the user move any column of a browse, you set the browse's COLUMN-MOVABLE attribute to TRUE.
- To let the user move a single column, you set the column's MOVABLE attribute to TRUE.
- To move a column at run time, the user drags a column label horizontally. (If the user drags a column label to either edge of the viewport, Progress scrolls the viewport.)

To move a browse column programmatically, you use the MOVE-COLUMN method of the browse, as shown earlier.

Changing the row height of the browse

To let the user change the row height of a browse, you set the browse's ROW-RESIZABLE attribute to TRUE.

To change the row height of a browse, the user vertically drags a row separator that appears at run time.

To change the row height programmatically, you set the browse's ROW-HEIGHT-CHARS or ROW-HEIGHT-PIXELS attribute.

Additional attributes

When you set up a resizable browse, whether for user or programmatic manipulation, you can set additional attributes.

When setting up a browse for user manipulation, you can set the following attributes:

- **SEPARATORS** (attribute of the browse) — Indicates if the browse displays separators between each row and each column.
- **SEPARATOR-FGCOLOR** (attribute of the browse) — Sets the color of the row and column separators, if they appear.

When setting up a browse to resize or otherwise manipulate programmatically, you can set the following attributes:

- **EXPANDABLE** (attribute of the browse) — Indicates whether Progress extends the rightmost browse column to the right edge of the browse, covering any white space that might appear.
- **AUTO-RESIZE** (attribute of the browse column) — Indicates whether Progress automatically resizes the browse column if a change occurs in its font or its label's font or text.

User manipulation events

When the user moves or resizes a browse or one of its components, Progress fires one or more of the following events:

- START-MOVE
- END-MOVE
- START-RESIZE
- END-RESIZE
- START-ROW-RESIZE
- END-ROW-RESIZE
- SELECTION
- DESELECTION

These events do not fire when you manipulate the browse programmatically.

You do not have to monitor these events. This chapter includes them only for your reference. For example, you might want to write trigger code for one or more of these events.

For more information on these events, see the Events Reference section of *OpenEdge Development: Progress 4GL Reference*.

Using browse objects in character interfaces

Browse objects in character interfaces operate in two distinct modes: row mode and edit mode.

In row mode, the user can select and change focus among rows, as well as navigate (tab) to widgets outside the browse. In edit mode, the user can edit and tab between enabled cells within the browse. A user can only enter edit mode in an updateable browse, but can operate in row mode in either a read-only or an updateable browse.

Character browse modes

Figure 12–3 shows a character browse in row mode.

| Update Credit Limits | | | | |
|----------------------|-------------------------|--------------|-----------|--|
| Cust-Num | Name | Credit-Limit | Balance | |
| * | 1 Lift Line Skiing | 6,000 | 42,568.00 | |
| * | 2 Urpon Frisbee | 27,600 | 17,166.00 | |
| * | 3 Hoops Croquet Co. | 75,000 | 66,421.00 | |
| * | 4 Go Fishing Ltd | 15,000 | 689.00 | |
| * | 5 Match Point Tennis | 11,000 | 0.00 | |
| * | 6 Fanatical Athletes | 38,900 | 37,697.00 | |
| * | 7 Aerobics valine KY | 13,500 | 10,439.00 | |
| * | 8 Game Set Match | 15,000 | 3,373.00 | |
| * | 9 Pihtiputaan Pyora | 29,900 | 25,792.00 | |
| * | 10 Just Joggers Limited | 22,000 | 16,621.00 | |

Enter data or press F4 to end.

Figure 12–3: Character browse in row mode

The asterisks (*) are row markers that indicate editable rows in an updateable browse. Row markers do not appear:

- In a read-only browse.
- In an editable browse defined with the NO-ROW-MARKERS option.

In row mode, the highlight indicates the row that has focus and the close angle brackets (>) are tick marks that indicate selected rows. In a single selection browse, the tick mark follows the highlight because focus and selection are the same. In a multiple selection browse, focus is independent of selection, and tick marks indicate the selected rows. Figure 12–4 shows a character browse in edit mode.

| Update Credit Limits | | | | |
|----------------------|-------------------------|--------------|-----------|--|
| Cust-Num | Name | Credit-Limit | Balance | |
| * | 1 Lift Line Skiing | 6,000 | 42,568.00 | |
| * | 2 Urpon Frisbee | 27,600 | 17,166.00 | |
| * | 3 Hoops Croquet Co. | 75,000 | 66,421.00 | |
| *> | 4 Go Fishing Ltd | 84,840 | 689.00 | |
| * | 5 Match Point Tennis | 11,000 | 0.00 | |
| * | 6 Fanatical Athletes | 38,900 | 37,697.00 | |
| * | 7 Aerobics valine KY | 13,500 | 10,439.00 | |
| * | 8 Game Set Match | 15,000 | 3,373.00 | |
| * | 9 Pihtiputaan Pyora | 29,900 | 25,792.00 | |
| * | 10 Just Joggers Limited | 22,000 | 16,621.00 | |

Please enter a Credit Limit

Figure 12–4: Character browse in edit mode

In edit mode, all rows become deselected except the edited row. The highlight indicates the edited cell within the edited row. As the user moves from row to row during editing, the tick mark and highlight both move to indicate the row and cell being edited.

Control keys

To control operation of a character browse, Progress provides a set of dedicated control keys. These control keys apply differently in row mode or edit mode. However, in both modes, any control key that changes (or attempts to change) the selected row fires the VALUE-CHANGED event.

These control keys correspond to Progress key functions that you can redefine. On Windows, you can redefine them in the registry or in an initialization file. On UNIX, you can redefine them in the PROTERMCAP file. For more information on redefining control key functions, see the chapter on user interface environments in *OpenEdge Deployment: Managing 4GL Applications*.

Table 12–2 describes the key functions, default key labels, and operation of the row mode control keys.

Table 12–2: Row mode control keys

(1 of 2)

| Key functions | Default key labels | Description |
|---------------|--------------------|--|
| " "1 | SPACE BAR | Fires the VALUE-CHANGED event. In multiple selection mode, this key also selects and deselects the row that has focus, alternately displaying and erasing the tick mark. |
| CURSOR-LEFT | CURSOR-LEFT | Scrolls the browse horizontally one column to the left. |
| CURSOR-RIGHT | CURSOR-RIGHT | Scrolls the browse horizontally one column to the right. |
| CURSOR-DOWN | CURSOR-DOWN | Moves focus down one row. ² |
| CURSOR-UP | CURSOR-UP | Moves focus up one row. ² |
| END | ESC-. | Moves focus to the last row in the browse. ² |
| HOME | ESC-, | Moves focus to the first row in the browse. ² |
| PAGE-DOWN | ESC-CURSOR-DOWN | Pages down one full page of data. ² |
| PAGE-UP | ESC-CURSOR-UP | Pages up one full page of data. ² |

Table 12–2: Row mode control keys

(2 of 2)

| Key functions | Default key labels | Description |
|---------------|--------------------|--|
| REPLACE | ESC-R | Changes the browse to edit mode, places focus in the first enabled cell of the currently focused row, and fires the VALUE-CHANGED event. In multiple selection mode, this key also selects the focused row and deselects any other selected rows. If you specify NO-ROW-MARKERS in the browse definition or set the ROW-MARKERS attribute to FALSE, REPLACE has no effect. |
| RETURN | RETURN | Fires the DEFAULT-ACTION event. |
| TAB | TAB | Leaves the browse and sets input focus to the next sibling of the browse in the tab order. |

¹ Note that Progress has no key function identifier for the space bar. In code, you reference a character string containing a single space (" ").

² In single selection mode, this also fires the VALUE-CHANGED event because selection follows focus.

Table 12–3 describes the key functions, default key labels, and operation of the edit mode control keys.

Table 12–3: Edit mode control keys

(1 of 2)

| Key function | Default key label | Description |
|--------------|-------------------|--|
| " "1 | SPACE BAR | Enters a space or zeroes numeric data in the focused cell. |
| CURSOR-LEFT | CURSOR-LEFT | Moves the cursor one character to the left in the cell. (Does not leave the cell.) |
| CURSOR-RIGHT | CURSOR-RIGHT | Moves the cursor one character to the right in the cell. (Does not leave the cell.) |
| CURSOR-DOWN | CURSOR-DOWN | Moves focus down to the next cell in the column and fires the VALUE-CHANGED event. |
| CURSOR-UP | CURSOR-UP | Moves focus up to the previous cell in the column and fires the VALUE-CHANGED event. |

Table 12–3: Edit mode control keys

(2 of 2)

| Key function | Default key label | Description |
|---------------------|--------------------------|---|
| BACK-TAB | CTRL-U | Moves focus to the previous enabled cell in the browse (right to left, bottom to top). ² When focus is on the first cell, BACK-TAB does not function. |
| EDITOR-TAB | CTRL-G | Moves focus to the next enabled cell in the browse (left to right, top to bottom). ² When focus is on the last cell, the EDITOR-TAB does not function. |
| END | ESC-. | Moves focus to the last cell in the current column and fires the VALUE-CHANGED event. |
| HOME | ESC-, | Moves focus to the first cell in the current column and fires the VALUE-CHANGED event. |
| PAGE-DOWN | ESC-CURSOR-DOWN | Pages down one full page of data and fires the VALUE-CHANGED event. |
| PAGE-UP | ESC-CURSOR-UP | Pages up one full page of data and fires the VALUE-CHANGED event. |
| REPLACE | ESC-R | Changes the browse to row mode, with selection set to the currently focused row. |
| RETURN | RETURN | Moves focus to the next cell in the current column and fires the VALUE-CHANGED event. |
| TAB | TAB | Leaves the browse and sets input focus to the next sibling widget of the browse in the tab order. |

¹ Note that Progress has no key function identifier for the space bar. In code, you reference a character string containing a single space (" ").

² If this action changes or attempts to change the selected row, it also fires the VALUE-CHANGED event.

Functional differences from the Windows graphical browse

The character browse shares most of the same functional capabilities of the Windows graphical browse, including all methods and most attributes and events.

However, there are differences from the Windows graphical browse in:

- Font management.
- Color management.
- Row and cell navigation.

Font management

Because there is no font management within character interfaces, all font attributes are inactive for the character browse.

Color management

For color terminals, the character browse supports the following attributes to manage its color:

- **LABEL-DCOLOR** — Specifies the color of a column label (like LABEL-FGCOLOR for the Windows browse).
- **COLUMN-DCOLOR** — Specifies the color of a single column (like COLUMN-FGCOLOR for the Windows browse).
- **DCOLOR** — Specifies the color of an individual cell (like FGCOLOR for the Windows browse). You can only specify the color of an individual cell as it appears in the view port. For more information on specifying individual cell color, see the “[Browse events](#)” section on page 12–19.

- **COLUMN-PFCOLOR** — Specifies the color of the enabled (updateable) column with input focus (unsupported for the Windows browse).
- **PFCOLOR** — Specifies the color of the updateable cell with input focus (handled by default processing for the Windows browse).

By default, the COLUMN-PFCOLOR and PFCOLOR values are both set from the INPUT color value. On Windows, you can specify this value in the registry or in an initialization file. On UNIX, you can specify this value in the PROTERMCAP file.

Row and cell navigation

Unlike the Windows graphical browse, the character browse does not support column searching (positioning to a row or cell by typing a character contained in that row or cell).

Unlike the Windows graphical browse, the character browse uses a different set of key functions to tab between updateable cells than between the browse and other sibling widgets. Tabbing between cells occurs only in the edit mode of the character browse using the EDIT-TAB and BACK-TAB key functions.

Tabbing forward from the character browse to a sibling object occurs in either row mode or edit mode using the TAB key function. However, tabbing backward from the character browse to a sibling object occurs only in row mode using the BACK-TAB key function.

Conclusion

This chapter has introduced you to some of the ways in which you can use the browse and customize its behavior. The browse is a complex object, and one that you can manipulate in many ways—more than can be completely documented in a single chapter. To explore the complete capabilities of the browse, you should consult the Browse Widget section of *OpenEdge Development: Progress 4GL Reference* or the online help. These provide both a list of all attributes and methods, and an explanation of each. Just by way of example of how extensive the browse capabilities are, Figure 12–5 shows some of the browse methods in the online Help.

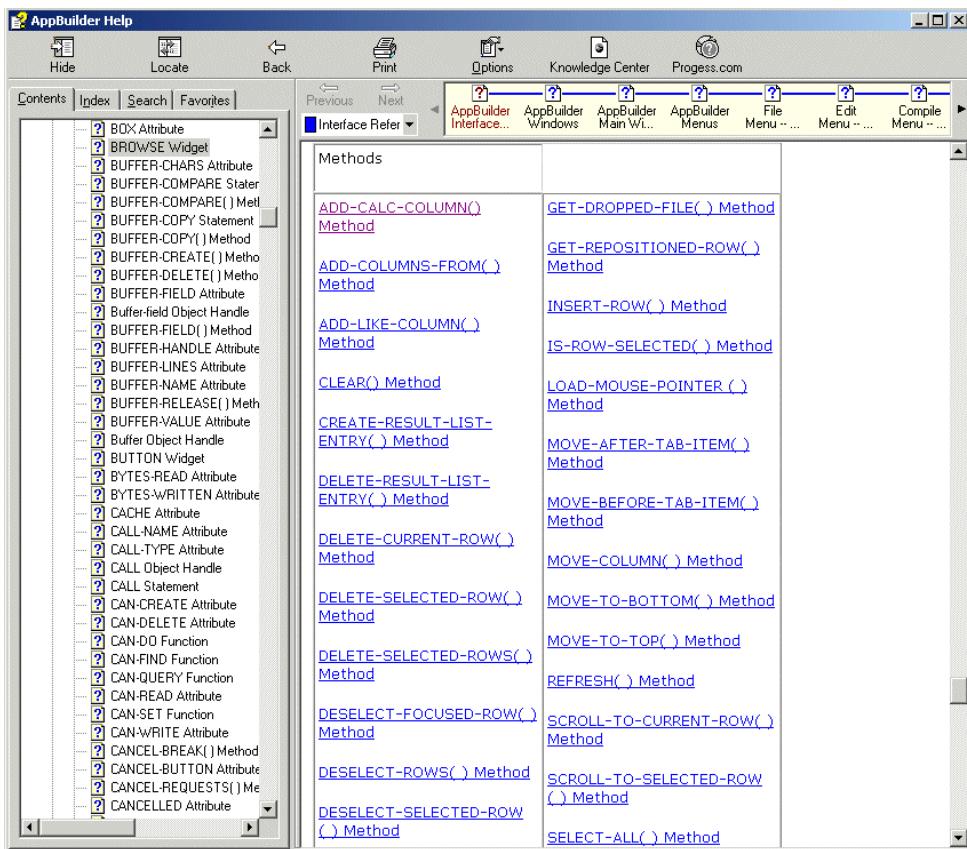


Figure 12–5: Browse widget online help topic

Advanced Use of Procedures in Progress

In [Chapter 2, “Using Basic 4GL Constructs,”](#) you learned how to write both internal and external procedures. In [Chapter 3, “Running Progress 4GL Procedures,”](#) you learned how to use the RUN statement to invoke one procedure from another. There’s a lot more to using procedures in Progress, however, and this chapter covers these additional topics. You’ll learn how to:

- Treat procedures as objects with standard behavior and independence from other procedures.
- Build what amounts to classes of behavior using multiple separate procedures that all contribute to the definition of the application’s behavior.
- Use procedures in ways that go beyond the top-down calling sequence you’ve been introduced to.

It includes the following sections:

- [RETURN statement and RETURN-VALUE](#)
- [Using persistent procedures](#)
- [Shared and global objects in Progress procedures](#)

RETURN statement and RETURN-VALUE

First you'll look at a small, but important, procedure topic. Whenever a procedure, whether internal or external, terminates and returns control to its caller, it returns a value to the caller. You can place a RETURN statement at the end of your procedure to make this explicit, and to specify the value to return to the caller:

```
RETURN [ return-value ].
```

The *return-value* must be a character value, either a literal or an expression. The caller (calling procedure) can access this *return-value* using the built-in RETURN-VALUE function, such as in this example:

```
RUN subproc (INPUT cParam).
IF RETURN-VALUE = ... THEN
.
.
```

If the called procedure doesn't return a value, then the value returned is either the empty string ("") or, if an earlier procedure RUN statement in the same call stack returned a value, then that value is returned.

The RETURN statement is optional in procedures. Because an earlier RETURN-VALUE is passed back up through the call stack if there's no explicit RETURN statement to erase it, it is good practice to have the statement RETURN "" at the end of every procedure to clear any old value, unless your application needs to pass a RETURN-VALUE back through multiple levels of a procedure call.

In addition, you can return to the calling procedure from multiple places in the called procedure by using multiple RETURN statements in different places. You could use this technique, for example, to return one *return-value* representing success (possibly the empty string) and other *return-values* from different places in the procedure code to indicate an error of some kind.

You can also use RETURN ERROR from a procedure to signal an error. You'll learn more about this option in [Chapter 16, “Updating Your Database and Writing Triggers.”](#)

Using persistent procedures

As you have seen in your work thus far, you can run one procedure from another. When you do this, you can pass INPUT parameters that are used by the procedure you run. When that procedure gets to the end of its 4GL code, it terminates and returns any OUTPUT parameters to the caller. This works the same for both external procedures (independent procedure source files) and for the internal procedures inside them.

These procedures form a *call stack*. On top of the stack is the main procedure you run from the editor or from the OpenEdge startup command line. Below this is another procedure the main procedure runs, which in turn can run another procedure that is added to the stack, and so on. As each procedure terminates, it returns control to the procedure above it and is removed from the stack. All the resources it uses are freed up, and the procedure effectively disappears from the session unless it is run again. [Figure 13–1](#) provides a rough sketch of how the procedure call stack can look.

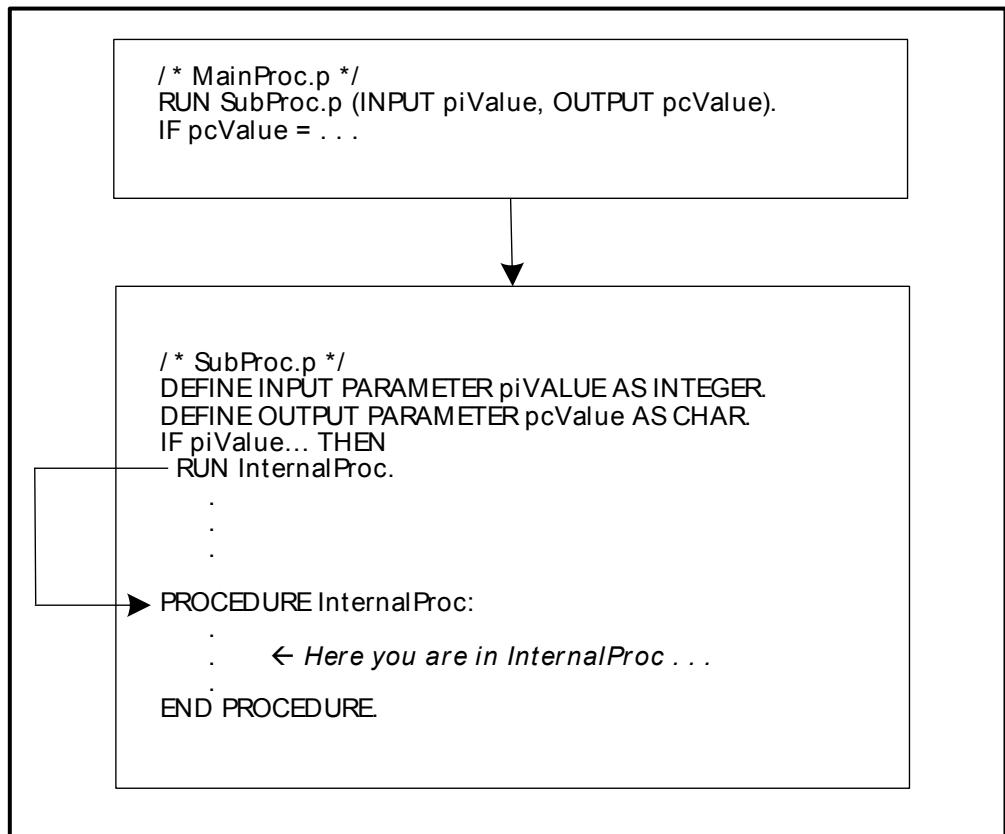


Figure 13–1: A procedure call stack

In this example, `MainProc.p` runs `SubProc.p`. `SubProc.p` runs an internal procedure (`InternalProc`) inside it. While the Progress interpreter is executing code in `InternalProc`, the call stack looks like the diagram shown in [Figure 13–2](#).

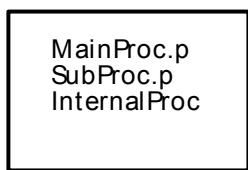


Figure 13–2: Call stack in InternalProc

What does this really mean? All three of these procedures have been instantiated and are currently in the session's memory. What is actually in memory is the r-code for the procedure, which the interpreter understands, whether this was a .r file you previously compiled and saved or whether Progress has to compile the procedure on the fly at run time from its source code.

It also means that all the elements that each procedure defines are currently in scope. Any variables, buffers, and other procedure objects have been instantiated along with the executable r-code. Progress has allocated the memory and other resources needed to execute them.

It also means that control of the application flow is fairly rigid. The code in `MainProc.p` cannot do anything at all while `SubProc.p` is running. Likewise, the code in the main block of `SubProc.p` cannot do anything at all while `InternalProc` is running. The three procedures together form a single thread that cannot be interrupted except to cancel the whole program execution.

When `InternalProc` returns to the rest of `SubProc.p`, any resources it defined, such as local variables, are freed up. `SubProc.p` can go on with the rest of its program logic. When `SubProc.p` returns to `MainProc.p`, all the resources allocated to it, including the in-memory copy of its executable code, are released and disappear. The code in `MainProc.p` has no way of accessing anything in `SubProc.p` except by receiving it back in the form of an OUTPUT parameter, because while `SubProc.p` is running, `MainProc.p` is completely blocked. When `SubProc.p` returns control to `MainProc.p`, it completely disappears.

This is how generations of Progress 4GL applications (along with applications in many other languages) were written. A typical character-mode application has a top-level menu of some kind defined in the equivalent of `MainProc.p`. Depending on what the user requests, some subprocessure runs and the main procedure waits until it completes. A subprocessure can then run some additional subprocessure that performs a service for it. The result is a very hierarchical application, both in its physical construction and in its user-visible execution. To be sure, Progress developers have created very sophisticated techniques for allowing users to jump easily from one part of such an application to another with shortcut keys and the like, but these techniques mostly mask the inherent programming style of the language at that time. In the world of character screen applications without a mouse or much in the way of user-initiated events, and where all the user terminals are connected to a single host process that's running the application itself and getting data from the database, this all works very well.

But in a more modern application, written for a graphical environment, designed to be distributed between client user interface sessions and separate server sessions that control the database, with the need to communicate with a variety of devices and other applications and to make the user much more in control of the application flow than before, this programming style is not sufficient. It is for these reasons that Progress includes the many event-driven language constructs that you've been introduced to: object events, the `ON event` statements, the `APPLY` statement, and so on.

The procedures in your application have the same kinds of requirements that the visual objects you've worked with do. They must have greater flexibility in how they interact with each other, without depending on a rigid hierarchy of execution.

Running a procedure PERSISTENT

For all these reasons, Progress lets you run a procedure so that it stays in memory for as long as you need it, without it being dependent on, or in any way subordinate to, the procedure that runs it. A procedure run in this way is called a *persistent procedure*. You use this syntax to run it:

```
RUN procedure-name PERSISTENT [ SET proc-handle ] [ ( parameters ) ].
```

The `PERSISTENT` keyword in the statement tells Progress to start the procedure and to leave it in memory, either until you delete it or until your session ends.

Optionally, you can use the SET phrase to save off the handle of the new procedure. You've seen handle variables already. Of particular significance is that persistent procedures have handles the same as other objects do, so that you can access them after they've been instantiated by a RUN PERSISTENT statement. In fact, even a procedure that you run without the PERSISTENT option has a handle, but you would rarely use it. As noted above, the calling procedure can't really execute any code until the procedure it runs terminates, and when that happens the subprocedure's handle goes away, so the calling procedure would never really have a chance to use it.

In practice, you will almost always use the SET *proc-handle* option, because the way you make use of a persistent procedure is to initiate it with a RUN statement and then to use its handle afterwards to run entry points inside it. The *proc-handle* must be a variable or temp-table field with the HANDLE data type, defined or accessible in the calling procedure.

THIS-PROCEDURE built-in function

Whenever you run a procedure, persistent or not, you can retrieve its procedure handle using the built-in function THIS-PROCEDURE. This is useful when you want to access attributes or methods of the current procedure. There are some examples of this later in the “[Useful procedure attributes and methods](#)” section on page 13–13.

In most cases, you use procedure handles and the THIS-PROCEDURE function when you run persistent procedures. However, keep in mind that every running procedure, whether it is persistent or not, has a procedure handle that is held in THIS-PROCEDURE. A non-persistent procedure can pass THIS-PROCEDURE as an INPUT parameter to another procedure, and the subprocedure can use that value to access internal procedures and other elements of the parent procedure. This is definitely not the norm for programming with procedure handles, but you can use it as a way to pass procedures down through a call stack, as the following example shows. This parent procedure passes its own procedure handle to another procedure as a parameter:

```
/* Procedure parentproc.p -- this runs another procedure and passes in its
own procedure handle. */
RUN childproc.p (INPUT THIS-PROCEDURE).

PROCEDURE parentInternal:
  DEFINE INPUT PARAMETER cString AS CHARACTER NO-UNDO.
  MESSAGE "The child sent the parent " cString VIEW-AS ALERT-BOX.
END PROCEDURE.
```

The child procedure can then use the parent's handle to run the internal procedure in the parent:

```
/* childproc.p -- called from parentproc.p, it turns around and uses
   the parent's procedure handle to run an internal procedure inside it. */
DEFINE INPUT PARAMETER hParent AS HANDLE      NO-UNDO.
RUN parentInternal IN hParent (INPUT "this child message").
```

The message shown in [Figure 13–3](#) results.



Figure 13–3: Message result of childproc.p

This technique has limited usefulness, because there is no way for the parent to access internal procedures or other elements of the child procedure it runs. Only the child can access the parent procedures's information.

Instantiating the persistent procedure

What happens when you run a procedure PERSISTENT? You already know that a procedure has a main block, which has all the code and definitions that are scoped to the procedure itself. Another way of saying this is that this is everything in the procedure file that *isn't* in an internal procedure or a trigger. When you run a procedure PERSISTENT, its main block executes and then returns to the caller, just as it would without the PERSISTENT keyword. The difference is that when you run it PERSISTENT, the procedure stays in memory so that you can run internal procedures in it later on. The diagram in [Figure 13–4](#) shows how this works.

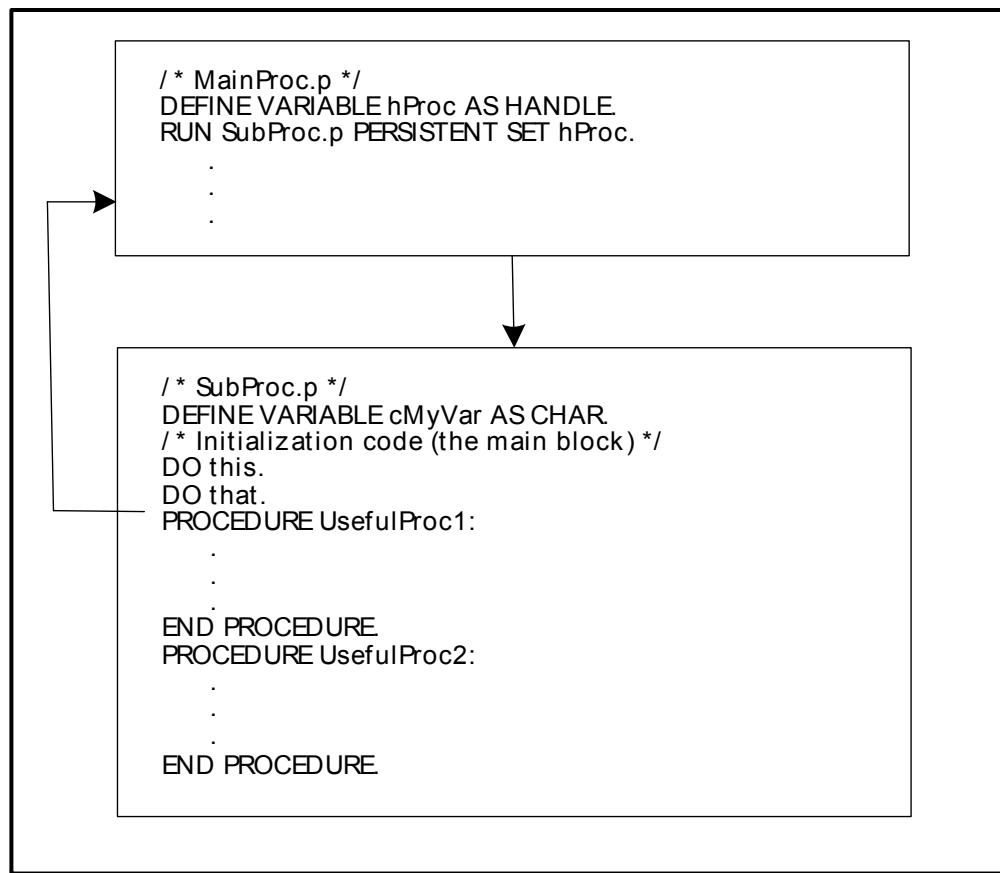


Figure 13–4: Instantiating a persistent procedure

The code in [Figure 13–4](#) executes as follows:

1. `MainProc.p` RUNs `SubProc.p` PERSISTENT and saves off its procedure handle in the `hProc` variable. The main block of `SubProc.p` defines a variable and then executes the startup code represented by the `DO this` and `DO that` statements.
2. The instantiation of the persistent procedure `SubProc.p` is complete. It returns control to `MainProc.p`, passing back through the `SET` phrase the procedure handle it's been given. `SubProc.p` now is removed from the call stack. At this point, and for the duration of `MainProc.p`, the `hProc` variable holds the procedure handle of the running instance of `SubProc.p`.

You'll see in the “[GET-SIGNATURE method](#)” section on page 13–14 how you can make use of this handle.

Now that `SubProc.p` is in memory, `MainProc.p` (or any other procedure with access to its handle) can make use of it by running `h-UsefulProc1` and `h-UsefulProc2` whenever it needs to.

Parameters and persistent procedures

You can pass parameters to a persistent procedure just as you can to any other procedure. If you pass INPUT parameters to it, they are available throughout the life of the persistent procedure. If you pass OUTPUT parameters to it, their values are returned to the caller at the end of the persistent procedure’s instantiation. This is represented in [Figure 13–4](#) by the arrow that returns control to `MainProc.p`. (Here and elsewhere in this discussion, you should understand that INPUT-OUTPUT parameters have the same characteristics as both INPUT and OUTPUT parameters where this is concerned.)

In practice, you should not normally pass parameters of either kind to a persistent procedure. This is because the best model for using persistent procedures is to initiate them with a `RUN` statement, and then to access the procedure’s contents afterwards with other statements. This is strictly a matter of programming style—there’s nothing wrong with passing parameters per se. Perhaps the best argument for *not* doing it is that if you instantiate a persistent procedure simply by running it with no parameters, you maximize the flexibility of how you access it. If it has parameters, then you must always be sure to pass the right parameters when you start it up, which can introduce maintenance problems if the parameter list must change.

Using a persistent procedure as a run-time library

The example in the previous section points to the first major use for persistent procedures. When you learned about internal procedures in [Chapter 2, “Using Basic 4GL Constructs,”](#) the examples all involved running an internal procedure from elsewhere in the same main procedure file (external procedure). But internal procedures are such a useful thing that you will want to run them from many different places. The only way to run an internal procedure that isn’t local to the caller is with a persistent procedure. In this way, running an internal procedure defined in some other procedure file is a two-step process:

1. Run the procedure file that contains it as a persistent procedure, or alternatively, to obtain the handle of an instance of the procedure that’s already running.
2. Run the internal procedure in that handle, using this syntax:

```
RUN internal-proc IN proc-handle [ ( parameters ) ].
```

If the internal procedure has parameters, you pass them in the usual way.

This allows you to think of a persistent procedure as a library of useful routines you want to be able to run from anywhere in your application. Because the rigid top-down hierarchy of the older application is gone, your application code can run any entry point in any running procedure instance from anywhere in the application. This greatly increases the flexibility of your application and its logic flow.

Obtaining a procedure handle for a persistent procedure

This should lead you to think about how you can make procedure handles available throughout your application. Clearly, the procedure that instantiates the persistent procedure can get its handle back easily with the SET phrase on the RUN statement. But your procedures can be more useful if their handles are more widely available.

Using the SESSION handle to locate running procedures

There is a very useful built-in handle available globally in your application and, among other advantages, is one way to identify all running persistent procedures. This is the SESSION handle. The SESSION handle has many attributes and methods, all of which you can learn about in the online help. You can also see a list of all SESSION attributes by accessing the **Session Attributes** tool from the PRO*Tools palette, as shown in Figure 13–5.



Figure 13–5: PRO*Tools palette with Session icon selected

Figure 13–6 shows the **Session Attributes** tool.

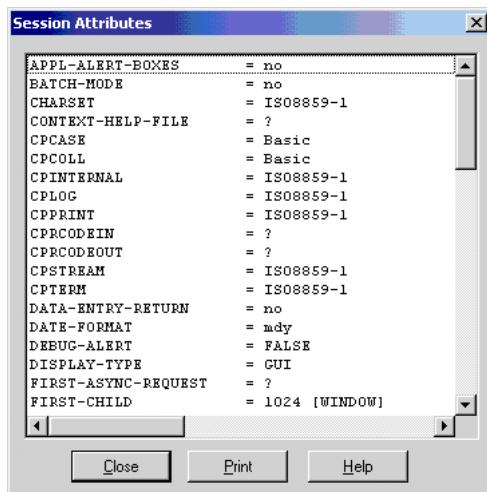


Figure 13–6: Session Attributes PRO*Tool

Some of these attributes have readable values that are meaningful to you when you look at them as a list. However, handles are not useful just as numbers (such as the FIRST-CHILD handle value, shown in Figure 13–6).

Your application, however, can make good use of these handles. If you want to examine a list of all running persistent procedures, you start with the SESSION:FIRST-PROCEDURE attribute. This attribute evaluates to the procedure handle of the first of a list of all running procedures. Note that there is no predictable sequence to the procedures in the list. All you can do is scan the list if you need to search for a procedure you're interested in. From here, you can walk through a sequence of all procedures using the NEXT-SIBLING attribute, which also returns a procedure handle, or with the PREV-SIBLING attribute if you want to walk backwards through the list. The last procedure in the list is available using the LAST-PROCEDURE attribute.

Here's a very simple example. h-FindUseful.p runs another procedure, h-UsefulProc.p, persistent. Then it searches the session's procedure list for an internal procedure it wants to run:

```
/* h-FindUseful.p -- searches for a persistent procedure with a useful
   routine to run. */
DEFINE VARIABLE hUseful AS HANDLE      NO-UNDO.
RUN h-UsefulProc.p PERSISTENT SET hUseful.

DEFINE VARIABLE hProc AS HANDLE      NO-UNDO.
hProc = SESSION:FIRST-PROCEDURE.
DO WHILE VALID-HANDLE(hProc):
  IF LOOKUP ("UsefulRoutine1", hProc:INTERNAL-ENTRIES) NE 0 THEN
    DO:
      RUN UsefulRoutine1 IN hProc.
      LEAVE.
    END.
    hProc = hProc:NEXT-SIBLING.
  END.

DELETE PROCEDURE hUseful.
```

In a more realistic example, of course, the same procedure that started h-UsefulProc.p would not be the one searching for it. h-UsefulProc.p defines some internal procedures that other procedures in the session would like to run:

```
/* h-UsefulProc.p -- has an internal procedure others want to find. */

PROCEDURE UsefulRoutine1:
  MESSAGE "It would be useful if you ran this."
  VIEW-AS ALERT-BOX.
END PROCEDURE.

PROCEDURE UsefulRoutine2:
  MESSAGE "This would be useful too."
  VIEW-AS ALERT-BOX.
END PROCEDURE.
```

When you run `h-FindUseful.p`, it locates the running instance of `h-UsefulProc.p` by looking at every procedure's INTERNAL-ENTRIES attribute, and runs the routine it's looking for, as shown in [Figure 13–7](#).

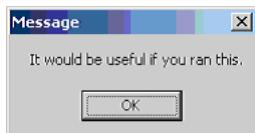


Figure 13–7: `h-UsefulProc.p` message

Useful procedure attributes and methods

There are many procedure attributes you can access through the procedure's handle. This section introduces you to a few of them.

FILE-NAME attribute

After you run a persistent procedure, you can query its FILE-NAME attribute to see the name of the procedure file, including any pathname it was run with. This attribute can be useful to identify a procedure so that you can run something in its handle or access other attributes.

PRIVATE-DATA attribute

There is a character attribute available on every handle, including procedure handles, called PRIVATE-DATA. It is there specifically so that you can store any value you want on the handle, to help identify the object it points to, or to store other information about the object. The PRIVATE-DATA attribute is not used by Progress in any way; it is strictly there for your application use. You could, for example, store a category of object in the PRIVATE-DATA handle when you run the procedure, or the value of one or more parameters that were passed in to the procedure to give it a distinctive function. Because you might run the same procedure file in numerous ways, for example with different input parameters or different settings that you establish in other ways, the PRIVATE-DATA attribute gives you the ability to identify it in any way you need to.

INTERNAL-ENTRIES attribute

The INTERNAL-ENTRIES attribute returns a comma-separated list of all the internal procedures in the procedure file, and also any user-defined functions, which you'll learn about a little later. This can help you identify the location of a routine you need to run.

GET-SIGNATURE method

Given the name of any internal procedure or function in a procedure file, you can get its *signature*, that is, the list of parameters it requires, using the GET-SIGNATURE method on the procedure handle. This is considered a method rather than an attribute simply because it requires an input parameter, the name of the routine you want the signature for. You can also pass in a blank argument to get the signature of the external procedure itself.

GET-SIGNATURE returns a string in this format:

```
type, fn-return-type, mode name data-type [ , mode name data-type ], . . .
```

The *type* is the type of entry point and its possible values are:

- **MAIN** for the external procedure itself (when you pass in an empty string).
- **PROCEDURE** for an internal procedure.
- **FUNCTION** for a user-defined function. You'll learn about user-defined functions in [Chapter 14, “Defining Functions and Building Super Procedures.”](#)

The *fn-return-type* is the data type that a function returns. For an internal procedure, this element of the signature is blank.

Following the second comma is a list of the parameter descriptions. Each description is a space-delimited list of three elements:

- The *mode* is INPUT, OUTPUT, or INPUT-OUTPUT.
- The *name* is the parameter name as defined in the entry point.
- The *data-type* is the data type of the parameter.

As an example, here is a very simple procedure file that contains an internal procedure and a user-defined function:

```
/* h-testsig.p -- tests GET-SIGNATURE */
DEFINE INPUT PARAMETER cVar AS CHAR.
PROCEDURE TestProc:
    DEFINE OUTPUT PARAMETER iValue AS INT.
    /* some procedure code */
END.
FUNCTION TestFunc RETURNS INTEGER (INPUT dValue AS DECIMAL):
    /* some function code */
END.
```

The `h-testsig.p` procedure takes an `INPUT` parameter, the `TestProc` internal procedure uses an `OUTPUT` parameter, and the `TestFunc` function takes an `INPUT` parameter and returns the data type `INTEGER`.

Here's another procedure that uses `GET-SIGNATURE` to get the signatures of all these routines:

```
/* Procedure h-mainsig.p -- uses GET-SIGNATURE to return the signatures
   of a procedure and its internal-entries */
DEFINE VARIABLE hProc      AS HANDLE      NO-UNDO.
DEFINE VARIABLE iProc      AS INTEGER     NO-UNDO.
DEFINE VARIABLE cEntries  AS CHARACTER   NO-UNDO.
DEFINE VARIABLE cMessage  AS CHARACTER   NO-UNDO.

RUN h-testsig.p PERSISTENT SET hProc (INPUT "aa").

cEntries = hproc:INTERNAL-ENTRIES.
cMessage = THIS-PROCEDURE:FILE-NAME + ":" + hproc:GET-SIGNATURE("") .
DO iProc = 1 TO NUM-ENTRIES(cEntries):
    cMessage = cMessage + CHR(10) +
               ENTRY(iProc, cEntries) + ":" +
               hproc:GET-SIGNATURE(ENTRY(iProc, cEntries)).
END.
MESSAGE cMessage VIEW-AS ALERT-BOX.
```

This procedure executes as follows:

1. `h-mainsig.p` runs `h-testsig.p` persistent and saves off its procedure handle.
2. `h-mainsig.p` gets the `INTERNAL-ENTRIES` attribute of the procedure, which should have the value `TestProp,TestFunc`.

3. It begins to build up a character string to display later with a MESSAGE statement. The first entry in the message string is THIS-PROCEDURE:FILE-NAME. Because the built-in handle function THIS-PROCEDURE evaluates to the handle of the currently running procedure, this should return its filename, h-mainsig.p. Passing the empty string to GET-SIGNATURE returns the signature of the external procedure itself.
4. The code loops through all the entries in the INTERNAL-ENTRIES attribute, and for each once, saves off its name and signature.
5. To simulate a series of SKIP keywords that you could put into a MESSAGE statement, the code adds a new line character after the signature of each entry. The CHR(*n*) Progress built-in function takes an INTEGER value that represents the ASCII character code of any character, whether printable or not, and returns that ASCII character. In this case, 10 represents the new line character.

Figure 13–8 shows the message you see when you load h-mainsig.p into an editor window and run it.



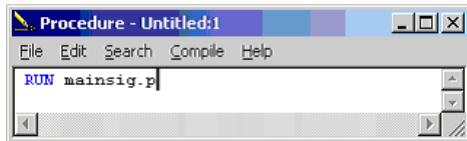
Figure 13–8: Message box for h-mainsig.p

Interesting. The value of THIS-PROCEDURE:FILE-NAME isn't what you expect. Instead you see what looks like a temporary filename. And that's exactly what it is. When you run a procedure from the Procedure Editor or from the AppBuilder just by loading it or typing code into the editor and then choosing the **Run** button or pressing **F2**, the tool doesn't actually run the saved procedure. In fact, you might not yet have saved the procedure file at all, or you might have changed it since you saved it. What you run is the code that is currently in the editor window or the AppBuilder's **Section Editor**, depending on which tool you're using. The tool gives this code a temporary filename, saves it off into that file, and runs it from there. That's why you see this filename in the message box.

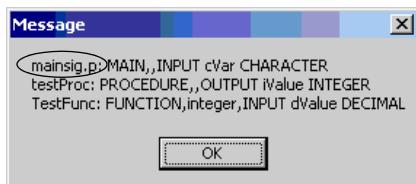


To see the filename you expected, rather than loading `h-mainsig.p` into an editor window and running it from there:

- Type this statement into a **New Procedure** window:



- Run this to see the result you expect:



Using a persistent procedure as shared code

Because multiple procedures can access the handle of a persistent procedure, they can all share that single instance of the procedure, if appropriate. This feature lets you implement procedures that act as libraries of shared routines that many other procedures need. In this case, you need to make sure that the code in the procedure is truly shareable. This means that routines in the procedure should not save any context information that is needed between calls, because there is often no way of predicting whether another procedure might run the same routines at that time.

For example, suppose one internal procedure in the shared persistent procedure instance saves off a value in a variable defined in the **Definitions** section of the procedure. The variable identifies the calling procedure in some way, storing an attribute that is used in the program logic. Because the variable is scoped to the whole procedure file, its value is persistent between calls to various internal procedures and is accessible to all of them. Now suppose that some other procedure runs the same routine in the persistent procedure, resetting the variable value. Then the original caller runs some second routine in the persistent procedure that needs that value. The value is then wrong, because the other requesting procedure placed its own request in between calls.

The general rule to avoid this kind of problem is: do not store any persistent data in a procedure that is going to be shared in this way. Use only variables and other values defined within the internal procedure that uses them, because these values go out of scope and are reset when the internal procedure call ends.

If you want to share a persistent procedure in your session, then there are a couple of ways you can do this:

- One way is to start all the persistent procedures your application needs at the very beginning of the session, or in the initialization of some module that uses them. You can save off their handles in a way similar to the examples you've already seen.
- Another technique is to structure your code in such a way that any procedure that wants to use the shared procedure needs to check to see whether a copy of it is already running, perhaps by looking it up in the same kind of procedure list or just by using the SESSION procedure list. If it's already running, the code uses the copy that's there. If it's not running yet, the code runs the procedure, saves off the handle to be available to others, and then uses it. There's another example of how to do this in code later in [Chapter 14, “Defining Functions and Building Super Procedures,”](#) in the section on super procedures.

Using a persistent procedure to duplicate code

In some cases, you might want to do just the opposite and create multiple running instances of the same procedure at the same time. Here's a simple example that extends the **Customers and Orders** window. Each time you select an **Order** in its browse, the procedure starts an instance of a separate window to display detail for the **Order**. The supposition is that you want to be able to see multiple **Orders** at the same time, so each **Order** you select is opened in a separate window and they all stay open until you close them or until you exit the main procedure.

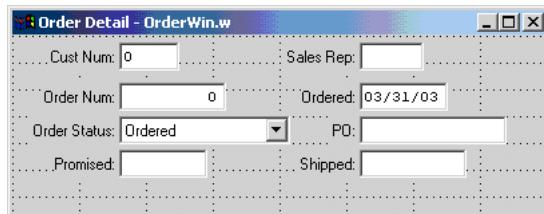


To use a persistent procedure to duplicate code:

1. Select **New→Window** in the AppBuilder. This gives you a new empty design window.
2. To give the window a title, press **CTRL-Click** on the frame to select the window itself and set the window title to **Order Detail**. C-Win is the AppBuilder's default name for the window (C for Container).



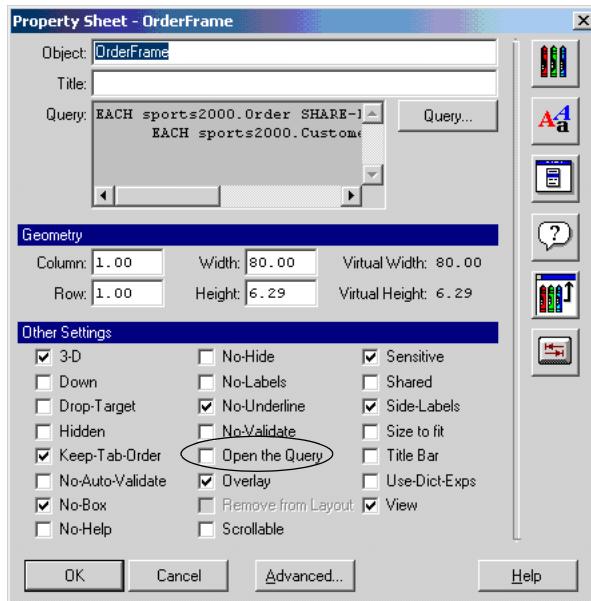
3. Choose the **DBFields** icon in the AppBuilder **Palette** and click on the new design window.
4. From the **Table Selector** dialog box, pick the **Order** table.
5. From the **Multi-Field Selector** dialog box, select some of the **Order** table fields, including the **OrderNum**, **CustNum**, **OrderDate**, **PromiseDate**, and **ShipDate**.
6. Arrange these fields in your new window so that they look something like this:



As you remember from [Chapter 4, “Introducing the OpenEdge AppBuilder,”](#) the AppBuilder generates a query and other supporting code for you whenever you drop database fields onto a window. In this case, you don't need the query because all you're going to do is display fields from an **Order** retrieved in the main window.

7. Double-click on the frame (outside of any field) to bring up its property sheet.

8. Uncheck the **Open the Query** toggle box so that the code doesn't try to open a new **Order** query when the window is opened:



Now you're going to pass in the current **Order** number as an input parameter to the procedure.

9. In the **Definitions** section, add a parameter definition for the **Order** buffer:

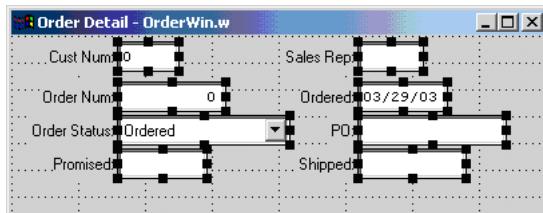
```
/* ***** Definitions *****/
/* Parameters Definitions --- */
DEFINE PARAMETER BUFFER Order FOR Order.
```

Note the special syntax you use when you pass a buffer as a parameter. The buffer is not defined explicitly as an INPUT or OUTPUT parameter. In effect, it acts as an INPUT-OUTPUT buffer. That is, both the calling and called procedure share the buffer so that changes made to the buffer in the called procedure are visible in the calling procedure.

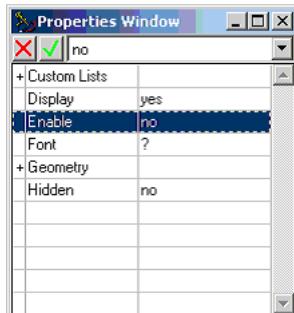
In this case, you don't change the name of the buffer because the AppBuilder gives you a frame definition based on the fields in the **Order** table. You could change those definitions to use a different buffer name or even to treat the fields just as fill-ins without any connection to the database, but the simplest thing for the example is just to use the definition the AppBuilder provides.

For this example, you don't need to change any data in the **Order** window (that comes in a later chapter), so the easiest way to disable all the fields is to use the **Properties Window**.

10. Choose all the fields in the **Order** window individually (with **CTRL-Click**) or by dragging a box around them with the mouse:



11. Select **Window→Properties Window** from the AppBuilder menu.
12. Double-click on the **Enable** property to set it to **no**:



13. Choose the green arrow to register the change.

If you also want to set the **Read-Only** attribute that puts a box around each field, you can only do this in the property sheet for each field individually.

This is all you have to do to the new window. The generated code displays the record automatically, and that's all this procedure needs to do.

14. Save the new procedure as **h-OrderWin.w**.



To add some code to the main window to start up an instance of **h-OrderWin.w** for each Order you want to see:

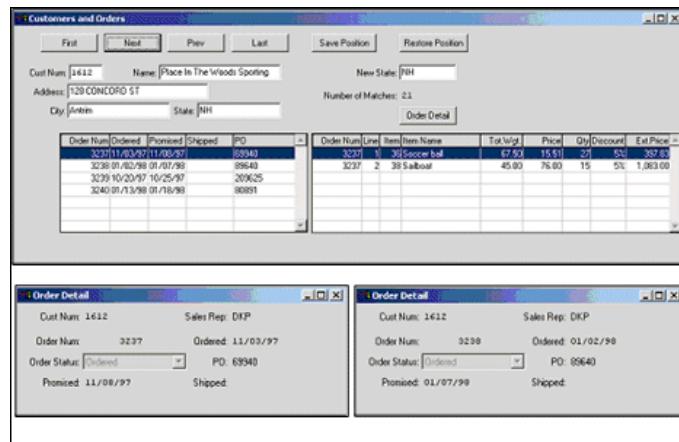
1. Open **h-CustOrderWin5.w** and save it as **h-CustOrderWin6.w**.
2. In **h-CustOrderWin6.w**, drop a new button onto the design window next to the **Order** browse.
3. Name it **BtnOrder** and give it a label of **Order Detail**.
4. In the **Section Editor**, give the new button this CHOOSE trigger:

```
DO:  
  DEFINE VARIABLE hOrder AS HANDLE      NO-UNDO.  
  RUN h-OrderWin.w PERSISTENT SET hOrder (BUFFER Order).  
END.
```

The code runs **h-OrderWin.w** as a persistent procedure, passing in the record buffer for the current **Order**. It sets the **hOrder** local variable to the procedure handle. Because you don't need the order handle outside of this trigger, you can define the **HANDLE** variable locally to the trigger.

In fact, the code isn't using the handle at all yet, but it will shortly. It is almost never the case that you run a persistent procedure without saving its handle for one purpose or another. In this case, you'll add code just below to clean up the procedure when the main window is closed.

5. Run the window and choose the **Order Detail** button several times to see different **Orders** displayed, each in its own window:



You've created multiple running instances of the same persistent procedure. Each has its own memory and its own data, so that they can display different **Order** records. If you want to identify them from other procedures, you can use the PRIVATE-DATA field to store off the **Order** number to go along with the procedure name, which is the same for each instance.

6. You can close the **Order Detail** windows either individually or when you close the main window, they all go away.

Deleting persistent procedures

Whenever your code is done using a persistent procedure, you must remember to delete it to clean up after yourself. Use this statement:

```
DELETE PROCEDURE procedure-handle [ NO-ERROR ].
```

The NO-ERROR option suppresses any error message, for example, if the procedure has already been deleted elsewhere. You can also use the VALID-HANDLE function, which you have seen in AppBuilder-generated code, to check whether a handle is still valid or not, such as in this example:

```
IF VALID-HANDLE(hProc) THEN  
  DELETE PROCEDURE hProc.
```

When you close one of the **Order** windows by clicking the close icon in the right-hand corner, the window disappears. Is its procedure really gone? Yes, but this doesn't happen automatically. It's very important for you to make sure that you remember to clean up persistent procedures when your application is done with them, if the AppBuilder doesn't do it for you.

In this case, the AppBuilder generated just the code you need to make sure the procedure is deleted when you close the window. You might remember this code from [Chapter 4, “Introducing the OpenEdge AppBuilder,”](#) but it should mean a lot more to you now because you understand about the THIS-PROCEDURE handle, the PERSISTENT attribute, and procedure handles. The code is in the `disable_UI` internal procedure. To review, here's the sequence again:

1. The main block, which executes as soon as the procedure starts up, sets up a trigger to RUN the internal procedure `disable_UI` on the CLOSE event:

```
/* The CLOSE event can be used from inside or outside the procedure to */  
/* terminate it. */  
ON CLOSE OF THIS-PROCEDURE  
  RUN disable_UI.
```

2. The AppBuilder generates code for the WINDOW-CLOSE event of the window, which is what fires when you click the **Close** icon in the window:

```
DO:  
/* This event will close the window and terminate the procedure. */  
APPLY "CLOSE":U TO THIS-PROCEDURE.  
RETURN NO-APPLY.  
END.
```

3. The APPLY "CLOSE" statement sets off the CLOSE event for the procedure handle itself. This runs `disable_UI`, which not only deletes the window with the `DELETE WIDGET` statement but also has the statement to delete the procedure, which might have looked cryptic when you first saw it, but which you should fully understand now:

```
/* Delete the WINDOW we created */  
IF SESSION:DISPLAY-TYPE = "GUI":U AND VALID-HANDLE(C-Win)  
THEN DELETE WIDGET C-Win.  
IF THIS-PROCEDURE:PERSISTENT THEN DELETE PROCEDURE THIS-PROCEDURE.  
END PROCEDURE.
```

4. The `PERSISTENT` attribute on the procedure handle identifies it as a persistent procedure, and the `DELETE PROCEDURE` statement deletes it by identifying its handle, `THIS-PROCEDURE`.

If the AppBuilder didn't do this for you, the procedure and all its memory and other resources would sit around until your session ended. This might create quite a mess, not just because of the memory it uses, but because of the possibility of records it might be holding and any other resources that could cause errors or contention in your application. The worst part is that the window itself would be gone, so you would have no visual clue that the procedure is still there.

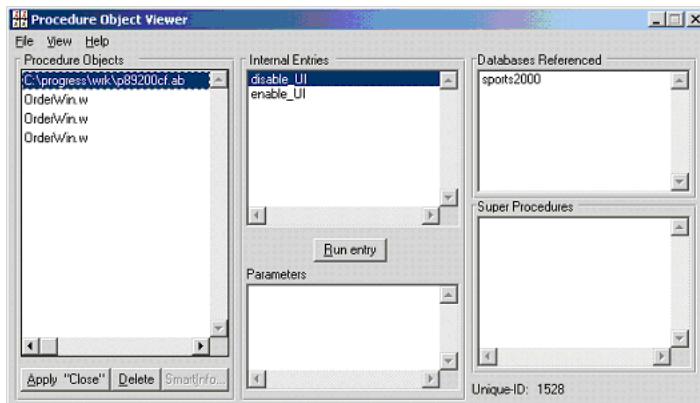


To test your application so you can see what procedures are in memory:

1. Run h-CustOrderWin6.w.
2. Select several **Orders** to display in their window.
3. Select the **Procedures** icon from the **PRO*Tools** palette:



The **Procedure Object Viewer** appears:



This tool shows that you have the main window and three copies of h-OrderWin.w running. Remember that whenever you run a procedure from the AppBuilder, it saves it to a temporary file and runs the temporary file, so you see a temporary filename such as p89200cf.ab in your temporary file directory instead of its actual procedure filename h-CustOrderWin6.w.

The **Procedure Object Viewer** window has a host of useful information in it, including all the INTERNAL-ENTRIES for any procedure you select from the list. You can also run any internal entry by choosing the **Run entry** button.

If you want to get rid of a running procedure from here, you can choose the **Delete** button. If it's an AppBuilder-generated procedure such as those you've been looking at, you can also choose the **Apply “Close”** button, which should execute all the proper cleanup code for the procedure that's associated with the CLOSE event.

4. Close the **Customers and Orders** window and select **View→Refresh List of Procedure Objects** in the **Procedure Object Viewer**.

You should see that all the running procedures, including both the main window and any **Order** windows you have open, are gone. So, are you done cleaning up?

Not yet! The AppBuilder is doing you another favor here, and it is one that is rather dangerous, because if you forget to test your application outside the AppBuilder, you might not see that you still have work to do. When you choose **Stop**, the AppBuilder deletes any persistent procedures that were started since you first chose the **Run** button, to make it easier for you to test parts of applications over and over. But when you run your application from outside the OpenEdge tools, you need to take care of your own cleanup. If you were to run h-CustOrderWin6.w from another procedure, open several **Order** windows, and then close h-CustOrderWin6, the **Order** windows (and their procedures, of course) would still be running.



To add code to delete those procedures when the main window that starts them up is deleted:

1. Go into the **Definitions** section for h-CustOrderWin6.w and add these definitions to the end of the section:

```
DEFINE VARIABLE cOrderHandles AS CHARACTER NO-UNDO.  
DEFINE VARIABLE iHandle      AS INTEGER    NO-UNDO.  
DEFINE VARIABLE hOrderWindow AS HANDLE     NO-UNDO.
```

Your new code uses these variables to save off a list of all the procedure handles of the procedures you create, and then later walks through them and deletes them.

2. Add another line to the CHOOSE trigger for the **BtnDetail** button to save off the new procedure handle in a list:

```

DO:
  DEFINE VARIABLE hOrder AS HANDLE      NO-UNDO.
  RUN h-OrderWin.w PERSISTENT SET hOrder (BUFFER Order).
  cOrderHandles = cOrderHandles +
    (IF cOrderHandles NE "" THEN "," ELSE "") + STRING(hOrder).

END.

```

To save a list of handles, you need to turn each handle into a STRING. The commas act as a delimiter between entries in the list. The IF-THEN-ELSE clause inside parentheses adds a comma only if there's already something in the list.

3. Go into the main block and add some code just before it runs `disable_UI` to clean up those procedure handles:

```

ON CLOSE OF THIS-PROCEDURE
DO:
  /* Cleanup any OrderLine windows that might still be open. */
  DO iHandle = 1 TO NUM-ENTRIES(cOrderHandles):
    hOrderWindow = WIDGET-HANDLE(ENTRY(iHandle, cOrderHandles)).
    IF VALID-HANDLE (hOrderWindow) THEN
      APPLY "CLOSE" TO hOrderWindow.
  END.
  RUN disable_UI.
END.

```

This DO block turns each handle value back into the right data type using the `WIDGET-HANDLE` function. (The function you use is the same one for all handles, even though this is a procedure handle, not a widget handle, for a visual object like a button.)

You could execute the `DELETE PROCEDURE hOrderWindow NO-ERROR` statement. The `NO-ERROR` option would let you use the `DELETE PROCEDURE` statement without bothering to check whether the handle is still valid. Remember that the user might or might not have closed it on his own.

But because the `h-OrderWin.w` procedure has special code to process the CLOSE event and do additional types of cleanup, it's always better to `APPLY "CLOSE"` to an AppBuilder-generated procedure window or any other procedure that uses the same convention. The `VALID-HANDLE` check makes sure that the window has not already been closed and the procedure deleted.

Examples: Communicating between persistent procedures

In this section, you extend the test windows to see a couple of different ways persistent procedures can communicate with one another.



- To add a fill-in field to the main window to locate and manipulate one of the Order windows:**

(You'll just add code to hide or view it alternately, just to show that you can access it.)

1. Open `h-OrderWin.w` and add this statement to the main block to save off the **Order Number** in the procedure's PRIVATE-DATA attribute:

```
ASSIGN THIS-PROCEDURE:PRIVATE-DATA = STRING(OrderNum)
      shipDate:BGCOLOR = dateColor(PromiseDate, ShipDate).
```

2. Open `h-CustOrderWin6.w` and add a fill-in to the window. Call it **iOrderWin** and give it the label **Show/Hide Order**.
3. Define this LEAVE trigger for the new fill-in:

```
DO:
  DEFINE VARIABLE hProc AS HANDLE      NO-UNDO.
  DEFINE VARIABLE hWin   AS HANDLE      NO-UNDO.
  hProc = SESSION:FIRST-PROCEDURE.
  DO WHILE VALID-HANDLE(hProc):
    IF hProc:FILE-NAME = "OrderWin.w" AND
       hProc:PRIVATE-DATA = iOrderWin:SCREEN-VALUE THEN
      DO:
        hWin = hProc:CURRENT-WINDOW.
        hWin:HIDDEN = IF hWin:HIDDEN THEN NO ELSE YES.
        LEAVE.
      END.
      hProc = hProc:NEXT-SIBLING.
    END.
  END.
```

This trigger block walks through the persistent procedure list managed through the SESSION handle. When it comes to an instance of h-OrderWin.w whose PRIVATE-DATA attribute matches the value in the fill-in, it saves off its CURRENT-WINDOW attribute, which is the handle of the procedure's window. If the window is currently hidden it is viewed, and vice versa.

4. Save the changes and test them out:
 - a. Run h-CustOrderWin6.w.
 - b. Choose the **Order Detail** button for a few different **Orders**.
 - c. Enter one of the **Order** numbers into the **Save/Hide Order** fill-in and tab out of it. The window displaying that **Order** should disappear, or reappear if it's already hidden.

This is a simple example of how to use the SESSION procedure list to locate a procedure you're interested in, and then either manipulate that procedure object or make a request of it in some way.

The next example goes in the opposite direction. Rather than locating one of the **Order** windows from the main window, you add code to the **Order** window procedure to request the handle of another procedure from the main window. It then uses that handle to reposition the other **Order** window. A new internal procedure in the main window provides an interface the **Order** window can use to get the handle it needs.

Using SOURCE-PROCEDURE to identify your caller

First, you'll learn another useful built-in function. You've seen the THIS-PROCEDURE handle function, which holds the handle of the current procedure. Another similar built-in handle function is SOURCE-PROCEDURE. Whenever Progress executes a RUN statement, the SOURCE-PROCEDURE handle in the called procedure returns the procedure handle of the caller.



To use SOURCE-PROCEDURE to identify your caller:

1. To capture the SOURCE-PROCEDURE when the **Order** window first starts up, add this definition to the **Definitions** section of h-OrderWin.w:

```
/* Local Variable Definitions --- */  
DEFINE VARIABLE hSource AS HANDLE      NO-UNDO.
```

2. Add another line to its main block, to save off the value on start-up:

```
ASSIGN THIS-PROCEDURE:PRIVATE-DATA = STRING(OrderNum)  
      hSource = SOURCE-PROCEDURE  
      shipDate:BGCOLOR = dateColor(PromiseDate, ShipDate).
```

3. Add a new fill-in to the window called **iOtherOrder**, with a **Label** of **Other Order**.
4. Add a new button called **BtnAlign**, with a **Label** of **Align Order**.

You can put a rectangle around them to group them if you want, like this:



5. Code this CHOOSE trigger for **BtnAlign**:

```

DO:
  DEFINE VARIABLE hOtherWin AS HANDLE      NO-UNDO.
  DEFINE VARIABLE hWnd AS HANDLE      NO-UNDO.
  RUN fetchOrderWin IN hSource
    (INPUT iOtherOrder:SCREEN-VALUE,
     OUTPUT hOtherWin).
  IF VALID-HANDLE(hOtherWin) THEN
    DO:
      ASSIGN hWnd = hOtherWin:CURRENT-WINDOW
      hWnd:ROW = THIS-PROCEDURE:CURRENT-WINDOW:ROW + 1
      hWnd:COLUMN = THIS-PROCEDURE:CURRENT-WINDOW:COLUMN + 3.
    END.
  END.

```

This code runs an internal procedure in its source (the main window), which takes an **Order** number as input and returns as output the handle of the window with that **Order** number in it. The code then uses the CURRENT-WINDOW attribute of that other window and the CURRENT-WINDOW attribute of THIS-PROCEDURE to align the other window just below and to the right of this one.

6. In **h-CustOrderWin6.w**, add another variable to the **Definitions** section:

```
DEFINE VARIABLE cOrderNumbers AS CHARACTER      NO-UNDO.
```

This variable will hold a list of all the **Order** numbers the procedure opens up windows for.

7. Add a statement to the **BtnOrder (Order Detail)** button to save off the **Order** number of each window as it's created, in addition to its procedure handle:

```

DO:
  DEFINE VARIABLE hOrder AS HANDLE      NO-UNDO.
  RUN h-OrderWin.w PERSISTENT SET hOrder (BUFFER Order).
  ASSIGN cOrderHandles = cOrderHandles +
    (IF cOrderHandles NE "" THEN "," ELSE "") + STRING(hOrder)
    cOrderNumbers = cOrderNumbers +
    (IF cOrderNumbers NE "" THEN "," ELSE "") + STRING(Order.OrderNum).

  END.

```

8. Define the new fetchOrderWin internal procedure in h-CustOrderWin6.w:

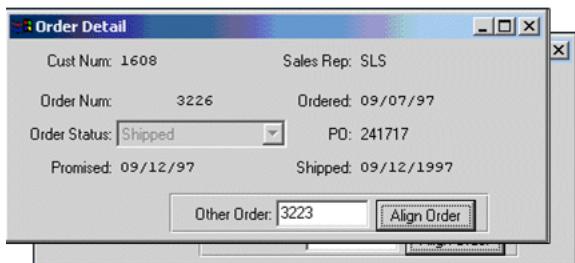
```
/*
-----*
Purpose: Returns the procedure handle to the instance of h-OrderWin.w
that displays the requested Order Number,
Notes:
-----*/
DEFINE INPUT  PARAMETER pcOrderNum AS CHARACTER  NO-UNDO.
DEFINE OUTPUT PARAMETER phOrderWin AS HANDLE      NO-UNDO.

DO iHandle = 1 TO NUM-ENTRIES(cOrderNumbers):
  IF ENTRY(iHandle, cOrderNumbers) = pcOrderNum THEN
    DO:
      phOrderWin = WIDGET-HANDLE(ENTRY(iHandle,cOrderHandles)).
      RETURN.
    END.
  END.
END PROCEDURE.
```

This code acts as an API call that other procedures like the instances of h-OrderWin can run to obtain data from the main window. It looks through the list of **Order** numbers and, when it finds the one that was requested, it passes back as output the matching procedure handle for that **Order**.

**To test out this latest change:**

1. **Run h-CustOrderWin6.w.**
2. Open a couple of **Order Detail** windows.
3. In one of them, enter the **Order** number displayed in the other, and choose the **Align Order** button (or tab out of the fill-in and press **ENTER** to do the same thing.) The windows should align like this:



Why did you use a button with a CHOOSE trigger instead of just defining a LEAVE trigger for the fill-in? In this case, without the button the fill-in would be the only enabled object in the window, so its LEAVE trigger would not fire. You need at least two enabled objects in a window in order to leave one of them. So the button gives you something to tab into and choose.

This latest example shows you in simple terms how you can provide an API in a procedure that other persistent procedures can use to get information from, or to invoke actions in, and a couple of different ways to obtain the handles of other running procedures you're interested in.

Now that you have learned the difference between non-persistent and persistent procedures, and how the stack of procedure calls in a typical older Progress application is turned into something more flexible with persistent procedures, it's time to learn about a programming concept that was very important in those older applications, but which you won't use often in new applications.

Shared and global objects in Progress procedures

In every DEFINE statement so far in this book, there is a part of the valid syntax that is deliberately left out. Now is the appropriate time to explain why this is so and why this syntax is not something you should use frequently in new applications.

The concept is *shared objects*, which allow multiple procedures to share a definition and the object it defines without passing the object as a parameter. Using a variable definition as an example, this is the basic syntax for shared objects:

```
DEFINE [ [ NEW ] SHARED ] VARIABLE cString AS CHARACTER NO-UNDO.
```

The first procedure in the call stack to reference a shared object defines the shared object as NEW SHARED. This means that Progress registers the definition and does not expect to find it already in the stack. Any subprocedures that want to share the object define it as SHARED. This means that Progress searches up the call stack for a matching definition and points the shared definition in the subprocedure at the same memory location as the NEW SHARED object. In this way, both procedures can see and modify the value of the object, rather the same as if it were an INPUT-OUTPUT parameter between the two procedures.

The two definitions must match exactly in every way that affects storage and treatment of the object. For example, one definition of a shared object with the NO-UNDO keyword does not match another definition of the same object without it, because the NO-UNDO keyword affects how Progress deals with changes to the object's value at run time.

To make sure that the definitions of shared objects in different procedures always match, and to make it easy to make changes to them when necessary, you can define an include file for the definition (or for a set of definitions to a whole list of shared objects). The NEW keyword, which is the one difference between the first definition of the object and all subsequent definitions further down the call stack, acts as an include file parameter. The first procedure to define the object passes NEW as an include file parameter, and subsequent procedures pass nothing. For example, this include file defines a shared variable:

```
/* inclvar.i */  
DEFINE {1} SHARED VARIABLE cMyVar AS CHARACTER NO-UNDO.
```

This main procedure includes the definition and makes it NEW SHARED:

```
{ inclvar.i NEW }.
```

And further definitions just reference the include file to define it as SHARED:

```
{ inclvar.i }
```

To see how this works, take another look at the simple example of `MainProc.p` and its subprocedure, `SubProc.p`, which illustrate the procedure call stack in a world of non-persistent procedures. Instead of passing parameters, this time the code shares a variable definition and a buffer definition, as shown in [Figure 13–9](#).

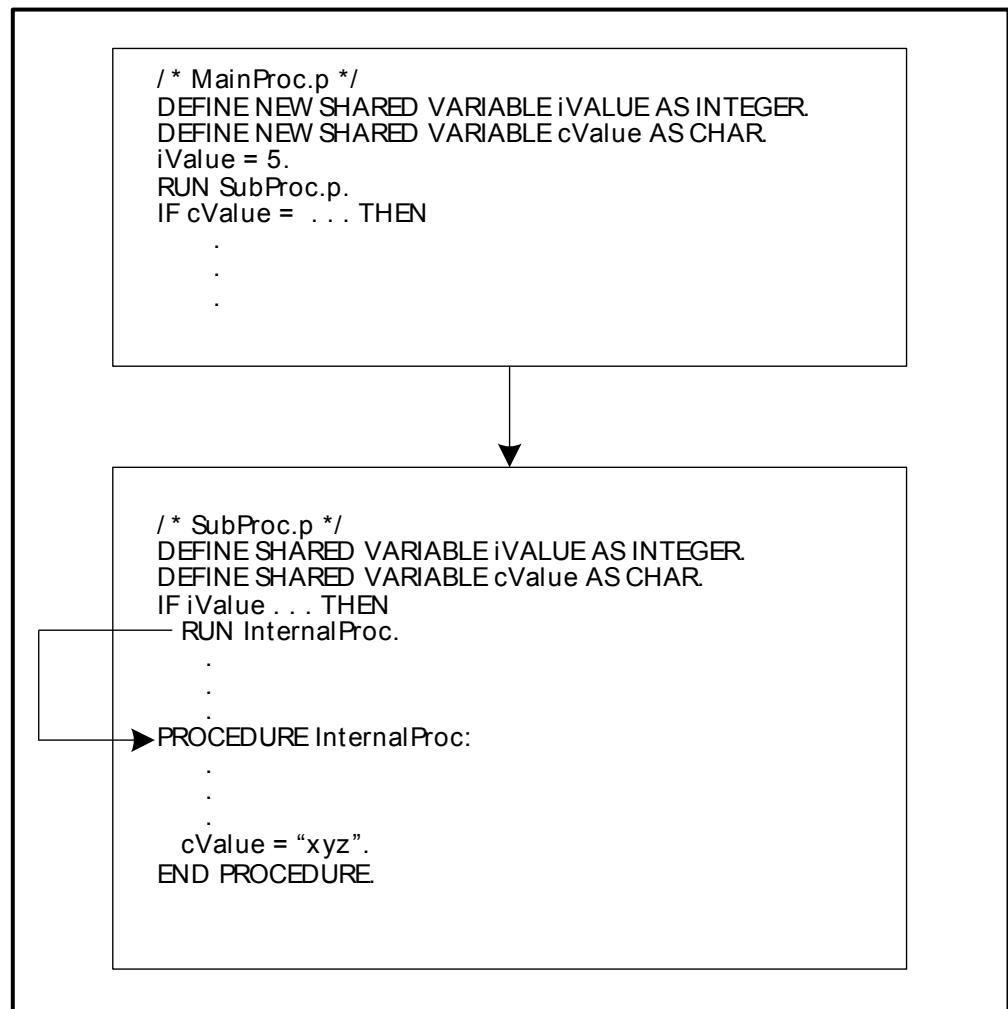


Figure 13–9: Mainproc.p and Subproc.p

The `NEW SHARED` variables defined in `MainProc.p` are available to `SubProc.p` because the latter defines them as `SHARED` variables. You don't need to redefine them in the `InternalProc` internal procedure because you defined them at the level of the entire `SubProc.p` procedure file, and therefore scoped to the whole procedure.

In general, a NEW SHARED object is available to be used, with a matching SHARED object definition, anywhere below the NEW SHARED definition in the call stack.

There are several reasons why shared objects are more useful than parameters. One such reason is that the procedures do not need to identify exactly which objects they will share, unlike a parameter list, which must spell out in full and must be the same in both the calling and the called procedures. For example, `MainProc.p` could define any number of objects as NEW SHARED. `SubProc.p` would not have to define all of them. It is free to define as SHARED whichever objects it needs to use. It can ignore the rest. Some other external procedure further down the call stack from `SubProc.p` could then define any of these as SHARED and use them, regardless of whether `SubProc.p` defined them or not.

It is perhaps worth noting, if only for historical reasons, that shared objects date back to a time when the Progress 4GL supported neither parameters nor internal procedures, let alone persistent procedures. Therefore, they were the only mechanism available for sharing values between procedures.

Objects you can define as shared include variables, buffers, queries, temp-tables, browses, and frames, among others.

Why you generally shouldn't use shared objects

If shared objects are such an important part of the Progress language, why shouldn't you use them? The reason has a lot to do with the persistent procedures you've been learning about in this chapter and which form the basis for most modern application code.

It is possible to share an object between a main procedure and a persistent procedure that it runs. But this would normally be a very unwise thing to do. Once your code runs a persistent procedure, any other procedure in the application with access to its handle can make use of its internal entries. And those procedures won't be able to use the shared object because they have no relationship on the procedure call stack to the persistent procedure. So the notion of a fixed parent-child relationship between one procedure and another is no longer there. In a modern, event-driven application, it is better to think of the various running procedures as peers cooperating together rather than as a hierarchy. You can use shared objects only in a fixed hierarchy, so shared objects are no longer an appropriate way to pass data from one object procedure to another.

Figure 13–10 shows a diagram of persistent procedures with another procedure added to the mix, just as a simple illustration.

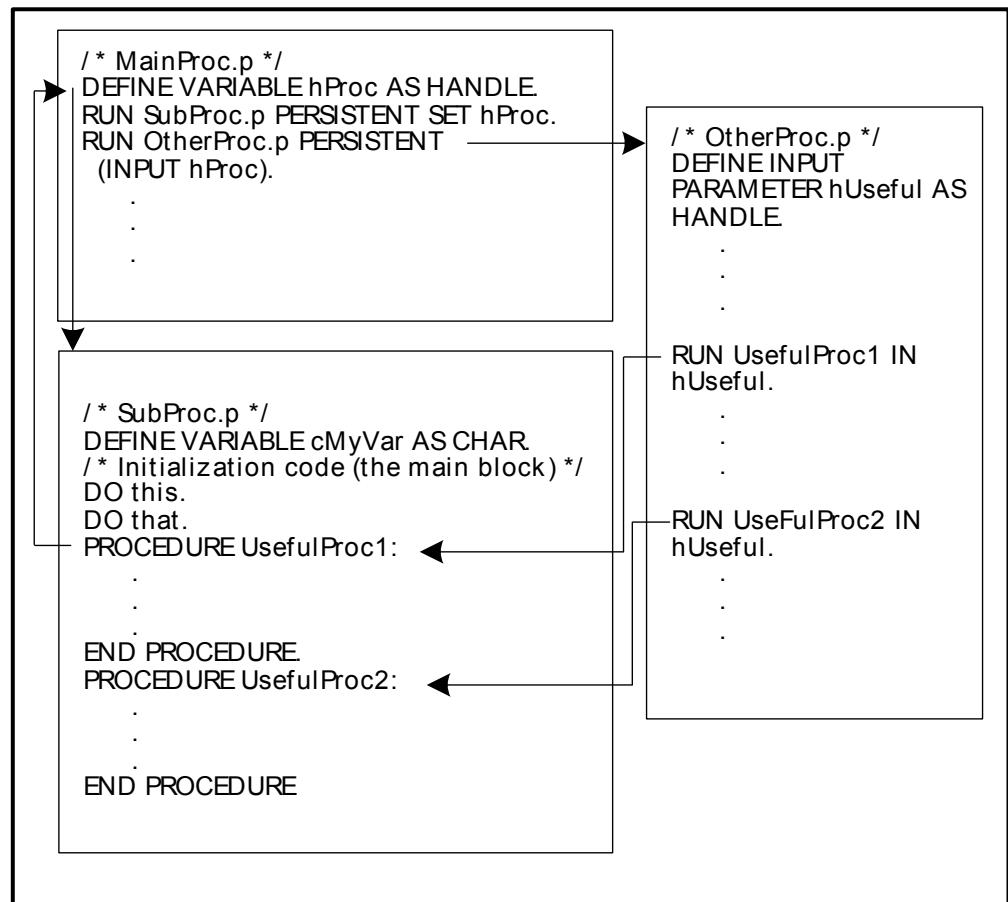


Figure 13–10: Persistent procedure example

MainProc.p runs both SubProc.p and OtherProc.p as persistent procedures. It passes SubProc.p's procedure handle to OtherProc.p as a parameter (or makes it available in some other way). Now OtherProc.p can make use of the internal entries in SubProc.p by running them in its procedure handle. But there's no other relationship between OtherProc.p and SubProc.p because they were run independently. So, there's no way they could make use of shared objects between them. This is the heart of why shared objects are now a problematic concept in event-driven applications and why you should generally avoid them.

Another reason why shared objects are of limited value in a modern application is that new applications are likely to be distributed across multiple machines and separate OpenEdge sessions. While you can pass most types of objects as parameters between sessions and between machines using the OpenEdge AppServer, there is no way to make use of the SHARED construct between sessions. This means that if you have two procedures today that use shared objects and tomorrow you decide that in your revised application architecture they should run on separate machines, you will have a lot more work to do to change the code so that it doesn't use the shared objects than you would if the data were already passed as parameters between the procedures. In general, code with many shared objects represents one of the biggest challenges for developers migrating existing older applications to a newer architecture.

Does this mean you should never use shared objects in a new application? The construct is still there and is still supported, for the sake of backward compatibility, but it is a good practice not to make use of it. Shared objects provide few advantages and several potential disadvantages to the future evolution of your code compared to other Progress constructs available to you today.

Global shared objects in Progress

There's another variation on the shared object theme, and this is *global shared objects*. The first procedure to define an object includes the GLOBAL keyword in the definition:

```
DEFINE [ [ NEW [ GLOBAL ] ] SHARED ] VARIABLE cString AS CHARACTER NO-UNDO
```

This makes the object definition available to the entire session. The same set of objects that can be defined as NEW SHARED can also be defined as NEW GLOBAL SHARED. For example, one procedure can have this global definition:

```
DEFINE NEW GLOBAL SHARED VARIABLE cEverywhere AS CHARACTER NO-UNDO.
```

Any other procedure in the Progress session that is executed *after that procedure* can have this definition and make use of the same variable:

```
DEFINE SHARED VARIABLE cEverywhere AS CHARACTER NO-UNDO.
```

One tricky aspect to this is the *after that procedure* requirement. In many applications, it is not easy to determine which of the procedures that might want to reference a value will execute first, and therefore should be the procedure that defines it. Or you might have many procedures that need to reference the value and not know which of them will execute at all in the course of a session.

Progress provides a work-around for this problem. If a procedure defines an object as NEW GLOBAL SHARED and another procedure has already registered this definition, then no error results. Progress ignores the NEW GLOBAL part of the second definition and accepts it as a reference to the existing SHARED object. This is different from nonglobal shared objects, where a second NEW SHARED reference causes a run-time error.

For this reason, it is common practice simply to use the NEW GLOBAL SHARED syntax in every procedure that needs to use the objects. Although this is somewhat contradictory to the way the nonglobal shared syntax works, it is reliable behavior.

When to consider using global shared objects

You might think that global shared objects are a better way to go in new applications than nonglobal shared objects, because multiple procedures that aren't necessarily running in a single predictable procedure call stack can all access them. Also, any of the procedures can define them first and others can define them and use them in any order. Thus, you could place a set of global definitions into an include file and include it in every procedure that needs them, without regard for which procedure has to use the NEW keyword in its definitions.

Up to a point this is correct, but there are good reasons why you should generally avoid global shared objects in new applications:

1. Simply having an include file that many procedures depend on is poor practice in a modern application. Any time the list of definitions in this file changes for any reason, every procedure that uses it must be recompiled and potentially redeployed. As you learn more about dynamic language constructs in later chapters, you learn useful alternatives to global shared variables that are more flexible and can change at run time without any code changes or recompilation at all.

2. Global shared objects create potential namespace conflicts and possible unintended consequences in your application, precisely because they *are* global. A global shared object is visible to every procedure in the entire session, and every procedure shares that name and its value. Global shared objects never go out of scope and they can't be deleted. They exist from the time the first definition is encountered until the session ends. An important aspect of a modern application is that it is as modular as possible, with code and data for a particular module isolated within a set of procedures that can run independently of other modules. What happens when two modules happen to use the same name for a global object? The result is that Progress defines just one object, with a single value, and two different sets of code are trying to use it for different purposes. Very insidious errors can result.

Thus, it's generally advisable that you should avoid global objects wherever possible. If you do think you want to use them, consider these important guidelines:

- A global shared variable or other object can be somewhat faster to access than an alternative that involves using possibly dynamic language statements to refer to a value defined elsewhere. All the procedures that use the global shared object are pointing directly to its memory just as if it was defined locally. Therefore, you can consider using global shared variables or other objects for a small number of values that are critical to your application, used with great frequency, and truly global in their use.
- Because global shared objects exist in a single namespace across the entire session, and because they can't be deleted, you should be very careful in naming them. Use a naming convention that prevents any chance of two different procedures using the same name for different purposes. Perhaps the easiest way to do this is to have a single include file for your entire application that holds all its global definitions. Any procedure that needs any of them includes this same file, so that (as long as they are recompiled together) they remain in sync and there isn't a chance of another developer inadvertently giving another global object the same name.

In the following chapter, you'll learn about a newer language construct that very definitely *is* of great use, your own 4GL functions, which you can often use as an alternative to internal procedures.

Defining Functions and Building Super Procedures

This chapter covers some other important topics concerning how you construct procedures. Specifically, it describes how to define functions of your own within your 4GL procedures to complement the built-in Progress functions you've been introduced to. It also details how to build super procedures, which provide standard libraries of application behavior. Finally, it describes the PUBLISH and SUBSCRIBE keywords that help procedures communicate with each other.

This chapter includes the following sections:

- [User-defined functions](#)
- [Defining a function](#)
- [Using super procedures in your application](#)
- [PUBLISH and SUBSCRIBE statements](#)

User-defined functions

As you have already seen in the previous chapter's discussion of INTERNAL-ENTRIES and the GET-SIGNATURE method, there is an alternative to the internal procedure as a named entry point within a procedure file. This is the *user-defined function*, sometimes referred to as a UDF or simply as a function. You are probably familiar with functions from other programming languages. Fundamentally, a function differs from a procedure in that it returns a value of a specific data type, and therefore can act in place of any other kind of variable or expression that would evaluate to the same data type. This means that you can place functions inside other expressions within a statement or within the WHERE clause of a result set definition. By contrast, you must invoke a procedure, whether external or internal, with a RUN statement. If it needs to return information, you have to use OUTPUT parameters that your code examines after the procedure returns. You should therefore consider defining a function within your application for a named entry point that needs to return a single value and that is convenient to use within larger expressions.

You can also compare user-defined functions with the many built-in functions that are a part of the Progress 4GL, such as NUM-ENTRIES, ENTRY, and many others that you've seen and used. Your functions can provide the same kinds of widely useful behavior as the built-in functions, but specific to your application.

If you use a function in a WHERE clause or other expression in the header of a FOR EACH block, Progress evaluates the function once, at the time the query with the WHERE clause is opened or the FOR EACH block is entered.

You can perhaps best think of a function as a small piece of code that performs a frequently needed calculation, or that checks a rule or does some common data transformation. In general, this is the best use for functions. However, a function can be a more complex body of code if this is appropriate.

Defining a function

This is the syntax you use to define the header for a function:

```
FUNCTION function-name [ RETURNS ] datatype [ ( parameters ) ] :
```

A function always must explicitly return a value of the data type you name. It can take one or more parameters just as a procedure can. Although this means that a function can return OUTPUT parameters or use INPUT-OUTPUT parameters just as a procedure can, you should consider that a function normally takes one or more INPUT parameters, processes their values, and returns its RETURN value as a result. If you find yourself defining a function that has OUTPUT parameters, you should reconsider and probably make it a procedure instead. One of the major features and benefits of a function is that you can embed it in a larger expression and treat its return value just as you would a variable of the same type. If there are OUTPUT parameters, this won't be the case as you must check their values in separate statements.

The body of the function can contain the same kinds of statements as an internal procedure. Just as with internal procedures, you cannot define a temp-table or any shared object within the function. It has two important additional restrictions as well:

- Within a function, you cannot define any input-blocking statements, such as a WAIT-FOR statement or any statement that prompts the user for input.
- You cannot reference a function within a Progress ASSIGN statement or other 4GL updating statement that could cause an index to be updated. The interpreter might not be able to deal with transaction-related code inside the function itself in the middle of the update statement. It is a very good practice to avoid using functions in any code statements that update the database.

A function must contain at least one RETURN statement to return a value of the appropriate data type:

```
RETURN return-value.
```

The *return-value* can be a constant, variable, field, or expression (including possibly even another function reference) that evaluates to the data type the function was defined to return.

A function must end with an END statement. Just as an internal procedure normally ends with an END PROCEDURE statement, it is normal to end a function with an explicit END FUNCTION statement to keep the organization of your procedure file clearer. The FUNCTION keyword is optional in this case but definitely good programming practice.

Although a function must always have a return type and return a value, a reference to the function is free to ignore the return value. That is, a 4GL statement can simply consist of a function reference without a return value, much like a RUN statement without the RUN keyword. This might be appropriate if the function returns a true/false value indicating success or failure, and in a particular piece of code, you are not concerned with whether it succeeded or not.

Whether a function takes any parameters or not, you must include parentheses, even if they're empty, on every function reference in your executable code.

Making a forward declaration for a function

In [Chapter 4, “Introducing the OpenEdge AppBuilder,”](#) you looked at the overall organization of a procedure file and used the AppBuilder-generated form as a good basis for your own procedure files. In fact, you should always use the AppBuilder to create your procedures if they have any internal entries at all, even if they have no visual content. The AppBuilder organizes the physical file for you and generates a lot of the supporting code you need.

If you follow the structural guidelines for a procedure file, you place all your internal entries—internal procedures and user-defined functions—at the end of the file. Normally, you need to reference the functions in your procedure file within the body of the code that comes before their actual implementation, either in the main block or in other internal entries in the file.

The Progress compiler needs to understand how to treat your function when it encounters it in the executable code. In particular, it needs to know the data type to return. It also does you a service by enforcing the rest of the function's signature—its parameter list—whenever it finds a reference to it. For this reason, Progress needs to know at least the definition of the function—its return type and parameters—before it encounters a use of the function elsewhere in the procedure. To do this and still leave the actual implementation of the function toward the end of the file, you can provide a *forward declaration* of the function, also called a *prototype*, toward the top of the procedure file, before any code that uses the function.

This is the syntax for a forward declaration of a function:

```
FUNCTION function-name [ RETURNS ] datatype [ ( parameters ) ]
{ FORWARD | [ MAP [ TO ] actual-name ] IN proc-handle | IN SUPER }
```

As you can see, the basic definition of the function name, return type, and parameters is the same as you would use in the function header itself. If you provide a forward declaration for a function, the parameter list is optional in the function header (though the RETURNS phrase is not). It's good practice, though, to provide it in both places.

A function prototype can point to an actual function implementation in one of three kinds of places:

- In the beginning of the procedure.
- In another procedure.
- In a super procedure.

Making a local forward declaration

If you use the FORWARD keyword in the function prototype, then this tells Progress to expect the actual function implementation later in the same procedure file. Here's a simple example of a function that converts Celsius temperatures to Fahrenheit:

```
/* h-ConvTemp1.p -- procedure to convert temperatures
   and demonstrate functions. */
FUNCTION CtoF RETURNS DECIMAL (INPUT dCelsius AS DECIMAL) FORWARD.
DEFINE VARIABLE dTemp AS DECIMAL      NO-UNDO.
REPEAT dTemp = 0 TO 100:
  DISPLAY dTemp LABEL "Celsius"
    CtoF(dTemp) LABEL "Fahrenheit"
    WITH FRAME f 10 DOWN.
END.

FUNCTION CtoF RETURNS DECIMAL (INPUT dCelsius AS DECIMAL):
  RETURN (dCelsius * 1.8) + 32.
END FUNCTION.
```

This procedure executes as follows:

1. The procedure makes a forward declaration of the CtoF conversion function, so that it can be used in the procedure before its implementation code is defined.
2. The function is used inside the REPEAT loop in the DISPLAY statement. Notice that it appears where any DECIMAL expression could appear and is treated the same way.
3. There is the actual implementation of the function, which takes the Celsius temperature as input and returns the Fahrenheit equivalent.

Figure 14–1 shows the first page of output from the h-ConvTemp1.p procedure.

The screenshot shows a Progress 4GL application window titled 'P'. Inside, there is a table with two columns: 'Celsius' and 'Fahrenheit'. The data is as follows:

| Celsius | Fahrenheit |
|---------|------------|
| 0.00 | 32.00 |
| 1.00 | 33.80 |
| 2.00 | 35.60 |
| 3.00 | 37.40 |
| 4.00 | 39.20 |
| 5.00 | 41.00 |
| 6.00 | 42.80 |
| 7.00 | 44.60 |
| 8.00 | 46.40 |
| 9.00 | 48.20 |

At the bottom of the window, a message says 'Press space bar to continue.'

Figure 14–1: Result of the ConvTemp.p procedure

You could leave the parameter list out of the function implementation itself, but it's good form to leave it in.

Making a declaration of a function in another procedure

Because functions are so generally useful, you might want to execute a function from many procedures when it is in fact implemented in a single procedure file that your application runs persistent. In this case, you can provide a prototype that specifies the handle variable where the procedure handle of the other procedure is to be found at run time. This is the second option in the prototype syntax:

```
[ MAP [ TO ] actual-name ] IN proc-handle
```

The *proc-handle* is the name of the handle variable that holds the procedure handle where the function is actually implemented. If the function has a different name in that procedure than in the local procedure, you can provide the MAP TO *actual-name* phrase to describe this. In that case, *actual-name* is the function name in the procedure whose handle is *proc-handle*.



To see an example of the CtoF function separated out in this way:

1. Create a procedure with just the function definition in it:

```
/* h-FuncProc.p -- contains CtoF and possible other useful functions. */
FUNCTION CtoF RETURNS DECIMAL (INPUT dCelsius AS DECIMAL):
    RETURN (dCelsius * 1.8) + 32.
END FUNCTION.
```

It could also have other internal entries to be used by others that have its procedure handle at run time.

2. Change the procedure that uses the function to declare it IN *hFuncProc*, its procedure handle. The code has to run *h-FuncProc.p* persistent or else access its handle if it's already running. In this case, it also deletes it when it's done:

```
/* h-ConvTemp2.p -- procedure to convert temperatures
   and demonstrate functions. */
DEFINE VARIABLE hFuncProc AS HANDLE      NO-UNDO.

FUNCTION CtoF RETURNS DECIMAL (INPUT dCelsius AS DECIMAL) IN hFuncProc.

DEFINE VARIABLE dTemp AS DECIMAL      NO-UNDO.

RUN h-FuncProc.p PERSISTENT SET hFuncProc.
REPEAT dTemp = 0 TO 100:
    DISPLAY dTemp LABEL "Celsius"
        CtoF(dTemp) LABEL "Fahrenheit"
        WITH FRAME f 10 DOWN.
END.

DELETE PROCEDURE hFuncProc.
```

3. Run this variant h-ConvTemp2.p, to get the same result as before. This time here's the end of the display, just to confirm that you got the arithmetic right:



An externally defined function such as this one can also reside in an entirely different OpenEdge session, connected to the procedure that uses the function using the OpenEdge AppServer. In this case, you declare the function in the same way but use the AppServer-specific ON SERVER phrase on the RUN statement to invoke the function. As with any AppServer call, you can't pass a buffer as a parameter to such a function. See *OpenEdge Application Server: Developing AppServer Applications* for more information.

Making a declaration of a function IN SUPER

The third option in the function prototype is to declare that it is found IN SUPER at run time. This means that the function is implemented in a *super procedure* of the procedure with the declaration. You learn about super procedures in the “[Using super procedures in your application](#)” section on page 14–14.

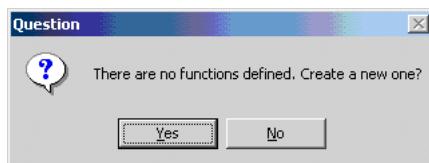
Using the AppBuilder to generate function definitions

If you want to build a procedure file of any size with a number of internal entries, whether internal procedures or functions, you should definitely use the AppBuilder to create it for you. The AppBuilder generates most of the supporting statements you've just read about here for user-defined functions, and provides a separate code section for each to make it easy to maintain your procedures.

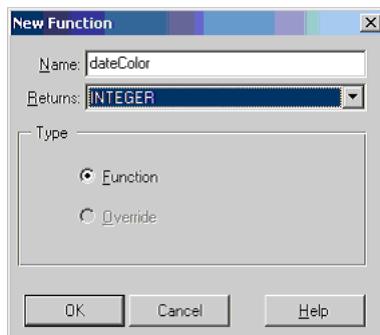


To see how to define a function in the AppBuilder and what it does for you:

1. Go into the **Section Editor** for the new h-OrderWin.w procedure.
2. Select **Functions** from the **Section** drop-down list. This message appears:



3. Answer **Yes**. The **New Function** dialog box appears:



4. Enter the function name **dataColor** and specify **INTEGER** for the **Returns** value, then choose **OK**.

5. Enter this definition for the function:

```

RETURNS INTEGER
( daFirst AS DATE, daSecond AS DATE ) :
/*
Purpose: Provides a standard warning color if one date
is too far from another.
Notes: The function returns a color code of:
-- yellow if the dates differ at all
-- purple if they are more than five days apart
-- red if they are more than ten days apart
*/
DEFINE VARIABLE iDifference AS INTEGER      NO-UNDO.
iDifference = daSecond - daFirst.
RETURN IF iDifference = ? OR iDifference > 10 THEN 12 /* Red */
ELSE IF iDifference > 5 THEN 13 /* Purple */
ELSE IF iDifference > 0 THEN 14 /* Yellow */
ELSE ?. /* Unknown value keeps the default background color. */

END FUNCTION.
```

The AppBuilder has generated the `FUNCTION dataColor` header syntax along with the `RETURNS` phrase, a placeholder `RETURN` statement and the `END FUNCTION` statement. You fill in the parameter definitions where the comment prompts you to do that. The function accepts two dates as `INPUT` parameters, calculates the difference between them (which is an integer value), and returns an integer value representing the appropriate background color for the second date. This is color code 12, 13, or 14, representing colors red, purple, and yellow, depending on how far apart the two dates are and whether either of them is undefined.

6. Go into the main block and add a line of code to invoke the function and assign the `BGCOLOR` attribute for background color to the date field when the row is displayed:

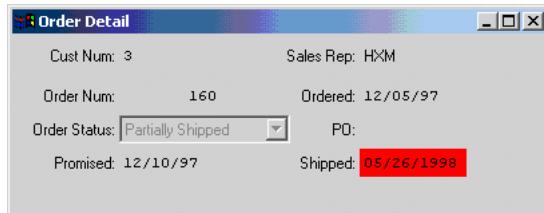
```

MAIN-BLOCK:
DO ON ERROR UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
  ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:
    RUN enable_UI.
    shipDate:BGCOLOR = dateColor(PromiseDate, ShipDate).
    IF NOT THIS-PROCEDURE:PERSISTENT THEN
      WAIT-FOR CLOSE OF THIS-PROCEDURE.
  END.
```

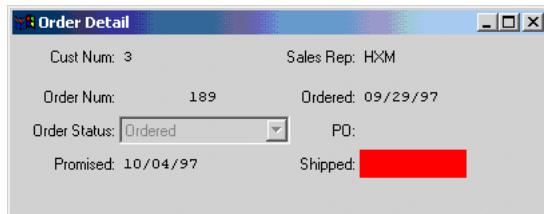
7. Save `h-OrderWin.w`, then Run `h-CustOrderWin6.w`.

There are only a few **Orders** where the **ShipDate** is later than the **PromiseDate**.

8. To find a few, enter a **New State** of **GA** for Georgia.
9. Choose the **Next** button to get to **Customer 3**.
10. Select **Order** number **160** in the browse and choose the **Order Detail** button to run h-OrderWin.w. You see the field is displayed in red, as it should be:



11. Select **Order** number **189**, which has no **ShipDate** at all, and choose **Order Detail**:



The **ShipDate** shows up in red here also because it has the unknown value.

Think about the code you just wrote. You defined an implementation for a function called `dateColor`, which goes to the bottom of the `h-OrderWin.w` procedure file, along with any internal procedures. Then you added a line of code to the main block that references the function.

The main block is above the function in the procedure, so why didn't Progress complain that there was no declaration for the function? The reason is that the AppBuilder generated it for you.

12. Go into the **Compile**→**Code Preview** window for `h-OrderWin.w` and search for **dateColor** (use the **CTRL-F** sequence for Find).

You'll see the function prototype in one of the header sections that comes before the main block or any other executable code:

```
/* ***** Function Prototypes ***** */
FUNCTION dateColor RETURNS INTEGER
( daFirst AS DATE, daSecond AS DATE ) FORWARD.
```

Thus, the AppBuilder not only helps you generate the header for the function code itself, but it also generates a prototype for you and organizes all the code into sections where you can easily inspect and maintain individual entries.

The AppBuilder copies the parameter definitions from your function header just as you entered it in the **Section Editor** so you cannot remove them from the function header, even though in a hand-coded procedure file you could define them in the prototype and leave them out of the function header itself.

Making run-time references with DYNAMIC-FUNCTION

Progress lets you construct a reference to a function at run time, using a built-in function called DYNAMIC-FUNCTION. Here's the syntax:

```
DYNAMIC-FUNCTION ( function [ IN handle ] [ , parameters ] )
```

Like a function reference itself, the DYNAMIC-FUNCTION function can appear anywhere in your procedure code where an expression of that data type could appear.

The first parameter to DYNAMIC-FUNCTION is the name of the function to invoke. This procedure can be a quoted literal or an expression, such as a variable that evaluates to a valid function name.

Following this, you can optionally include an IN *handle* phrase to direct Progress to execute the function in a persistent procedure handle. In this case, the *handle* must be an actual field or variable name of HANDLE data type, not an expression.

If the function itself takes any parameters, you pass those as additional parameters to DYNAMIC-FUNCTION, in the same form that you would pass them to the function itself.

DYNAMIC-FUNCTION gives you the flexibility to have code that can execute different function names depending on the situation, since the *function* parameter can be a variable. You might have, for example, several different functions that do parallel work for different categories of data in a loop you're iterating through. However, that is of perhaps limited value because all the functions you invoke must have the same signature.

The most common use of DYNAMIC-FUNCTION is for one of two other reasons:

- If you want to reference a function without going to the trouble of defining a prototype for it, you can do this with DYNAMIC-FUNCTION. Because Progress has no way of knowing what the name of the function is or its return type, it cannot look for a prototype for it and therefore does not give you an error as it would if you had a static reference to a function with no prior declaration. By the same token, it cannot provide you with any helpful warnings if your reference to the function is not valid.
- If you want to invoke a function in a number of different persistent procedures, you can easily do this with DYNAMIC-FUNCTION since the procedure handle to run it is part of the function reference, and not defined in the prototype. In this way, you can run a function in different persistent procedure instances depending on the circumstances. If each of those procedures represents an application object (such as a window, frame, browse, or query), then it can be very powerful to invoke the same function in different procedure handles representing those objects.

Using super procedures in your application

Super procedures were introduced to the Progress language in Version 9 as a way to provide standard libraries of application behavior that can be inherited by other procedures and, where necessary, overridden or specialized by individual application components. They give an object-oriented flavor to Progress programming that was not available before.

A *super procedure* is a separately compiled Progress procedure file. It's entry points can effectively be added to those of another procedure so that a RUN statement or function reference in the other procedure causes the Progress interpreter to search both procedures for an internal procedure or function to run. There is also a RUN SUPER statement that lets you implement multiple versions of a single entry point, each in its own procedure file, and have them all execute at run time to provide the application with complex behavior defined at a number of different levels.

In this section, you learn how to build super procedures and how to define application behavior in them that many other procedures can use in a consistent way.

Super procedure language syntax

On the face of it, there is nothing special about a super procedure. That is, there is nothing specific in the 4GL syntax of a Progress procedure file to identify it as a super procedure. Rather, it is how the procedure is referenced by *other* procedures that makes it a super procedure. Having said this, there are definite guidelines to follow when you build a procedure file that you want to use as a super procedure. These guidelines are discussed in the next section. First, you'll examine the syntax used for super procedures.

A Progress procedure file must be running as a persistent procedure before it can be made a super procedure of another procedure file. So the first step is that the code somewhere in the application must run the procedure PERSISTENT:

```
RUN superproc PERSISTENT SET superproc-hd1.
```

ADD-SUPER-PROCEDURE method

Any application procedure with access to the procedure handle of the super procedure can then add it as a super procedure to itself:

```
THIS-PROCEDURE:ADD-SUPER-PROCEDURE( superproc-hd1  
[ , search-directive ] ).
```

Or, in the more general case, it is possible to add a super procedure to any known procedure handle:

```
proc-hd1:ADD-SUPER-PROCEDURE( superproc-hd1 [ , search-directive ] ).
```

In addition, you can add a super procedure at the Session level, using the special SESSION handle, in which case its contents are available to every procedure running in the OpenEdge session:

```
SESSION:ADD-SUPER-PROCEDURE( superproc-hd1 [ , search-directive ] ).
```

The optional *search-directive* can be either SEARCH-SELF (the default) or SEARCH-TARGET. The significance of this option is discussed in the “[Super procedure guidelines](#)” section on page 14–18.

REMOVE-SUPER-PROCEDURE method

Once you add a super procedure, you can also remove it from its association with the other procedure, whether you reference it as THIS-PROCEDURE, SESSION, or some other handle:

```
proc-hd1:REMOVE-SUPER-PROCEDURE( superproc-hd1 ).
```

You can execute multiple ADD-SUPER-PROCEDURE statements for any given procedure handle (including SESSION). The super procedure handles form a stack, which is searched in Last In First Out (LIFO) order when Progress encounters a RUN statement or a function reference at run time. That is, the super procedure added last is searched first to locate the entry point to run. At any time, you can retrieve the list of super procedure handles associated with a procedure using the SUPER-PROCEDURES attribute of a procedure handle:

```
proc-hd1:SUPER-PROCEDURES
```

This attribute evaluates to a character string holding the list of super procedure handles (starting with the last one added, therefore indicating the order in which they are searched) as a comma-separated character string.

Changing the order of super procedures

You can rearrange the order of super procedures in two ways:

1. If the ADD-SUPER-PROCEDURE method is executed for a procedure handle already in the stack, Progress moves it to the head of the list (that is, to the position of being searched first). If the procedure was *already* first in the stack, no change occurs and no error results. For example, this sequence of statements results in the SUPER-PROCEDURES attribute returning the handle of Super2 followed by the handle of Super1:

```
hProc:ADD-SUPER-PROCEDURE(hSuper1).
hProc:ADD-SUPER-PROCEDURE(hSuper2).
hProc:ADD-SUPER-PROCEDURE(hSuper2).
```

And this sequence results in the SUPER-PROCEDURES attribute returning the handle of Super1 followed by the handle of Super2:

```
hProc:ADD-SUPER-PROCEDURE(hSuper1).
hProc:ADD-SUPER-PROCEDURE(hSuper2).
hProc:ADD-SUPER-PROCEDURE(hSuper1).
```

2. You can also rearrange the order of super procedure handles on the stack by invoking a sequence of REMOVE-SUPER-PROCEDURE and ADD-SUPER-PROCEDURE methods. Each REMOVE-SUPER-PROCEDURE method removes that handle from the list, wherever it is. And each ADD-SUPER-PROCEDURE method adds the named handle to the head of the list. A handle is never added to the list a second time.

Invoking behavior in super procedures

The Progress interpreter effectively adds the contents of the super procedures to the name space of the procedure that added it. Therefore, you can invoke the internal procedures and functions defined in a super procedure simply by referencing them as if they were actually implemented in the other procedure. The interpreter locates the routine and executes it.

In addition, because Progress compiles each procedure separately, each has its own *compile-time* name space, so you can define the same routine in one or more super procedures and also in the other procedures to which they are added. This functionality lets you build hierarchies of behavior for a single entry point name, effectively creating a set of *classes* that implement different parts of an application's standard behavior. A *local* version of an internal procedure can invoke the same routine in its (first) super procedure using this statement:

```
RUN SUPER [ ( parameters ) ].
```

If the internal procedure takes any parameters (INPUT, OUTPUT, or INPUT-OUTPUT) it must pass the same parameter types to its super procedure in the RUN SUPER statement. Note that these parameter *values* do not have to be the same. You might want to change the parameter values before invoking the behavior in the next version of the internal procedure, depending on your application logic.

Likewise, a user-defined function can invoke behavior in a super procedure using the expression:

```
SUPER ( [ parameters ] ).
```

This invokes the same function name in the super procedure and passes any parameters to it just as an internal procedure RUN SUPER statement does. The SUPER() expression returns from the super procedure whatever value and data type the function itself returns.

Each super procedure in turn can invoke the next implementation of that same routine up the stack by using the same SUPER syntax.

You must place a RUN SUPER statement inside an implementation of the invoked internal procedure and you must use exactly the same calling sequence. You can place any other 4GL definitions or executable code before or after the SUPER reference. This placement lets you invoke the inherited behavior before, after, or somewhere in the middle of the local specialization of the routine.

Super procedure guidelines

There is nothing specific that identifies a Progress procedure file as a super procedure. However, some guidelines can help in using super procedures effectively and without confusion.

The best general statement is that it is a good practice to consider some procedure files to be application *objects*, which implement specific application behavior, and others to be service procedures that operate in the background to provide standard behavior for a whole class of application objects. These latter procedures should be your super procedures. Said another way, a super procedure should be a library of standard behavior that many individual application objects can use. What does this mean in practice?

Guideline 1: Use a single super procedure stack

Super procedures should generally not have a super procedure stack of their own. The stack of super procedures (if there is more than one) for an application object should be defined by the object procedure itself. This is done by always using the SEARCH-TARGET keyword in the ADD-SUPER-PROCEDURE statements. This gives you maximum flexibility to define and modify the stack as needed without the individual super procedures having to be aware of each other.

An example can help clarify how this works. In this example, you have an application object procedure and two super procedures from which it inherits standard behavior, as illustrated in Figure 14–2.

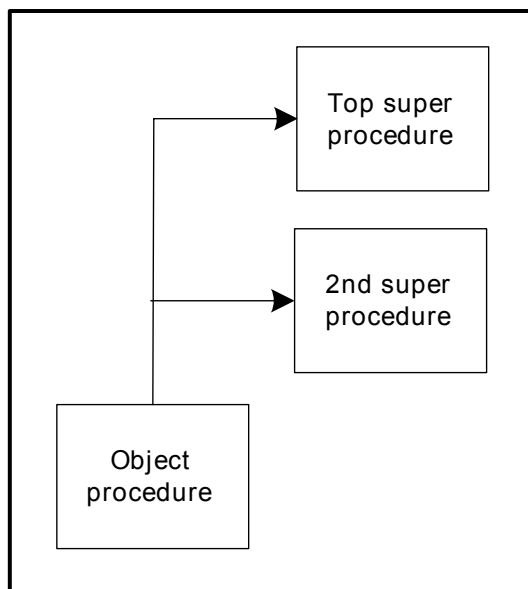


Figure 14–2: An object procedure and two super procedures

This is accomplished by running all the procedures persistent and then executing these statements from the object procedure:

```
THIS-PROCEDURE:ADD-SUPER-PROCEDURE(hTop, SEARCH-TARGET).  
THIS-PROCEDURE:ADD-SUPER-PROCEDURE(h2nd, SEARCH-TARGET).
```

The object procedure executes the following statement:

```
RUN startMeUp.
```

The Progress interpreter searches for an internal procedure named `startMeUp` first in the object procedure itself, then in the code of the last super procedure added (`h2nd`), and finally in the code of the first super procedure added (`hTop`). There is an implementation of this internal procedure in all three places and they are all intended to run in sequence. The code in the top super procedure is the most general and is executed last. The code in the 2nd super procedure is more specific and is executed second. And the code in the object procedure itself is specific to that particular object and is executed first. [Figure 14–3](#) shows the sequence of control that takes place at run time.

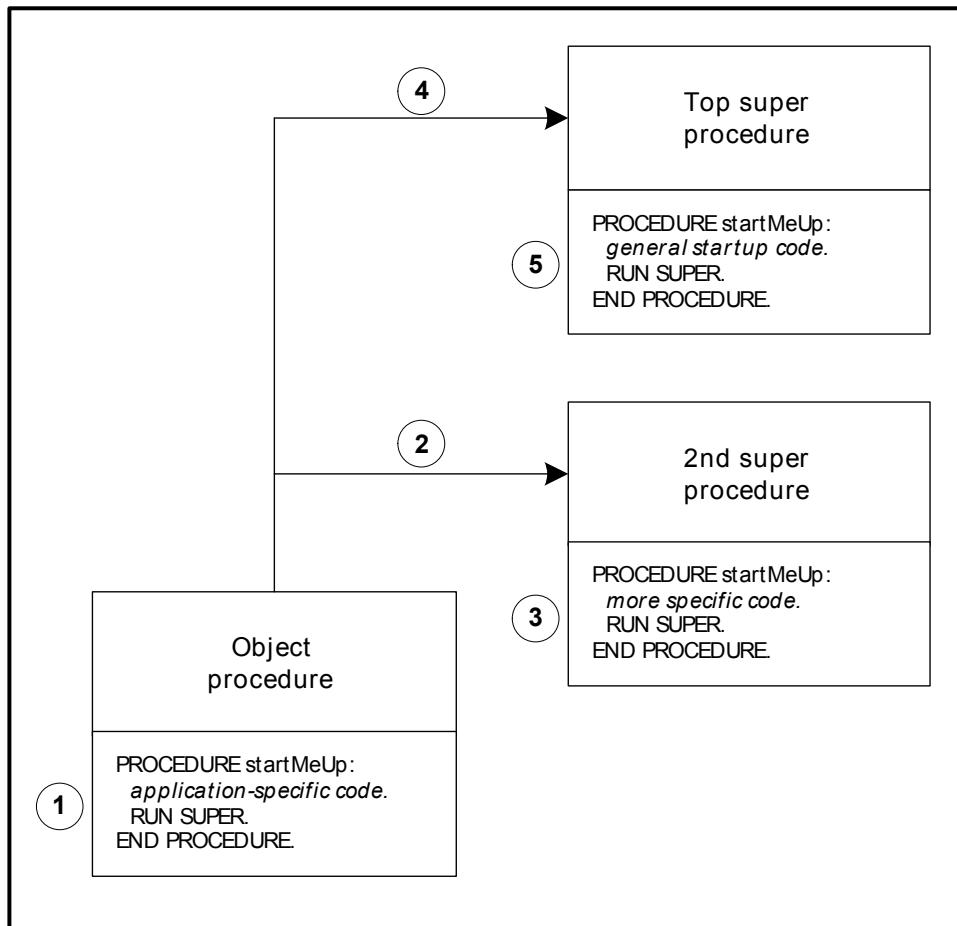


Figure 14–3: Super procedure stack execution

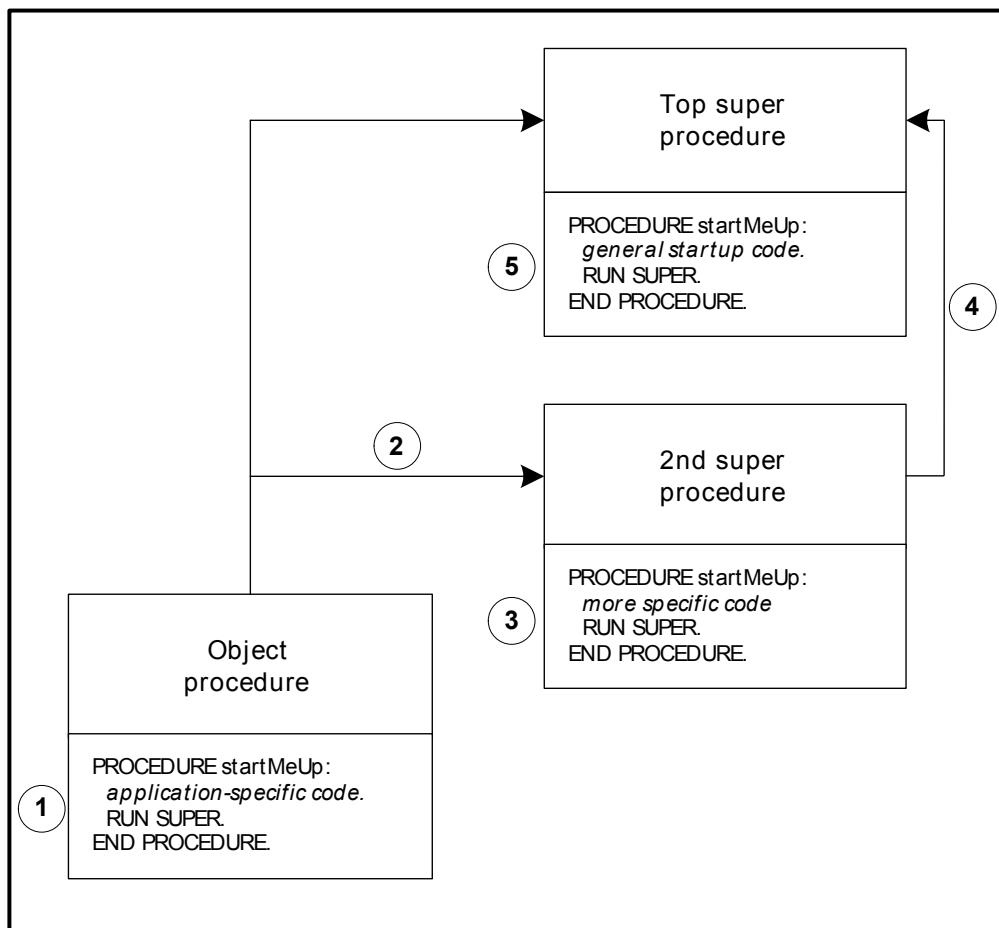
The code executes as follows:

1. The interpreter locates the implementation of `startMeUp` in the object procedure instance and runs it. The `startMeUp` procedure executes its application-specific code. The `RUN SUPER` statement causes the interpreter to search up the super procedure stack for another version of the same internal procedure name.
2. The interpreter searches the instance of the 2nd super procedure.
3. The interpreter finds and executes the version of `startMeUp` in the 2nd super procedure. It runs some code and then does another `RUN SUPER`. Here is where the use of `SEARCH-TARGET` becomes significant. If you didn't specify this keyword on the `ADD-SUPER-PROCEDURE` method, the default would be `SEARCH-SELF`, which means that the `RUN SUPER` statement in the 2nd super procedure would cause the interpreter to search up the 2nd super procedure's *own super procedure stack* for another version of `startMeUp`. Because the 2nd super procedure has no super procedures of its own, the interpreter would find nothing and the `RUN SUPER` statement would return an error.
4. Instead, the use of `SEARCH-TARGET` causes the interpreter to go back to the object procedure where `startMeUp` was run in the first place and continue to search up its stack. This causes it to search the top super procedure.
5. Finally, the interpreter locates and executes the top-most version of `startMeUp`.

Now compare this with what is required if you do not use `SEARCH-TARGET`. For the `RUN SUPER` statement in the 2nd super procedure to execute properly, you need a double set of super procedure definitions, with this statement in the initialization code for the second super procedure:

```
THIS-PROCEDURE:ADD-SUPER-PROCEDURE(hTop).
```

Figure 14–4 diagrams this code.

**Figure 14–4: Super procedure stack execution without SEARCH-TARGET**

Now the **RUN SUPER** statement in the 2nd super procedure causes the interpreter to search up its own stack (4), finding the top-level code to execute (5).

So what's the problem with this approach? This potentially complicates the application considerably. Each super procedure must have its own initialization code to establish its own procedure stack, which must duplicate that portion of the object procedure's stack above itself (somewhat akin to the old song *The Twelve Days of Christmas*, each super procedure moving down the chain from the top must duplicate the chain from itself on up). This is not only more work than the other approach, but it also makes it more difficult to build any flexibility into the scheme. What if some object procedure wants to have a different procedure stack? This isn't really possible because the super procedures themselves duplicate the stack and therefore make it almost impossible to change from object procedure to object procedure. Or what if an object procedure wants to insert an additional super procedure into the stack conditionally? This also can't really be done. Simply put, unless there are specific circumstances dictating otherwise, it is the best practice always to use the SEARCH-TARGET keyword.

Guideline 2: Use TARGET-PROCEDURE to refer back to the object procedure

If code in a super procedure is to be general-purpose, so that it can serve any object procedure needing it, it must always refer back to the object procedure for handles or other values it needs to make calculations or execute code on behalf of the object. There are two ways you can do this:

1. You can pass any needed values into the procedure call as input parameters. This is a simple technique but not necessarily an effective one. For one thing, putting values into parameters hard-codes the list of needed values forevermore. This can lead to serious maintainability problems. Keep in mind that not only every reference to the internal procedure or function, but also every RUN SUPER statement inside it, must duplicate the same parameter list. Any change to that list will be a maintenance nightmare. Also, specifying the needed parameters violates the object-oriented nature of the relationship between the two procedures in that it is forcing the object procedure to know something about the super procedure's implementation of the internal procedure, namely what information it needs to operate on.

Beyond this, it is often awkward to specify parameters for another reason. The object procedure itself probably doesn't need the information itself—it's presumably going to be available to it somehow, through local variable names or whatever else—so the parameter won't be used by a localization of the routine in the object procedure, if there is one. The parameter is *only* needed when the local procedure passes control to a separately compiled procedure that doesn't have access to the local variable. This makes use of the parameter look awkward.

2. A better and more object-oriented technique is to allow the super procedure to refer back to values that the object procedure makes available. This technique provides more independence between the user of the standard behavior and the implementation of that behavior. The traditional Progress programming technique of using **SHARED** variables doesn't work in this case, because there is no top-down hierarchy of procedures to let Progress make those values available. The alternative is to allow the super procedure to refer back into the object procedure in some other way.

As an example, Progress Dynamics uses a combination of two techniques that illustrate ways to do this, both using the **TARGET-PROCEDURE** built-in function.

When an object procedure runs a routine such as `startMeUp` and the interpreter locates it in a super procedure, then within the `startMeUp` code inside the super procedure the **TARGET-PROCEDURE** function evaluates to the procedure handle of the object procedure where the routine was originally run. Alternatively, if there is a local implementation of `startMeUp` in the object procedure, which then executes a **RUN SUPER** statement, then the super procedure code can likewise use **TARGET-PROCEDURE** to obtain the procedure handle of the original object.

Given this handle, the super procedure code can do one of two kinds of things:

- It can run a procedure or function **IN TARGET-PROCEDURE** that returns the desired value. In Progress Dynamics, this is done using a convention of defining functions with the name `getproperty`, where the *property* name is the logical name of the value needed. For every piece of public data that an object makes available to other procedures, the procedure must define such a function. Alternatively, you can define a single function that takes the name of the property as an input parameter and returns its value. Within the limitations of what the object determines to be public data, any other procedure can retrieve needed values at will.
- A second kind of technique is to make the values available through the object's procedure handle. For example, Progress Dynamics stores the handle of a temp-table buffer containing all of a SmartObject's properties in the **ADM-DATA** attribute of the object's procedure handle. User application code can use the similar **PRIVATE-DATA** attribute to store this kind of information as well: either a list of the values themselves, in some delimited form, or a handle of a buffer where they are stored.

Keep in mind that it is the nature of how handles are used in the Progress language that once a procedure object in a given OpenEdge session has the handle of something defined in another procedure instance, it can operate on that handle exactly as the other procedure can. This rule applies to handles of visual controls such as fields, browses, buttons, dynamic buffer handles, dynamic query handles, and dynamic temp-table handles. This is part of what makes the super procedure mechanism so powerful, though it compromises the object-oriented nature of the procedures as well-isolated individual objects. Once you let another procedure gain access to your procedure by giving away, say, the frame handle for the frame your procedure defines, everything else related to that is freely available.

Another important point to keep in mind about TARGET-PROCEDURE is that Progress re-evaluates it every time a new procedure or function name is invoked. For example, throughout the little example in this section, the value of TARGET-PROCEDURE inside any version of `startMeUp` is the procedure handle of the procedure in which `startMeUp` was originally invoked (that is, the handle of the procedure where the original RUN statement was located or, if that original RUN statement was `RUN startMeUp IN some-other-proc-hd1`, then the value of `some-other-proc-hd1`). This is true no matter how many nested levels of RUN SUPER statements you go through. However, as soon as some other routine is run, the value of TARGET-PROCEDURE changes to be the procedure handle where that routine was run. Once any and all versions of that new routine execute and control returns to some version of `startMeUp`, then the value of TARGET-PROCEDURE pops back to what it was before.

For this reason, it is important for super procedures to invoke other routines IN TARGET-PROCEDURE if there is any chance that the newly run routine needs to refer to TARGET-PROCEDURE itself. For example, if the `startMeUp` code in the 2nd super procedure needs to run another internal procedure called `moreStartupStuff`, then *even if moreStartupStuff is also implemented in the 2nd super procedure*, you should invoke it by using the `RUN moreStartupStuff IN TARGET-PROCEDURE` statement if it needs to refer to TARGET-PROCEDURE itself (for example, to retrieve another property value from the object procedure). If you don't do this, then the value of TARGET-PROCEDURE inside `moreStartupStuff` becomes the h2nd super procedure handle, which is not useful. This statement also lets the object procedure localize `moreStartupStuff` if it needs to. If you don't desire this behavior (that is, if you want this subprocedure to be invisible to object procedures), then you should define `moreStartupStuff` as PRIVATE, and then pass the value of TARGET-PROCEDURE into it as a parameter if it's going to be needed.

Guideline 3: Make super procedure code shareable

If you write a super procedure as a library of general-purpose code, it makes sense that in most cases it should be shareable. The use of TARGET-PROCEDURE facilitates this. If the routines in a super procedure always refer back to the TARGET-PROCEDURE, then they always get data values from the procedure they currently support, that is, the procedure they were invoked from. To make sure that stale data isn't left over from call to call, the general rule is to have *no* variables or any other definitions scoped to the super procedure main block. You should place all definitions within each individual internal procedure or function, so that they safely go out of scope when the routine exits.

This also means that you should structure your application so that only one instance of each super procedure starts for a session. Again Progress Dynamics, as an example, uses the simple mechanism of checking existing procedure handles and their filenames to see if the super procedure is already running, as shown in this procedure, used by all SmartObjects:

```
PROCEDURE start-super-proc :
/*-----
 Purpose:      Procedure to start a super proc if it's not already
               running, and to add it as a super proc in any case.
 Parameters:   Procedure name to make super.
 Notes:        NOTE: This presumes that we want only one copy of an ADM
               super procedure running per session, meaning that they
               are stateless (i.e., that every call is independent of
               every other call). This is intended to be the case
               for ours, but may not be true for all super procs.
-----*/
DEFINE INPUT PARAMETER pcProcName AS CHARACTER NO-UNDO.
DEFINE VARIABLE          hProc      AS HANDLE    NO-UNDO.

hProc = SESSION:FIRST-PROCEDURE.
DO WHILE VALID-HANDLE(hProc) AND hProc:FILE-NAME NE pcProcName:
    hProc = hProc:NEXT-SIBLING.
END.
IF NOT VALID-HANDLE(hProc) THEN
    RUN VALUE(pcProcName) PERSISTENT SET hProc.
THIS-PROCEDURE:ADD-SUPER-PROCEDURE(hProc, SEARCH-TARGET).

RETURN.

END PROCEDURE.
```

Guideline 4: Avoid defining object properties and events in super procedures

This guideline is related to the previous ones about always referring back to TARGET-PROCEDURE for any data needed for an operation done by a super procedure. It also relates to keeping all super procedure data local to the individual internal procedure or function. You should define a property or other persistent data value in a super procedure *only if it is truly global.* (That is, if it is to be shared by all object procedures that use that super procedure.) This should definitely be the exception.

Similarly, it is a good rule that super procedures should not subscribe to or publish named events directly. You learn about the Progress PUBLISH and SUBSCRIBE syntax in the “[PUBLISH and SUBSCRIBE statements](#)” section on page 14–37. A PUBLISH or SUBSCRIBE statement should always be done on behalf of the object procedures they serve. Super procedure code might need to subscribe object procedures to named events to set up relationships between procedures.

Forgetting to carry out program actions relative to TARGET-PROCEDURE is one of the most common mistakes in using super procedures.

Guideline 5: Never run anything directly in a super procedure

As discussed earlier, the whole mechanism of using TARGET-PROCEDURE as a means to identify the handle of the object on whose behalf an action is being taken depends on the routine that executes the action being invoked IN the object procedure. The Progress interpreter then takes care of the work of locating the routine in a super procedure and executing it there. This works properly only if application code never runs any routines directly in a super procedure handle. The example shown in [Figure 14–5](#) helps to illustrate this point.

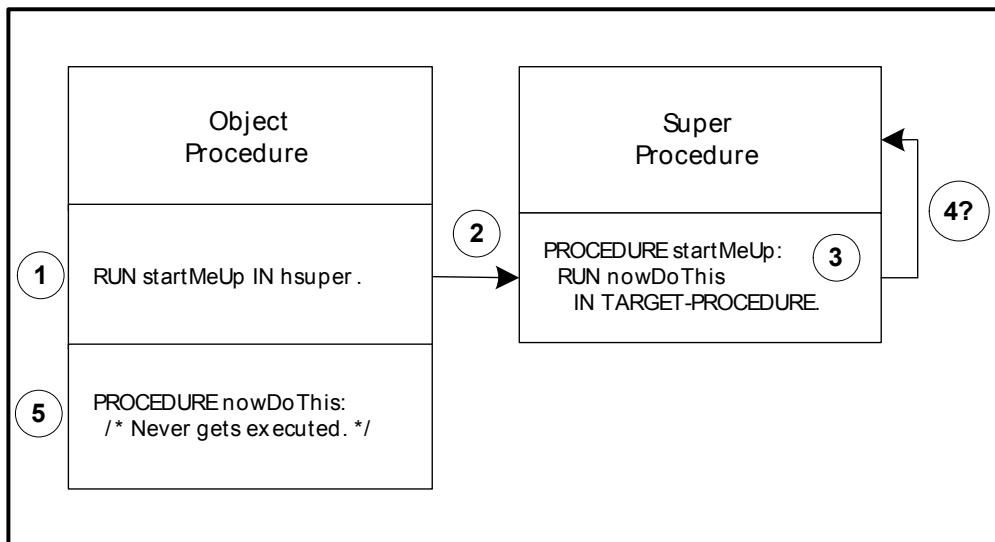


Figure 14–5: Running a procedure within a super procedure

Here's the (undesired) sequence of events:

1. Initialization code in the object procedure runs `startMeUp`, but runs it explicitly IN the handle of its super procedure, rather than letting the interpreter do this implicitly.
2. Progress does what it's told and runs `startMeUp IN hSuper.`
3. Because `startMeUp` was specifically run `IN hSuper`, then `hSuper` becomes the TARGET-PROCEDURE, the handle of the procedure in which the routine was originally invoked.

4. Thus, any reference to TARGET-PROCEDURE causes the interpreter to search the super procedure itself for the routine to execute.
5. Progress never finds and executes the version of nowDoThis back in the object itself.

For the super procedure to get back to the object procedure, it needs to use the SOURCE-PROCEDURE built-in function, but even this won't always work if the original RUN statement is located anywhere except in the actual source code for the object procedure. So, the whole relationship between the object procedure and its supporting super procedure becomes messed up, and you can expect confusing results.

Note that even in a case where you are defining a global property in a super procedure, as discussed in the previous section, an object that wants to get at that property value should still do it indirectly, by requesting it from itself, and letting the interpreter locate the routine that supplies the value in the super procedure. [Figure 14–6](#) shows an example.

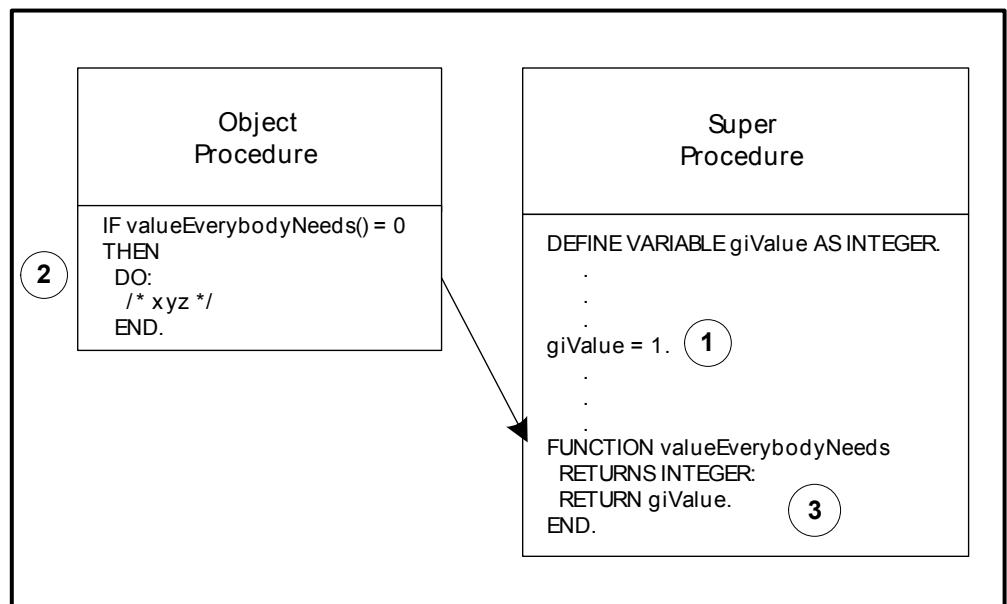


Figure 14–6: A global property in a super procedure

The sequence of events in [Figure 14–6](#) executes as follows:

1. The main block of the super procedure defines the value to return to any requesting object procedure, outside the scope of an internal procedure or function. This is not actually a Progress GLOBAL variable (which you should avoid), but because it is effectively global to all procedures using this super procedure, in the sense that its value is persistent across calls from many objects, you can use a naming convention of preceding the name with a `g` to make it clear that it's defined at the scope of the whole super procedure.
2. An object procedure requests the value by executing a function that's defined in the super procedure. But the function is invoked in the object procedure itself, not directly in the handle of the super procedure. The interpreter locates the function in the super procedure and executes it there. Invoking the function directly in the super procedure by using the syntax `DYNAMIC-FUNCTION ('valueEverybodyNeeds' IN hSuper)` is a bad idea. First, because it messes up any attempt by the function code to reference `TARGET-PROCEDURE`, and, second, because it hard-codes into the object procedure the fact that the supporting function is implemented in this particular super procedure. This makes the application more difficult to maintain.
3. The supporting function returns the value, which is then used by the object procedure.

This simple example presumes that there is a function prototype defined in the object procedure so that the compiler knows how to evaluate the function. You can accomplish this easily by using the **ProtoGen** tool in the **PRO*Tools** palette of the AppBuilder, which generates a Progress include file containing a prototype for every routine defined in the super procedure. Your code can then include this file in the object procedure to make the prototypes available.

Using session super procedures

In addition to associating super procedures with a specific object procedure, you can also add them to the entire OpenEdge session so that every procedure that executes in the session can take advantage of them. You do this by executing the `ADD-SUPER-PROCEDURE` method from the `SESSION` handle:

```
SESSION:ADD-SUPER-PROCEDURE( hSuper ).
```

The search order the interpreter uses to locate internal procedures and functions now becomes this:

1. The procedure file the routine is run in (that is, either the procedure that actually contains the RUN statement or function reference or, if you use the RUN IN syntax or the DYNAMIC-FUNCTION . . . IN syntax, the procedure handle following the IN keyword).
2. Super procedures added to that procedure handle, starting with the last one added.
3. Super procedures added to the SESSION handle, starting with the last one added.

In addition, in the case of a RUN statement, if an internal procedure of that name is not located anywhere, the final step of the search is for a compiled external procedure (.r file) of that name.

You should exercise caution when adding super procedures to the session, precisely because their contents become available to absolutely every procedure run in the session. Nevertheless, this technique can be effective in some cases, especially for extending the behavior of existing application procedures without having to edit them to put in ADD-SUPER-PROCEDURE statements. If the existing procedure was originally intended to run a compiled external procedure (.r file) and does not explicitly include the .p or .r extension on the filename reference in the RUN statement, then the external .r file can be replaced by a session super procedure that contains an *internal* procedure of the same name. This changes the behavior of the application without making any changes whatsoever to the existing application files (not even recompiling them).

Super procedure example

To show some of the principles of super procedures in action, you can create one that manipulates windows in a simple way. The super procedure needs to have a single entry point, an internal procedure called alignWindow, to position all windows to the same column.



To create an example super procedure:

1. From the AppBuilder, select **New→Structured Procedure**.
2. Make sure the **Procedures** toggle box is checked so that this nonvisual program template is in the list.

3. Add a new procedure called alignWindow:

```
/*
-----  
Purpose: Aligns all windows to the same column position.  
-----  
TARGET-PROCEDURE:CURRENT-WINDOW:COL = 20.0.  
RETURN.  
END PROCEDURE.
```

The TARGET-PROCEDURE handle is the handle of the window procedure that this is a super procedure for. The code simply sets its current window's column position to 20. Note the use of chained attribute references. You can chain together as many object attributes as you need in a single expression, as long as all the attributes (except the last) evaluate to handles. Also, the COL attribute is of type DECIMAL, so don't forget the decimal on the value 20.0.

Structured procedures don't have any visualization, so there's no real design window for them in the AppBuilder. Instead, you get a window with a tree view with all the code sections.

4. Expand the tree view and double-click on any section to bring it up in the **Section Editor**:



Now you need a standard procedure that knows how to start a super procedure. It needs to determine whether it's already running, run it if it's not there, and then make it a super procedure of the requesting procedure.

5. Define a new external procedure called h-StartSuper.p with this code:

```
/*
 * h-StartSuper.p -- starts a super procedure if not already running,
 * and returns its handle. */
DEFINE INPUT  PARAMETER cProcName AS CHARACTER  NO-UNDO.

DEFINE VARIABLE hProc AS HANDLE      NO-UNDO.

/* Try to locate an instance of the procedure already running. */
hProc = SESSION:FIRST-PROCEDURE.
DO WHILE VALID-HANDLE(hProc):
    IF hProc:FILE-NAME = cProcName THEN
        LEAVE. /* found it. */
    hProc = hProc:NEXT-SIBLING.
END.
/* If it wasn't found, then run it. */
IF NOT VALID-HANDLE(hProc) THEN
    RUN VALUE(cProcName) PERSISTENT SET hProc.
/* In either case, add it as a super procedure of the caller. */
SOURCE-PROCEDURE:ADD-SUPER-PROCEDURE(hProc, SEARCH-TARGET).
```

This code looks for a running instance of the procedure name passed in, using the SESSION procedure list. If it's not there, it runs it. Then it adds it as a super procedure of the SOURCE-PROCEDURE, which is the procedure that ran h-StartSuper.p.

6. Add statements to the main block of both h-CustOrderWin6.w and h-OrderWin.w to get h-StartSuper.p to start h-WinSuper.p and then to run alignWindow:

```
MAIN-BLOCK:
DO ON ERROR UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
    ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:
        RUN h-StartSuper.p (INPUT "h-WinSuper.p"). /* add this super procedure
*/
        RUN alignWindow.
        RUN enable_UI.
        ASSIGN cState = Customer.State
            iMatches = NUM-RESULTS("CustQuery").
        DISPLAY cState iMatches WITH FRAME CustQuery.
        APPLY "VALUE-CHANGED" TO OrderBrowse.
END.
```

To review what's happening when you run h-CustOrderWin6.w again, look at the diagram in Figure 14–7.

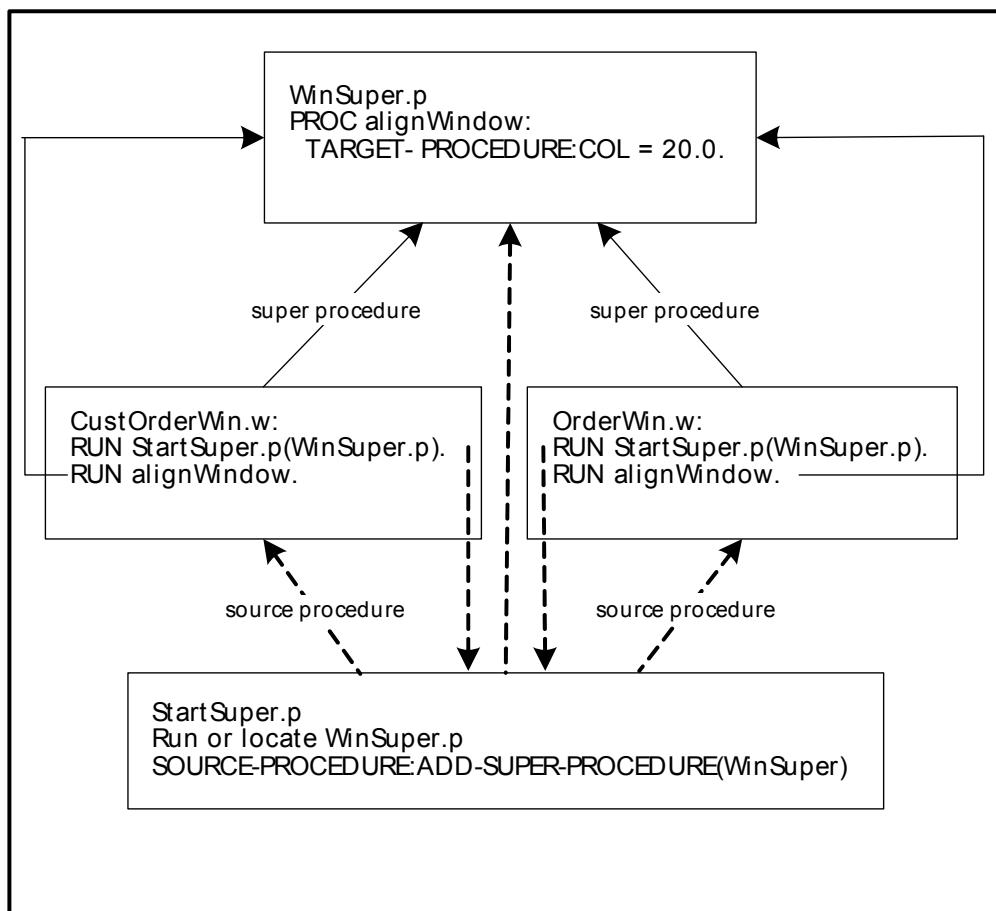


Figure 14–7: Running the sample procedure with a super procedure

The dotted lines represent transient relationships that are only present during startup. Both h-CustOrderWin6.w and h-OrderWin.w run h-StartSuper.p to get their super procedure. When each runs it in turn, SOURCE-PROCEDURE points back to the procedure that ran it. h-StartSuper.p establishes h-WinSuper.p as the super procedure for each one in turn, and then goes away because it was not run persistent itself. It was only needed to set up the relationships.

Once the persistent procedures are all set up, each one runs alignWindow. There's no alignWindow procedure in either of the .w's, so Progress searches the super procedure stack and locates it in h-WinSuper.p and runs that code. Now alignWindow can refer to TARGET-PROCEDURE to access the window handle inside each of the .w's.



To add another small layer of complexity:

- Add an implementation of alignWindow in h-CustOrderWin6.w:

```
/*
Purpose: Local version of alignWindow for h-CustOrderWin
          to set ROW as well as column.
*/
RUN SUPER.
ASSIGN THIS-PROCEDURE:CURRENT-WINDOW:ROW = 10.0.
END PROCEDURE.
```

This code uses a RUN SUPER to invoke the standard behavior in h-WinSuper.p, and then adds some more code of its own, in this case to set the ROW attribute of the window.

So the net effect of these two versions of alignWindow is to set both the COL and the ROW.

- Do the same thing in h-OrderWin.w. Define a version of alignWindow that sets ROW to something different from the local code in h-CustOrderWin6.w:

```
/*
Purpose: Local version of alignWindow for OrderWin sets Row to 20.0.
Parameters: <none>
Notes:
*/
RUN SUPER.
THIS-PROCEDURE:CURRENT-WINDOW:ROW = 20.0.
END PROCEDURE.
```

Now all **Order** windows come up initially in the same place, on top of one another. You can drag them around to see all of them. The diagram in [Figure 14-8](#) represents this second situation.

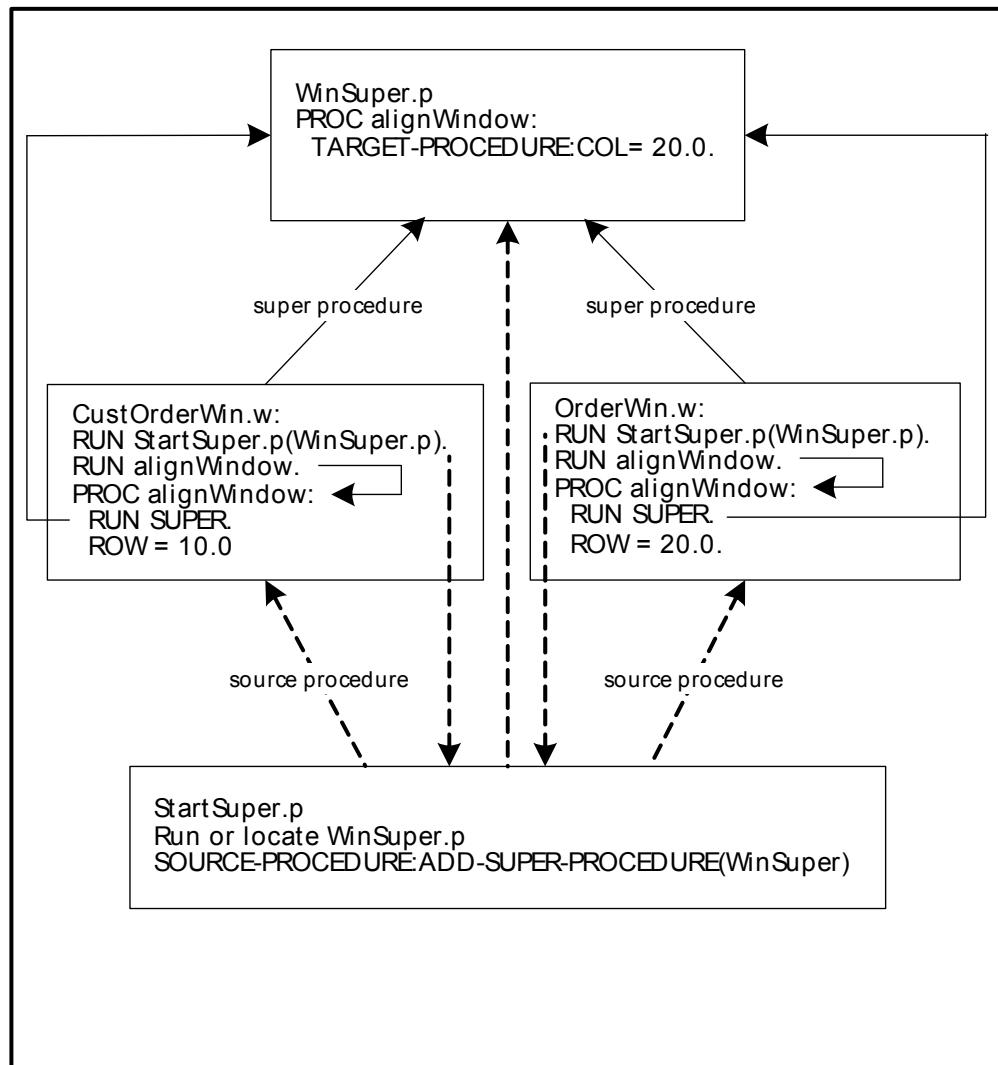


Figure 14–8: Variation on running the sample procedures with a super procedure

When Progress executes the `RUN alignWindow` statement, it finds the internal procedure locally and executes it. The local version first invokes the standard behavior in the super procedure and then extends it with its own custom code.

Super procedures might seem like a complicated mechanism, but consider that they are intended to allow you to provide standard behavior for many procedure objects. This means that you can carefully craft the super procedure code for a type of behavior, and then every other procedure that uses that super procedure inherits the behavior automatically and transparently. This is the power of super procedures.

PUBLISH and SUBSCRIBE statements

When you have multiple procedures running in a session, you might want them to communicate with each other without being strictly bound to each other. In other words, one procedure might want to send a message to other procedures that are interested in receiving that message without knowing or caring just how many such procedures there are, what their procedure handles are, or what they intend to do with the information. Likewise, a procedure might want to post an interest in receiving a message on a subject without necessarily knowing or caring where the message comes from.

Progress supports PUBLISH and SUBSCRIBE keywords for this purpose. One procedure can publish a named event when something of interest happens and other procedures running in the same session can subscribe to that same event name, either in a particular procedure handle or anywhere in the session. When the event occurs, an internal procedure in each subscriber runs. If there are no subscribers, then nothing happens and no error results. If there are many subscribers, they all get the message. In effect, a PUBLISH statement amounts to Progress executing this pseudo-code:

```
FOR EACH subscriber-handle:  
    RUN event-procedure IN subscriber-handle NO-ERROR.  
END.
```

The order in which multiple subscribers receive an event is undefined.

Just as with any procedure call, a PUBLISH statement can include one or more parameters. These are normally INPUT parameters, although under some circumstances other parameter modes can be useful.

Subscribing to an event

Since a procedure has to subscribe to an event before anything happens when another procedure publishes it, you'll look at the SUBSCRIBE statement first. Here is its syntax:

```
SUBSCRIBE [ PROCEDURE subscriber-handle ]
[ TO ] event-name-expr
{ IN publisher-handle | ANYWHERE }
[ RUN-PROCEDURE local-internal-procedure ]
[ NO-ERROR ]
```

By default, a SUBSCRIBE statement registers the request on behalf of the external procedure that contains the SUBSCRIBE statement. The value of the procedure handle is the built-in Progress handle THIS-PROCEDURE. For the subscriber to receive the event, it must be running at the time the event occurs, so normally this means that you should only include a SUBSCRIBE statement in a procedure that is run persistent.

You can, however, create a service procedure that subscribes other procedures to events. In that case, you can include the PROCEDURE *subscriber-handle* phrase and the SUBSCRIBE is done on behalf of that other procedure handle.

The *event-name-expr* is a string expression holding the name of the event to publish. This is a standard Progress name of the same type as an internal procedure name. The default action when the event is published is to run an internal procedure in the subscriber with the same name as the event.

When you subscribe to an event, you can either subscribe to it in a specific running procedure handle that is available, or you can use the ANYWHERE keyword to indicate that you want to be notified when this event occurs anywhere in your session. If you specify the IN *publisher-handle* phrase, then the publisher must be a procedure that is already running and that remains running until it publishes the event. There is no way to subscribe to events that are published by procedures in other OpenEdge sessions.

One common practice is to subscribe to events that are published by the procedure that started the subscriber. The handle of that procedure is available in the SOURCE-PROCEDURE handle. An example in the “[PUBLISH/SUBSCRIBE example](#)” section on page 14–41 shows you how this works.

A procedure can also subscribe to events in the SESSION handle. Other procedures can then publish events from the SESSION handle, thus using it as a kind of central coordinating point for events.

If the name of the internal procedure you want to run in response to the event must be different from the event name itself, then you include the RUN-PROCEDURE *local-internal-procedure* phrase in the SUBSCRIBE statement. You must implement the internal procedure in the subscriber (or one of its super procedures, if any).

If it is possible that the *subscriber-handle* or *publisher-handle* might not be valid at the time the statement is executed, you can include the NO-ERROR phrase to suppress any error messages that would result from this. In this case, the SUBSCRIBE has no effect, and the ERROR-STATUS handle holds a status and message describing the error.

Publishing an event

Once there are one or more subscribers to an event, then a procedure can publish that event and each of the subscribers receives it. The publisher does not have to preregister the intention of publishing the event, even though the subscriber must register the intent to receive it. This is the syntax for the PUBLISH statement:

```
PUBLISH event-name [ FROM publisher-handle ]
[ ( parameter [ , . . . ] ) ].
```

As with the SUBSCRIBE statement, the default is to publish the event from the procedure that contains the PUBLISH statement. A procedure can also publish events on behalf of another procedure by including the FROM *publisher-handle* phrase. Any subscriber that has subscribed to the event of the same name in the *publisher-handle*, whether implicit, explicit, or ANYWHERE, receives it.

Passing parameters

The publisher can pass parameters in the same way as for a RUN statement. Ordinarily, any such parameters are INPUT parameters so that the publishing procedure can pass one or more values into each subscriber. INPUT parameters are the norm because the publisher normally does not want to know the specifics of who the subscribers are and therefore is unlikely to be interested in output from them. If the publisher wants to get a specific response back, then it is better to use a RUN statement with a known procedure handle to run it in.

In particular, an OUTPUT parameter is almost always a bad idea because its value is received only from the last subscribing procedure to execute. If there are multiple subscribers, the OUTPUT parameter returned by the rest of them is discarded. Since there is no way to determine which of multiple subscribers will run last, this is not a reliable or useful mechanism.

On the other hand, there are times when it can be useful to pass an INPUT-OUTPUT parameter. In this case, each subscriber in turn receives the current value of the parameter and can act on it and modify it. The next subscriber receives the modified value and can modify it further.

Finally, the publisher receives it as output from the final subscriber and can see the collective result. This is useful in cases where all the subscribers need to contribute something to a final total, perhaps adding a value to the current value of the parameter or appending something to the end of a delimited list. The publisher then sees the result of all the calls. Again, there is no determined order for the subscribers to be called, but if the order of execution is not important, then an INPUT-OUTPUT parameter is sometimes useful.

The final subscriber can also pass back a string in a RETURN statement that is received by the publisher as the RETURN-VALUE just as if the publisher had used a RUN statement. Again, there is no way to know which subscriber will be the final one called, so each subscriber could append text to the existing RETURN-VALUE in a way similar to using an INPUT-OUTPUT parameter in a form such as the one in the following example. In this code, the RETURN statement appends the value of the local CHARACTER variable cMyIdent to the current RETURN-VALUE:

```
RETURN RETURN-VALUE + " " + cMyIdent.
```

Each subscriber can see the value returned by the previous subscriber in its RETURN-VALUE. If this is the final statement in each subscriber's internal procedure, then the publisher can look at the value of RETURN-VALUE and see all the accumulated values of cMyIdent for all the subscribers.

The PUBLISH statement always executes NO-ERROR. If there are no subscribers, no error message results. If one or more subscribers have a parameter list that does not match the parameters in the PUBLISH statement, then those subscribers do not receive the event and, likewise, no error results. For this reason, it is important to make sure that subscribers have the proper calling sequence. Otherwise, their procedures will not be run and it might not be clear to you why the PUBLISH statement seems to have failed.

Cancelling a subscription

You can also cancel the effect of a SUBSCRIBE statement by using the UNSUBSCRIBE statement:

```
UNSUBSCRIBE [ PROCEDURE subscriber-handle ]
[ TO ] { event-name | ALL } [ IN publisher-handle ]
```

As with the SUBSCRIBE statement, the default is that UNSUBSCRIBE cancels one or more events for the current procedure. Otherwise, you can use the PROCEDURE *subscriber-handle* phrase. You can unsubscribe a specific *event-name* or ALL events. If you include the IN *publisher-handle* phrase, then the *event-name* subscription or ALL events, as specified, are canceled only in that handle. If you specify ALL events without a *publisher-handle*, then all the *subscriber-handle*'s subscriptions are canceled.

PUBLISH/SUBSCRIBE example

The h-CustOrderWin6.w procedure runs a separate window called h-OrderWin.w to display details from the current **Order**. You can run this **Order** window multiple times.



To see how you can use PUBLISH and SUBSCRIBE statements to extend the communication between these procedures:

1. Open h-OrderWin.w and save it as h-OrderWin1b.w.
2. In h-OrderWin1b.w define this internal procedure called **ShipDateChange**:

```
/*
-----Procedure ShipDateChange:
Purpose:    receives the event of the same name to update the
            ShipDate field display when the field value changes.
Parameters: Order Number and new ShipDate SCREEN-VALUE.
Notes:     ShipDateChange is PUBLISHED by h-CustOrderWin6.w
-----*/
DEFINE INPUT PARAMETER iOrderNum AS INTEGER      NO-UNDO.
DEFINE INPUT PARAMETER cShipDate AS CHARACTER    NO-UNDO.

IF STRING(iOrderNum) = Order.OrderNum:SCREEN-VALUE IN FRAME OrderFrame
THEN
DO:
    Order.ShipDate:SCREEN-VALUE IN FRAME OrderFrame = cShipDate.
    shipDate:BGCOLOR = dateColor(PromiseDate, DATE
(ShipDate:SCREEN-VALUE)).
END.

END PROCEDURE.
```

This code responds to an event published by the main window whenever the value of the **ShipDate** field changes. It receives both the **Order Number** and the new **ShipDate** as INPUT parameters. If the **Order Number** matches the one displayed by this instance of OrderWin, then the displayed **ShipDate** is changed to the value passed in, and the `dateColor` procedure is run to flag it with a new background color if its value requires this warning signal.

3. Add this SUBSCRIBE statement to the procedure's main block:

```
ASSIGN THIS-PROCEDURE:PRIVATE-DATA = STRING(OrderNum)
      hSource = SOURCE-PROCEDURE
      shipDate:BGCOLOR = dateColor(PromiseDate, ShipDate).
SUBSCRIBE TO "ShipDateChange" IN SOURCE-PROCEDURE.
IF NOT THIS-PROCEDURE:PERSISTENT THEN
  WAIT-FOR CLOSE OF THIS-PROCEDURE.
```

This sets up every running instance of `h-OrderWin.w` to receive the `ShipDateChange` event when the `SOURCE-PROCEDURE` `h-CustOrderWin` publishes it. There might be multiple instances of `OrderWin` that have subscribed to the event. Under other circumstances, there could just as easily be multiple, completely different versions of the procedure `ShipDataChange` in multiple different procedure files that do different things with the changed **ShipDate** information. The publisher has no reason to know or care.

4. In `h-CustOrderWin`, go into the property sheet for the **OrderBrowse** and enable the **ShipDate** column.
5. Define this LEAVE trigger for the **ShipDate** field:

```
DO:
  IF Order.ShipDate NE DATE(Order.ShipDate:SCREEN-VALUE IN BROWSE
OrderBrowse)
    THEN PUBLISH "ShipDateChange"
      (INPUT Order.OrderNum,
       INPUT Order.ShipDate:SCREEN-VALUE) .
END.
```

If the field value changes, the published event notifies all subscribers.

6. Run the main window and choose the **Order Details** button several times for different **Orders** to bring up multiple instances of `OrderWin`.

7. Modify the **ShipDate** for one of those **Orders**, then tab out of the field.

The corresponding **OrderWin** instance for that **Order** shows the new **ShipDate**, possibly with a different color, depending on the value of **ShipDate**, as this sequence shows:

- a. Run **h-CustOrderWin6.w** and choose the **Order Details** button for several **Orders**:

| Order Num | Ordered | Promised | Shipped | PO |
|-----------|----------|----------|---------|--------|
| 3237 | 11/03/97 | 11/08/97 | | 69940 |
| 3238 | 01/02/98 | 01/07/98 | | 89640 |
| 3239 | 10/20/97 | 10/25/97 | | 209625 |
| 3240 | 01/13/98 | 01/18/98 | | 80891 |
| | | | | |
| | | | | |
| | | | | |

One of those orders shows that there's no **ShipDate**:

Order Detail

| | | | |
|-----------------------------------|----------|-------------|----------|
| Cust Num: | 1612 | Sales Rep: | DKP |
| Order Num: | 3238 | Ordered: | 01/02/98 |
| Order Status: | Ordered | PO: | 89640 |
| Promised: | 01/07/98 | Shipped: | |
| Other Order: <input type="text"/> | | Align Order | |

- b. Enter a **ShipDate** for the **Order**, then tab out of the field:

| Order Num | Ordered | Promised | Shipped | PO |
|-----------|----------|----------|------------|--------|
| 3237 | 11/03/97 | 11/08/97 | | 69940 |
| 3238 | 01/02/98 | 01/07/98 | 01/09/1998 | 89640 |
| 3239 | 10/20/97 | 10/25/97 | | 209625 |
| 3240 | 01/13/98 | 01/18/98 | | 80891 |
| | | | | |
| | | | | |
| | | | | |

Now the **Order Detail** window for that **Order** shows the new date:

Order Detail

| | | | |
|-----------------------------------|----------|-------------|------------|
| Cust Num: | 1612 | Sales Rep: | DKP |
| Order Num: | 3238 | Ordered: | 01/02/98 |
| Order Status: | Ordered | PO: | 89640 |
| Promised: | 01/07/98 | Shipped: | 01/09/1998 |
| Other Order: <input type="text"/> | | Align Order | |

Conclusion

You've learned a lot about the Progress 4GL up to this point, but you still haven't made a single change to a database record. It's about time to get to that! The 4GL allows you to define and scope database updates and their transactions with a flexibility unmatched by any other programming environment. In the next chapter you learn how to do this.

Handling Data and Locking Records

The most important aspect of any enterprise application is updating and maintaining data its database. This book deliberately delays that all-important topic until now so that you have the necessary background in the Progress 4GL to understand and manage database updates. In this chapter, you'll learn about data handling and record locks.

This chapter discusses these topics from the perspective of a distributed application, one where the user interface and the logic that actually update the database are running in different sessions, normally on different machines. You don't have to run your application in this distributed mode, but you should definitely design it this way, so that as your requirements change, you are prepared to take advantage of multiple user interfaces and other requirements that demand that your application run on different platforms, one where the user interface is running and one or more where the database is directly connected.

This distributed environment affects some of the most basic precepts of how you construct Progress 4GL procedures. When the language was first developed, a typical run-time environment was a host-based system with a number of character-screen terminals connected directly to a single computer. Later, this environment was extended to the client/server model, with a network of PCs usually running a graphical interface connected over a network to a server machine where the database was located. In the client/server model, the notion of a direct connection to the database was still maintained across the network so that individual client sessions could run as if they had a local connection to the database, reading and writing individual records directly to and from the database and maintaining locks on records as they were viewed in the client session.

In a distributed application this model changes considerably. There's no longer a direct connection from a client session to the database. Records are read from the database into temp-tables and passed to the client for display and possible update. The client session can no longer hold locks on records it's using, so the server-side code needs to verify whether updated records passed back from the client have been modified by another user since they were read. Likewise, the server-side business logic cannot easily hold record locks while the records are being viewed in the client session. All of these considerations have a fundamental impact on how you design your application and how you write your 4GL procedures.

To help you understand a bit of the historical perspective of how the Progress 4GL has evolved, this chapter begins by introducing you to all the update-related statements in the language, and then explains how they are or are not still relevant to the new distributed paradigm.

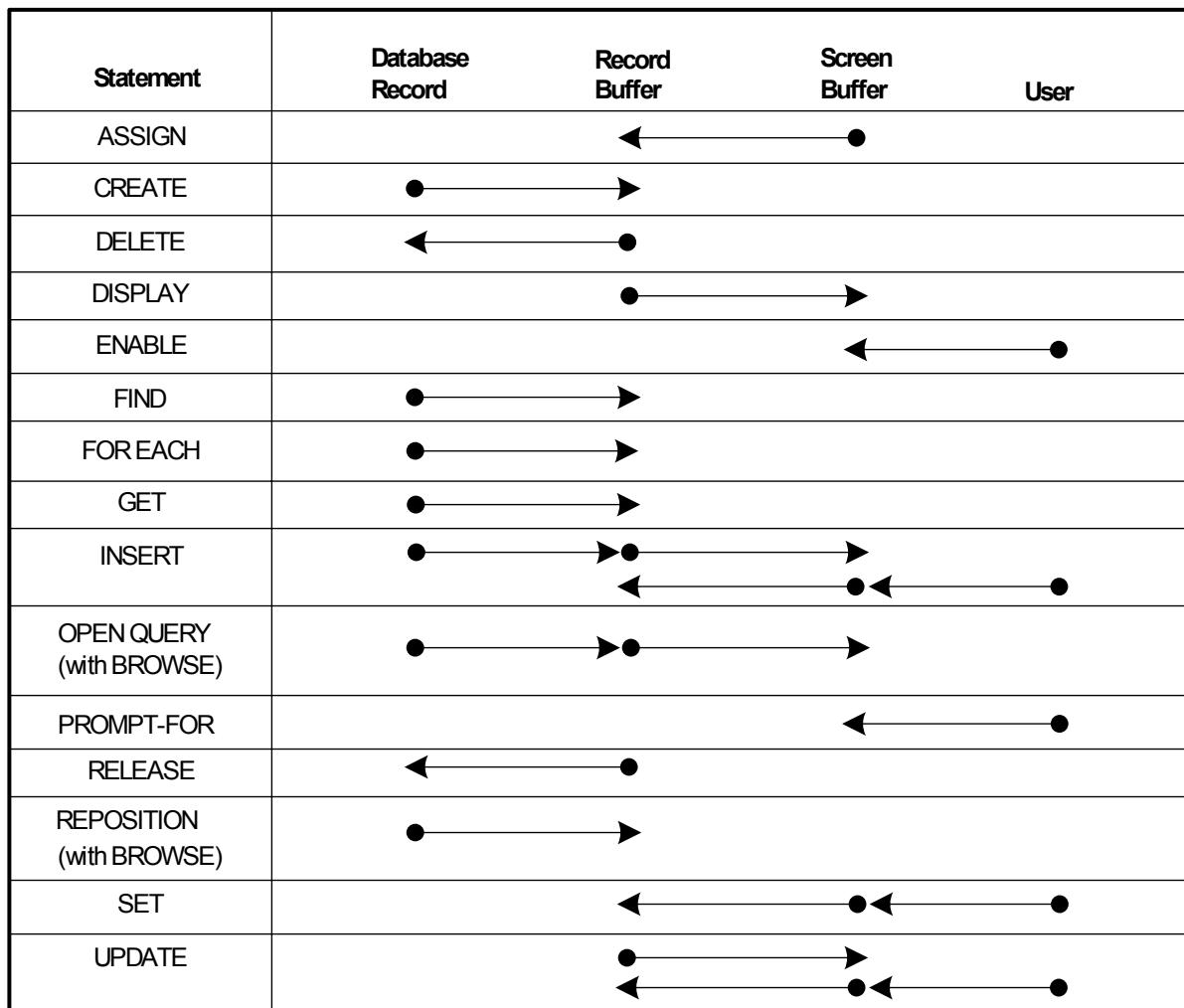
This chapter includes the following sections:

- [Overview of data handling statements](#)
- [Record locking in Progress](#)

Overview of data handling statements

You have already been introduced to many of the 4GL statements that manage data, either by reading records from a data source and making them available to the application in record buffers or by displaying data to the screen and saving changes from the screen. The chart in [Figure 15–1](#) summarizes the scope of each of these statements relative to these four layers of interaction:

- A record in a database table or temp-table.
- A record in a record buffer.
- A record in the screen buffer as the user sees it.
- An action by the user on the screen buffer.

**Figure 15–1: Data handling statements**

From the chart in Figure 15–1, you can see that some single statements span multiple layers in the system. This makes these statements very economical (in terms of syntax requirements) in a host-based or client/server system, but problematic in a distributed system where the user and the screen buffer do not exist in the same session with the database. This means that some of these statements are not of great use in a distributed application and, as a result, you rarely (if ever) use them. These include:

- **INSERT *table-name*.** — You saw a brief example of this statement in Chapter 6, “Procedure Blocks and Data Access.” As you can see from Figure 15–1, the INSERT statement starts at the database level with the creation of a new record, which Progress

then displays (with its initial values) in the screen buffer. The statement pauses to let the user enter values into the record's fields, and then saves the record back to the record buffer.

- **OPEN QUERY *query-name*** (with a browse on a query against a database table) — As you have seen in [Chapter 12, “Using the Browse Object,”](#) there is a direct association between a browse object and its query. When you open the query, records from the underlying table are automatically displayed in the browse.
- **SET *table-name* or *field-list*.** — The SET statement accepts input from the user in the screen buffer for one or more fields and assigns them directly to the record buffer.
- **UPDATE *table-name* or *field-list*.** — The UPDATE statement displays the current values for one or more fields to the screen, accepts changes to those fields from the user, and assigns them back to the record buffer.

Each of these statements is implicitly (if not explicitly) followed by a record RELEASE that actually writes the record with its changes back to the database. The exact timing of that record release is discussed later in the [“Making sure you release record locks”](#) section on page 15–16, but this means that these statements carry data all the way from the user to the database. Since this can't be done in a single session of a distributed application, you won't generally use these statements with database tables.

When can you still use them? As you've already seen, Progress provides the temp-table as a way to present and manage a set of records independent of the database and, in particular, as the way you should pass data from server to client and back. So you could use any of these statements against a temp-table in the client session, because the Database Record layer is, in fact, just the temp-table record in the client session. In particular, opening a query against a client-side temp-table and seeing that query's data in a related browse is the standard way to present tables of data on the client.

You've already seen the following statements in action:

- **ASSIGN *table-name* or *field-list*.** — You can use this statement to assign one or more fields from the screen buffer to the record buffer, for example, on a LEAVE trigger for a field. You can also use it to do multiple programmatic assignments of values in a single statement, as in the ASSIGN *field = value* *field = value* . . . statement.
- **DISPLAY *table-name* or *field-list*.** — This statement moves one or more values from a record buffer to the screen buffer. You could use this statement in a client procedure with a temp-table record, or with a list of local variables.

- **ENABLE *field-list*.** — This statement enables one or more fields in the screen buffer to allow user input. Its counterpart, DISABLE, disables one or more fields.
- **FIND *table-name*.** — This statement locates a single record in the underlying table and moves it into the record buffer. You could use this statement in server-side logic using a database table, or in a client-side procedure using a temp-table.
- **FOR EACH *table-name*:** — This block header statement iterates through a set of related records in the underlying table and moves each one in turn into the record buffer. You could use this statement in server-side logic on a database table or client-side logic on a temp-table.
- **GET *query-name*.** — This statement locates a single record in an open query and moves it into the record buffer.
- **REPOSITION** (with browse) — The REPOSITION statement on a browsed query moves the new record into the record buffer.

There remain several new statements you'll learn about in the next chapter. These include:

- **CREATE *table-name*.** — This statement creates a new record in a table and makes it available in the record buffer.
- **DELETE *table-name*.** — This statement deletes the current record in the record buffer, removing it from the underlying table.
- **RELEASE *table-name*.** — This statement explicitly removes a record from the record buffer and releases any lock on the record, making it available for another user.

In addition to these transaction-oriented statements, the PROMPT-FOR statement enables a field in the screen buffer and lets the user enter a value for it. In an event-driven application, you do not normally want the user interface to wait on a single field and force the user to enter a value into that field before doing anything else, so the PROMPT-FOR statement is also not a frequent part of a modern application. The INSERT, SET, and UPDATE statements similarly have their own built-in WAIT-FOR, which demands input into the fields before the user can continue. For this reason, these statements are also of limited usefulness in an event-driven application, even when you are updating records in a client-side temp-table.

Record locking in Progress

When you read records from a database table, Progress applies a level of locking to the record that you can control so that you can prevent conflicts where multiple users of the same data are trying to read or modify the same records at the same time. This locking doesn't apply to temp-tables since they are strictly local to a single Progress session and never shared between sessions.

When you read records using a FIND statement, a FOR EACH block, or the GET statement on a query that isn't being viewed in a browse, by default Progress reads each record with a SHARE-LOCK. The SHARE-LOCK means that Progress marks the record in the database to indicate that your session is using it. Another user (and the general term *user* here and elsewhere means code in any type of OpenEdge session that is accessing data, whether it's through an actual user interface or not) can also read the same record in the same way—that's why this is called a SHARE-LOCK.

You can also specify a keyword on your FIND or FOR EACH statement, or on the OPEN QUERY statement for a query, to read records with a different lock level. If you intend to change a record, you can use the EXCLUSIVE-LOCK keyword. This marks the record as being reserved for your session's exclusive use where any changes are concerned, including deleting the record. If any other user has a SHARE-LOCK on the record, an attempt to read it with an EXCLUSIVE-LOCK fails. Thus, a SHARE-LOCK assures you that while others can read the same record you have read, they cannot change it out from under you.

Record locking examples

A few simple examples can help illustrate how Progress handles different kinds of locks. When you start the AppBuilder and connect to the Sports2000 database, you are running in single-user mode, as the only user of the database.



To set up your session so you can test locking:

1. Exit your session to free up your single-user connection to the Sports2000 database.
2. Make sure your path includes the `bin` directory under your OpenEdge install directory.
3. From a Windows Command Prompt window, change your directory to your working directory, or wherever your local Sports2000 database is located.

4. Type the command: **proserve Sports2000**

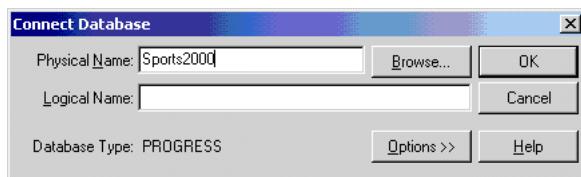
You should see a series of messages as the server starts up.

5. Restart the AppBuilder.

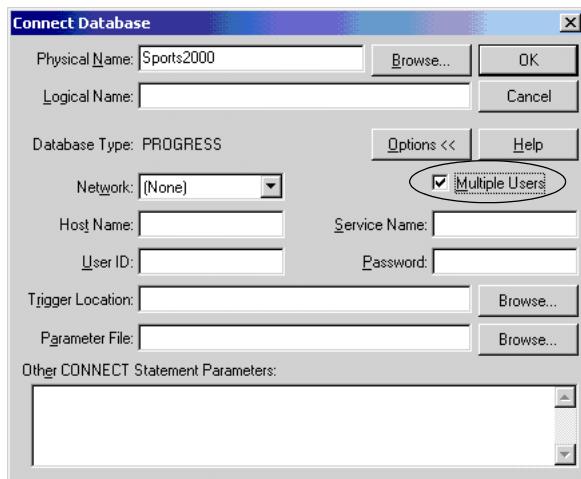
6. From the **Tools** menu, select **Database Connections**.

7. Choose the **Connect** button.

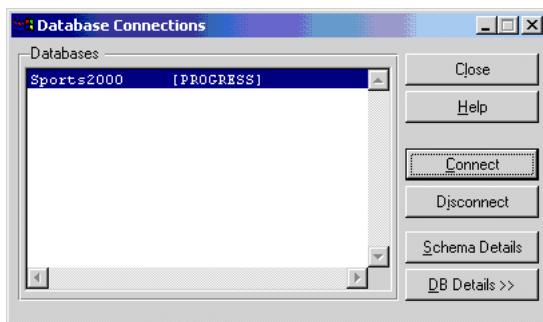
8. Type **Sports2000** as the **Physical Name**, then choose the **Options** button:



9. Check on the **Multiple Users** toggle box so that you connect to the server in multi-user mode:



10. Choose **OK**, then close the **Database Connections** dialog box:

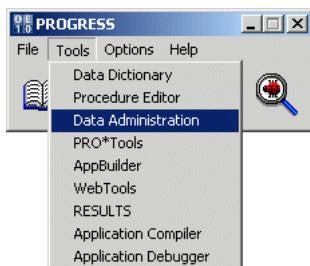


Now your session is connected to the database server. To test the effects of record locking in a multi-user environment, you need to create a second OpenEdge session.



To start up another session:

1. Start the **OpenEdge Desktop**.
2. From the **Tools** menu, select **Data Administration**:



3. From the **Data Administration** menu, select **Database**, then choose **Connect**.
4. Go through the same sequence of steps as before to connect in multi-user mode to the same Sports2000 database server.

Now you're ready to test locking conflicts.

Using EXCLUSIVE-LOCKS

Your first test involves using EXCLUSIVE-LOCKS.



To test locking conflicts when using an EXCLUSIVE-LOCK:

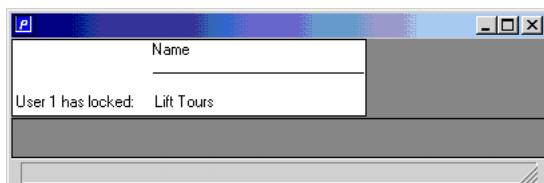
1. Bring up a **New Procedure Window** from the AppBuilder in your first session.
2. Enter this code to retrieve the first **Customer** record with an exclusive lock:

```
FIND Customer 1 EXCLUSIVE-LOCK.  
DISPLAY "User 1 has locked:" NAME.  
WAIT-FOR CLOSE OF THIS-PROCEDURE.
```

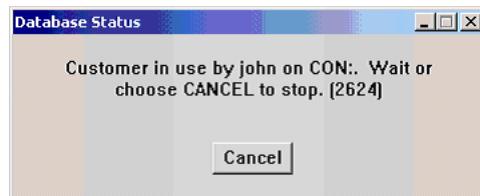
3. From the **Desktop** in your second session, bring up the **Procedure Editor**.
4. Enter the same procedure, but with a message that says **User 2** instead of **User 1**:

```
FIND Customer 1 EXCLUSIVE-LOCK.  
DISPLAY "User 2 has locked:" NAME.  
WAIT-FOR close OF THIS-PROCEDURE.|
```

5. Run the first procedure. The window for the first session comes up:



6. Run the procedure in the second session. Because it's trying to get an exclusive lock on a record already locked by the other process, you get a message telling you that the record is in use and that you must either wait or cancel out of the FIND statement:



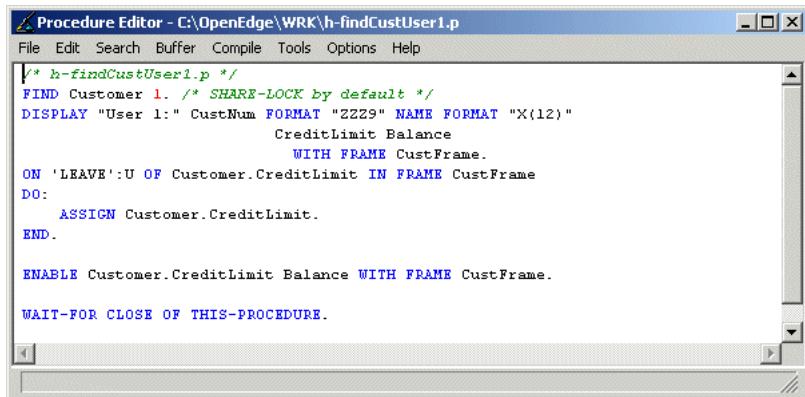
If you close the first window, the second process can now read and lock the **Customer** record:



This example illustrates the most basic rule of record locking. Only one user can have an EXCLUSIVE-LOCK on a record at a time. Any other user trying to lock the same record must either wait for it or cancel the request.

Using and upgrading SHARE-LOCKS

If you remove the EXCLUSIVE-LOCK keyword from the FIND statements in both procedures, Progress reads the record in each case with a SHARE-LOCK by default. The version of the procedure shown in [Figure 15–2](#) is saved in the examples as h-findCustUser1.p. For this test, the second user runs the same procedure with the displayed string “**User 2**”.



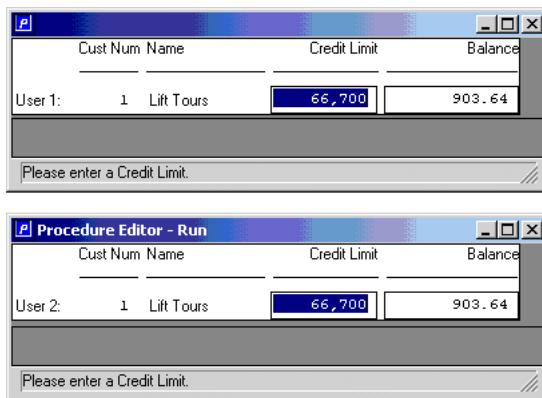
```
Procedure Editor - C:\OpenEdge\WRK\h-findCustUser1.p
File Edit Search Buffer Compile Tools Options Help
/*
 * h-findCustUser1.p
 */
FIND Customer 1. /* SHARE-LOCK by default */
DISPLAY "User 1:" CustNum FORMAT "ZZZ9" NAME FORMAT "X(12)"
      CreditLimit Balance
      WITH FRAME CustFrame.
ON 'LEAVE':U OF Customer.CreditLimit IN FRAME CustFrame
DO:
  ASSIGN Customer.CreditLimit.
END.

ENABLE Customer.CreditLimit Balance WITH FRAME CustFrame.

WAIT-FOR CLOSE OF THIS-PROCEDURE.
```

Figure 15–2: h-findCustUser1.p procedure

You can run both procedures at the same time, and they can both access the **Customer** with a SHARE-LOCK, as shown in [Figure 15–3](#).



| | Cust Num | Name | Credit Limit | Balance |
|------------------------------|----------|------------|--------------|---------|
| User 1: | 1 | Lift Tours | 66,700 | 903.64 |
| Please enter a Credit Limit. | | | | |

| | Cust Num | Name | Credit Limit | Balance |
|------------------------------|----------|------------|--------------|---------|
| User 2: | 1 | Lift Tours | 66,700 | 903.64 |
| Please enter a Credit Limit. | | | | |

Figure 15–3: Result of two sessions running h-findCustUser1.p procedure

However, if you enter a new value in the **CreditLimit** field for either one and tab out of the field, Progress tries to upgrade the lock to an EXCLUSIVE-LOCK to update the record. This attempt fails with the message shown in [Figure 15–4](#) because the other user has the record under SHARE-LOCK.

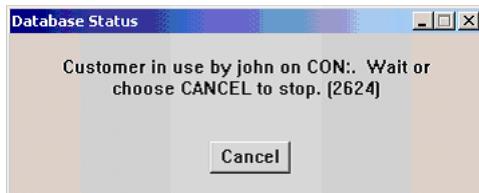


Figure 15–4: SHARE-LOCK status message

If both users try to update the record, they both get the lock conflict message. This situation is called a *deadly embrace*, because neither user can proceed until one of them cancels out of the update, releasing the SHARE-LOCK so that the other can upgrade the lock and update the record.

To avoid this kind of conflict, it is better to read a record you intend to update with an EXCLUSIVE-LOCK in the first place. If you do this in a server-side business logic procedure, which in a modern application is always the case, you won't see a message if the record is locked by another session. Your session simply waits until the lock is freed. If your record locks are confined to server-side procedures with no user interface or other blocking statements, then the problem of a record being locked indefinitely won't ever happen, and a brief wait for a record is not normally a problem.

Using the NO-WAIT Option with the AVAILABLE and LOCKED functions

Just for the record, there are options you can use to make other choices in your procedures in the case of lock contention. If for some reason you do not want your procedure to wait for the release of a lock, you can include the NO-WAIT keyword on the FIND statement or FOR EACH loop. Normally, you should also include the NO-ERROR keyword on the statement to avoid the default “record is locked” message from Progress. Following the statement, you can use one of two built-in functions to test whether your procedure got the record it wanted. The following variant of the procedure for User 2 uses the AVAILABLE keyword to test for the record, and is saved as h-findCustUser2.p. Because the NO-WAIT option causes the FIND statement in this procedure to fail and execution to continue if the record is already locked by another session, the record is not in the buffer, and the AVAILABLE(Customer) function returns false:

```
/* h-findCustUser2.p */
FIND Customer 1 EXCLUSIVE-LOCK NO-WAIT NO-ERROR.
IF NOT AVAILABLE(Customer) THEN
DO:
    MESSAGE "That Customer isn't available for update."
    VIEW-AS ALERT-BOX.
END.
ELSE DO:
    DISPLAY "User 2:" CustNum FORMAT "ZZZ9" NAME FORMAT "X(12)"
        CreditLimit Balance
        WITH FRAME CustFrame.
    ON 'LEAVE':U OF Customer.CreditLimit IN FRAME CustFrame
    DO:
        ASSIGN Customer.CreditLimit.
    END.

    ENABLE Customer.CreditLimit Balance WITH FRAME CustFrame.
    WAIT-FOR CLOSE OF THIS-PROCEDURE.
END.
```

If you run h-findCustUser1.p and h-findCustUser2.p, in that order from different sessions, the second session displays the MESSAGE statement, as shown in [Figure 15–5](#).

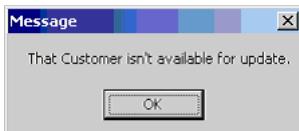


Figure 15–5: EXCLUSIVE-LOCK message

This message occurs because h-findCustUser1.p has read the record with a SHARE-LOCK and h-findCustUser2.p has attempted to read it with an EXCLUSIVE-LOCK, which fails.

Alternatively, if you want to distinguish between the case where the record is not available because it has been locked by another user and the case where the record wasn't found because the selection was invalid in some way, you can use the LOCKED function:

```
FIND Customer 1 EXCLUSIVE-LOCK NO-WAIT NO-ERROR.  
IF LOCKED(Customer) THEN  
    MESSAGE "That Customer isn't available for update."  
    VIEW-AS ALERT-BOX.  
ELSE IF NOT AVAILABLE(Customer) THEN  
    MESSAGE "That Customer was not found."  
    VIEW-AS ALERT-BOX.  
.  
.  
.
```

Once again, because SHARE-LOCKS are of very limited use in application procedures that are distributed or might be distributed between sessions, it is good practice to bypass this method of reading records altogether and always read records with an EXCLUSIVE-LOCK if you know that your procedure updates them immediately.

Also, because the NO-WAIT option is designed for possible record contention in an environment where a user might hold a record lock for a long time while viewing the record on screen or going out for lunch, this option is not normally needed in an application where all database access is confined to the server, where there should be no user interaction with locked records. If you code your application properly to make sure that record locks are not held unnecessarily, then lock contention should almost never be an issue.

Reading records with NO-LOCK

If you want to read records regardless of whether they are locked by other users, you can use a third NO-LOCK option. This option lets you read a record without ever being prevented from doing so by another user's lock. If you do not intend to update the record and are simply reading the data, this is an appropriate option. You must always be aware that reading records with NO-LOCK means that you might read parts of an incomplete transaction that has written some but not all of its changes to the database. In some cases, you can even read a record that has been newly written to the database with its key fields and index information, but not with the changes to other fields in the record. NO-LOCK is the default locking level for queries that have an associated browse, since you would normally not want to lock a whole set of records simply to browse them.

You cannot upgrade a record's lock level from NO-LOCK to EXCLUSIVE-LOCK. If you try to update a record you've read with NO-LOCK, you get an error message from Progress, such as the one shown in [Figure 15–6](#).



Figure 15–6: NO-LOCK error message

You must FIND the record again with an EXCLUSIVE-LOCK if you need to update it.

Making sure you release record locks

Under no circumstances do you want to hold a record lock longer than you need it. The best way to make sure you get the locking you want is to be explicit about it. Do not fall back on Progress defaults, which try to give you reasonable behavior when you don't specify the behavior you want, but which cannot always anticipate what your procedure really requires. Here are two guidelines for using locks:

- **Never read a record before a transaction starts, even with NO-LOCK, if you are going to update it inside the transaction.** If you read a record with NO-LOCK before a transaction starts and then read the same record with EXCLUSIVE-LOCK within the transaction, the lock is automatically downgraded to a SHARE-LOCK when the transaction ends. Progress does *not* automatically return you to NO-LOCK status.
- **If you have any doubt at all about when a record goes out of scope or when a lock is released, release the record explicitly when you are done updating it with the RELEASE statement.** If you release the record, you know that it is no longer locked, and that you cannot unwittingly have a reference to it after the transaction block that would extend the scope of the lock, or even the transaction.

If you observe these two simple guidelines, your programming will be greatly simplified and much more reliable. Here are a few rules that you simply don't need to worry about if you acquire records only within a transaction and always release them when you're done:

- A SHARE-LOCK is held until the end of the transaction *or the record release, whichever is later*. If you avoid SHARE-LOCKS and always release records when you're done updating them, the scary phrase *whichever is later* does not apply to your procedure.
- An EXCLUSIVE-LOCK is held until transaction end *and then converted to a SHARE-LOCK if the record scope is larger than the transaction and the record is still active in any buffer*. Again, releasing the record assures you that the use of the record in the transaction does not conflict with a separate use of the same record outside the transaction.
- When Progress backs out a transaction, it releases locks acquired within a transaction *or changes them to SHARE-LOCK if it locked the records prior to the transaction*. If you don't acquire locks or even records outside a transaction, you don't need to worry about this special case.

Lock table resources

SHARE-LOCKS and EXCLUSIVE-LOCKS use up entries in a lock table maintained by the database manager. The possible number of entries in the lock table defaults to 500. You can change this number using the Lock Table Entries (-L) startup parameter for your OpenEdge session. Progress stops a program if it attempts to access a record that causes it to overflow the lock table.

Optimistic and pessimistic locking strategies

In a traditional host-based or client/server application, you can enforce what is referred to as a *pessimistic locking* strategy. This means that your application always obtains an EXCLUSIVE-LOCK when it first reads any record that might be updated, to make sure that no other user tries to update the same record.

In a distributed application, this technique simply can't work. If you read a single record or a set of records on the server, and pass them to a client session for display and possible update, your server-side session cannot easily hold locks on the records while the client is using them. When the server-side procedure ends and returns the temp-table of records to the client, the server-side record buffers are out of scope and the locks released. In addition, you would not *want* to maintain record locks for this kind of duration, as it would lead to likely record contention.

Note: It is *possible* to write server-side code so that one procedure holds record locks while another procedure returns a set of records to the client, but you should normally not do this.

Instead, you need to adopt an *optimistic locking* strategy. This means that you always read records on the server with NO-LOCK, *if* they are going to be passed to another session for display or processing. When the other session updates one or more records and passes them back, presumably in another copy of the temp-table that sent the records to the client, the server-side procedure in charge of handling updates must:

1. Find again each updated record in the database with an EXCLUSIVE-LOCK, using its key or its RowID.
2. Verify that the record hasn't been changed by another user or at least verify that the current changes do not conflict with other changes.
3. Assign the changes to the database record.

If another user *has* changed the record, then the application must take appropriate action. This might involve rejecting the new changes and passing the other changes back for display, or otherwise reconciling the two sets of changes, depending on the application logic and the nature of the data.

Using FIND CURRENT and CURRENT-CHANGED

When you have a record in a record buffer, you can re-read it from the database to see if it has changed since it was read, using the FIND CURRENT statement or, for a query, the GET CURRENT statement. You can then use the CURRENT-CHANGED function to compare the record currently in the buffer with what is in the database. This is a part of your optimistic locking strategy. The simple example that follows, saved as h-findCurrent.p, shows how the syntax works:

```
/* h-findCurrent.p */
DEFINE FRAME CustFrame Customer.CustNum Customer.NAME FORMAT "x(12)"
          Customer.CreditLimit Customer.Balance.

ON "GO" OF FRAME CustFrame
DO: /* When the user closes the frame by pressing F2, start a transaction: */
DO TRANSACTION:
  FIND CURRENT Customer EXCLUSIVE-LOCK.
  IF CURRENT-CHANGED(Customer) THEN
    DO:
      MESSAGE "This record has been changed by another user." SKIP
      "Please re-enter your changes." VIEW-AS ALERT-BOX.
      DISPLAY Customer.CreditLimit Customer.Balance WITH FRAME CustFrame.
      RETURN NO-APPLY. /* Cancel the attempted update/GO */
    END.
    /* Otherwise assign the changes to the database record. */
    ASSIGN Customer.CreditLimit Customer.Balance.
  END.
  RELEASE Customer.
END.

/* To start out with, find, display, and enable the record with no lock. */
FIND FIRST Customer NO-LOCK.
DISPLAY Customer.CustNum Customer.NAME Customer.CreditLimit Customer.Balance
      WITH FRAME CustFrame.
ENABLE Customer.CreditLimit Customer.Balance WITH FRAME CustFrame.
/* Wait for the trigger condition to do the update and close the frame. */
WAIT-FOR "GO" OF FRAME CustFrame.
```

The code executed first is at the bottom of the procedure, after the trigger block. It finds a desired **Customer** NO-LOCK, so as to avoid lock contention, then displays it and any enabled fields for input. If the user changes the **CreditLimit** or **Balance** in the frame and presses **F2**, which fires the **GO** event for the frame, the code re-reads the same record with an EXCLUSIVE-LOCK and uses CURRENT-CHANGED to compare it with the record in the buffer. Note that because the changes haven't been assigned to the record buffer yet, the record in the buffer and the one in the database should be the same if no one else has changed it. If someone has, then the procedure displays a message, displays the changed values, and executes a RETURN NO-APPLY to cancel the **GO** event. Otherwise, it assigns the changes.

The DO TRANSACTION block defines the scope of the update. The RELEASE statement after that block releases the locked record from the buffer to free it up for another user. You'll learn more about these statements in the next chapter.



To test this procedure:

1. Run this procedure in both sessions. Either session can update the **CreditLimit** or **Balance**, because neither session has the record locked. One session displays it:

| Cust Num | Name | Credit Limit | Balance |
|----------|------------|--------------|---------|
| 1 | Lift Tours | 66,700 | 903.64 |

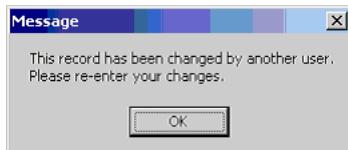
Please enter a Credit Limit.

2. In the other session, update it and save the change by pressing F2:

| Cust Num | Name | Credit Limit | Balance |
|----------|------------|--------------|---------|
| 1 | Lift Tours | 66,800 | 903.64 |

Procedure complete. Press space bar to continue.

3. If you try to update the same record in the first session, you see a message:



The change made by the other session is displayed and, because you now see the record as it's saved in the database, you can now re-enter your change and save it.

In a distributed application, it's much more difficult to manage this type of situation. In the event that a server procedure gets a modified record back from another session that has already been changed, it must somehow send a response back to that session to inform it that there is a conflict. The procedure that detects the conflict does not have access to the user interface and so cannot display a message or change values directly. This is why Progress provides SmartObjects and the Progress Dynamics framework, which handle passing data from server to client, manage record locks and record contention, and communicate messages and updates back to the client from the server transparently. But it does all this using the same Progress 4GL statements that you can use in procedures you write from scratch.

In the next two chapters, you'll explore how to write your server-side logic so as to manage database transactions and record locks properly.

16

Updating Your Database and Writing Triggers

In this chapter, you'll learn about updating your database and writing triggers—topics you need to understand to build the business logic that is the heart of your application.

This chapter includes the following sections:

- [Transactions in Progress](#)
- [Building updates into an application](#)
- [Defining database triggers](#)

Transactions in Progress

A *transaction* is a set of related changes to the database that the database either completes in its entirely or discards, leaving no modification to the database. In other contexts this might be referred to as a *physical transaction* or *commit unit*. In the most basic case, Progress assures that if you assign multiple field values to a database table, it either applies all of those changes or none of them. But in many cases, you need to update multiple related records in the database with the assurance that all of the changes are made together. For example, you might update an **Order**, its **OrderLines**, and the **Customer** for the **Order** in a single transaction. In the business logic for that operation you update the **Customer Balance** to reflect the total of all the **OrderLines**. You want to know that the adjustment you make to the **Customer Balance** is always written to the database along with the **OrderLine** records that are the detail for that adjustment, and that some subset of these changes are never made without the rest of them. If the changes can't be completed for any reason (whether it is a validation error that your application detects, a database error, a system hardware failure, or anything else), the database needs to make all the changes successfully or back out any partial changes so that the records are restored to their state before the transaction began. Many users need to be able to update the same database concurrently with the same assurance. Progress transactions and the integrity of the OpenEdge database, together with record locking, assure that this is always the case. Your application procedures define the scope of every transaction that updates the database so that you have complete control over the unit of work that is reliably committed or rolled back.

Transaction blocks

Transactions in Progress are scoped to blocks. You're already familiar with the concept of building procedures up out of nested blocks of procedural statements and how record scoping is affected by those blocks. Transaction scoping works in much the same way and is closely tied to the scoping of records to blocks. Some blocks in Progress procedures are transaction blocks and some aren't, according to these rules:

- You can explicitly include the **TRANSACTION** keyword on a **FOR EACH** or **REPEAT** block, or on a **DO** block with the optional error-handling phrase beginning **ON ERROR**. Any block that uses the **TRANSACTION** keyword becomes a transaction block.
- Any block that directly updates the database or directly reads records with **EXCLUSIVE-LOCK** likewise becomes a transaction block. This can be a procedure block, a trigger block, or each iteration of a **DO**, **FOR EACH**, or **REPEAT** block.

A direct database update can be, for example, a CREATE, DELETE, or ASSIGN statement. A block is said to be directly reading records with EXCLUSIVE-LOCK if at least one of the FIND or FOR EACH statements that has the EXCLUSIVE-LOCK keyword is not embedded in an inner block enclosed within the block in question.

A DO block without the NO-ERROR phrase does not automatically have the transaction property. Also, a procedure that would start a transaction on its own can run from another procedure that has already started a transaction. In this case, you enclose the called procedure in the larger transaction of the calling procedure. Once a transaction is started, all database changes are part of that transaction until it ends. Each user or session can have just one active transaction at a time. If one transaction block is nested within another, the inner block forms a subtransaction that you can programmatically undo if it causes an error or fails some validation constraint. You'll learn about subtransactions later in [Chapter 17, “Managing Transactions.”](#)

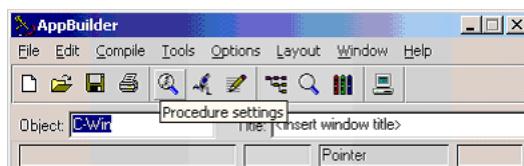
Building updates into an application

In this section, you build a new window to display data and capture updates, along with a separate procedure to validate those updates and write them to the database. Because you should get into the habit of not mixing user interface procedures and data management procedures, the UI and the database access are in separate procedures. You'll use temp-tables to pass data back and forth and display data from those temp-tables in the user interface.



To define temporary tables for your window:

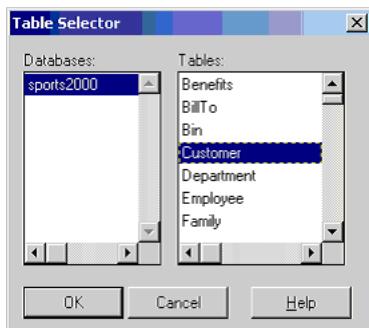
1. In the AppBuilder, select **New→Window**.
2. Choose the **Procedure settings** button:



First, you need to define three temp-tables that hold data received from the database. If you let the AppBuilder generate the definitions for them, you can then provide a user interface for them just as you can for database tables.

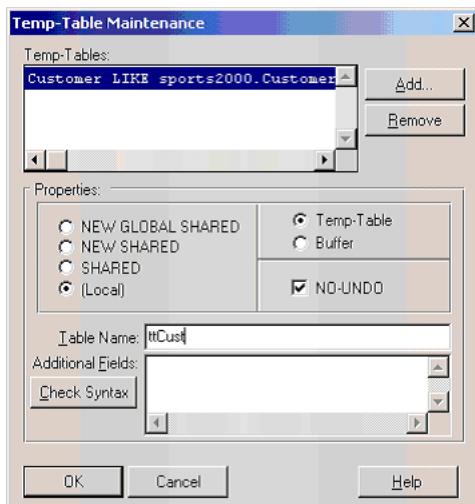
3. In the **Procedure Settings** dialog box, choose the **Temp-Table Definitions** button .

4. Choose the **Add** button, then select the **Customer** table from the **Table Selector** dialog box:



By default this generates a temp-table definition for a table with the same name, **Customer**, that is exactly LIKE the database table.

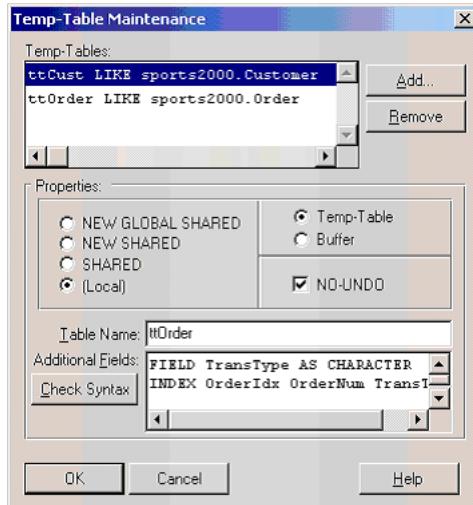
5. To change this default, type **ttCust** as the **Table Name**:



Note that there is a **NO-UNDO** toggle box that is checked on by default. Leave this on. This option adds the NO-UNDO keyword to the temp-table definitions, which is appropriate because Progress does not need to roll back any changes to the temp-tables themselves as part of a transaction.

6. Choose **Add** again, then this time select the **Order** table.

7. In the editor labeled **Additional Fields**, add a new field definition for a CHARACTER field called **TransType**. Your procedure uses this to keep track of changed **Order** records.
8. In the same editor, define an index for your temp-table called **OrderIdx** with the **OrderNum** and **TransType** fields:



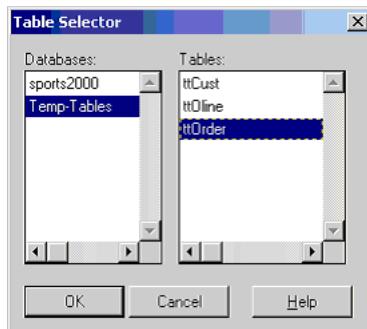
9. Repeat Step 3 through Step 8 for the **OrderLine** table. Add a temp-table for **OrderLine** called **ttOline** with a **TransType** field and an **INDEX OlineIdx** on **OrderNum**, **LineNum**, and **TransType**.
10. Choose **OK** to save all these new temp-table definitions.



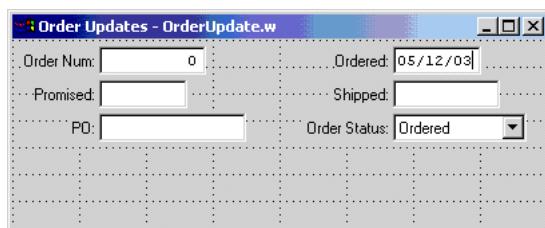
To set up the user interface of your window:

1. In the AppBuilder Palette, choose the **DB-Fields** icon and then click on your design window.

When you use the AppBuilder to define temp-tables for a procedure, it keeps track of them by treating them as if they were in a special database called Temp-Tables, so you see this listed along with the actual database you're connected to:



2. Select **Temp-Tables** from the **Databases** list and **ttOrder** from the **Tables** list.
3. From the **Multi-Field Selector** dialog box, choose the fields: **OrderNum**, **OrderDate**, **PromiseDate**, **ShipDate**, **PO**, and **OrderStatus**, and lay them out in the frame.
4. Give your window the title **Order Updates**.
5. Give the window's frame the name **OrderFrame**.
6. Save this procedure as **h-OrderUpdate.w**:

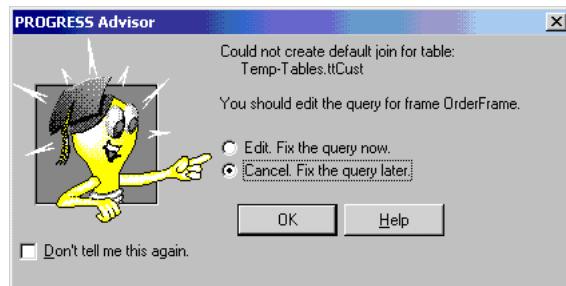


Note that because you used the **LIKE** keyword to define your **ttOrder** temp-table based on the **Order** database table, its fields inherit all the attributes of the corresponding database fields, including their label, data type, and display type.

7. Add a rectangle under the **Order** fields and give it a **Background Color** of brown just to create a divider between the **Order** fields and the rest of the frame.
8. Add **DB-Fields** from the temp-table **ttCust**.

When you go to do this, the AppBuilder tries to define a default join between **ttCust** and the **ttOrder** temp-table you've already used in the frame. It's unable to do this for temp-tables so it puts up an Advisor message to this effect.

9. Select the **Cancel** option to tell the AppBuilder not to worry about the default join, then choose **OK**:



Your procedure will receive the correct **Customer** for an **Order** from the database procedure it uses.

10. Arrange the fields from **ttCust** in the lower part of the frame.
11. Disable the **ttCust** fields. These are used only to display values from the **Order**'s **Customer**.

When you're done, your design window should look something like this:



To add a browse for the Order's OrderLines:

1. Select the **Browse** icon from the AppBuilder **Palette** and drop it onto the bottom of the design window.

2. Add **ttOline** from the list of **Available Tables**.

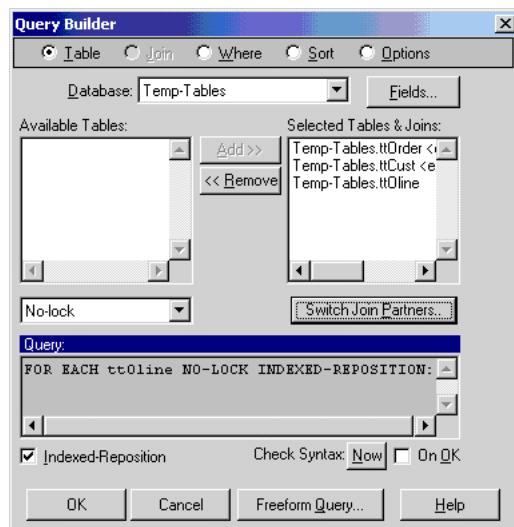
Once again the AppBuilder tries to provide you with a default join that you won't need.

3. Select **ttOline** from the **Selected Tables** list and click the **Switch Join Partners** button.

4. In the **Select Related Table** dialog box, select **(None)**:



5. Choose **OK**. The **Query Builder** shows **ttOline** without a join to another table:

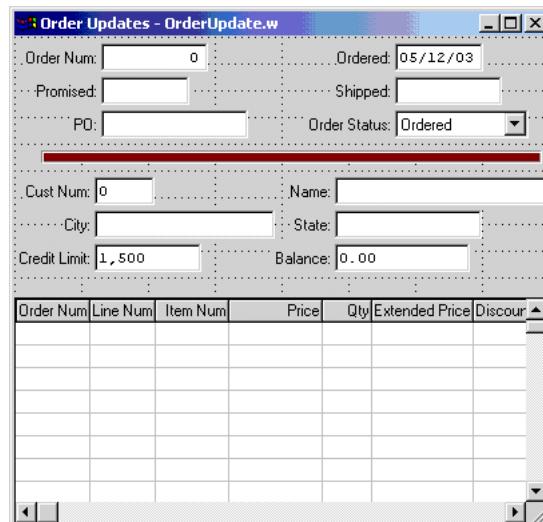


6. Choose the **Fields** button in the **Query Builder**. Select all of the **ttOline** fields in an order such as: **OrderNum**, **Line Num**, **Item Num**, **Price**, **Qty**, **Extended Price**, and **Discount**.

Note that the fields from the **ttCust** and **ttOrder** tables are also in the list, but you don't want them in the browse.

7. Check on the **Enable** toggle box for the columns **ItemNum**, **Price**, **Qty**, **Discount**, and **OrderLineStatus** so that the user can change these values for an **Order**'s lines.
8. Change the browse **Object Name** to **OlineBrowse**.

Now the window should look like this:



To write code that populates the fields and the browse and lets you update the Order:

1. In the **Definitions** section, define a handle called **hLogic**:

```
/* Local Variable Definitions --- */  
DEFINE VARIABLE hLogic AS HANDLE      NO-UNDO.
```

This variable holds the procedure handle of the procedure you write next where all the logic is to read records for the **Order** from the database and accept changes back from the client.

2. In the main block, add a line to run the `h-OrderLogic.p` logic procedure as a persistent procedure and save its handle:

```

MAIN-BLOCK:
DO ON ERROR  UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
  ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:
  RUN enable_UI.
RUN h-OrderLogic.p PERSISTENT SET hLogic.
IF NOT THIS-PROCEDURE:PERSISTENT THEN
  WAIT-FOR CLOSE OF THIS-PROCEDURE.
END.

```

3. Add two buttons to the right of the **OrderNum** field: call one **btnFetch** with the label **Fetch**, and call the other **btnSave** with the label **Save**.

You'll use these buttons to retrieve data from the logic procedure and then to return any changes. The user can enter an **Order** number and then chooses **Fetch** to retrieve it with its **Customer** and **OrderLines**, make changes, and then send the changes back to the logic procedure by choosing **Save**.

4. In the CHOOSE trigger for **btnFetch**, define a buffer named **bUpdateOline** for the **ttOline** table:

```

DO:
  DEFINE BUFFER bUpdateOline FOR ttOline.

```

You'll use this buffer to have two records for each **OrderLine**: one with any changes that are made and one that saves the original values before changes.

5. Empty the three temp-tables to prepare for fetching a requested **Order** and its customer and lines:

```

/* Remove any data leftover from previous calls. */
EMPTY TEMP-TABLE ttCust.
EMPTY TEMP-TABLE ttOrder.
EMPTY TEMP-TABLE ttOline.

```

6. Run an internal procedure called `fetchOrder` in the logic procedure's handle. This needs to pass in the value in the **OrderNum** field. It gets back three temp-tables for the **Order** and its related data:

```
/* Retrieve all related records for the Order. */
RUN fetchOrder IN hLogic
  (INPUT INTEGER(ttOrder.OrderNum:SCREEN-VALUE),
   OUTPUT TABLE ttOrder,
   OUTPUT TABLE ttCust,
   OUTPUT TABLE ttOLine) .
```

7. Add code so that if the **Order** isn't found or if there's some other error, this comes back in the RETURN-VALUE:

```
/* Check for an error such as order-not-found. */
IF RETURN-VALUE NE "" THEN
DO:
  MESSAGE RETURN-VALUE.
  RETURN NO-APPLY.
END.
```

8. Because there's always exactly one **ttOrder** record (the one the user requested) and one **ttCust** record for that **Order**, you can just FIND these in the temp-tables that came back to bring them into their respective buffers, and then display their field values:

```
FIND ttOrder.
FIND ttCust.
DISPLAY ttOrder.OrderNum ttOrder.OrderDate ttOrder.PromiseDate
      ttOrder.ShipDate ttOrder.PO ttOrder.OrderStatus
      ttCust.CustNum ttCust.NAME ttCust.City ttCust.State
      ttCust.CreditLimit ttCust.Balance
WITH FRAME OrderFrame.
```

9. For each of the **ttOlines**, you need to create a copy of the record that holds any updates that are made to it. This is so that you can also keep the original *before image* of each record to compare with the database, to see if the record has been changed by another user. The copy that can be updated is marked with a **TransType** of "U":

```
/* For each OrderLine, create a record to hold any updates. */
FOR EACH ttOline WHERE ttOline.TransType = "":
    CREATE bUpdateOline.
    BUFFER-COPY ttOline TO bUpdateOline ASSIGN TransType = "U".
END.
```

10. To display the **OrderLine** records the user can update, you just open the browse's query:

```
/* Display just the updatable records in the browse. */
OPEN QUERY OlineBrowse FOR EACH ttOline WHERE ttOline.TransType = "U".
END.
```



To create the separate logic procedure that retrieves data from the database and later applies any updates to the database:

1. Select **New→Structured Procedure** in the AppBuilder to create a procedure you'll call **h-OrderLogic.p**.
2. In the **Definitions** section, repeat the temp-table definitions. Make the temp-tables NO-UNDO (which is how you defined them in the calling procedure):

```
/* ***** Definitions *****/
DEFINE TEMP-TABLE ttCust NO-UNDO LIKE Customer.
DEFINE TEMP-TABLE ttOrder NO-UNDO LIKE Order FIELD TransType AS CHARACTER
    INDEX OrderIdx OrderNum TransType.
DEFINE TEMP-TABLE ttOline NO-UNDO LIKE OrderLine FIELD TransType AS
    CHARACTER
    INDEX OlineIdx OrderNum LineNum TransType.
```

You define them NO-UNDO because changes to the records in the temp-tables themselves do not need to be rolled back by the Progress transaction mechanism. Defining them as NO-UNDO saves Progress the overhead of preparing to roll back changes.

In a larger application, these could become include files used in all the procedures that reference these tables.

3. Add the `fetchOrder` internal procedure to load all the needed data for the `OrderNum` passed in:

```
/*
-----Procedure fetchOrder:
Purpose:  Return an Order, its Customer, and all its related OrderLines
          to the caller.

-----*/
DEFINE INPUT  PARAMETER piOrderNum AS INTEGER      NO-UNDO.
DEFINE OUTPUT PARAMETER TABLE FOR ttOrder.
DEFINE OUTPUT PARAMETER TABLE FOR ttCust.
DEFINE OUTPUT PARAMETER TABLE FOR ttOline.

FIND Order WHERE Order.OrderNum = piOrderNum NO-LOCK  NO-ERROR.
IF NOT AVAILABLE(Order) THEN
    RETURN "Order not found".

EMPTY TEMP-TABLE ttOrder.
CREATE ttOrder.
BUFFER-COPY Order TO ttOrder.
FIND Customer OF Order.
EMPTY TEMP-TABLE ttCust.
CREATE ttCust.
BUFFER-COPY Customer TO ttCust.
EMPTY TEMP-TABLE ttOline.
FOR EACH OrderLine OF Order:
    CREATE ttOline.
    BUFFER-COPY OrderLine TO ttOline.
END.

RETURN "".
END PROCEDURE.
```

Now the retrieval end of your sample procedure window should work.

- To test it, run h-OrderUpdate.w, type an **Order Number**, and then choose the **Fetch** button:



You can modify fields in the **Order** record and in one or more of the browse rows for **OrderLines**. Remember that you are not making changes to the database when you do this, because your user interface is just working with temp-tables. So you need to write trigger code for the **Save** button and a procedure in the logic procedure to handle the updates.



To add the code that handles the database updates:

- Add this trigger code for the **Save** button **btnSave**. It defines a second buffer for each of the updateable temp-tables, and a variable to hold the result of a buffer compare:

```

DO:
  DEFINE BUFFER b0ldOrder FOR ttOrder.
  DEFINE BUFFER b0ld0line FOR tt0line.

  DEFINE VARIABLE cCompare AS CHARACTER NO-UNDO.

```

The procedure hasn't saved changes for the **Order** into the temp-table record from the screen buffer yet.

2. Create a temp-table record to hold the updates and then assign all the screen fields, saving the original version in the separate buffer b0ldOrder:

```
/* Create an Update record in the Order temp-table for any changes. */
FIND b0ldOrder.
CREATE ttOrder.
BUFFER-COPY b0ldOrder TO ttOrder.
ASSIGN ttOrder.PromiseDate ttOrder.ShipDate ttOrder.PO
      ttOrder.OrderStatus ttOrder.TransType = "U".
```

3. Include the following code to check whether any fields were actually changed by comparing the two records. If there are no changes, it deletes the before image as a signal to the SAVE procedure:

```
/* Check to see if anything was changed, and if not, then delete the
   before image record. */
BUFFER-COMPARE ttOrder EXCEPT TransType TO b0ldOrder SAVE cCompare.
IF cCompare = "" THEN
  DELETE b0ldOrder.
```

The code on the **Fetch** button creates an update record for every **OrderLine**.

4. Add code to check which records were actually updated and delete the before image for those records that weren't. This code tells the save procedure which records changed and allows the window to browse all the **OrderLines** by selecting those marked with a "U":

```
/* For every OrderLine, compare the original and Update records to see
   if anything was in fact changed; if not, delete the original before
   saving. */
FOR EACH b0ld0line WHERE b0ld0line.TransType = "U":
  FIND tt0line WHERE tt0line.OrderNum = b0ld0line.OrderNum AND
    tt0line.LineNum = b0ld0line.LineNum AND tt0line.TransType = "".
  BUFFER-COMPARE b0ld0line EXCEPT TransType TO tt0line SAVE cCompare.
  IF cCompare = "" THEN /* If there were no changes, */
    DELETE tt0line. /* delete the unchanged version of the rec. */
  END.
```

5. Run a saveOrder procedure to return the changes. The **Order** and **OrderLine** tables are passed as INPUT-OUTPUT parameters to allow the logic procedure to return either changes made by another user, if the update is rejected for that reason, or the final versions of all the records, in case they are further changed by update logic:

```
RUN saveOrder IN hLogic  
    (INPUT-OUTPUT TABLE ttOrder,  
     INPUT-OUTPUT TABLE ttOLine).
```

6. Add code so that the RETURN-VALUE indicates the **Order** was changed out from under you. The new values are displayed:

```
IF RETURN-VALUE = "Changed Order" THEN  
DO:  
    MESSAGE "The Order has been changed by another user.".  
    FIND ttOrder WHERE ttOrder.TransType = "U".  
    DISPLAY ttOrder.OrderDate ttOrder.PromiseDate ttOrder.Shipdate  
          ttOrder.PO ttOrder.OrderStatus WITH FRAME OrderFrame.  
    RETURN NO-APPLY.  
END.
```

7. Add code that, if any **OrderLines** changed, returns and displays the updates made by another user. Otherwise, it reopens the browse query to display the final versions of all the **OrderLines**, including any changes made in the logic procedure. Those changes are all in the temp-table record versions marked with a "U":

```
ELSE IF RETURN-VALUE = "Changed Oline" THEN  
MESSAGE "One or more OrderLines have been changed by another user.".  
  
OPEN QUERY OlineBrowse FOR EACH ttOLine WHERE ttOLine.TransType = "U".  
END. /* END trigger block */
```



To create another internal procedure called **saveOrder** in the logic procedure **h-OrderLogic.p**:

1. In the save procedure, define the INPUT-OUTPUT parameters for the two updated tables:

```
/*
-----+
Procedure saveOrder:
Purpose:    Accepts updates to an Order and its related records and
            saves them to the database, returning any field values
            calculated during the update process.
-----*/
DEFINE INPUT-OUTPUT  PARAMETER TABLE FOR ttOrder.
DEFINE INPUT-OUTPUT  PARAMETER TABLE FOR ttOline.
```

2. Define a second buffer for each of them to allow original values to be compared with the database:

```
DEFINE BUFFER bUpdateOrder FOR ttOrder.
DEFINE BUFFER bUpdateOline FOR ttOline.
```

3. Define explicit buffers for the database tables to make sure that no record is inadvertently scoped to a higher level in the procedure (which is always a good idea in writing code that does updates):

```
DEFINE BUFFER Order FOR Order.
DEFINE BUFFER OrderLine FOR OrderLine.
```

4. Add a character variable that holds a record of any differences found by a buffer compare:

```
DEFINE VARIABLE cCompare AS CHARACTER  NO-UNDO.
```

5. Open a transaction block so that all the comparisons with the original database records and all the update are made together. Then try to find the before image of the **Order** temp-table record. If it's there, then the **Order** was changed. Next, find the corresponding database record using the table's unique primary key. You do this with an EXCLUSIVE-LOCK because you'll later update this database record if no one else has changed it. This also assures that no one else can change it after you first read it. You compare the two and reject the update if the record has been changed by someone else:

```
DO TRANSACTION ON ERROR UNDO, LEAVE:  
  FIND ttOrder WHERE ttOrder.TransType = "" NO-ERROR.  
  IF AVAILABLE (ttOrder) THEN  
    /* If this rec is there then the Order was updated on the client. */  
    DO:  
      FIND Order WHERE Order.OrderNum = ttOrder.OrderNum EXCLUSIVE-LOCK.  
      BUFFER-COMPARE ttOrder TO Order SAVE cCompare.  
      IF cCompare NE "" THEN  
        DO: /* Somebody else has changed the record since we read it. */  
          BUFFER-COPY Order TO ttOrder. /* Return the changes. */  
          RETURN "Changed Order".  
        END. /* END DO IF cCompare Not "" */
```

6. If it hasn't been changed, then you find the **Update** version of the **ttOrder** and copy your changes from there to the database record:

```
ELSE DO: /* FIND the updated tt rec and save the changes. */  
  FIND bUpdateOrder WHERE bUpdateOrder.TransType = "U".  
  BUFFER-COPY bUpdateOrder TO Order. /* Save our changes. */  
END. /* END ELSE DO */  
END. /* END OF AVAILABLE ttOrder */
```

7. Do the same for any changed **OrderLines**. For each before-image record, which you read into the `tt0line` buffer, find the changed version in the `bUpdate0line` buffer, find the corresponding database record, compare the before version with the database, and reject the update if someone else has changed it:

```
/* For each OrderLine that has a before-image (unchanged) record,
   make sure it hasn't been changed by another user. */
FOR EACH tt0line WHERE tt0line.TransType = "":
   /* Bring the updated version into the other buffer. */
   FIND bUpdate0line WHERE bUpdate0line.TransType = "U" AND
      bUpdate0line.OrderNum = tt0line.OrderNum AND
      bUpdate0line.LineNum = tt0line.LineNum.
   FIND OrderLine WHERE OrderLine.OrderNum = tt0line.OrderNum AND
      OrderLine.LineNum = tt0line.LineNum EXCLUSIVE-LOCK.
   BUFFER-COMPARE tt0line TO OrderLine SAVE cCompare.
   IF cCompare NE "" THEN
      DO: /* Somebody else has changed the record since we read it.
         Copy the changes to the Update version to display on the client. */
         BUFFER-COPY OrderLine TO bUpdate0line. /* Return the changes. */
         RETURN "Changed Oline".
   END. /* END DO IF cCompare NE "" */
```

8. Otherwise, apply your changes to the database:

```
ELSE DO: /* Save our OrderLine changes. */
   BUFFER-COPY bUpdate0line TO OrderLine.
```

There's some additional code here that needs explanation. You release the **OrderLine**, which forces it to be written immediately to the database without waiting for the iteration of the `FOR EACH tt0line` block within the transaction. As the record is written out, any database trigger procedures for the table execute. Trigger procedures let you execute standard update logic when a database record is modified, created, or deleted, so that it is always run no matter where the update occurs within your application. You learn about how to write and use trigger procedures in the “[Defining database triggers](#)” section on page 16–22. The trigger is a kind of side effect to the update.

9. To see its effects, you need to re-read the record after the RELEASE forces the trigger to fire and then bring any changes the trigger made back into the temp-table, where it can be returned to the client and displayed:

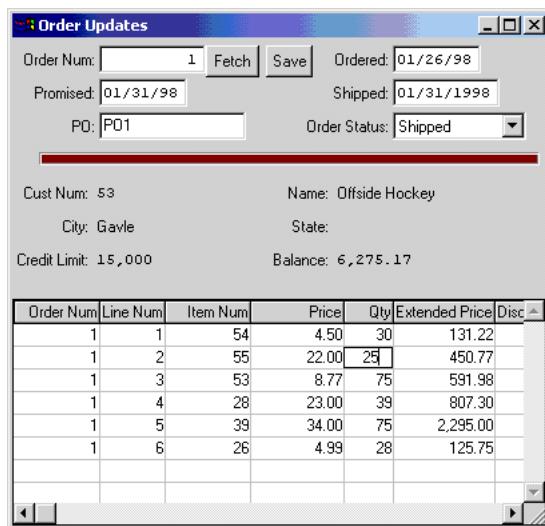
```
RELEASE OrderLine.  
/* Re-find the db record to capture any changes made by a trigger. */  
FIND OrderLine WHERE OrderLine.OrderNum = bUpdateOline.OrderNum AND  
    OrderLine.LineNum = bUpdateOline.LineNum NO-LOCK.  
BUFFER-COPY OrderLine TO bUpdateOline.  
END.      /* END ELSE DO If we updated the OrderLine */  
END.      /* END DO FOR EACH ttOline */  
END.      /* END DO Transaction */  
  
END PROCEDURE.
```



To test your logic procedure:

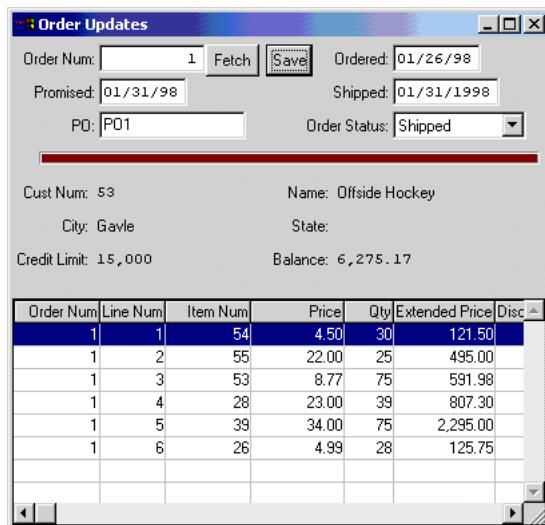
1. Save h-OrderLogic.p.
2. Rerun the **OrderUpdate** window.
3. Select **Order 124** again (or any other **Order** you like), then choose **Fetch**.
4. Make a change to one or more of the **Order** fields. For example, enter a purchase order number (**PO**).

5. Change the Price or Qty for some of the OrderLines:



6. Choose the Save button.

Notice that the **Extended Price** field (which is not enabled for input) changes for any **OrderLine** whose **Price** or **Qty** you changed:



This is the effect of the database trigger procedure for the **WRITE** event on the **OrderLine** table. Re-reading the changed record into the temp-table, sending it back to the client window, and reopening the browse query displayed those changes. Later, you'll learn how to write these trigger procedures yourself.

Defining database triggers

There are certain basic operations that you would always like to have occur when an application updates a record in a database table, regardless of where in a procedure the update occurs. In this way, you can be assured that code that acts as a validation check on an attempted update or a side effect of an update is executed consistently, without having to remember to add the code to every place in the application that changes that table. You can create special Progress 4GL procedures, called *database trigger procedures*, that contain this common code. You can register them in the **Data Dictionary** so that Progress knows to always run them when a corresponding database event occurs. They are also called *schema trigger procedures* because their names are registered in the database schema. It's important to understand that the procedures themselves are not actually stored in the schema.

For the most part, you should use these triggers for basic referential integrity checks to maintain the consistency and integrity of your application data. For example, the **CustNum** field in the **Order** table acts as a foreign key pointing to a **Customer** record with the same **CustNum** value that is the parent record for the **Order**. You would normally never want a user to save an **Order** (to the database) that didn't have a valid **Customer Number**. Your application code itself should enforce this, but you might want to have a check at a lower level so that you can be certain it always executes.

A trigger can also change other database records or other field values in the updated record as a side effect of a change. For example, you might want to delete all **OrderLines** for an **Order** if the user deletes the **Order**. Alternatively, you might want to check whether there are any outstanding **OrderLines** before *allowing* the user to delete the **Order**. Or you might want to calculate values for other fields, in the same table or in other tables, when one field value changes. The **ExtendedPrice** field that you saw updated in the **OrderLine** table is an example of this kind of trigger.

Database trigger guidelines

You can write just about any 4GL code you want into a trigger procedure, but there are a few guidelines that will help you use them widely and effectively. You must decide what the appropriate practice will be for your own application. These are recommendations that are generally useful:

- **Use trigger procedures sparingly, for truly global and non-changing integrity checks.** Resist the temptation to write real business logic into a trigger procedure. Basic integrity constraints are themselves a part of your business logic, of course, but you can think of business logic in this sense as being rules that are complex, subject to change, different for different customers or user groups, or otherwise difficult to pin down precisely.
- **Always remember that trigger procedures execute on the server-side of a distributed application.** Never put messages into a trigger procedure, for example, because they are not seen on the client. A trigger procedure should never have any user interface or contain any statements that could possibly block the flow of the application, such as a statement that requests a value.
- **Remember that trigger procedures operate in the background of an application.** That is, you don't see the trigger code itself when you are looking through the procedures that make up your application, so it is important that they not have surprising or strange effects. If a trigger verifies a **Customer Number** in an **Order** record in a consistent way, developers come to see this as a welcome check and understand why it is there and what it is doing. If you put complex or obscure code into a trigger procedure, you might confuse developers who cannot understand why the application procedures they are coding or looking at are executing in a strange way.
- **Return errors in a standard way from all your triggers.** If a trigger procedure does an integrity check, it must be able to reject the record update that fired it. Without being able to display a message, your procedure must generate the error in a way that application code can deal with it consistently. One method is to RETURN ERROR with a message that becomes the RETURN-VALUE of the trigger and code your application to be prepared to handle errors of this kind, by taking the RETURN-VALUE and turning it into a standard message box on the client, for example.

- **Write your applications so that errors from triggers are as unlikely as possible.** You should use integrity procedures as a last defense for your application to make sure that casually written procedures don't compromise your database. The heart of your application logic should enforce all integrity at a level that is visible to the application. Where appropriate, you can take specific actions when errors occur, and when updates change other database values that the user might need to see. For example, the user interface for your **Order Entry** screen probably should present the user with a lookup list of some sort to choose a **Customer** from, where the **Customer Name** or other identifying information verifies that the **Customer Number** is correct. If you do this, then it is very unlikely that an invalid **CustNum** will find its way back to an actual update to be detected and rejected by a trigger procedure.

Database events

There are five database events you can associate with a trigger procedure:

- **CREATE** — Executes when Progress executes a CREATE or INSERT statement for a database table, after the new database record is created. You can use the procedure to assign initial values to some of the table's fields, such as a unique key value.
- **DELETE** — Executes when Progress executes a DELETE statement for a database table, before the record is actually deleted. You can use the procedure to check for other related records that would prevent deletion of the current one, to delete those related records if you wish, or to adjust values in other records in other tables to reflect the delete.
- **WRITE** — Executes when Progress changes the contents of a database record. More specifically, it occurs when a record is released, normally at the end of a transaction block, or when it is validated. This book recommends against using the field validation expressions that the Data Dictionary allows you to define because these have the effect of mixing validation logic with user interface procedures. The **WRITE** trigger happens in conjunction with those validations, if they exist, when the record is released or you execute an explicit **VALIDATE** statement. The **WRITE** trigger can replace those kinds of validations without combining validation with the UI.
- **ASSIGN** — Monitors a single field rather than an entire database record, so you can use it to write field-level checks. An **ASSIGN** trigger executes at the end of a statement that assigns a new value to the field, after any necessary re-indexing. If a single **ASSIGN** statement (or **UPDATE** statement, but you know *not* to use that anymore) contains several field assignments, Progress fires all applicable **ASSIGN** triggers at the end of the statement. If any trigger fails, Progress undoes all the assignments in the statement.

- **FIND** — Executes when Progress reads a record using a FIND or GET statement or a FOR EACH loop. A FIND trigger on a record executes only if the record first satisfies the full search condition on the table, as specified in the WHERE clause. FIND triggers do not fire in response to the CAN-FIND function. If a FIND trigger fails, Progress behaves as though the record has not met the search criteria. If the FIND is within a FOR EACH block, Progress simply proceeds to the next iteration of the block. Generally, you should not use FIND triggers. They are expensive—consider that you are executing a compiled procedure for every single record read from that table anywhere in your application. Also, they are typically used for security to provide a base level of filtering of records that the user should never see. For various reasons, including the fact that a CAN-FIND function does not execute the FIND trigger, this security mechanism is not terribly reliable. You are better off building a general-purpose filtering mechanism into your application architecture in a way that is appropriate for your application, rather than relying on the FIND trigger to enforce it.

There are also trigger events to support replication of data, so that any change to a database can be copied to another database. These REPLICATION-CREATE, REPLICATION-DELETE, and REPLICATION-WRITE events are described in *OpenEdge Development: Progress 4GL Reference*.

Trigger procedure headers

A trigger procedure must be an external 4GL procedure. That is, it must be a procedure file of its own, not an internal procedure within some larger procedure file. It can contain just about any 4GL code. It is identified by a special header statement at the top of the procedure file.

CREATE, DELETE, and FIND headers

The header statement for a CREATE, DELETE, or FIND procedure has this syntax:

```
TRIGGER PROCEDURE FOR { FIND | CREATE | DELETE } OF table-name.
```

This statement effectively defines a buffer automatically with the same name as the table, scoped to the trigger procedure.

WRITE header

The header statement for a WRITE trigger has this syntax:

```
TRIGGER PROCEDURE FOR WRITE OF table-name
[ NEW [ BUFFER ] new-buffer-name ]
[ OLD [ BUFFER ] old-buffer-name ].
```

When executing a WRITE trigger, Progress makes two record buffers available to the trigger procedure. The NEW buffer contains the modified record that is being validated. The OLD buffer contains the most recent version of the record before the latest set of changes was made. This is the template record that holds initial values for a table if it is a newly-created record, the record from the database if the record has not been validated or the most recently validated record if it has been validated. The default for the new buffer is the automatically-created buffer named for the table itself, which makes the NEW phrase optional. If you wish to compare the new buffer to the old, you must use the OLD phrase to give the old one a name. The BUFFER keyword is just optional syntactic filler.

You can make changes to the NEW buffer, but the OLD buffer is read-only.

You can determine whether the record being written is newly created using the Progress NEW function, which returns true if Progress has not written the record to the database before, and false otherwise:

```
NEW table-name
```

For example:

```
IF NEW Customer THEN . . .
```

ASSIGN header

The header statement for an ASSIGN trigger has this syntax:

```
TRIGGER PROCEDURE FOR ASSIGN
{ OF table.field }
| NEW [VALUE] new-field { AS data-type | LIKE other-field }
[ OLD [VALUE] old-field { AS data-type | LIKE other-field2 } ] .
```

If you use the OF form, the expression *table.field* identifies the field, but you can in fact refer to any field in the record where the field has been changed.

If you need to compare the field value before and after it was changed, you must use the NEW and OLD phrases to give those versions of the field names. If you do this, you cannot refer to the rest of the record buffer. You can change the NEW field, and this changes the field value in the record, but changing the OLD field value has no effect. The VALUE keyword here is just optional syntactic filler.

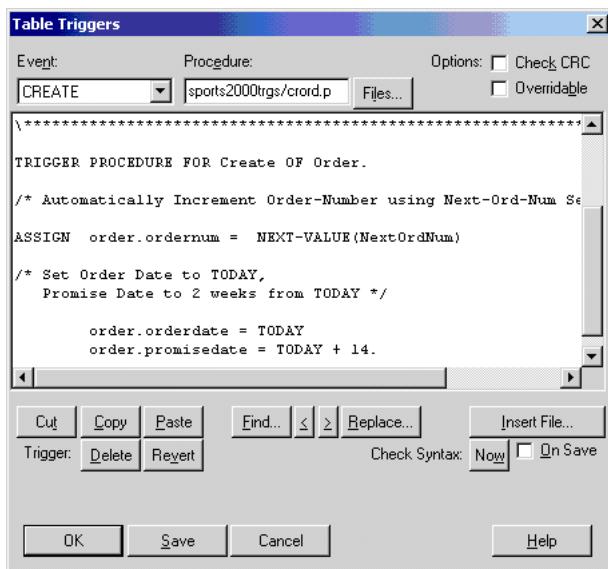
Accessing and defining triggers

You can create a trigger procedure in the Procedure Editor, but you attach it to your database in the Data Dictionary, which provides an editor designed to let you code your triggers there as well.

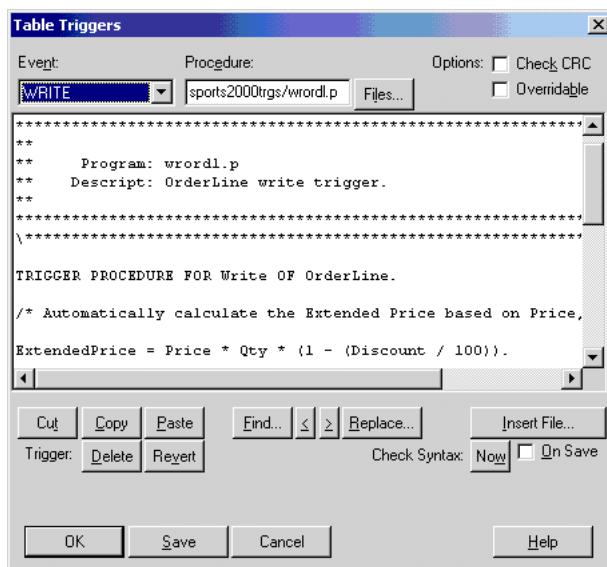


To access triggers:

1. From the AppBuilder menu, select **Tools**→**Data Dictionary** and then select a database table such as **OrderLine**.
2. Choose the **Triggers** button in the **Table Properties** dialog box. The **Table Triggers** dialog box appears:



3. To see the **WRITE** trigger that calculated the **Extended Price** for the **OrderLine** when it changed, select **WRITE** from the **Event** drop-down list:



When you create a new trigger procedure in the Data Dictionary, it supplies the header statement for you. Then you simply write the rest of the code for the procedure. You must give the procedure a name and specify a pathname either in the **Procedure** field or by selecting the **Files** button. It is this name that the Data Dictionary associates with the procedure so that Progress executes it at the right times. As you can see, the trigger procedures for the Sports2000 database are located in the directory sports2000trgs under the install directory for OpenEdge. The write trigger for the **OrderLine** table is called **wrordl.p**. You can use any naming convention you want for your trigger procedures.

You can choose the **Help** button to access descriptions of all the other buttons here. This section mentions just the two **Options** that are shown as toggle boxes in the corner of the dialog box.

A trigger procedure provides a certain measure of security that a validation check, or an effect elsewhere in the database, occurs reliably whenever a certain type of update occurs. If you wish to protect yourself against a trigger procedure being replaced by another procedure that doesn't do the same job, you can check on the **Check CRC** toggle box. If this option is on, then Progress stores in the metaschema, along with the trigger procedure name, a unique Cyclic Redundancy Check (CRC) identifier for the *compiled* version of the trigger procedure. Progress raises an error if the r-code file it encounters at run time doesn't match or if there is no compiled version of the procedure.

You can check on the other toggle box, **Overridable**, if you want to let the trigger procedure be overridable by a trigger local to a specific application procedure, called a *session trigger*.

Session triggers are discussed briefly in the “[Session triggers](#)” section on page 16–33. Session triggers allow you to provide the effects of a trigger but without making it global to the entire application. Among other things, they let you override a trigger procedure with behavior more appropriate to a particular application module. If you don’t check on the **Overridable** toggle box, then Progress raises an error if a session trigger executes that tries to override the behavior of this trigger procedure.

Having said all this, look at the statement in `wrord1.p` that calculates the **Extended Price**:

```
ExtendedPrice = Price * Qty * (1 - (Discount / 100)).
```

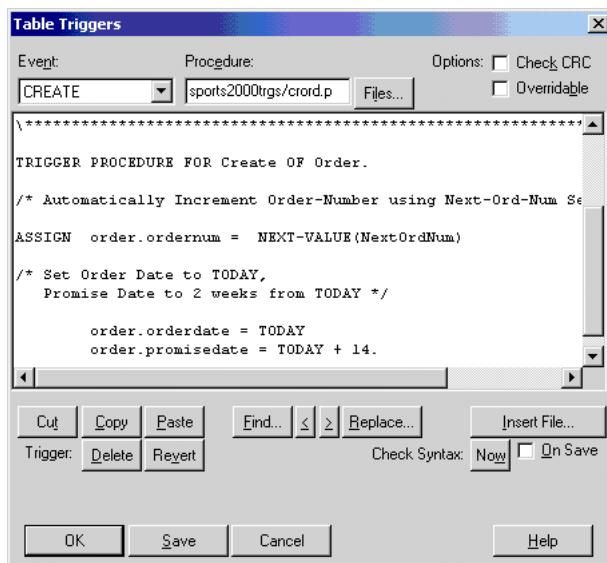
Is this a good use for a trigger? Probably not, because it definitely violates the guideline that trigger procedures shouldn’t contain real business logic. In any real application, this kind of price calculation is complex and variable, depending on any number of factors. It’s probably better to provide access to the price calculation algorithm in the application module that controls **OrderLine** maintenance and to make sure that your application is put together in such a way that the code is always executed when it needs to be. Burying the code in a trigger is not a good thing. Generally, the Sports2000 trigger procedures can serve as examples of how to write triggers, but are often not *good* examples. This is partly because the database is simplified in ways that are not always realistic and, partly because many of these example procedures predate the architecture for distributed applications and other features that have changed the way you build applications. (Many contain MESSAGE statements, for example, which is definitely a bad idea.)

Using database sequences in CREATE triggers

This section describes database sequences in CREATE triggers.

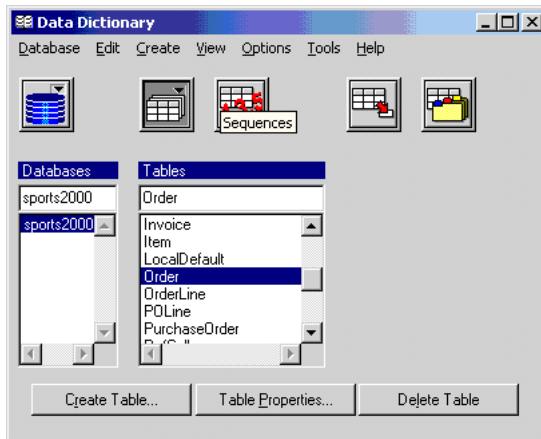
- To see one of the basic uses for triggers, look at an example trigger procedure:

1. Cancel out of the **Table Triggers** dialog box and select the **Order** table.
2. Choose the **Triggers** button in its **Table Properties** dialog box. The **Table Triggers** dialog box appears again:



To assign each **Order** a unique number, the procedure uses a **Database Sequence** that stores the latest value assigned to an **Order**.

3. Define sequences in the Data Dictionary by choosing the **Sequences** button in the **Data Dictionary** main window:



The online help tells you more about how to create sequences in your own database. Basically, each sequence is an Integer value maintained for you in the database. When you define it, you simply give the sequence a name, a starting value, a maximum value, and a value to increment by. Then, each time Progress encounters the NEXT-VALUE function in your application, it increments the sequence and returns the new value, as in the example from this trigger:

```
ASSIGN order.ordernum = NEXT-VALUE(NextOrdNum)
```

You can also use the CURRENT-VALUE function to access the current sequence value without incrementing it. Also, there is a CURRENT-VALUE statement that allows you to assign a new value to the sequence:

```
CURRENT-VALUE ( sequence-name ) = integer-expression.
```

You should use the CURRENT-VALUE statement only to reset a sequence in a database that is not being actively used. You should never put such a statement in your application code to assign individual sequence values, as this is not reliable in a multi-user environment.

Both NEXT-VALUE and CURRENT-VALUE allow you to specify a logical database name as an optional second argument if the sequence name might not be unique among all your connected databases.

Assigning a unique Integer key to a new record is the most common use of CREATE triggers.

You can also use a CREATE trigger to assign other initial values that need to be expressions that can't be represented in the Data Dictionary when you define fields. This trigger assigns the value of the built-in function TODAY to the **OrderDate**, and TODAY + 14 as the default **Promised Date**:

```
order.orderdate = TODAY  
order.promisedate = TODAY + 14
```

Session triggers

In addition to defining schema trigger procedures that are always executed when an operation occurs on a table, you can also define trigger blocks within your application that act on these events, much as you can define triggers for user interface events. These triggers are not of great utility, but there may be circumstances where you need the same block of code to execute regardless of which of a number of different update or find statements against a table are executed in a portion of your application.

A session trigger is in scope while the procedure that defines it is running. The code in the trigger executes in the context of the procedure that defines it, regardless of where the event occurs that fires the trigger. Therefore, it can access local variables and other procedure objects not available to the procedure where the event occurs.

The syntax for session triggers is modeled on the syntax for user interface events:

```
ON event OF object [ reference-clause ] [ OVERRIDE ]  
{ trigger-clock | REVERT }
```

The *event* can be CREATE, WRITE, DELETE, FIND, or ASSIGN, as for schema triggers.

The *object* is a database table name in the case of CREATE, DELETE, FIND, and WRITE triggers, or a database field qualified by its table name for the ASSIGN trigger.

The *reference-clause* applies only to the WRITE and ASSIGN triggers. For the WRITE trigger it is:

```
[ NEW [BUFFER] new-buffer-name ] [ OLD [BUFFER] old-buffer-name ]
```

For the ASSIGN trigger it is:

```
OLD [VALUE] old-value-name [ COLUMN-LABEL label | FORMAT format |  
INITIAL constant | LABEL string | NO-UNDO ]
```

If you include the OVERRIDE option, then this trigger block executes in place of any schema trigger for the same event and object. This is allowed only if you checked on the **Overridable** toggle box when you defined the schema trigger in the Data Dictionary. Otherwise, an error results when the trigger block is executed.

The *trigger-block* is a single statement or DO END block just as for user interface triggers. The trigger block executes as would a call to an internal procedure in the same place in the code.

If Progress encounters an additional session trigger for an event and object when a trigger is still in scope for that combination, the new one replaces the old one for the duration of the new trigger's scope. Alternatively, if you use the REVERT option in a session trigger block, the current session trigger for the event and object is cancelled and the previously executed session trigger definition is reactivated. If there is no other session trigger on the stack, then the session trigger is deactivated altogether.

General trigger considerations

In general, a schema trigger procedure or session trigger executes within the larger context of the procedure containing the statement that causes the trigger to fire, just as if the procedure or block of code had been run following the triggering statement. Keep the following considerations in mind about how triggers operate:

- Progress does not allow database triggers on events for metaschema tables and fields (which have an initial underscore in their names and a negative internal file number in the schema).
- You cannot delete a record in a buffer passed to a database trigger or change the current record in the buffer with a statement such as FIND NEXT or FIND PREV.

- An action within one trigger procedure can execute another trigger procedure. For example, if a trigger assigns a value to a field and you have also defined an ASSIGN trigger for that field, the ASSIGN trigger executes. You must take care that this does not result in either unwanted conflicts between the actions of the triggers or a possible loop.
- By their nature, triggers are executed within transactions (except possibly for a FIND trigger). Whatever action is encoded in the trigger becomes part of the larger transaction.
- For all blocks in a database trigger, the default ON ERROR handling is ON ERROR UNDO, RETURN ERROR, rather than the usual Progress default of ON ERROR UNDO, RETRY. You learn more about the ON ERROR phrase in the next chapter.
- You can store collections of precompiled Progress procedures in a single file called a *procedure library*. If you collect together your application's schema triggers into a procedure library, you can use the –trig startup parameter to identify either the name of the procedure library for triggers or the operating system directory where they reside. See *OpenEdge Deployment: Startup Command and Parameter Reference* for more information.
- When you dump and load database records, you might want to disable the schema triggers of the database, both to avoid the overhead of the triggers and to deal with the likely possibility that integrity constraints enforced by the triggers might not be satisfied until the database load is complete. See *OpenEdge Development: Basic Database Tools* for more information on disabling triggers.
- For information on how SQL access to your database interacts with 4GL schema trigger procedures, see *OpenEdge Data Management: SQL Development*.

In the following chapter about transactions, you'll learn how Progress uses locks to control multi-user access to data, and what the scope of those locks is relative to the transactions they're used in.

Managing Transactions

This chapter continues the discussion of database transactions. It includes the following sections:

- [Controlling the size of a transaction](#)
- [Using the UNDO statement](#)

Controlling the size of a transaction

You've already learned which statements start a transaction automatically. To summarize, these are:

- FOR EACH blocks that directly update the database.
- REPEAT blocks that directly update the database.
- Procedure blocks that directly update the database.
- DO blocks with the ON ERROR or ON ENDKEY qualifiers (which you'll learn more about later) that contain statements that update the database.

You can also control the size of a transaction by adding the TRANSACTION keyword to a DO, FOR EACH, or REPEAT block. This can force a transaction to be larger, but because of the statements that start a transaction automatically, you cannot use it to make a transaction smaller than Progress would otherwise make it.

Take another look at the update procedure `saveOrder`, in the sample logic procedure `orderlogic.p`, to see how transaction blocks are affected by changes to the procedure.

As written, there is a DO TRANSACTION block around the whole update of both the **Order** and any modified **OrderLines**:

```
DO TRANSACTION ON ERROR UNDO, LEAVE:  
  DO:  
    /* Order update block */  
  END.  
  FOR EACH tt0line:  
    /* OrderLine update block */  
  END.  
END. /* END Transaction block. */
```

The update of the **Order** and its **OrderLines** happens as a single transaction. If any errors are encountered in any of the updates, the entire transaction is backed out.

- To verify this, you can generate a listing file as you've done earlier:



When you run this **COMPILE** statement, your listing file tells you, among other things, where all the transactions begin and end. This is very valuable information. You should always use a listing file in any complex procedure to make sure that your record scope and transaction scope are as you intended.

Taking a look at this listing file, you can see that the **DO TRANSACTION** block is a top-level block within its procedure, marked with a 1. The **DO** block inside it, where the **Order** is actually updated, is block level 2:

```

144 1    DO TRANSACTION ON ERROR UNDO, LEAVE:
145 1        FIND ttOrder WHERE ttOrder.TransType = "" NO-ERROR.
146 1        IF AVAILABLE (ttOrder) THEN
            /* If this rec is there then the Order was updated on the client. */
147 1
148 2    DO:

```

Further down, you can see that the **FOR EACH** block that operates on the **OrderLines** is also a nested block, at level 2 within the main **DO TRANSACTION** block:

```

164 2    FOR EACH tt0line WHERE tt0line.TransType = "":
165 2        /* Bring the updated version into the other buffer. */

```

Now if you look at the end of the file, you see a summary of all the blocks. Here's an excerpt from that part of the listing. It shows that the procedure blocks for the internal procedures **fetchOrder** and **saveOrder** are *not* transaction blocks:

| File Name | Line | Blk. | Type | Tran | Blk. | Label |
|----------------------|------|-----------|------|-----------|------------|-------|
| ...ter8\orderlogic.p | 82 | Procedure | No | Procedure | fetchOrder | |
| ...ter8\orderlogic.p | 111 | For | No | | | |
| ...ter8\orderlogic.p | 124 | Procedure | No | Procedure | saveOrder | |

Buffers: bUpdateOrder
 sports2000.Order

This is a good thing. You *never* want your transactions to default to the level of a procedure, because they are likely to be larger than you want them to be. This means that record locks are held longer than necessary and that more records might be involved in a transaction than you intended.

Next you see that Progress identifies the first DO block at line 144 as a transaction block. This is because it has an explicit TRANSACTION qualifier on it. The nested DO block two lines later is not a transaction block because a DO block by itself does not mark a transaction.

The FOR EACH block at line 164 is also marked as a transaction block:

```
...ter8\orderlogic.p 144 Do      Yes
...ter8\orderlogic.p 146 Do      No
...ter8\orderlogic.p 151 Do      No
...ter8\orderlogic.p 156 Do      No
...ter8\orderlogic.p    164 For    Yes
    Buffers: sports2000.OrderLine
        bUpdateOline
```

What does this mean? Is this really a separate transaction? The answer is no, because the FOR EACH block is nested inside the larger DO TRANSACTION block. This code tells you that the FOR EACH block would be a transaction block (because this is the nature of FOR EACH blocks that perform updates). However, because it is nested inside a larger transaction, it becomes a *subtransaction*. Progress can back out changes made in a subtransaction within a larger transaction when an error occurs, and you can also do this yourself, as you'll learn a little later in the “[Subtransactions](#)” section on page 17–23.

Making a transaction larger

In this section, you experiment with increasing the size of a transaction.



To see the effect of removing the DO TRANSACTION block from saveOrder:

1. Comment out the DO TRANSACTION statement and the matching END statement at the end of the procedure.
2. Recompile and generate a new listing file.

3. Take a look at the final section. You can see that, without the DO TRANSACTION block, the entire saveOrder procedure become a transaction block:

| File Name | Line Blk. | Type Tran | Blk. Label |
|----------------------|-----------|-------------------------------|----------------------|
| ...ter8\orderlogic.p | 82 | Procedure No | Procedure fetchOrder |
| ...ter8\orderlogic.p | 111 | For | No |
| ...ter8\orderlogic.p | 124 | Procedure Yes | Procedure saveOrder |
| | | Buffers: bUpdateOrder | |
| | | sports2000.Order | |
| ...ter8\orderlogic.p | 146 | Do | No |
| ...ter8\orderlogic.p | 151 | Do | No |
| ...ter8\orderlogic.p | 156 | Do | No |
| ...ter8\orderlogic.p | 164 | For | Yes |
| | | Buffers: sports2000.OrderLine | |
| | | bUpdateOline | |

Why did this happen? A DO block by itself, without a TRANSACTION or ON ERROR qualifier, does not start a transaction. Therefore, Progress has to fall back on the rule that the entire procedure becomes the transaction block. In this particular case, this does not really make a big difference because the update code for **Order** and **OrderLine** is essentially the only thing in the procedure. But, as emphasized before, this is a very dangerous practice and you should always avoid it. If you generate a listing file and see that a procedure is a transaction block, you need an explicit transaction within your code. In fact, you should *always* have explicit transaction blocks in your code and then verify that statements outside that block are not forcing the transaction to be larger than you intend.

Making a transaction smaller

Try one more experiment.

- To decrease the size of a transaction, uncomment the DO TRANSACTION statement and its matching END statement. Move the END statement up to just after the end of the DO block for the **Order** record. Now your procedure structure looks like this:

```
DO TRANSACTION ON ERROR UNDO, LEAVE:  
  DO:  
    /* Order update block */  
  END.  
END.      /* END Transaction block. - This used to be at the end of the proc. */  
  
FOR EACH tt0line:  
  /* OrderLine update block */  
END.
```

How does this affect your transactions? Now that you removed the FOR EACH block from the larger transaction, it becomes a true transaction block of its own at block level 1, as the new listing file shows:

```
166   1   FOR EACH tt0line WHERE tt0line.TransType = "":  
167   1   /* Bring the updated version into the other buffer. */
```

This means that the two transaction blocks identified by the listing are now separate and distinct transactions:

| File Name | Line | Blk. | Type | Tran | Blk. | Label |
|----------------------|------|-----------|----------------------|-----------|------------|-------|
| ...ter8\orderlogic.p | 82 | Procedure | No | Procedure | fetchOrder | |
| ...ter8\orderlogic.p | 111 | For | No | | | |
| ...ter8\orderlogic.p | 124 | Procedure | No | Procedure | saveOrder | |
| | | Buffers: | bUpdateOrder | | | |
| | | | sports2000.Order | | | |
| ...ter8\orderlogic.p | 144 | Do | | Yes | | |
| ...ter8\orderlogic.p | 146 | Do | | No | | |
| ...ter8\orderlogic.p | 151 | Do | | No | | |
| ...ter8\orderlogic.p | 156 | Do | | No | | |
| ...ter8\orderlogic.p | 166 | For | | Yes | | |
| | | Buffers: | sports2000.OrderLine | | | |
| | | | BUpdateOline | | | |

If Progress encounters an error in the update of the **Order**, it leaves that block and continues on with the **OrderLines**. If the **OrderLines** update succeeds, then the newly modified **OrderLines** are in the database, but the failed **Order** update is not. Likewise, if the **Order** block succeeds but there is an error in the **OrderLines** block, then the **Order** update is in the database but the **OrderLines** update is not. You must decide when you put your procedures together how large your updates need to be to maintain your data integrity. In general, you should work to keep your transactions as small as possible so that you do not lock more records or lock records for longer periods of time than is absolutely necessary. But your transactions must be large enough so that related changes that must be committed together either all get into the database or are all rejected.

Transactions and trigger and procedure blocks

If your code starts a transaction in one procedure and then calls another procedure, whether internal or external, the entire subprocedure is contained within the transaction that was started before it was called. If a subprocedure starts a transaction, then it must end within that subprocedure as well, because the beginning and end of the transaction are always the beginning and end of a particular block of code.

Since a database trigger procedure is an external procedure called under special circumstances—in response to an update event elsewhere in the application—it follows the same rule. There is always a transaction active when a database trigger is called (except in the case of the FIND trigger), so the trigger procedure is entirely contained within the larger transaction that caused it to be called.

Trigger blocks beginning with the `ON event` phrase are treated the same as an internal procedure call. If there is a transaction active when the block is executed in response to the event, then its code is contained within that transaction.

Checking whether a transaction is active

You can use the built-in `TRANSACTION` function in your procedures to determine whether a transaction is currently active. This logical function returns true if a transaction is active and false otherwise. You might use this, for example, in a subprocedure that is called from multiple places and which needs to react differently depending on whether its caller started a transaction. (When you have a single procedure, you had better not need this function to tell you whether you've started a transaction or not!)

The NO-UNDO keyword on temp-tables and variables

You were instructed very early on in this book to define almost all variables using the NO-UNDO keyword. Also, in this chapter's example, the temp-tables for **Customer**, **Order**, and **OrderLine** are NO-UNDO. Why is this?

When you define variables, Progress allocates what amounts to a record buffer for them, where each variable becomes a field in the buffer. There are in fact *two* such buffers, one for variables whose values can be undone when a transaction is rolled back and one for those that can't. There is extra overhead associated with keeping track of the before-image for each variable that can be undone, and this behavior is rarely needed. If you are modifying a variable inside a transaction block (and it is important for your program logic that sets that variable's value that it be undone if the transaction is undone), then you should define the variable *without* the NO-UNDO keyword.

Here's a trivial example of when this might be useful. This little procedure lets you create and update **Customer** records in a REPEAT loop, and then shows you how many were created:

```
DEFINE VARIABLE iCount AS INTEGER.  
  
REPEAT :  
    CREATE Customer.  
    iCount = iCount + 1.  
    DISPLAY Customer.CustNum WITH FRAME CustFrame 5 DOWN.  
    UPDATE Customer.Name WITH FRAME CustFrame.  
END.  
DISPLAY iCount "records created." WITH NO-LABELS.
```

The REPEAT block defines the scope of a transaction. Each time Progress runs through the block is a separate transaction and, as each iteration completes successfully, the record created in that block is committed to the database. If you run the procedure, the REPEAT loop lets you enter **Customers** until you press **ESCAPE**, which in an OpenEdge session running on MS Windows is mapped to the END-ERROR key label. Each time it goes through the block, Progress creates a **Customer**, displays its new **CustNum** (assigned by the CREATE trigger for **Customer**), and prompts you for a **Name**. It is at this point that you can press **ESCAPE** when you're done entering **Customers**. This undoes and leaves the current iteration of the REPEAT block. Because each iteration of the REPEAT block is a separate transaction, the final **Customer** you created is undone—it's erased from the database.

But what about the `iCount` variable? Since this was defined as undoable (the default), the final change to its value is rolled back along with everything else, and its value is the actual number of records created and committed to the database, as shown in [Figure 17–1](#).

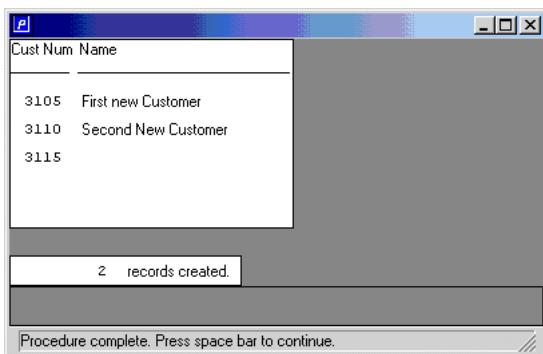


Figure 17–1: Example of creating and updating Customer records

As you enter the REPEAT block for the third time, Progress creates a third new **Customer** and increments `iCount` from 2 to 3. When you press **ESCAPE**, the final iteration of the REPEAT block is undone, and **Customer 3115** is deleted from the database. The value of `iCount` is restored to 2, which was its value before that iteration of the block. (Note that the key value 3115 cannot be undone or reused, however, because it comes from a database sequence, and for performance reasons, these are not under transaction control).

If the variable were defined **NO-UNDO**, then after it is incremented from 2 to 3 its value would not be restored when the final transaction is undone, and the final **DISPLAY** statement would show its value as 3.

Relatively few variables need to be undone in this way, so to maximize performance you should define all other variables as **NO-UNDO**. (The Progress Editor macros do this for you when you type DVI, DVC, etc. into the Editor.) So why isn't **NO-UNDO** the default? Quite simply, it didn't at first occur to the developers of the language that most variables should be defined this way, so the first versions of Progress went out with undo variables as the default. Because of the commitment to maintaining the behavior of existing applications, the default has not changed with new releases.

The same consideration applies to temp-tables. Because temp-tables are really database tables that are not stored in a persistent database, they have almost all of the capabilities of true database tables, including the ability to be written to disk temporarily, the ability to have indexes, and the ability to participate in transactions. As with variables, your temp-tables are more efficient if you define them as **NO-UNDO** so that they are left out of transactions. Consider whenever you define a temp-table whether it really needs to be part of your transactions. If not, include the **NO-UNDO** keyword in its definition.

Using the UNDO statement

Progress undoes a transaction automatically if it detects an error at the database level, for example, because of a unique key violation. In many cases, your application logic also wants to undo a transaction when you detect a violation of your business logic. The UNDO statement lets you control when to cancel the effects of a transaction on your own. It also lets you define just how much of your procedure logic to undo.

Here is the syntax of the UNDO statement:

```
UNDO [ label ] [ , LEAVE [ label ] | , NEXT [ label ] | , RETRY [ label ]  
      | , RETURN [ ERROR | NO-APPLY ] [ return-value ] ]
```

In its simplest form, you can use the UNDO keyword as its own statement. In this case, Progress undoes the innermost containing block with the error property, which can be:

- A FOR block.
- A REPEAT block.
- A procedure block.
- A DO block with the TRANSACTION keyword or ON ERROR phrase.

You can change this default by specifying a block *label* to undo. You can place a block name anywhere in your procedure. The block name must adhere to the same rules as a variable name and it must be followed by a colon. If you use this block name in an UNDO statement, it identifies how far back you want Progress to undo transactional work.

The default action on an UNDO is to attempt to retry the current block. In Progress, a block of code that prompts the user for field values can be retried; that is, having entered invalid data, the user can re-enter different values. If you are writing well-structured procedures that do not mix user interface elements with database logic, then retrying a block never results in a user entering different values. Progress recognizes this and changes the action to a NEXT of an iterating block, or a LEAVE of a noniterating block, so this is effectively the default for transactions not involving user interface events.

You can change this default action as well. If you specify LEAVE, you can name the block to leave. This defaults to the block you undo.

If you specify NEXT within an iterating block, then after the UNDO Progress proceeds to the next iteration of either the block whose label you specify or the block you undo as a default. Use the NEXT option if each iteration of a repeating block is its own transaction (the default) and if you want Progress to keep trying to apply changes to other records, for example, as Progress walks through a set of records in a FOR EACH block.

If you specify RETRY, then you can retry either the current block (the default) or the same block you applied the UNDO statement to. Again, in a properly structured application, you do not need to use the RETRY option.

Finally, you can RETURN out of the current procedure. You can RETURN ERROR, which raises the Progress error condition, or you can use RETURN NO-APPLY to cancel the effect of the last user-interface event. Once again, this is not an option you would normally use in server-side business logic. The RETURN option can also specify a *return-string*, which becomes the RETURN-VALUE in the procedure you return to.

You can also specify UNDO as an option on a DO, FOR, or REPEAT block, as you did in your saveOrder example procedure:

```
DO TRANSACTION ON ERROR UNDO, LEAVE:
```

Using the UNDO statement in the sample procedure

In this section, you see how the UNDO statement can affect the operation of the saveOrder procedure in `orderlogic.p`. You'll try several variations to the business logic, in succession, to illustrate the ways you can control the scope of your transaction and how to react when it fails.

Using UNDO, LEAVE on a block

In this section, you try an example that uses UNDO, LEAVE on a block.



To undo and leave the block that updates the OrderLine table:

1. To make this a transaction block of its own, put the END statement for the DO TRANSACTION block after the **Order** is updated, as you did earlier:

```
    .
    .
    .
ELSE DO: /* FIND the updated tt rec and save the changes. */
    FIND bUpdateOrder WHERE bUpdateOrder.TransType = "U".
    BUFFER-COPY bUpdateOrder TO Order. /* Save our changes. */
END.      /* END ELSE DO */
END.      /* END OF AVAILABLE ttOrder */
/* END DO Transaction */
```

This makes the FOR EACH block that follows a separate transaction.

2. Add a new variable definition in `saveOrder` for an error message to return:

```
DEFINE VARIABLE cMessage AS CHARACTER NO-UNDO.
```

3. Add the highlighted lines, shown below, to the FOR EACH block that updates **OrderLines**:

```

FOR EACH tt0line WHERE tt0line.TransType = "":
    .

    .

/* Find corresponding bUpdate0line */
    FIND OrderLine WHERE OrderLine.OrderNum = tt0line.OrderNum AND
        OrderLine.LineNum = tt0line.LineNum EXCLUSIVE-LOCK.

    .

    BUFFER-COPY bUpdate0line TO OrderLine. /* Save our OrderLine
changes. */
    RELEASE OrderLine.
/* Re-find the db record to capture any changes made by a trigger. */
    FIND OrderLine WHERE OrderLine.OrderNum = bUpdate0line.OrderNum AND
        OrderLine.LineNum = bUpdate0line.LineNum NO-LOCK.
    BUFFER-COPY OrderLine TO bUpdate0line.
    IF bUpdate0line.ExtendedPrice > (tt0line.ExtendedPrice * 1.2) THEN
        DO:
            cMessage = "Line " + STRING(OrderLine.LineNum) +
                ": Can't increase price by more than 20%."
            UNDO, LEAVE.
        END.
    .

    .

END.          /* END DO FOR EACH tt0line */

```

This code checks to make sure that the **ExtendedPrice** for an **OrderLine** is not increased by more than 20%. If this limit is exceeded, then the current iteration of the block is undone and the block is left.

On each iteration, the FOR EACH block makes a **tt0line** record current. Your code uses the second buffer, **bUpdate0line**, to locate the updated version of that **OrderLine** temp-table record. It then finds the **OrderLine** in the database and copies the updates to it. Next it releases the database record to force its WRITE trigger to fire, which recalculates the **ExtendedPrice** field. Then it again finds the database record and copies it back into the **bUpdate0line** buffer to capture the effects of the trigger code, in particular the new **ExtendedPrice** value. Only now can your program logic compare this to the original value in the **tt0line** buffer to see if it exceeds the limit. If it does, then you store off a message, then undo and leave the FOR EACH block.

At this point, following the UNDO statement, the whole database change that you wrote out and then read back in is gone—Progress has restored the database record to its values before the transaction began.

4. Add code so that after leaving the block, you check if you have an error message. If so, your code needs to re-find the **OrderLine** record with its original values, and copy it back into the **bUpdateOline** buffer to return to the client for display. It then returns the message as the return value for the procedure:

```
IF cMessage NE "" THEN
  DO:
    FIND OrderLine WHERE OrderLine.OrderNum = bUpdateOline.OrderNum AND
      OrderLine.LineNum = bUpdateOline.LineNum NO-LOCK.
    BUFFER-COPY OrderLine TO bUpdateOline.
    RETURN cMessage.
  END.
```

Why did you have to re-find the **OrderLine** record from the database? The UNDO released it from its buffer, so it's no longer available after the block ends. Then why did you *not* have to re-find the temp-table record? You defined the temp-table as NO-UNDO so it was left out of normal transaction handling. The temp-table buffer is scoped to the whole procedure, so the record it contains is still available after the block ends. If you had defined the temp-table without NO-UNDO, then the **bUpdateOline** record would have been released, as well as the database **OrderLine** record, and you would have had to re-establish it in the temp-table as well. This is an illustration of the kind of overhead you save by using NO-UNDO temp-tables, and also of the sometimes unintended consequences of having undo capability on a temp-table that doesn't require it.

The simple diagram in [Figure 17–2](#) illustrates the scope of the transactions.

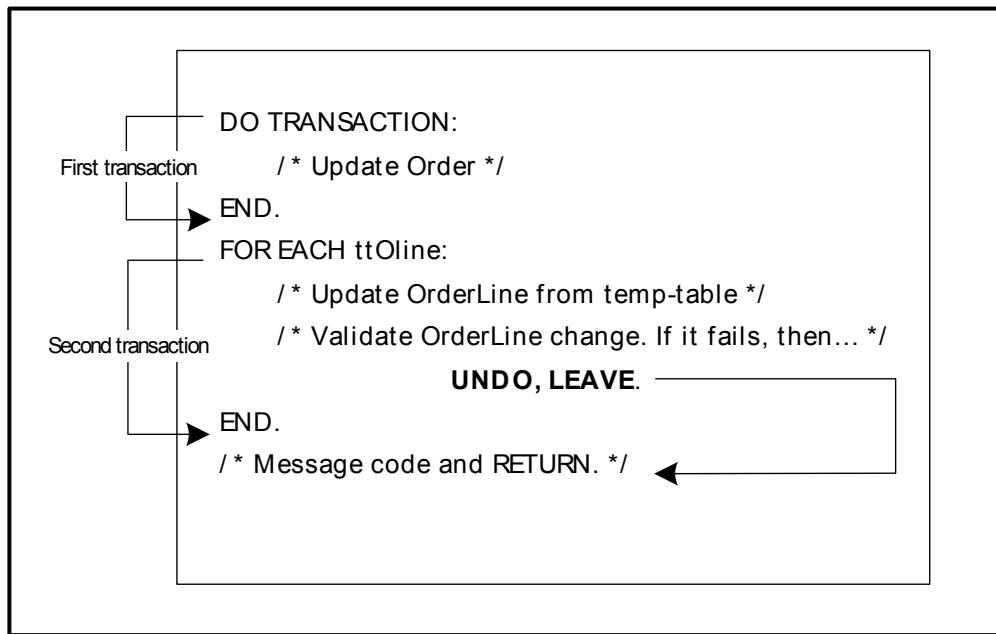


Figure 17–2: Example transaction scope

In [Figure 17–2](#), the first transaction, which saves changes to the **Order**, completes at the **END** statement for the **DO TRANSACTION** block. At the end of the **FOR EACH** block, the transaction to save the current **OrderLine** ends, committing those changes to the database, releasing the **OrderLine** record, and then going back to the beginning of the block. Each **OrderLine** is saved in a separate transaction.

The UNDO statement rolls back the transaction for the current **OrderLine** and leaves the FOR EACH block, which skips any remaining **ttOline** records. But any previously committed **OrderLine** changes remain in the database. For example, in Figure 17–3, the user changes the **Price** for **Line 1** from 7.49 to 7.60, for **Line 2** from 23.00 to 30.00, and for **Line 3** from 13.98 to 13.50.

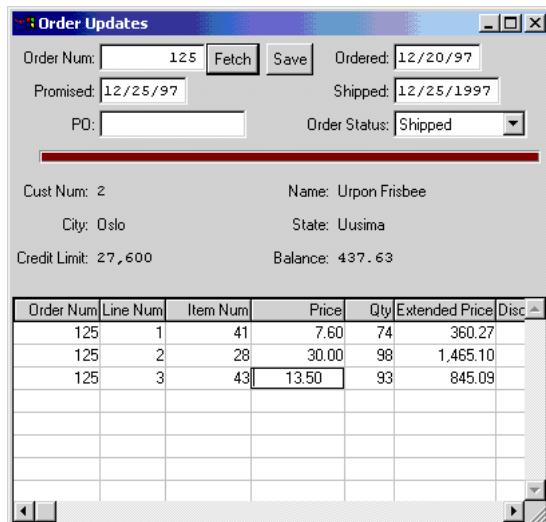


Figure 17–3: Order Updates example

The first and third changes are valid, but the second one is not. It increases the **ExtendedPrice** by too much. So when the user chooses **Save**, the user sees an error message for **Line 2**, as shown in Figure 17-4.

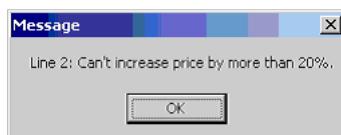


Figure 17–4: Order Updates example message

But if the user presses the **Fetch** button to refresh all the **OrderLines**:

- The change to **Line 1** is committed, because it is in its own transaction.
 - The change to **Line 2** is undone, because it failed the constraint.
 - The change to **Line 3** is never even applied because the code left the block after the error on **Line 2**.

Figure 17–5 shows the result.

Figure 17–5: Order Updates example (after Fetch)

This might not be the behavior you want.

On the one hand, you might want all **OrderLines** to be committed together or all undone if any one fails. In another case, you might want to process each **OrderLine** as a separate transaction, but keep going if one of them fails. Look at both of those cases.

In the first case, you want all the **OrderLine** updates to either succeed or fail together. If any one fails its validation, all are rolled back.



To see an example of the first case:

1. Define the transaction scope to be greater than a single iteration of the FOR EACH block by putting a new DO TRANSACTION block around it. Then add a label for that new block to identify how much to undo in your UNDO statement:

```
OlineBlock:
DO TRANSACTION:
  FOR EACH ttOline WHERE ttOline.TransType = "":
```

2. Change the UNDO statement to undo the entire larger transaction and to leave that outer block as well:

```
UNDO OlineBlock, LEAVE OlineBlock.
```

Remember that the default is to undo and leave the innermost block with the error property, the FOR EACH block.

3. Add another END statement to match the new DO TRANSACTION statement that begins the new block:

```
END. /* END ELSE DO If we updated the OrderLine */
END. /* END DO FOR EACH ttOline */
END. /* END new DO TRANSACTION block */
```

Note that a block label, such as **OlineBlock:**, does not require a matching END statement. It is simply an identifier for a place in the code and does not actually start a block of its own.

4. Make a change to the code that restores the original values to the temp-table if there's an error. Because the error might not be on the line that's current when you leave the block, you need to re-read all the **OrderLines** for the **Order** and buffer-copy their values back into the update copies of each of the temp-table records, in a FOR EACH loop:

```
IF cMessage NE "" THEN
DO:
  FOR EACH OrderLine WHERE OrderLine.OrderNum = bUpdateOline.OrderNum
  NO-LOCK:
    FIND bUpdateOline WHERE OrderLine.LineNum = bUpdateOline.LineNum
      AND bUpdateOline.TransType = "U".
    BUFFER-COPY OrderLine TO bUpdateOline.
  END.
  RETURN cMessage.
END.
```

Now if you make changes to three **OrderLines**, and the second of the three is invalid, then all three changes are rolled back because they're all part of one large transaction. You see this reflected in your window.

Figure 17–6 shows a sketch of what this variation looks like.

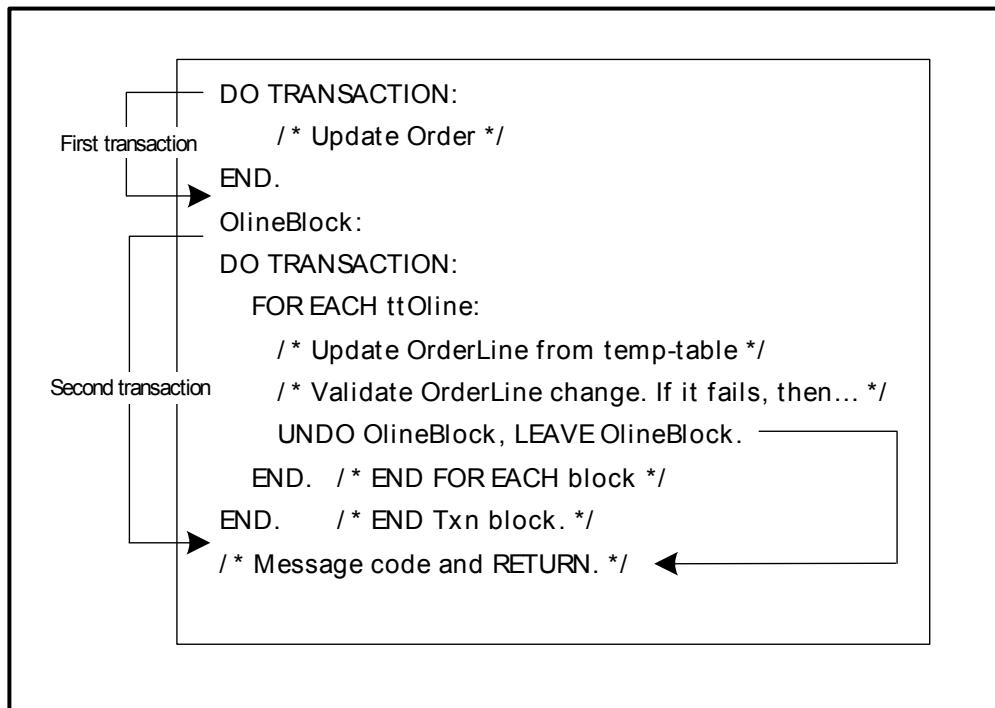


Figure 17–6: Variation of transaction scope

Now look at the second case. You'd like each **OrderLine** committed independently of the others, and you'd like to keep going if one is invalid. In this case, you can use the **NEXT** option on the **UNDO** statement instead of **LEAVE**. If an error is found, the current transaction is undone and your code continues on to the next **ttOline** record.



To try this variation:

1. Remove the **OlineBlock** block label, along with the **DO TRANSACTION** block header and its matching **END** statement, from around the **FOR EACH** block.
2. Change the **UNDO**, **LEAVE** statement to **UNDO**, **NEXT**.

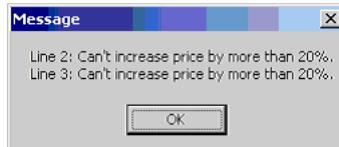
Since it is now possible to get errors on more than one **OrderLine** at a time, you should be prepared to accumulate multiple error messages in your message string.

3. Append each new message to the end of the string by using the plus sign (+) as a concatenation symbol for the character variable (**cMessage = cMessage + . . .**).

4. Put a newline character at the end of each message, using the CHR function to append the ASCII character whose numeric value is 10 to the string:

```
IF bUpdate0line.ExtendedPrice > (tt0line.ExtendedPrice * 1.2) THEN
DO:
  cMessage = cMessage +
    "Line " + STRING(OrderLine.LineNum) +
    ": Can't increase price by more than 20%." + CHR(10).
  UNDO, NEXT.
END.
```

5. Run the window.
6. Enter a valid price for **Line 1** and invalid prices for **Lines 2 and 3**. You see error messages for both of these:



You can also see that the valid change for **Line 1** is kept because you're back to making each iteration of the FOR EACH block its own transaction.

Figure 17–7 shows a sketch of this variation.

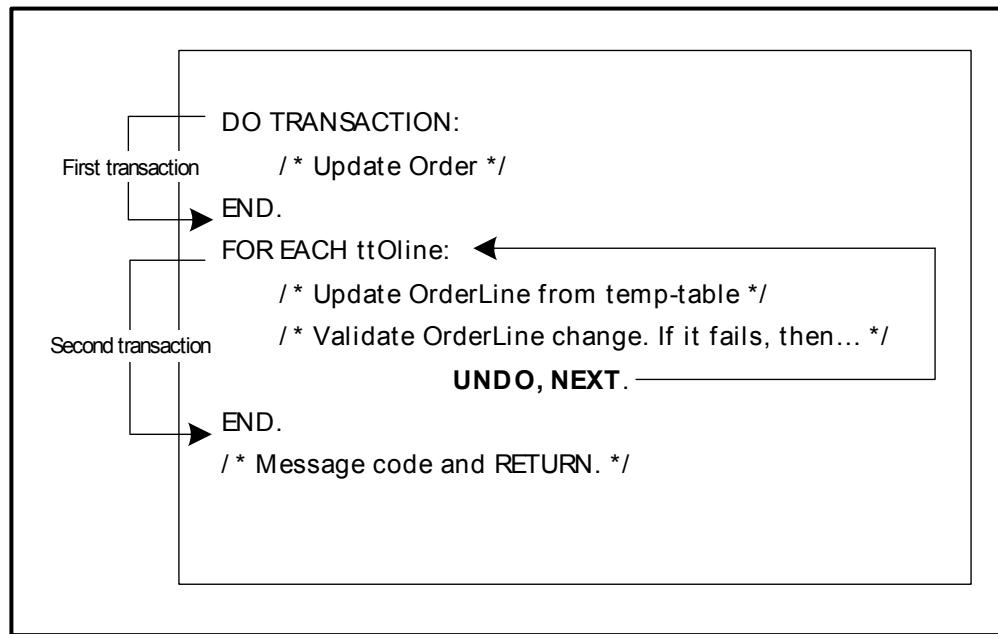


Figure 17–7: Another variation of transaction scope

Subtransactions

You separated the **Order** update and the **OrderLine** updates out into two separate transactions by moving the end of the DO TRANSACTION block up after the **Order** block.



To look at what happens if you combine them all again:

1. Define a new label for the DO TRANSACTION block:

TransBlock:

```
DO TRANSACTION ON ERROR UNDO, LEAVE:  
  FIND ttOrder WHERE ttOrder.TransType = "" NO-ERROR.
```

2. Move this block's END statement back all the way down to the end of the FOR EACH block, then change the UNDO, LEAVE statement to undo and leave that entire block:

```
IF bUpdateOline.ExtendedPrice > (ttOline.ExtendedPrice * 1.2) THEN  
DO:  
  cMessage = cMessage +  
    "Line " + STRING(OrderLine.LineNum) +  
    ": Can't increase price by more than 20%." + CHR(10).  
  UNDO TransBlock, LEAVE TransBlock.  
END.  
END. /* END ELSE DO If we updated the OrderLine */  
END. /* END DO FOR EACH ttOline */  
END. /* END DO Transaction */
```

Now the transaction structure looks like the diagram in [Figure 17–8](#).

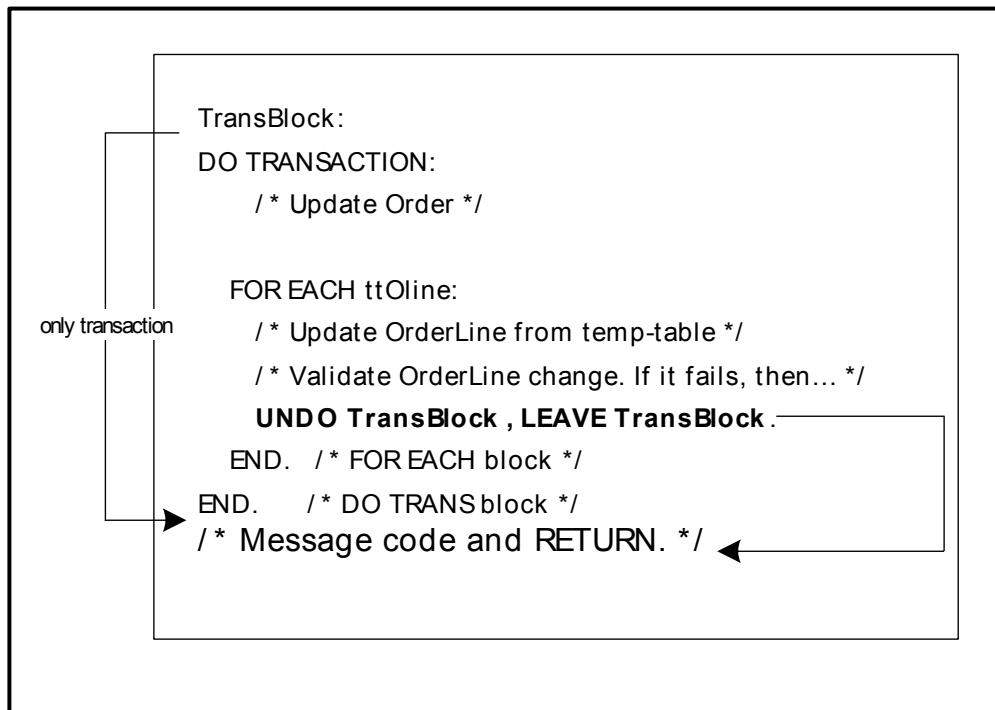


Figure 17–8: Third variation of transaction scope

If there's an error on any **OrderLine**, then the whole transaction is backed out, *including any change to the Order*.



To test this:

1. Edit one of the fields in the **Order**, such as the **PO**, and then make an invalid change to one of its **OrderLines**.
2. Choose **Save**. You see an error message.
3. Choose **Fetch** to refresh the **Order**.

The changes you made to the **Order** have been undone along with changes to the **OrderLines**. (Note that the code isn't set up to refresh the **Order** display if the transaction fails. This is an exercise you can do yourself.)

But what if you want to undo a *portion* of a transaction? Progress supports the capability to do this. If your application has multiple nested blocks, each of which would be a transaction block if it stood on its own, then the outermost block is the transaction and all nested transaction blocks within it become *subtransactions*. A subtransaction block can be:

- A procedure block that is run from a transaction block in another procedure.
- Each iteration of a FOR EACH block nested within a transaction block.
- Each iteration of a REPEAT block nested within a transaction block.
- Each iteration of a DO TRANSACTION, DO ON ERROR, or DO ON ENDKEY block inside a transaction block.

If an error occurs during a subtransaction, all the work done since the beginning of the subtransaction is undone. You can nest subtransactions within other subtransactions. You can use the UNDO statement to programmatically undo a subtransaction.

In the sample logic procedure, for example, with the END statement moved to the end, the FOR EACH block is really a subtransaction within the main transaction. An error inside this inner block undoes only the change made in that block. Likewise, if you change the UNDO statement back to UNDO, NEXT, then the **Order** changes are saved and only the current **OrderLine** changes are undone, as shown in [Figure 17–9](#).

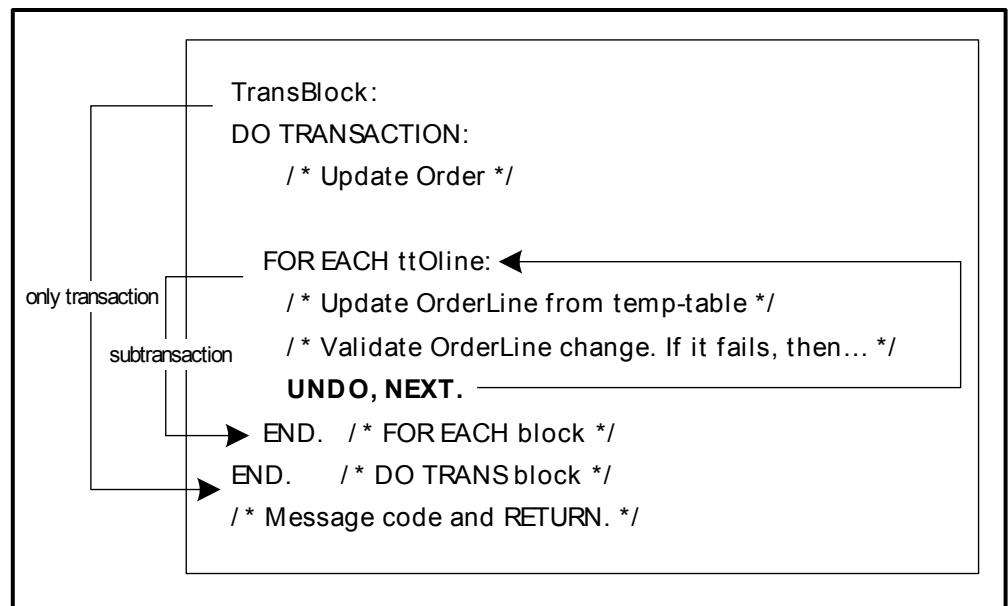


Figure 17–9: Example subtransaction

Note that a FOR EACH, REPEAT, or procedure block that does not directly contain statements that either modify the database or read records using an EXCLUSIVE-LOCK does *not* start a transaction on its own. However, if contained inside a transaction block, it *does* start a subtransaction.

Transaction mechanics

It's important to have at least a basic understanding of how Progress handles transactions and subtransactions, although a detailed discussion of this topic is beyond the scope of this book. During a transaction, information on all database activity occurring during that transaction is written to a *before-image* (or BI) file that is associated with the database and located on the server with the other database files. The information written to the before-image file is coordinated with the timing of the data written to the actual database files. That way, if an error occurs during the transaction, Progress uses the before-image file to restore the database to the condition it was in before the transaction started. Information written to the before-image file is not buffered. It is written to disk immediately, so that there is minimal loss of information in the event of a system crash.

Space in the before-image file is allocated in units called *clusters*. Progress automatically allocates new clusters as needed. After all changes associated with a cluster are committed and written to the database itself, Progress can reuse the cluster. Therefore, the disk space used by the before-image file depends on several factors including the cluster size, the scope of your transactions, and when physical writes are made to the database files. An action such as creating a huge number of database records in a batch procedure within a single transaction creates an enormous before-image file. You should avoid such actions.

When Progress encounters a transaction block nested within another transaction block, it starts a subtransaction. All database activity occurring during that subtransaction is written to a *local-before-image* (or LBI) file. Unlike the database BI file, Progress maintains one LBI file for each user. If an error occurs during the subtransaction, Progress uses this local-before-image file to restore the database to the condition it was in before the subtransaction started. In any case where a full transaction is not being backed out, Progress uses the local-before-image file to back out, not only subtransactions, but also changes to variables not defined as NO-UNDO.

Because the LBI file is not needed for crash recovery, it does not have to be written to disk in the same carefully synchronized fashion as does the before-image information. This minimizes the overhead associated with subtransactions. The LBI file is written using normal buffered I/O. The amount of disk space required for each user's LBI file depends on the number and size of subtransactions started that are subject to being done. It is advisable that you minimize the use of subtransactions, as well as the scope of your overall transactions, not just to simplify the handling of these files but also to minimize record contention with other users.

Using the ON phrase on a block header

In addition to placing one or more UNDO statements within a transaction, you can also specify the default undo processing as part of the block header of a FOR, REPEAT, or DO block. The default action is to retry the block that was undone or, if there is no user interaction that could change the data, to leave the block.

The UNDO phrase as a part of a block header has the same syntax as the UNDO statement itself. You can specify the action to be LEAVE, NEXT, RETRY (with or without a label), or RETURN ERROR or NO-APPLY.

There are four special conditions Progress recognizes:

- The ERROR condition
- The ENDKEY condition
- The STOP condition
- The QUIT condition

Handling the ERROR condition

Your application can encounter an error condition whenever Progress cannot execute a statement properly, such as trying to find a record that does not exist or create a record with a duplicate value in a unique index. Progress has a built-in error message associated with each such error condition and, by default, displays it (or writes it to an error log on an AppServer).

For example, if your procedure executes a FIND statement for a nonexistent **Customer**, you get the error shown in Figure 17–10.



Figure 17–10: FIND error message

The error number in parentheses lets you locate the message number under the Help menu to get more information on the error. It can also help you when you are reporting unexpected errors to technical support.

You can also generate the ERROR condition programmatically using the RETURN ERROR statement, either as part of the ON phrase of a block header or as a statement of its own. If your application has associated a keyboard key with the ERROR condition then Progress also raises ERROR when the user presses that key.

ERROR-STATUS system handle

In most cases, you do not want raw error messages to be shown to users, even when it is their mistake that causes the error. The alert box in [Figure 17–10](#), for example, is not a very friendly or informative way to present an error to a user. Even more important, it is essential that your procedures anticipate all possible error conditions whether they are caused by a user action or not, and respond to them, in some cases by suppressing an error message altogether. In addition, your application must define a mechanism for returning errors generated in an AppServer session back to the client, because by default Progress messages simply go to the server log file and are never seen.

To check for errors programmatically, you use the ERROR-STATUS system handle. Many Progress statements support the NO-ERROR option as the last keyword in the statement. If you specify this option, that statement does not generate the ERROR condition. Instead, if the statement cannot execute properly, execution continues with the next statement. You can then check whether an error occurred by examining the attributes of the ERROR-STATUS system handle.

The ERROR-STATUS handle contains information on the last statement executed with the NO-ERROR option. The logical attribute ERROR-STATUS:ERROR tells you whether an error occurred. Because in some cases a single error can return multiple messages, the NUM-MESSAGES attribute tells you how many messages there are. The GET-MESSAGE(<msg-num>) method returns the text of the message, and the GET-NUMBER(*msg-num*) method returns the internal message number. Here's a simple example:

```
DEFINE VARIABLE iMsg AS INTEGER      NO-UNDO.  
  
FIND Customer WHERE CustNum = 9876 NO-ERROR.  
IF ERROR-STATUS:ERROR THEN  
DO iMsg = 1 TO ERROR-STATUS:NUM-MESSAGES:  
    MESSAGE "Error number: " ERROR-STATUS:GET-NUMBER(iMsg) SKIP  
          ERROR-STATUS:GET-MESSAGE(iMsg)  
    VIEW-AS ALERT-BOX ERROR.  
END.
```

Because there is no **Customer 9876**, you get an error and your code displays the message box shown in Figure 17–11.

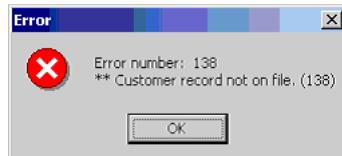


Figure 17–11: Example error message

Because you are intercepting the error, you can handle it more gracefully than this and also have your program logic proceed accordingly. You can check the message number using the GET-NUMBER method and put code in your application to deal with each of the possible error conditions.

Remember also that Progress provides special built-in functions, such as AVAILABLE and LOCKED, to make it easier for you to tell when certain common errors have occurred:

```
FIND Customer WHERE CustNum = 9876 NO-ERROR.
IF NOT AVAILABLE Customer THEN
  MESSAGE "So sorry, but this Customer does not seem to be there."
  VIEW-AS ALERT-BOX INFORMATION.
```

Figure 17–12 shows the result.



Figure 17–12: Example information message

Note that the ERROR-STATUS handle holds only error conditions and messages for the most recently executed statement with the NO-ERROR option. It does not accumulate errors over multiple statements. The ERROR-STATUS remains in effect (and checkable by your code) until the next statement with the NO-ERROR option.

When the ERROR condition occurs anywhere outside of a trigger block, Progress looks for the closest block with the error property and undoes and retries that block. As discussed earlier, if it is not meaningful to retry the block, Progress proceeds to the next iteration if it is a repeating block or else leaves the block.

Because this is the default, the following transaction header from the sample `saveOrder` procedure could simply be `DO TRANSACTION`: and have the same effect:

```
DO TRANSACTION ON ERROR UNDO, LEAVE:
```

Error handling

If an error occurs in a database trigger block, Progress undoes the trigger block and returns `ERROR`.

If you use the `NO-ERROR` option on statements within a block, you are suppressing not only the visible error message but also the `ERROR` condition itself. Therefore, if you do this, it becomes your responsibility to check for errors and respond to them correctly. This might include issuing an `UNDO` statement of your own. The `ON` phrase in the header simply changes the default action for untrapped conditions.

RETURN statement and ON . . . RETURN phrase

In any `RETURN` statement, whether it returns an `ERROR` or not, you can return a text string to the calling procedure. This string is accessible in the caller through the `RETURN-VALUE` built-in function. Thus, a procedure can use a single `RETURN` statement to raise the `ERROR` condition in the caller, return a message, or both:

```
RETURN [ ERROR ] [ return-value-string ] .
```

Likewise, the caller can check for the `ERROR-STATUS:ERROR` condition, or a `RETURN-VALUE`, or both, depending on the agreement between caller and callee as to how they communicate with one another. The `RETURN-VALUE` function retains its value even through multiple `RETURN` statements. Thus, while it is not required, it is advisable always to have a `RETURN` statement at the end of every procedure, if only to clear the `RETURN-VALUE`. A simple `RETURN` statement is the same as `RETURN ""`. If you want to pass the `RETURN-VALUE` up the call stack, you should do this explicitly using the statement `RETURN RETURN-VALUE`.

The same is true of the `RETURN` option as part of the `ON` phrase in a block header. It can return a `return-value`, raise `ERROR`, or both.

ENDKEY condition

The ENDKEY condition occurs when the user presses a keyboard key that is mapped to the ENDKEY keycode when input is enabled. It also occurs if an attempt to find a record fails or you reach the end of an input stream. Because your applications should not normally mix data input with transaction blocks in the same procedures, you do not frequently have cause to use the ON ENDKEY phrase on a transaction block.

STOP condition

Progress supports a STOP statement that lets the user terminate or restart your application altogether if an unrecoverable error occurs. You can trap the STOP condition in your block header statements as well. The STOP condition occurs when a Progress STOP statement executes or when the user presses the keyboard key mapped to that value. The STOP key, by default, is mapped to **CTRL-BREAK** on MS Windows and **CTRL-C** on UNIX.

When the STOP condition occurs, by default Progress undoes the current transaction (if any). If the user starts the application from an OpenEdge tool, such as the Procedure Editor, Progress terminates the application and returns to that tool. Otherwise, if the user starts the application using the Startup procedure (-p) startup option on the OpenEdge session, Progress reruns the startup procedure.

Progress raises the STOP condition when an unrecoverable system error occurs, such as when a database connection is lost or an external procedure that your code runs cannot be found. You cannot put the NO-ERROR condition on a RUN statement for an external procedure, as you can for an internal procedure. Therefore, the only way to trap such an error is to put the statement in a DO ON STOP block such as this:

```
DO ON STOP undo, LEAVE:  
    RUN foo.p.  
END.  
MESSAGE "The procedure to support your last action cannot be found."  
        SKIP "Please try another option." VIEW-AS ALERT-BOX.
```

If the procedure isn't found you still get an error, as shown in [Figure 17–13](#).



Figure 17–13: Procedure not found error message

But your procedure continues executing and you can deal with the error, as shown in [Figure 17–14](#).



Figure 17–14: Example message for procedure not found condition

If you anticipate that this might happen, it is better to use the SEARCH function to determine in advance whether Progress can find the procedure in the current PROPATH:

```
IF SEARCH("foo.p") = ? THEN
  MESSAGE "The procedure to support your last action cannot be found."
  SKIP "Please try another option." VIEW-AS ALERT-BOX.
ELSE RUN foo.p.
```

System and software failures

Following a system hardware or hardware failure that it cannot recover from, Progress undoes any partially completed transactions for all users. This includes any work done in any complete or incomplete subtransaction encompassed within the uncommitted transaction.

If Progress loses a database connection (for example, because a remote server fails), client processing can still continue. In this case, the following actions occur:

- Progress raises the STOP condition. For this special instance of the STOP condition, you cannot change the default processing. Progress ignores any ON STOP phrases.
- Progress deletes any running persistent procedure instances that reference the disconnected database.
- Progress undoes executing blocks, beginning with the innermost active block and working outward. It continues to undo blocks until it reaches a level above all references to tables or sequences in the disconnected database.
- Progress changes to the normal STOP condition. From this point on, Progress observes any further ON STOP phrases it encounters in your procedures.

Progress continues to undo blocks until it reaches an ON STOP phrase. If no ON STOP phrase is reached, it undoes all active blocks and restarts the top-level procedure.

QUIT condition

Progress also supports a QUIT statement to terminate the application altogether. Progress raises the QUIT condition only when it encounters a QUIT statement. The default handling of the QUIT condition differs from STOP in these ways:

- Progress commits, rather than undoes, the current transaction.
- Even if the user specifies the -p startup option to define the main procedure in your application, it returns to the operating system rather than trying to rerun the startup procedure. In other words, Progress quits the session unconditionally.

Summary

Having completed this chapter, you now have a thorough grounding in all the essentials of Progress programming with static statements. In the next three chapters, you'll learn how to use dynamic constructs to define both the user interface and the data management code for parts of your application. Then, in the final chapter, you'll learn some useful advanced 4GL constructs as well as some best practices guidelines for building efficient and maintainable procedures.

Using Dynamic Graphical Objects

In [Chapter 8, “Defining Graphical Objects,”](#) you learned about visual objects you can define in Progress, such as fill-in fields, buttons, and images. In other chapters, you have studied how to define and use record buffers, queries, and temp-tables. For all of these, both visual objects and data management objects, you use the `DEFINE` statement to describe the object to Progress. The compiler then builds a structure to support the object and makes that structure part of the compiled r-code.

In addition to defining these objects at compile time, you can create them and define their attributes programmatically at run time. This adds great flexibility to your application, as you can create objects specifically to respond to the needs of a procedure under particular circumstances. This might include creating dynamic data representation objects to deal with a variety of different kinds of fields that you can’t determine in advance. Or it could mean creating a temp-table whose fields aren’t known until run time and then creating a query to manage the data in that temp-table.

This chapter discusses:

- The `CREATE` statement, which you use for all of these kinds of objects.
- The use of handle variables to hold the descriptor for the object when it’s created.
- How you can use graphical dynamic objects in your application.

This chapter includes the following sections:

- [Creating visual objects](#)
- [Object handles](#)
- [Managing dynamic objects](#)
- [Creating a dynamic browse](#)
- [Creating a dynamic menu](#)

Creating visual objects

You can create all the kinds of visual objects you've seen in this book. There is a uniform syntax for all of them:

```
CREATE object-type handle [ IN WIDGET-POOL pool-name ]
[ASSIGN attribute = attribute-value [attribute = attribute-value] . . . ]
[ trigger-phrase ] .
```

When you create an object dynamically you must associate it with a HANDLE variable (or possibly a handle field in a temp-table). This is the only way to reference the object after you create it. Unlike a static object, it has no name.

The WIDGET-POOL phrase lets you define a special storage area in memory where you want the object's description to be located. The “[Using named widget pools](#)” section on page 18–14 describes this phrase in more detail.

Assigning object attributes

The optional ASSIGN phrase allows you to assign one or more attribute values for the object at the time you create it. The *attribute-value* for each *attribute* can be a constant or it can be an expression. You can use the same CREATE statement for multiple objects of the same type that need to have different attribute values. Alternatively, you can assign attribute values after you create the object by using this syntax:

```
handle:attribute = attribute-value
```

The attributes you can assign to a dynamic object are largely the same as those you can assign in a static definition. You can find a complete list of all the attributes supported by each object in *OpenEdge Development: Progress 4GL Reference* material or in the online help, under the **object-type Widget** topic (for example, **Button Widget**). In the description of many of these attributes, there are one or more special restrictions attached to the attribute:

- **Readable only** — These attributes can be read and used in expressions using the `handle:attribute` syntax, but their values cannot be assigned. In some cases, this is because the attribute value is part of the definition of the object and it would not make sense to change it. Examples of these attributes are:
 - **DYNAMIC** — Always true for dynamic objects and always false for static objects.
 - **HANDLE** — Holds the value of the object's handle.
 - **TYPE** — Evaluates to the *object-type*.

In other cases, an attribute value cannot be assigned directly because it is set indirectly using some other method or statement. For example, most visual objects have a TAB-POSITION object, which holds the sequential position of the object within its frame. You cannot set this directly, but rather you must use the MOVE-BEFORE-TAB-ITEM or MOVE-AFTER-TAB-ITEM methods on an object to change its tab position, which is then reflected in the value of its TAB-POSITION attribute.

- **Can be set only before the widget is realized** — You learned about realized objects in Chapter 8, “Defining Graphical Objects.” Some attribute values cannot be changed after the object has been realized. An example of this is the DEFAULT attribute for a button. The default button for a frame is the button that receives a RETURN event for its frame when the user presses RETURN or ENTER, which in turn executes its CHOOSE trigger. Once a button has been established as the default button for the frame and the button has been realized, when the frame is viewed, this attribute can't be changed.
- **Graphical interfaces only** — Progress supports most visual objects in character environments as well as GUI, but some object attributes can only be supported in graphical interfaces. An example is the BGCOLOR (background color) of a fill-in field. A fill-in field in a character environment does not use this attribute.

- **Character interfaces only** — Likewise, some attributes are supported only in character interfaces. An example is the DCOLOR of a button or other visual object, which is its display color.
- **Windows only** — Progress today supports a graphical interface only for the Microsoft Windows platform, so effectively any attributes marked as Windows only are the same as graphical interfaces only.

Assigning triggers to a dynamic object

You can assign one or more triggers to the object using the optional *trigger-phrase*, in the same way as you did for static objects in Chapter 9, “Using Graphical Objects in Your Interface.”

Object handles

You’ve seen a few examples of the use of handles in earlier chapters. You can define a variable to hold the handle of an object with the `DEFINE VARIABLE` statement:

```
DEFINE VARIABLE handle-name AS HANDLE [ NO-UNDO ].
```

You can also define temp-table fields as type `HANDLE`, so it would be possible to assign the handle of an object to a field in a temp-table record.

As you can see from the `CREATE` statement syntax, the only way to identify a dynamic object is to associate it with a handle. It does not have a name as a static object does. Progress builds a data structure to the object when it executes the `CREATE` statement and the handle becomes a pointer to that structure. You retrieve or set attribute values through the handle, and execute methods on the object through the handle. In Chapter 8, “Defining Graphical Objects,” you learned how to use attributes and methods by appending a colon and the attribute or method name to the object name, such as in the expression `bMyButton:LABEL`. For dynamic objects you do the same thing with the object’s handle, as shown in this sequence:

```
DEFINE VARIABLE hButton AS HANDLE      NO-UNDO.  
CREATE BUTTON hButton ASSIGN LABEL = "Test Button".  
MESSAGE "Label: " hButton:LABEL SKIP  
        "Type: " hButton:TYPE SKIP  
        "Handle:" hButton:HANDLE SKIP  
        "Dynamic?: " hButton:DYNAMIC  
        VIEW-AS ALERT-BOX.
```

Figure 18–1 shows the result.

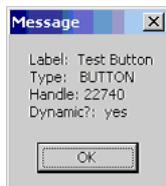


Figure 18–1: Test button message

As you can see, the handle can be represented as an integer value, but you cannot do any kind of arithmetic with object handles or manipulate handle values in any other way.

Each handle value represents a unique instance of the object you create. As you learned in [Chapter 8, “Defining Graphical Objects,”](#) when you use the DEFINE statement to define a static object, it does not have a unique identity until it is realized. Depending on how it is realized, the same object definition can have multiple distinct run-time instances. A handle always points to a single unique instance of an object.

Static object handles

Static objects have handles just as dynamic objects do. As soon as a statically defined object is realized, it is given a handle just like a dynamic object. You can access this handle value using the HANDLE attribute of the static object. In this way, you can access an object's attributes and methods using its handle just as you can for dynamic objects, as an alternative to using the object name. For example, the following code defines a handle variable and a static button. It then tries to display the button's handle in the procedure's default frame:

```
DEFINE VARIABLE hButton AS HANDLE      NO-UNDO.
DEFINE BUTTON bStaticButton LABEL "Static Button".
DISPLAY bStaticButton:HANDLE LABEL "Button handle:".
WAIT-FOR CLOSE OF THIS-PROCEDURE.
```

However, this code doesn't compile, as shown in [Figure 18–2](#), because it tries to reference the button handle without the button being realized.



Figure 18–2: Button handle error message

The button itself hasn't been realized in a frame and therefore has no handle. You can't display its handle, or for that matter any of its other attributes, using the `widget:attribute` syntax because Progress has no way of identifying what the attributes of the object are until it can attach a handle to it. If you add the button itself to the DISPLAY statement, then you can reference its handle and other attributes:

```
DEFINE VARIABLE hButton AS HANDLE      NO-UNDO.
DEFINE BUTTON bStaticButton LABEL "Static Button".
DISPLAY bStaticButton bStaticButton:HANDLE LABEL "Button Handle:".
WAIT-FOR CLOSE OF THIS-PROCEDURE.
```

Figure 18–3 shows the result.



Figure 18–3: Static button example result

Now that the static object has been realized, you can assign its handle to a variable and access its attributes through it. Here the code changes the button label and enables the button by setting its SENSITIVE attribute to YES:

```
DEFINE VARIABLE hButton AS HANDLE      NO-UNDO.  
DEFINE BUTTON bStaticButton LABEL "Static Button".  
DISPLAY bStaticButton bStaticButton:HANDLE LABEL "Button Handle:".  
hButton = bStaticButton:HANDLE.  
ASSIGN hButton:LABEL = "Enabled!"  
    hButton:SENSITIVE = YES.  
WAIT-FOR CLOSE OF THIS-PROCEDURE.
```

Figure 18–4 shows the result.



Figure 18–4: Enabled button example result

Managing dynamic objects

When you define a static object with the `DEFINE` statement, Progress knows everything it needs to know about the object at compile time. In addition to setting up the object's description in r-code at compile time, Progress defines a scope for the object. [Chapter 8, “Defining Graphical Objects,”](#) discussed object scope. Any defined object has a name, and a reference to that object has to be within its scope. Outside its scope the object name has no meaning and causes a compile-time error.

By contrast, a dynamic object has no particular scope in terms of procedure blocks. It can be referenced and used by any part of the application that has access to its handle from the time it is created until it is deleted. You can explicitly delete an individual dynamic object or you can place that object into a pool of memory where its storage is allocated and then delete the object by deleting its memory pool. It's especially important to keep in mind that the existence of a dynamic object has nothing to do with the scope of the handle variable you associate it with. For example, look at the following two-line procedure:

```
DEFINE VARIABLE hButton AS HANDLE      NO-UNDO.
CREATE BUTTON hButton.
```

When the procedure terminates, the `hButton` variable goes out of scope and you can no longer refer to it. Does this mean that the dynamic button is gone as well? No! The memory Progress allocates for the button is still there, but you have no way of referring to it (or deleting it). It occupies memory until your session ends.

Likewise, you can lose your access to an object by resetting the handle. For example:

```
DEFINE VARIABLE hButton AS HANDLE      NO-UNDO.
CREATE BUTTON hButton ASSIGN LABEL = "Lost Button!".
hButton = ?.
```

Oops! You've reset your handle variable but the object is still there. You just can't find it anymore. Or consider this example:

```
DEFINE VARIABLE hButton AS HANDLE      NO-UNDO.
DEFINE VARIABLE iCounter AS INTEGER    NO-UNDO.
DO iCounter = 1 TO 10:
  CREATE BUTTON hButton ASSIGN LABEL = "Button #" + STRING(iCounter).
END.
MESSAGE hButton:LABEL.
```

Figure 18–5 shows the result.



Figure 18–5: Example button message

You've just created ten buttons and lost contact with all but the last of them. This is not a good idea. Thus, the first rule of managing dynamic objects is to make sure that you don't lose track of them. Once you lose a valid handle to a dynamic object you can't access it and it just sits in memory taking up space.

The next sections describe ways to manage memory and clean up dynamic objects so that this kind of memory leak doesn't happen in your application.

Deleting dynamic objects

You can delete a dynamic object with the `DELETE OBJECT` statement:

```
DELETE OBJECT handle.
```

You can also use the `WIDGET` keyword in place of `OBJECT` in this statement. The *handle* is a variable or temp-table field of type `HANDLE`, which was previously used in a `CREATE` statement to create the object. If there is no object currently associated with the handle, Progress returns an error when it tries to execute the statement. In cases where your statement might be attempting to delete an object that has not been created or which has already been deleted, and you don't want to be informed of this at run time, you can avoid such an error by checking the handle in advance with the `VALID-HANDLE` function:

```
IF VALID-HANDLE(hButton) THEN  
  DELETE OBJECT hButton.
```

Alternatively, you can include the `NO-ERROR` keyword on the `DELETE` statement to suppress any error message:

```
DELETE OBJECT hButton NO-ERROR.
```

In either case, don't forget the OBJECT keyword. If you do, Progress thinks you're trying to delete a record from a table:

```
DEFINE VARIABLE hButton AS HANDLE      NO-UNDO.  
.  
.DELETE hButton.      /* Don't do this! */
```

Figure 18–6 shows the result.

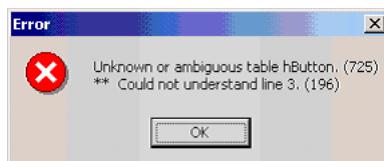


Figure 18–6: Unknown table error message

It's also important that you understand that you cannot delete a static object using its handle. Consider, for example, this sequence of statements:

```
DEFINE VARIABLE hButton AS HANDLE      NO-UNDO.  
DEFINE BUTTON MyButton.  
DISPLAY MyButton.  
hButton = MyButton:HANDLE.  
DELETE OBJECT hButton.
```

If you run this procedure you get the error shown in Figure 18–7 at run time.

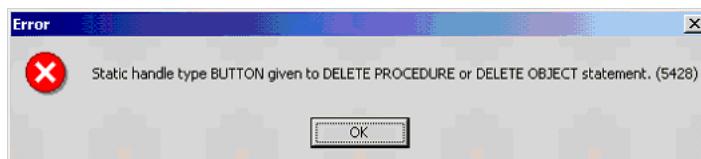


Figure 18–7: Static handle error message

Progress deletes static objects for you when they go out of scope. You cannot delete them yourself.

Using widget pools

Another way to manage the memory dynamic objects use is through widget pools. A *widget pool* is a pool of run-time memory that dynamic objects are created in. Every dynamic object you create is assigned to a widget pool.

Progress creates a single unnamed widget pool for each client session. This *session pool* is the default pool for all dynamic objects created during the session. It is automatically deleted when the session ends.

You can also create your own widget pools with the CREATE WIDGET-POOL statement:

```
CREATE WIDGET-POOL [ pool-name [ PERSISTENT ] ] [ NO-ERROR ] .
```

If you don't specify a *pool-name*, then you create an unnamed widget pool. Any dynamic objects you create are then assigned, by default, to the most recently created unnamed widget pool. The scope of the pool is the scope of the procedure that created it. When that procedure terminates, the widget pool is deleted along with all the dynamic objects that were created in it. This can be a very simple and powerful way of handling much of your memory management of dynamic objects. In fact, the standard template the AppBuilder uses for a window procedure puts this statement at the top of every window procedure you create:

```
/* Create an unnamed pool to store all the widgets created  
by this procedure. This is a good default which assures  
that this procedure's triggers and internal procedures  
will execute in this procedure's storage, and that proper  
cleanup will occur on deletion of the procedure. */
```

```
CREATE WIDGET-POOL .
```

In this way, all the dynamic objects you create from within the procedure are deleted when the procedure terminates. The CREATE WIDGET-POOL statement is in the **Definitions** section, which is editable, so if you need some other treatment of your dynamic objects, you are free to change it.

Note that this and other standard code in a procedure you create using the AppBuilder isn't actually generated by the AppBuilder. It is part of the template procedure file the AppBuilder uses as a starting point for a procedure of that type. These template procedure files are located in the `src/template` directory and identified by a set of text files with the `.cst` extension in the same directory. The AppBuilder reads and parses these `.cst` files on startup to build its **Palette** and the contents of its **New** dialog box. For example, the starting point for a window procedure, such as `CustOrderWin`, is the `window.w` file. For a structured procedure, such as `h-WinSuper.p`, it is `procedur.p`. If you want to learn more about how to define your own template procedure files, you should consult [OpenEdge Development: AppBuilder](#).

Sometimes you might need to manage objects more explicitly than just with an unnamed widget pool. In this case, you can give a name to a pool. Once you do this, you can then create objects and allocate them explicitly to that pool, as you saw in the syntax for the `CREATE object` statement. For example, this sequence of statements creates a named widget pool and then a button in that pool:

```
DEFINE VARIABLE hButton AS HANDLE      NO-UNDO.  
CREATE WIDGET-POOL "ButtonPool".  
CREATE BUTTON hButton IN WIDGET-POOL "ButtonPool".
```

Note, first of all, that the pool name is a character expression. You can specify either a quoted string literal or a variable or other character expression that holds the name.

When you create the button, you allocate it to the specific widget pool named `ButtonPool`. You can then later delete this pool using the `DELETE WIDGET-POOL` statement:

```
DELETE WIDGET-POOL [ pool-name ] [ NO-ERROR ] .
```

The memory for the button and any other dynamic objects you allocated to the pool goes away without disturbing other dynamic objects in other pools, including the unnamed pool.

You can only assign dynamic objects to a named widget pool, so if you want something other than the default allocation, you need to name your widget pools.

Using named widget pools

Here's a simple example of using a named widget pool:

```
DEFINE VARIABLE hButton AS HANDLE      NO-UNDO.
CREATE WIDGET-POOL "ButtonPool".
CREATE BUTTON hButton IN WIDGET-POOL "ButtonPool".
DELETE WIDGET-POOL "ButtonPool".
MESSAGE "What is the button handle value?" hButton SKIP
      "Is the button still there?" VALID-HANDLE(hButton).
```

The code creates a widget pool named `ButtonPool`. Then it creates a button in that pool. It then deletes the pool. The handle variable itself is still defined, of course, but its value now points to an invalid object because the memory is gone, as indicated by the message shown in [Figure 18–8](#).

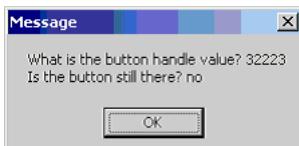


Figure 18–8: Button handle message

By default, a widget pool is deleted when the procedure that creates it terminates. If you create a named widget pool, you can use the `PERSISTENT` keyword to keep the pool around after the procedure terminates. This creates another level of memory management responsibility for you, because now you need to remember to delete the widget pool when you're done with it. Otherwise, it lasts until the end of the session just like the default pool does.

The `NO-ERROR` keyword can prevent an error message if you try to create a pool with a name that is already in use or if you try to delete one that has already been deleted.

Now look at some variations on this theme. In this example, the code creates a named widget pool and a button in that pool, but then deletes the unnamed pool:

```
DEFINE VARIABLE hButton AS HANDLE      NO-UNDO.
CREATE WIDGET-POOL "ButtonPool".
CREATE BUTTON hButton IN WIDGET-POOL "ButtonPool".
DELETE WIDGET-POOL.
MESSAGE "What is the button handle value?" hButton SKIP
      "Is the button still there?" VALID-HANDLE(hButton).
```

A `DELETE WIDGET-POOL` statement without a pool name deletes the unnamed widget pool created most recently in that routine, where *routine* means a main procedure block, internal procedure, or trigger block. Progress does not display an error if there is no unnamed pool to delete, as you can see from this example. When you run the code, the button is still there because its pool wasn't deleted, as shown in [Figure 18–9](#).

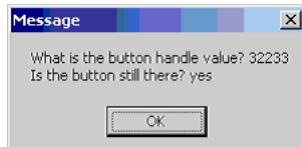


Figure 18–9: Button handle message

The default session pool is never deleted until the session ends.

In this next variation, you create the button in the unnamed pool and then delete the named pool:

```
DEFINE VARIABLE hButton AS HANDLE"          NO-UNDO.
CREATE WIDGET-POOL "ButtonPool".
CREATE BUTTON hButton. /* No longer IN WIDGET-POOL "ButtonPool". */
DELETE WIDGET-POOL "ButtonPool".
MESSAGE "What is the button handle value?" hButton SKIP
      "Is the button still there?" VALID-HANDLE(hButton).
```

Is the button still there after you delete the named pool? Yes, as shown in [Figure 18–10](#), because it wasn't allocated in the named pool you deleted.

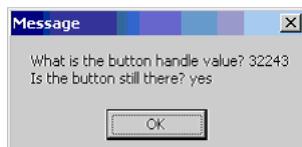


Figure 18–10: Another button handle message

Remember that a nonpersistent widget pool, whether named or unnamed, is automatically deleted when its procedure goes out of scope. Thus, a `DELETE WIDGET-POOL` statement at the end of such a procedure is optional. But it's certainly not a bad idea to include the statement to confirm that the pool is going away with the procedure.

Using unnamed widget pools

Any unnamed widget pool you create becomes the default pool until it is deleted or until you create another unnamed pool. Any unnamed pools you create are scoped to the routine in which they are created. This routine can be a main procedure block, an internal procedure, or trigger block. A subprocedure or trigger inherits, as its default pool, the most recent unnamed widget pool created in the calling procedure unless it creates an unnamed pool of its own. When execution of a routine ends, or it goes out of scope, any unnamed pools created in the routine are automatically deleted. Because a persistent procedure goes out of scope only when it is explicitly deleted, an unnamed widget pool created in one can persist as long as the procedure does.

You might use an unnamed pool to ensure that all objects created in the default pool in a procedure you run are deleted when that procedure returns or goes out of scope, as in this example:

```
CREATE WIDGET-POOL.  
RUN subprocedure.p.  
DELETE WIDGET-POOL.
```

In this example, the CREATE WIDGET-POOL statement creates a new default pool. Any objects created in the default pool within subprocedure.p are placed in this pool. After subprocedure.p completes, the pool is deleted along with any objects subprocedure.p might have created.

On the other hand, in a persistent procedure, you can use an unnamed pool to ensure that dynamic objects are *not* deleted after the procedure returns from its main block. Otherwise, if the calling procedure deletes the pool that was current when it ran the persistent procedure, it also deletes any dynamic objects for the persistent context.

Are widget pools static or dynamic objects?

It may strike you as you read through this discussion that there is a bit of an inconsistency in how Progress manages widget pools as opposed to other kinds of objects. A widget pool seems to be a dynamic object in its own right, but it doesn't have a handle. You use a CREATE statement to create a pool and a DELETE statement to get rid of it, but you reference it only by name and never by a handle. This is a valid observation, and one that you simply need to accept. A widget pool is definitely a dynamic object. It is created only when the CREATE statement is executed, just like other dynamic objects. It has no definition that the compiler is aware of as true static objects do. But it is true that you refer to it by name and not by a handle.

Manipulating the objects in a window

In this section, you'll learn how to locate and identify the objects in a window by their handles. Because both static and dynamic objects have handles, and attributes and methods you can manipulate through those handles, this material applies to both static and dynamic objects. Later in this chapter, you'll add new dynamic objects to the sample window.

As discussed in the [Chapter 14, “Defining Functions and Building Super Procedures,”](#) one of the principal reasons why you might need to locate and manipulate objects by their handles is to write generic code to adjust the appearance or behavior of objects from another procedure, such as a super procedure. The sample super procedure `h-WinSuper.p` showed a very simple example of that.

Next, you'll write another super procedure to make some more substantial adjustments to the objects in the test window by “walking the widget tree” of the window.



To write another super procedure:

1. Open the version of `h-CustOrderWin1.w` from Chapter Four as a starting point, as you've done before and save it as `h-CustOrderWin7.w`.
2. In the main block of `h-CustOrderWin7.w`, add two lines of code, one to use `h-StartSuper.p` to start or locate a new super procedure called `h-dynsuper.p` and the second to run an internal procedure that Progress locates in `h-dynsuper.p`:

```

MAIN-BLOCK:
DO ON ERROR UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
  ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:
    RUN enable_UI.
    RUN h-StartSuper.p("h-dynsuper.p").
    RUN changeFields.
    IF NOT THIS-PROCEDURE:PERSISTENT THEN
      WAIT-FOR CLOSE OF THIS-PROCEDURE.
  END.

```

3. Create a new structured procedure in the AppBuilder.
4. Create a new internal procedure called `changeFields`.

5. Define these variables that the procedure needs:

```
/*
-----  

Procedure changeFields:  

Purpose:    Locates the fields and browse in the target frame  

            and changes some of their attributes.  

Parameters: <none>  

Notes:  

-----*/  

DEFINE VARIABLE hWindow AS HANDLE      NO-UNDO.  

DEFINE VARIABLE hFrame   AS HANDLE      NO-UNDO.  

DEFINE VARIABLE hObject  AS HANDLE      NO-UNDO.  

DEFINE VARIABLE hColumn  AS HANDLE      NO-UNDO.
```

These handles allow you to identify the procedure's current window, its frame, each field or other object inside the frame, and finally the columns of the order browse in the frame. To understand how to walk down through the frame to locate objects, you need to look at the hierarchy of the objects the handles point to.

First, look at how a procedure's window handles are established. Progress defines a single default window for a session, and uses this window to display objects in when no other window is specified. There is a built-in system handle, called DEFAULT-WINDOW, that holds the handle of this window. There is also a system handle, called CURRENT-WINDOW, that Progress uses as the default for parenting frames, dialog boxes, and messages. Initially, the CURRENT-WINDOW is the same as the DEFAULT-WINDOW.

There is no DEFINE WINDOW statement in Progress. All other windows in a session are dynamic windows that you create with the CREATE WINDOW statement. When you start a procedure, you can set the CURRENT-WINDOW system handle to be a window created by your procedure. There is also a CURRENT-WINDOW attribute for the THIS-PROCEDURE system handle, which holds the procedure handle of the procedure containing the reference to THIS-PROCEDURE. Setting the CURRENT-WINDOW attribute sets the default parenting for all frames, dialog boxes, and messages used in that procedure without affecting windows defined by other procedures.

The standard code the AppBuilder uses for a window includes statements to set the CURRENT-WINDOW system handle to the dynamic window the AppBuilder code creates, as you saw in [Chapter 5, “Examining the Code the AppBuilder Generates.”](#) It also sets the CURRENT-WINDOW attribute of THIS-PROCEDURE to the same window:

```
/* Set CURRENT-WINDOW: this will parent dialog-boxes and frames.      */  

ASSIGN CURRENT-WINDOW      = {&WINDOW-NAME}  

THIS-PROCEDURE:CURRENT-WINDOW = {&WINDOW-NAME}.
```

Other procedures can access the window of a procedure that adheres to this convention by referencing its CURRENT-WINDOW attribute. Therefore, the first thing the changeFields procedure does is obtain that window handle. Because it is a super procedure of h-CustOrderWin7.w, the TARGET-PROCEDURE system handle within h-dynsuper.p has the same value as THIS-PROCEDURE does within h-CustOrderWin7.w:

```
hWindow = TARGET-PROCEDURE:CURRENT-WINDOW.
```

If the h-CustOrderWin7.w procedure did not set its CURRENT-WINDOW attribute, you would need to use some other mechanism to identify it. Because this is the most straightforward way to associate a window with a procedure, you should adopt this convention, whether you use this standard AppBuilder template or not.

A procedure can create more than one window. Only one window can be the procedure's CURRENT-WINDOW. If your procedures use more than one window, then you need to use some other convention to locate other windows in the procedure. For example, you could store a character string representing a list of other window handles in the window's PRIVATE-DATA attribute. The standard for procedures you build in the AppBuilder, as well as in the SmartObjects that are the components you can build applications from, is to create only one window in a procedure.

The next level down from a window is the frame or frames that are parented to the window. To identify the first frame in a window, you use its FIRST-CHILD attribute:

```
hFrame = hWindow:FIRST-CHILD.
```

If a window has more than one frame, you can access the other frames by following the NEXT-SIBLING of the first frame, which chains the frames together.

As you might expect, you go down to the objects in a frame by accessing its FIRST-CHILD attribute. But there is another level of object in between the frame and the fields and other objects in the frame. This level is called a field group.

Remember that Progress supports the notion of a DOWN frame, which displays multiple instances of the same fields to show multiple records from a result set in a manner similar to what a browse does. This visualization is used primarily by older character-mode applications where there is no browse. For this reason most graphical applications use only one-down frames. Regardless of this, the notion of this iteration through multiple instances of the same fields is still present in all frames, and each instance of the fields in the frame is a field group. Therefore, when you reference the FIRST-CHILD attribute of a frame, you are accessing its first field group:

```
hObject = hFrame:FIRST-CHILD. /* This is the field group */
```

Unless your frame has a DOWN attribute greater than one, the field group is not very interesting in itself. If the frame has more than one field group, you can use the NEXT-SIBLING chain to get at each of them. Otherwise, you just proceed down another level to get to the first object in the frame:

```
hObject = hObject:FIRST-CHILD.
```

In changeFields, you can combine all these steps into a single ASSIGN statement:

```
ASSIGN hWndow = TARGET-PROCEDURE:CURRENT-WINDOW
hFrame = hWndow:FIRST-CHILD
hObject = hFrame:FIRST-CHILD /* This is the field group */
hObject = hObject:FIRST-CHILD.
```

The ASSIGN statement is more efficient than a sequence of individual assignments. The steps in the sequence are executed in order, just as they appear. For instance, the value of the hObject variable assigned in the third step (to the field group) can then be used in the fourth step to assign the same variable a new value equal to the first object in the group.

Figure 18–11 is a pictorial representation of how these different objects and their handles are related.

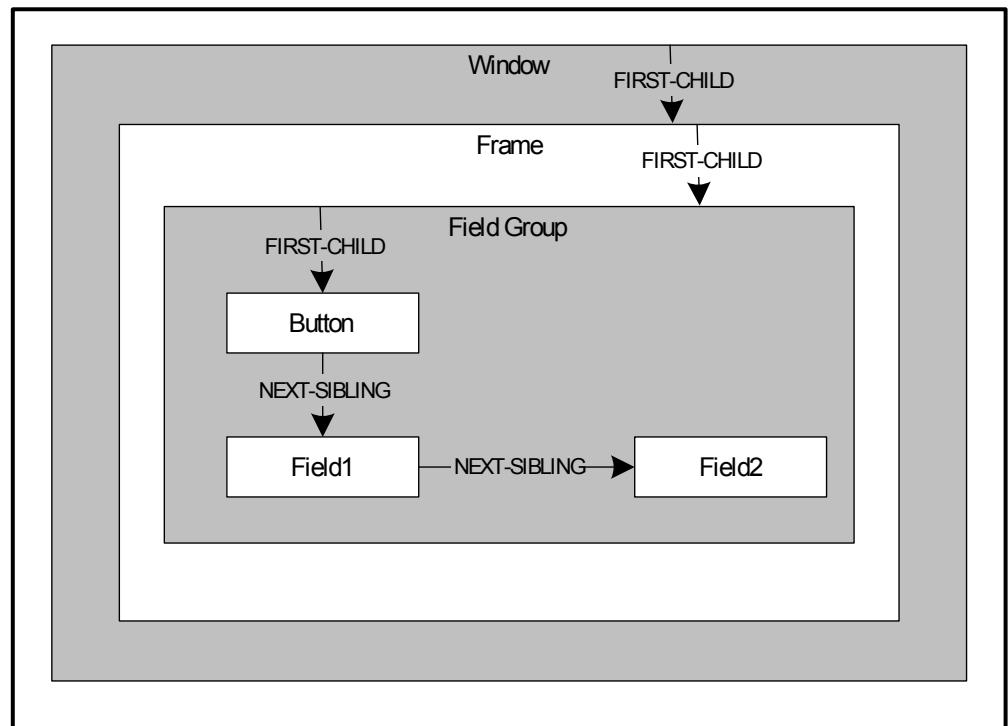


Figure 18–11: Relationships between objects in a window

Reading and writing object attributes

Once you have the handle to an object, you can change its appearance and behavior through its handle. To locate all the objects in the sample procedure's frame, you start with the field group's FIRST-CHILD, which is now in the `hObject` variable, and walk through the chain of NEXT-SIBLING objects as long as the object handle remains valid.

For example, assume you want to identify each fill-in field in the frame. For each one that is an integer field, you want to disable the field and set its background color to a dark gray. For each other fill-in field, you want to set the background color to green to highlight the field for data entry.

To start with, `changeFields` looks at the `TYPE` attribute of each object to see if it is a `FILL-IN`. If it is, then it checks the `DATA-TYPE` attribute to see if the field is an `INTEGER`. If it is, then it sets its `SENSITIVE` attribute to `false` and its `BGCOLOR` attribute to 8, which represents the color gray. Otherwise, if the field is not an integer, it sets the `BGCOLOR` attribute to 10, which is the color green:

```
DO WHILE VALID-HANDLE(hObject);
  IF hObject:TYPE = "FILL-IN" THEN
    DO:
      IF hObject:DATA-TYPE = "INTEGER" THEN
        ASSIGN hObject:SENSITIVE = NO
        hObject:BGCOLOR = 8.
      ELSE hObject:BGCOLOR = 10.
    END.
```

Identifying the columns of a browse

Next, you need to know how to identify the columns in the **Order** browse. The browse is a single Progress object with its own handle, but the columns in the browse have handles as well. The browse acts as a container for those columns much as a frame does for the fields and other objects it contains.

To get the handle to the first column in a browse, you use its `FIRST-COLUMN` attribute. The chain of columns is linked by the `NEXT-COLUMN` attribute of each column.

This next block of code checks to see if the current object in the frame is a browse. If it is, then it moves the browse to the left by changing its COLUMN attribute, and widens it by six characters by setting the WIDTH-CHARS attribute. It then walks through the columns, checking each one's data type. If a column is a date, it widens it by four characters. You use the same DO WHILE VALID-HANDLE block header as for the frame itself to walk through all the columns in the browse:

```
ELSE IF hObject:TYPE = "Browse" THEN
DO:
    ASSIGN hObject:COLUMN = hObject:COLUMN - 3
    hObject:WIDTH-CHARS = hObject:WIDTH-CHARS + 6.
    hColumn = hObject:FIRST-COLUMN.
    DO WHILE VALID-HANDLE(hColumn):
        IF hColumn:DATA-TYPE = "DATE" THEN
            hColumn:WIDTH-CHARS = hColumn:WIDTH-CHARS + 4.
            hColumn = hColumn:NEXT-COLUMN.
    END.
END.
```

Finally, you need to remember to move on to the next object in the frame before ending the original DO block:

```
hObject = hObject:NEXT-SIBLING.
END.

END PROCEDURE.
```

If you forget this step, your procedure goes into an infinite loop when you run it, and you'll need to press **CTRL-BREAK** to end it.

Figure 18–12 shows what you see when you run the window.

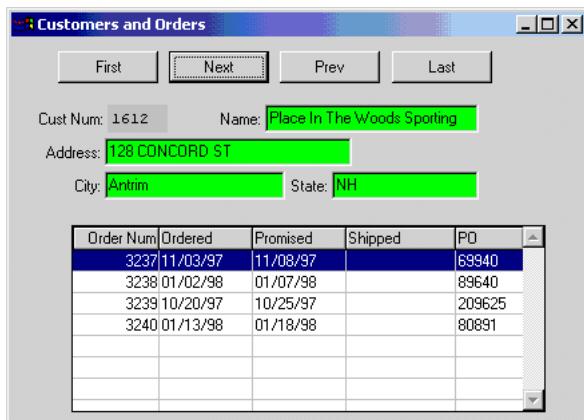


Figure 18–12: Updated sample window

Using the CAN-QUERY and CAN-SET functions

In this kind of code, where you are walking through a frame that might contain many different kinds of objects, you might need to verify not only whether the current object handle is valid, but also whether it is valid to set or query a particular attribute. If you don't, you might get an error at run time. The code you've been looking at checks that an object is a fill-in before it checks the DATA-TYPE, but suppose for a moment that the original TYPE check wasn't there:

```
IF hObject:DATA-TYPE = "INTEGER" THEN
    ASSIGN hObject:SENSITIVE = NO
        hObject:BGCOLOR = 8.
ELSE hObject:BGCOLOR = 10.
```

When Progress tries to retrieve the DATA-TYPE of an object that doesn't have this attribute, such as the browse, you get the error shown in Figure 18–13.

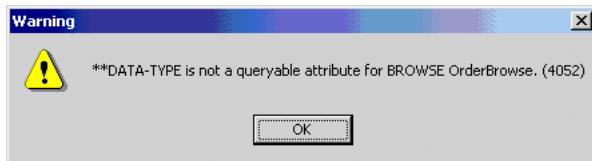


Figure 18–13: DATA-TYPE error message

To avoid this error, in cases where you can't be sure whether the attribute matches the object type, you can use the CAN-QUERY function to check whether something is a readable attribute before your code does so. CAN-QUERY takes two arguments, a valid object handle and a character expression that evaluates to an attribute name. This code example eliminates the error:

```
IF CAN-QUERY(hObject, "DATA-TYPE") AND  
    hObject:DATA-TYPE = "INTEGER" THEN
```

```
    .  
    .
```

Likewise, you can check in advance whether something is a writable attribute using the CAN-SET function, which also takes an object handle and attribute name as arguments:

```
IF CAN-SET (hObject, "SENSITIVE") AND  
    CAN-SET(hObject, "BGCOLOR") THEN  
    ASSIGN hObject:SENSITIVE = NO  
    hObject:BGCOLOR = 8.
```

You'll find these functions useful especially in cases where the attribute name itself is a variable, so that you can't be sure when you write the code whether all possible values will be valid.

To see a list of all the valid attributes you can set or query for an object, use the LIST-QUERY-ATTRS and LIST-SET-ATTRS functions. Each function takes a valid object handle as an argument:

```
IF hObject:TYPE = "FILL-IN" THEN  
    MESSAGE LIST-QUERY-ATTRS(hObject).
```

As you can see in [Figure 18–14](#), the list of attributes for most objects is quite large. You can find out about all of them in the online help, as well as in the third volume of [OpenEdge Development: Progress 4GL Reference](#).



Figure 18–14: Result of LIST-QUERY-ATTRS function example

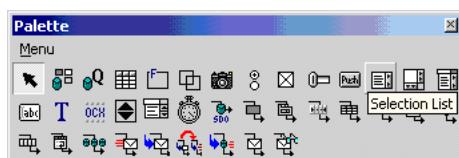
Adding dynamic objects to a window

To see dynamic objects in action, you can add a few to the test window. The goal is to let the user select one or more fields from the **OrderLine** table to display alongside the **Order** browse. Because these fields are the user's choice at run time, they are dynamic fill-ins.



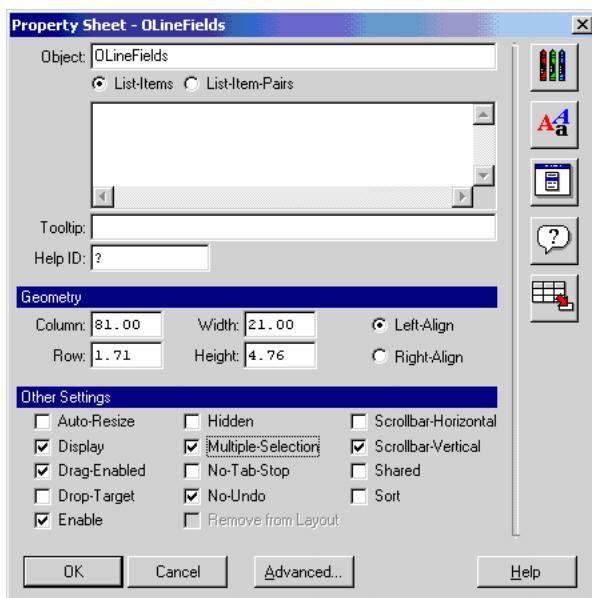
To add dynamic objects to the sample window:

1. Open the **h-CustOrderWin7.w** test window in the AppBuilder. Widen it somewhat to make room for some additional objects.
2. Pick the **Selection List** object from the **Palette**:



3. Drop the **Selection List** onto the window to the right of the **Customer** fields.

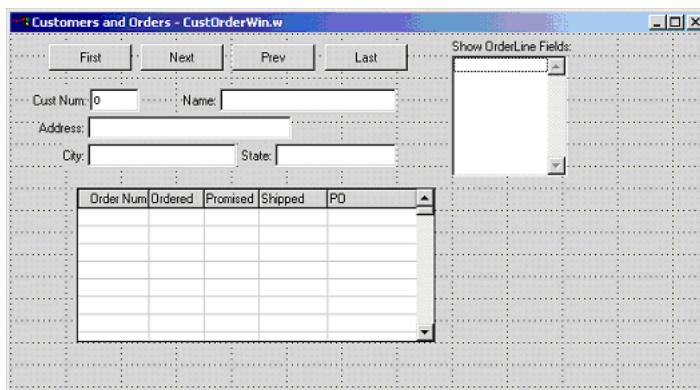
4. Double-click on the **Selection List** to bring up its property sheet:



5. Name the object **OlineFields**.
6. Check on the **Multiple-Selection** toggle box. This option allows the user to select more than one entry from the object at run time.

Notice that there is a choice between **List-Items** and **List-Item-Pairs** in the property sheet. Selection lists, combo boxes, and radio sets all provide this choice. If you set up the object to use **List-Items**, then the values displayed and selected in the object are the actual values stored in the field in the underlying variable or database field. In this case, the initial value of the object, which establishes the list of choices, is a simple comma-separated list of those values. If you set up the object to use **List-Item-Pairs**, then the values displayed are paired with another set of values that are the ones actually stored in the variable or field. In this case, the initial value of the object is a comma-separated list of alternating displayed and stored values. You use this option if the value stored is a coded value that is not meaningful to the user, and the user should instead choose from a more meaningful set of labels for those values. In this case, the default choice of **List-Items** is appropriate. Instead of setting the list as its initial value in its property sheet, you'll establish the list of **OrderLine** fields at run time.

7. Choose **OK** to save your changes to the property sheet.
8. Select the **Text** object from the **Palette**, drop it onto the window above the selection list, and give it a value of **Show OrderLine Fields**:



Using a buffer handle and buffer field handles

In the next chapter, you'll learn how to define dynamic data management objects such as buffers, and queries. In the meantime, it is useful to know that these objects, whether static or dynamic, have handles and attributes just like any other Progress object.



To see how to use buffer handle and buffer field handles:

1. Create a new internal procedure called **initSelection**.
2. Begin the internal procedure with these definitions:

```
/*
Purpose:      Set the selection list to a list of all the fields in the
              OrderLine table.
Parameters:   <none>
*/
DEFINE VARIABLE hBuffer AS HANDLE      NO-UNDO.
DEFINE VARIABLE iField   AS INTEGER    NO-UNDO.
```

To build a list of all the fields in the **OrderLine** table at run time, you need to be able to walk through a list of those fields in the **OrderLine** record buffer.

3. To do this, you first need to use the HANDLE attribute to get the buffer handle:

```
hBuffer = BUFFER OrderLine:HANDLE.
```

Notice that you need to include the BUFFER keyword in the statement so that Progress knows how to identify the literal value **OrderLine**.

Next, you need to know that the buffer object has an attribute called NUM-FIELDS that conveniently tells you how many fields there are in the buffer.

4. Use this code to start a block that walks through all those fields:

```
DO iField = 1 TO hBuffer:NUM-FIELDS:
```

There is another Progress object that represents a single field in a buffer. You get the handle of a particular field object using the buffer's BUFFER-FIELD attribute.

BUFFER-FIELD takes a single argument, which can be either the sequential field position within the buffer or the field name.

5. In this procedure, since you just want to walk through all the fields to build up a list, use its position to identify it:

```
hBuffer:BUFFER-FIELD(iField)
```

6. Finally, you need to know that, like other objects, the BUFFER-FIELD has various attributes you can query or set. In this case you want the NAME attribute. The Progress 4GL (beginning in Progress Version 9.1D) lets you chain multiple colon-separated references together in a single expression, such as this:

```
hBuffer:BUFFER-FIELD(iField):NAME
```

The only requirement is that each of the elements in the expression until the last one must be a handle, since each element is in turn an attribute of the handle that the expression yields at that point in its evaluation. Thus the expression above represents this sequence:

Starting with the handle to the buffer, retrieve the handle of the buffer field that is in position iField. Then retrieve the NAME attribute of that field handle.

You could even leave out the earlier step of saving off the buffer handle in a variable and put that into the expression as well, as in:

```
BUFFER OrderLine:BUFFER-FIELD(iField):NAME
```

In this example, you did not do this because the buffer handle is referenced in several different statements in this little procedure, so it's more efficient and makes the code a bit more readable to save the value off once and reuse it.

Populating a list at run time

There are two methods you can use on a selection list, combo box, or radio set to set the value list at run time: ADD-FIRST and ADD-LAST. Each has the same two forms:

```
object-handle:ADD-FIRST(item-list)
object-handle:ADD-LAST(item-list)
object-handle:ADD-FIRST(label, value)
object-handle:ADD-LAST(label, value)
```

Use ADD-FIRST to add items to the beginning of the list and ADD-LAST to add them to the end. If the object uses the simple **List-Items** form, in which the actual object values are displayed, then use the *item-list* form of the method to add one or more items to the list. If the object uses the **List-Item-Pairs** form, then use the second form of the method to specify a single *label* followed by the single *value* it represents.

These types of objects have a DELIMITER attribute to allow you to set a delimiter between items other than the default comma, in case one of the values or labels contains a comma.

The objects also support a logical SORT attribute, which initially is false. If you set SORT to true, then displayed items are sorted by their label. In this case, there is no meaningful difference between using ADD-FIRST and ADD-LAST.

The methods return true if the operation succeeded, and false if for any reason it failed.



To use the ADD-LAST method to add each of the OrderLine field names to the end of the selection list:

1. Enter this code to complete the `initSelection` procedure:

```
/*
Purpose:      Set the selection list to a list of all the fields in the
              OrderLine table.
Parameters:   <none>
*/
DEFINE VARIABLE hBuffer AS HANDLE      NO-UNDO.
DEFINE VARIABLE iFields AS INTEGER     NO-UNDO.

hBuffer = BUFFER OrderLine:HANDLE.
DO WITH FRAME CustQuery:
  DO iFields = 1 TO hBuffer:NUM-FIELDS:
    OLineFields:ADD-LAST(hBuffer:BUFFER-FIELD(iFields):NAME).
  END.
END.

END PROCEDURE.
```

2. To display these values in the list when the window is viewed, add a RUN statement to the procedure's main block:

```
MAIN-BLOCK:
DO ON ERROR UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
  ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:
  RUN enable_UI.
  RUN h-StartSuper.p("h-dynsuper.p").
  RUN changeFields.
  RUN initSelection.
  IF NOT THIS-PROCEDURE:PERSISTENT THEN
    WAIT-FOR CLOSE OF THIS-PROCEDURE.
END.
```

Creating dynamic fields

Before you look at the code for this example that actually creates and manages the dynamic objects, this section summarizes some basic principles of setting up dynamic field-level objects.

Frame parenting

You can place a dynamic field-level object in either a static or a dynamic frame. To do this, assign the frame handle to the FRAME attribute of the object. For a field-level object, there is also a PARENT attribute. Remember that there is a field group object that acts as the immediate container for field-level objects, in effect in between the frame and the individual objects. The PARENT attribute points to the field group, not the frame. Progress automatically puts the object into a field group when you assign its FRAME attribute and also when you assign the object a default tab position, if it can receive input.

Object positioning

To arrange objects in a frame, you must explicitly position each one by setting the appropriate vertical (ROW or Y) and horizontal (COLUMN or X) attributes. However, Progress *does* assume the topmost and leftmost position in the frame if you do not set a placement attribute for the object. This means that, if you place multiple dynamic objects into a frame without positioning them properly, they all wind up on top of one another.

Object sizing

You can size an object, depending on its object type and data type, using either the various height and width attributes or the FORMAT attribute. Because of the imprecise nature of width calculations for values displayed in variable-width fonts, you might have to adjust either the format or the width to be appropriate for the values you’re displaying. This is no different than for static objects in a graphical environment. Be aware that a FORMAT of “X(20)” is not necessarily the same width in terms of screen real estate as a WIDTH-CHARS of 20. Generally, Progress uses a formula for calculating a format that is slightly more generous (that is, yields a slightly greater width) than the formula for calculating the width from WIDTH-CHARS. Some amount of trial and error might be necessary to arrive at the right format or width for the type of data typically displayed in a field. Capital letters, for example, are on average much wider than lowercase letters, so a field that is displayed all in capitals likely needs a greater width than a lowercase or mixed-case field.

The sample code in the “[Using the FONT-TABLE to make the labels colon-aligned](#)” section on page 18–39 provides an example of using a built-in system handle called FONT-TABLE to calculate the display width of a specific string in a specific font, which you can also use to assign an appropriate width.

Label handling

You must provide a separate text object as a label for dynamic data representation objects, including fill-ins, combo boxes, editors, radio sets, selection lists, sliders, and text fields. When you drop fill-ins into the AppBuilder’s design window, for example, the AppBuilder is actually generating separate, dynamic label objects so that you see how the label will look at run time, since it is creating dynamic objects at design time to build up the contents of what will become a static frame and static field-level objects when you save it and it generates code for the frame and object definitions. When you create your own dynamic objects, you have to supply the dynamic text label yourself.

If you want a side label for a fill-in field, or one of the other dynamic data representation object types, you must create a separate text object and then assign its handle to the SIDE-LABEL-HANDLE attribute of the object it is a label for. For any other type of label, such as vertical columns, you must create and manage the text object completely separately. You must also position text objects used as labels explicitly, even for side labels. Progress assigns no positioning information for dynamic side labels, as it does for button or toggle box labels. The SIDE-LABEL-HANDLE attribute on the fill-in does not actually provide any automatic services such as moving the label together with the field. It is simply a useful way to help you navigate between the field object and its label object when you need to.

The example code described in the “[Adding dynamic fields to the test window](#)” section on page 18–34 shows you how to define side labels for dynamic fill-ins.

Data handling

Unlike static data representation objects, dynamic objects have no field or variable implicitly associated with them. You must explicitly assign data between an object’s SCREEN-VALUE attribute and the field or variable you use for data storage. This allows you to use a single object to represent multiple fields or variables at different times, if you wish, limited only by the object and data type.

Data typing

Some dynamic objects support entry and display data types other than CHARACTER. In particular, fill-ins and combo boxes support the full range of Progress entry and display data types. It's important to understand that, for dynamic objects, this support is for entry validation and display formatting purposes only. The SCREEN-VALUE attribute always holds the data in character format, no matter what the object's data type. You must make all necessary data type conversions using the appropriate functions (STRING, INTEGER, DECIMAL, etc.) when assigning data between the object's SCREEN-VALUE and the field or variable you use for data storage.

Adding dynamic fields to the test window

The first thing you need to add to h-CustOrderWin7.w to create dynamic fields is a variable or other storage to hold their handles. Because the whole purpose of the exercise is to allow the user to select a variable number of fields to display, there is no reasonable way to store each one's handle in a separate variable.

Storing a list of handles

You could store the object handles in a HANDLE variable array that has an EXTENT, but this is almost certainly a bad idea. The first rule of using a variable with an extent is that you should do it only when the proper value for the extent is clear, based on the nature of the data it is holding, such as values for the seven days in a week or the twelve months in a year. If you just try to pick a value that seems big enough, you will often regret it later when that number turns out to be too small for some case you hadn't anticipated.

The method used in the example is just to store the handles in a list, in character form. For a modest number of values, this is quite reasonable, and the conversion effort back and forth between a handle and its character representation is not significant.

Always keep in mind the alternative of using a temp-table to store a set of values during program execution. Although the overhead of having to perform a FIND on what amounts to a special database table may seem significant, in fact temp-tables are extremely fast. Most or all of the records you need to work with will likely be in memory anyway and, with the ability to index fields that you need to retrieve or filter on, even a large temp-table should provide very good performance. A temp-table is well suited to situations where the number of possible values you need to keep track of can grow large. How large is *large*? There's no precise answer to this, but it is probably a good rule of thumb that if you're storing more than a few dozen values, it is cleaner and possibly faster to use a temp-table. A temp-table is also the right choice when you need to store several related pieces of information for each item, each of which can become a field in the temp-table definition.

For this example you simply use a character variable. Its value needs to persist for the life of the procedure, because the handles are saved off by one internal procedure or trigger block and used by another.



To create dynamic fields in the sample window:

1. Define the `cFieldHandles` variable in the **Definitions** section of `h-CustOrderWin7.w`, which scopes the variable definition to the whole procedure:

```
/* Local Variable Definitions --- */  
DEFINE VARIABLE cFieldHandles AS CHARACTER NO-UNDO.
```

2. Write a block of code to execute whenever a new **Order** is selected. This is the **VALUE-CHANGED** event for the browse, which you've used in an earlier variation of this procedure.

The code in the **VALUE-CHANGED** trigger needs to find the first **OrderLine** for the **Order**. For the sake of simplicity, the example does not navigate through all the **OrderLines**, but you could easily extend it to do this. Then, it looks at the existing list of dynamic field handles (if any) and clears them out by setting their **SCREEN-VALUE** to blank:

```
/* ON VALUE-CHANGED OF OrderBrowse */  
DO:  
  DEFINE VARIABLE iField AS INTEGER      NO-UNDO.  
  DEFINE VARIABLE hField AS HANDLE       NO-UNDO.  
  
  FIND FIRST OrderLine OF Order.  
  DO iField = 2 TO NUM-ENTRIES(cFieldHandles) BY 2:  
    hField = WIDGET-HANDLE(ENTRY(iField, cFieldHandles)).  
    IF VALID-HANDLE(hField) THEN  
      hField:SCREEN-VALUE = "".  
  END.  
END.
```

3. Define a LEAVE trigger for the **OLineFields** selection list. The trigger uses these variables:

```
DEFINE VARIABLE iField      AS INTEGER      NO-UNDO.
DEFINE VARIABLE hField      AS HANDLE       NO-UNDO.
DEFINE VARIABLE hLabel      AS HANDLE       NO-UNDO.
DEFINE VARIABLE cFields     AS CHARACTER    NO-UNDO.
DEFINE VARIABLE cField      AS CHARACTER    NO-UNDO.
DEFINE VARIABLE hBufField   AS HANDLE       NO-UNDO.
DEFINE VARIABLE dRow        AS DECIMAL     NO-UNDO INIT 8.0.
```

4. To allow for the case where this is not the first time the user has selected a list of fields, add code that first deletes the existing fields using their object handles, which are stored in a list in the **cFieldHandles** variable:

```
DO iField = 1 TO NUM-ENTRIES(cFieldHandles):
  hField = WIDGET-HANDLE(ENTRY(iField,cFieldHandles)).
  DELETE OBJECT hField NO-ERROR.
END.
```

Remember that if you neglect to do this, each new request would add more objects to the session that aren't being used anymore. The **NO-ERROR** qualifier on the **DELETE OBJECT** statement simply suppresses any error message in the event that the object has already been deleted in some other way.

How about when the procedure is terminated? Do you need code to delete the dynamic fields that are around at that time to prevent a memory leak? The answer is no, but only because of the widget pool created in the **Definitions** section, which cleans up all dynamic objects created by the procedure when the procedure terminates. That's why the widget pool convention is so valuable. Without the widget pool created for the procedure, you could leave dynamic objects in memory for the duration of the session, even after the procedure exits.

Since this code is the LEAVE trigger for the selection list, the field's **SCREEN-VALUE** attribute holds the value the user selected. In the case of a multiple-selection list such as this, the value is actually a comma-separated list of all the entries the user selected.

5. Save this value in a variable to keep the rest of the code from having to refer to the **SCREEN-VALUE** attribute over and over again:

```
cFields = OLineFields:SCREEN-VALUE.
```

6. Add a block to iterate through all the selections. You saw earlier how the BUFFER-FIELD attribute on a buffer handle can take the ordinal position of the field in the buffer as an identifier. You can also pass the field name, as the code does here. Once you've retrieved the handle of the selected field, the code can query a number of different field attributes through that handle:

```
DO iField = 1 TO NUM-ENTRIES(cFields):
  ASSIGN cField = ENTRY(iField, cFields)
  hBufField = BUFFER OrderLine:BUFFER-FIELD(cField).
```

7. Create the text label for the fill-in. As you learned earlier, the label must be a separate text object:

```
CREATE TEXT hLabel1
  ASSIGN FRAME = FRAME CustQuery:HANDLE
    DATA-TYPE = "CHARACTER"
    FORMAT = "X(" + STRING(LENGTH(hBufField:LABEL) + 1) + ")"
    SCREEN-VALUE = hBufField:LABEL + ":"
    HEIGHT-CHARS = 1
    ROW = dRow
    COLUMN = 85.0.
```

The CREATE statement parents it to the frame, sets its data type, calculates a format and value for it using the LABEL attribute of the current buffer field, and positions it in the frame. The HEIGHT-CHARS of 1 makes the label text align properly with the value displayed next to it. The COLUMN positions it next to the browse, and the row is incremented each time through the loop to define a distinct position for each field.

8. Create the fill-in object itself:

```
CREATE FILL-IN hField
  ASSIGN DATA-TYPE = hBufField:DATA-TYPE
    FORMAT = hBufField:FORMAT
    FRAME = FRAME CustQuery:HANDLE
    SIDE-LABEL-HANDLE = hLabel1
    COLUMN = 85.0 + LENGTH(hBufField:LABEL) + 4
    ROW = dRow
    SCREEN-VALUE = hBufField:BUFFER-VALUE
    HIDDEN = NO.
```

The data type, format, and value all come from the buffer field object handle. The SIDE-LABEL-HANDLE attribute connects this fill-in to its handle object. The COLUMN setting allows room for the label before displaying the field value. The SCREEN-VALUE assigns the value from the buffer field's BUFFER-VALUE attribute. The HIDDEN attribute makes sure the field is viewed along with the frame that contains it.

9. Increment the row counter to set the position of the next field, and save off the handles of the labels and fill-ins in a list:

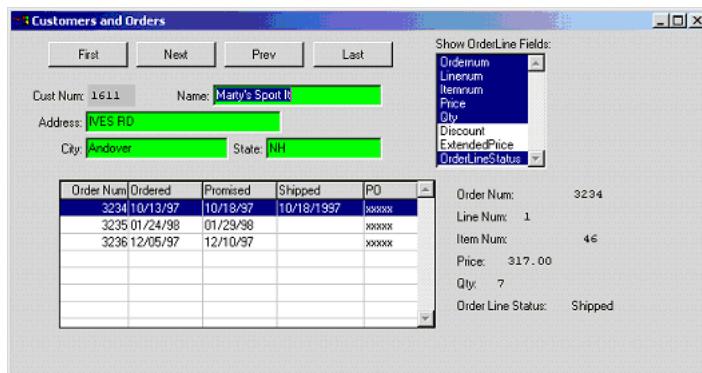
```
ASSIGN dRow = dRow + 1.0
cFieldHandles = cFieldHandles +
    (IF cFieldHandles = "" THEN "" ELSE ",") +
    STRING(hLabel) + "," + STRING(hField).
END. /* END DO iField */
```

10. Make sure that the VALUE-CHANGED trigger for the **Order** browse fires whenever a different record is displayed. This includes when the procedure first starts up, so make this addition to the main block:

```
MAIN-BLOCK:
DO ON ERROR UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
    ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:
        RUN enable_UI. RUN h-StartSuper.p("h-dynsuper.p").
        RUN changeFields.
        APPLY "VALUE-CHANGED" TO OrderBrowse.
        RUN initSelection.
        IF NOT THIS-PROCEDURE:PERSISTENT THEN
            WAIT-FOR CLOSE OF THIS-PROCEDURE.
    END.
```

11. Make the same addition to each of the navigation button triggers, as you have done to another version of the procedure in [Chapter 7, “Record Buffers and Record Scope.”](#)

12. Run the window. Now you can select one or more fields from the selection list, tab out of it, and see those fields displayed as dynamic fill-ins with dynamic labels next to the browse:



If a few of the fields seem to be positioned rather far to the right (the **Order Num** for instance), it's because they are right-justified numeric fields with overly generous display formats as defined in the Data Dictionary. Specifically, the **OrderNum** and **ItemNum** fields are defined in the schema with a long format that uses the **Z** character to format leading zeros. The **Z** tells Progress to replace leading zeroes with spaces, which pushes the displayed value out to the right. Others, such as the **Price**, are formatted with the **>** character, which tells Progress to suppress leading zeroes, effectively left-justifying the value. This is just a result of the formatting choices made by the database designer and has nothing to do with the display of dynamic values.

Using the FONT-TABLE to make the labels colon-aligned

This display looks all right as far as it goes, but in many cases you want your labels to appear right-justified rather than left-justified. In other words, you want the colons that end each label to be vertically aligned, so that all the field values can begin at the same column position to the right of that. How can you do this?

Progress provides a built-in system handle, called **FONT-TABLE**, which is an object representing the current font. There are four useful methods you can apply to this handle to calculate the actual size of a value when it's displayed: **GET-TEXT-WIDTH-CHARS**, **GET-TEXT-HEIGHT-CHARS**, **GET-TEXT-WIDTH-PIXELS**, and **GET-TEXT-HEIGHT-PIXELS**. In an alternative version of the trigger code for the selection list, you can use this function to align the labels on their colons.

**To colon-align your labels:**

1. Change the column setting in the CREATE TEXT statement for the label to this:

```
/* COLUMN = 85.0. -- modified to do colon-aligned labels */
COLUMN = 100.0 - FONT-TABLE:GET-TEXT-WIDTH-CHARS(hBuffField:LABEL + ":").
```

Instead of starting at column 85 and positioning the label to the right, the statement starts where the labels should *end* (column position 100), and subtracts the label width as calculated by the method on the FONT-TABLE. This gives the right starting position for the label object.

2. Change the column assignment for the fill-in to be fixed at column 102:

```
/* COLUMN = 85.0 + LENGTH(hBuffField:LABEL) + 4 -- changed for
colon-aligned */
COLUMN = 102
```

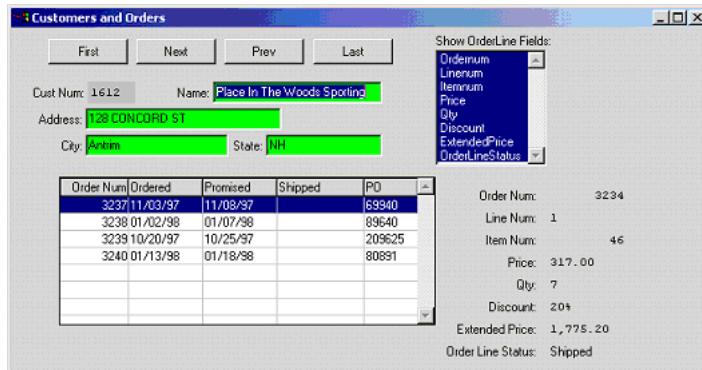
This places each displayed value at the same position, two positions to the right of the label.

There's a third change you have to make as well. As discussed in the “Object sizing” section on page 18–32, the format calculation for the label is likely to provide a format somewhat larger than the actual display width. This is a deliberate adjustment Progress makes to provide a format for a field that is large enough to display most values without truncation. In the case of your labels, however, you might find that if you just use the label format to determine the width, the display width is a bit too large and overwrites the beginning of some of the displayed values with blanks. To correct this, you need to specify an accurate WIDTH-CHARS attribute value for the label, so that it is just large enough to display itself without truncation but not so large that it overwrites the value that follows it.

3. Use the same FONT-TABLE method to calculate the WIDTH-CHARS of the text label object, adding this assignment to the CREATE TEXT statement:

```
WIDTH-CHARS = FONT-TABLE:GET-TEXT-WIDTH-CHARS(hBuffField:LABEL + ":")
```

4. Run the window. With these changes, you see a different display where the labels of the dynamic fields are colon-aligned:



5. Select another row in the browse, and the field values are set to blank by the VALUE-CHANGED trigger on the browse. What you see, however, is that because the displayed fields are not all CHARACTER fields, Progress applies the field's format to the blank value, which in the case of a numeric field is the value zero, and is displayed in various ways by the different field formats:



Clearly this procedure is an interesting example of using dynamic fields but not a terribly useful application window. You would probably want to be able to select a list of fields once and use them to display **OrderLine** values for any **Order** you select, rather than having the fields blanked out after one **OrderLine** is displayed. And there must be a way to navigate through the **OrderLines** of an **Order** rather than just seeing the first one. Feel free to extend the test procedure to provide these abilities.

Since there's room in the window to display all the **OrderLine** fields, the whole notion of making them dynamic fill-ins is also of limited use. A more realistic example of how you can use this technique in an application would be for a table with a great many fields, only a few of which each user needs to work with. After you learn about dynamic data management objects in the next chapter, you will also be able to allow the user to select a completely different table to display in place of the **OrderLines**.

Field format versus width

The relationship between format and width that the example highlights might seem confusing. Keep these basic guidelines in mind:

- The **FORMAT** is the maximum number of characters the user is allowed to type into the field at run time.
- The **WIDTH-CHARS** (or **WIDTH-PIXELS** if you measure your objects that way) is the actual screen real estate allotted to the object, whether it's for display or for data entry purposes. You can use the **FONT-TABLE** methods to calculate a precise width for a single displayed value, but you must estimate an appropriate width for a field that can be used to display many values, for example as you navigate through many records in a table.
- If you assign a format but not a width, Progress estimates a width based on the format. This normally is slightly greater than the width as based on **WIDTH-CHARS**, because Progress tries to provide additional space to prevent truncation of, in particular, short values with wide characters (such as a two-character capitalized state abbreviation, for instance). Therefore, you must assign an explicit width (using **WIDTH-CHARS** or **WIDTH-PIXELS**) if you want to make sure the display space isn't bigger than it needs to be, for example to prevent overwriting an object displayed next to the object you're sizing.

- If, on the other hand, you assign a width but not a format, Progress uses a default display format for the data type. For a CHARACTER field, for example, this is “X(8)”. This is not likely to be appropriate, so you almost always need to assign a format.
- If the format is larger than the display width for a CHARACTER fill-in field that is enabled for input, the user can type as many characters as the format allows. If the display space runs out, the fill-in automatically scrolls to allow the user to enter more data. This is very often a desirable user interface design, which displays a reasonable number of characters for a field but allows the user to type more when necessary (and have it all stored in the underlying field).
- If the format for a CHARACTER fill-in is *smaller* than the display width and the user tries to type more characters than the format allows, Progress prevents this. The user gets only a warning bell as he continues to type, even though there is still visible display space available that was allocated by the width attribute. This is a very *bad* user interface design, which leads to the following basic guiding principle.
- *When in doubt, make the format generously large and the width appropriate for the display.* Limit the format size for enabled fields only when it is important to limit the number of characters stored in the underlying variable or database field. Remember that all Progress CHARACTER variables and fields are inherently variable width, so the format of a database field does not allocate a fixed storage size, but only what a particular field value uses for each record.

Using multiple windows

As you've already learned, there is no `DEFINE WINDOW` statement in Progress. Any windows you build for your application are dynamic windows. This section summarizes some of the window handles and attributes that can be useful to you in designing an application with multiple windows.

You create a window with the `CREATE WINDOW` statement, which has the same syntax as every other `CREATE` statement you've seen.

Window system handles

The simple examples in this book that don't use a window you create (or that is created for you in code the AppBuilder generates) use the default window that is part of every session. This has a system handle called `DEFAULT-WINDOW`. This handle is not something you would ordinarily use in a real application.

You've seen the `CURRENT-WINDOW` system handle, which holds the handle of the window used by default for parenting frames, dialog boxes, and message alert boxes. The `CURRENT-WINDOW` attribute of a procedure overrides `CURRENT-WINDOW` for the context of that procedure only, without changing the value of the session-wide system handle. The statements in the standard AppBuilder window template that set both `CURRENT-WINDOW` and the `CURRENT-WINDOW` procedure attribute to the procedure's window provide a good default for parenting of objects created and used in that procedure.

Another useful system handle is `ACTIVE-WINDOW`, which holds the handle of the window that has received the most recent input focus in the application. This handle can help you assure that a dialog box or message alert box appears parented to the window where the user is currently working, even if it is not the current window of the procedure that executes the code to display the dialog box or message.

Useful window attributes, methods, and events

[Table 18–1](#) describes the numerous attributes, methods, and events that you can use to control the appearance and behavior of windows in your application.

Table 18–1: Window attributes, methods, and events

(1 of 2)

| Type | Name | Description |
|-----------|--|---|
| Attribute | TITLE | Specifies the window title. |
| | HEIGHT-CHARS WIDTH-CHARS HEIGHT-PIXELS WIDTH-PIXELS | Specifies the standard height and width of the window. |
| | MIN-HEIGHT-CHARS MIN-WIDTH-CHARS MIN-HEIGHT-PIXELS MIN-WIDTH-PIXELS | Specifies the minimum height and width of the window. |
| | MAX-HEIGHT-CHARS MAX-WIDTH-CHARS MAX-HEIGHT-PIXELS MAX-WIDTH-PIXELS | Specifies the maximum height and width of the window. |
| | VIRTUAL-HEIGHT-CHARS VIRTUAL-WIDTH-CHARS VIRTUAL-HEIGHT-PIXELS VIRTUAL-WIDTH-PIXELS | Specifies the maximum display area of the window. |
| | MESSAGE-AREA MESSAGE-AREA-FONT | Defines a message area and its font at the bottom of the window, where messages that are not qualified with the VIEW-AS ALERT-BOX phrase are displayed. |
| | STATUS-AREA STATUS-AREA-FONT | Defines a status area and its font at the bottom of the window. |
| | SCROLL-BARS | Defines whether scroll bars appear when the window is resized. |
| | MENUBAR, POPUP-MENU | Associates a menu or pop-up menu with a window. |
| | PARENT | Establishes parent-child relationships between windows. |

Table 18–1: Window attributes, methods, and events

(2 of 2)

| Type | Name | Description |
|--------|-------------------|--|
| Method | LOAD-ICON() | Takes the name of an image file with the .ico extension and displays the image in the title bar of the window, in the task bar when the window is minimized, and when the user selects the window using ALT-TAB. |
| | LOAD-SMALL-ICON() | Takes the name of an image file for an icon in the title bar or the task bar. The .ico file for either method can be an image of 16x16 or 32x32 pixels. |
| Event | WINDOW-MINIMIZED | Fires when a window is minimized. |
| | WINDOW-MAXIMIZED | Fires when a window is maximized. |
| | WINDOW-RESTORED | Fires when a window is restored after being minimized. |
| | WINDOW-RESIZED | Fires when a keyboard or mouse event starts to change the window's size. The window's RESIZE attribute must be true for this to occur. |
| | WINDOW-CLOSE | Fires when the user selects the standard icon used to close the window. In fact, Progress does not take any action automatically when the WINDOW-CLOSE event occurs—it does not even close the window! For this reason, you need to define a trigger for the WINDOW-CLOSE event, as described in the next section. |

WINDOW-CLOSE event example

The simplest example of an action on the WINDOW-CLOSE event is this:

```
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

If the WAIT-FOR statement is the last executable statement in a nonpersistent procedure, the event satisfies the WAIT-FOR and the procedure terminates.

The AppBuilder window template uses indirection to direct the WINDOW-CLOSE event for the window to the procedure itself, using this standard ON WINDOW-CLOSE trigger:

```
DO:  
  /* This event will close the window and terminate the procedure. */  
  APPLY "CLOSE":U TO THIS-PROCEDURE.  
  RETURN NO-APPLY.  
END.
```

You've already seen the rest of the steps in this sequence, with the CLOSE event on the procedure running `disable_UI` and deleting the procedure itself.

Creating window families

By default, when you create a window, Progress parents that window transparently to the window system. In this way, windows you create are siblings of each other. You can also parent a window to another window by setting the one window's PARENT attribute to the handle of the other. Windows that are parented to another window form a *window family*. A window parented directly to the window system can be called the *root window* of a window family. Windows parented by any child window, in turn, form a *child window family*. A window can be parented to only one other window at a time, but can have multiple child windows.

Window families share a number of properties that make them convenient for both applications and users to manage:

- **Coordinated viewing and hiding** — When you view any member of a window family (by setting its VISIBLE attribute to true, setting its HIDDEN attribute to false, or using the VIEW statement), the whole window family is viewed unless the HIDDEN attribute is true for at least one other member. If HIDDEN is true for a parent or other ancestor of the window that is viewed, no windows in the family are viewed. The only effect is that the HIDDEN attribute for the window you try to view is set to false. If, on the other hand, any child or descendent window has HIDDEN set to true, then all the windows in the family are viewed, except the descendent window and its descendants. When you hide a member of a window family by setting its VISIBLE attribute to false, that window and all its descendants are hidden, but their HIDDEN attributes are not changed. In this way, you can hide and view an entire family of windows by changing the VISIBLE attribute of the top window. You can leave any window or part of a hierarchy of windows out of the hiding and viewing by setting the HIDDEN attribute of the window at the head of that part of the hierarchy to true.
- **Coordinated minimizing and restoring** — When you minimize a window, all of its descendants disappear from view, unless they are already minimized. Any minimized descendants appear separately in the taskbar and can be restored individually. When you restore a parent window, any of its hidden descendants are redisplayed.
- **Coordinated close events** — If a parent window receives a WINDOW-CLOSE event, it propagates the PARENT-WINDOW-CLOSE event to all of its descendent windows. Progress supports these as two separate events so that a procedure can react in a special way to the parent of its window being closed. The WINDOW-CLOSE event does not propagate any events upward to its parent window.

Creating a dynamic browse

The most complex graphical object is the browse object. You can create a browse dynamically and specify programmatically all its attributes, including what table and query's records are displayed, what columns it displays, which columns are enabled for input, and its visible attributes such as size and position.

You can assign most of a dynamic browse's attributes in the `CREATE BROWSE` statement:

```
CREATE BROWSE browse-handle ASSIGN attribute = value . . .
```

In addition, you can specify some attribute values individually following the `CREATE` statement:

```
browse-handle:attribute = value
```

Here are some of the principal attributes you can set either in the `CREATE BROWSE` statement or in a separate assignment on the browse handle:

- **FRAME** — You must associate the browse with a frame it is visualized in.
- **X and Y, or ROW and COLUMN** — You should specify a position for the browse. Otherwise, it is positioned in the upper-left corner of the frame. As for other objects, X and Y are in pixels, ROW and COLUMN are in character rows.
- **WIDTH or WIDTH-PIXELS** — You should specify a width for the browse using the `WIDTH` attribute in characters or the `WIDTH-PIXELS` attribute in pixels.
- **HEIGHT, HEIGHT-PIXELS or DOWN** — You should specify a height for the browse using either the character `HEIGHT` or `HEIGHT-PIXELS` attribute or the `DOWN` attribute. `DOWN` specifies the number of rows to display. Keep in mind that if you use the `HEIGHT` or `HEIGHT-PIXELS` attribute, then any horizontal scrollbar uses part of that height. The overall height of the browse remains the same whether there is a horizontal scrollbar or not. If you specify `DOWN`, that determines the number of rows of data to display, and any horizontal scrollbar is added to that height.
- **ROW-HEIGHT** — Set this decimal attribute if you want each row to be different from the default, which is calculated based on the font.
- **SENSITIVE** — You should generally make a browse sensitive so that the user can use the scroll bar.

- **QUERY** — You must provide the browse with the handle of a query that provides its data to display.
- **VISIBLE or HIDDEN** — You must set either HIDDEN to false, if you want the browse to be viewed when its parent frame is viewed, or VISIBLE to true to force the frame and the browse to be viewed.
- **READ-ONLY** — Set this option to true if you are not going to enable columns in the browse.
- **SEPARATORS** — Set this option to true if you want lines between the columns and rows of the browse.
- **ROW-MARKERS** — Set this option to false if you do not want row markers at the beginning of each row.
- **NO-VALIDATE** — Set this option to true to prevent Progress from compiling field-level validation phrases from the schema into the browse.
- **TITLE** — Set this string if you want the browse to have a title bar.
- **MULTIPLE** — Set this option to true if you want to enable multiple selection of rows.

Some attributes can only be assigned before the browse is realized. In the case of a dynamic browse, this is generally when the browse is made visible, which you can do in one of two ways:

- By setting its VISIBLE attribute to true, which forces its containing frame to be viewed.
- By setting its HIDDEN attribute to false, which views the browse when its parent frame is viewed.

In a CREATE BROWSE statement, the attributes of the browse in the ASSIGN list are assigned in sequence. So, as this single statement is executed, if Progress encounters the VISIBLE = TRUE or HIDDEN = FALSE phrases (and its parent frame is visible), the browse is realized at that time. If you try to assign an attribute later in the statement, or in a separate statement, which has to be assigned before the browse is realized, you get an error at run time. For example, if you don't want the browse to have row markers at the beginning of each row, you set the ROW-MARKERS attribute to FALSE. If you put this assignment into the CREATE statement after an attribute that realizes the browse, you get the error shown in [Figure 18–15](#).

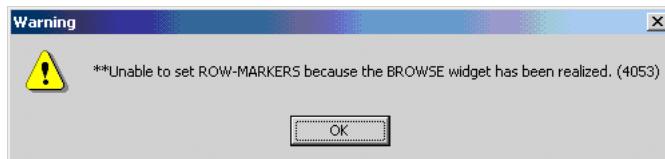


Figure 18–15: ROW-MARKERS error message

If this occurs, you have to reorder the assignments in the CREATE BROWSE statement. Generally, you should put the assignment that realizes the browse at the end of the statement or in a separate attribute assignment statement.

In the same way, you can modify most attributes in separate statements after the browse has been realized, except for attributes such as ROW-MARKERS.

The one thing you cannot set within the CREATE BROWSE statement is the list of columns to display. There are three methods on the browse handle that set the column list after the browse has been created.

ADD-COLUMNS-FROM method

If you want all or most of the columns from the query's buffer to be displayed in the browse, use the ADD-COLUMNS-FROM method:

```
browse-handle:ADD-COLUMNS-FROM (buffer-name [ , except-list ] ).
```

If you specify a comma-separated *except-list* expression, the columns in the list are not included in the browse. All other columns from the *buffer-name* are included. If the browse's query uses more than one buffer, you can invoke ADD-COLUMNS-FROM more than once on different buffers used by the query.

ADD-LIKE-COLUMN method

If you want to add columns individually, you can use the ADD-LIKE-COLUMN method:

```
browse-handle:ADD-LIKE-COLUMN (field-name-expr | buffer-field-handle)
```

This method adds one column at a time to a browse, based on a field name string expression or field handle in a buffer. You can add any number of columns to a browse by making successive calls to this method.

ADD-CALC-COLUMN method

The ADD-CALC-COLUMN method creates a single column based on a list of specified properties rather than deriving it from a specific field in a buffer. This is typically used as a placeholder column for a calculated value:

```
[ column-handle = ] buffer-handle:ADD-CALC-COLUMN
  (datatype-exp , format-exp , initial-value-exp , label-exp [ , pos ] )
```

Here are descriptions of the ADD-CALC-COLUMN options:

- **column-handle** — The handle of the column object returned from the method. You might want to capture this value to associate the column with a ROW-DISPLAY trigger that populates it.
- **datatype-exp** — A literal string or character expression evaluating to the data type for the column.
- **format-exp** — A string or character expression evaluating to the format for the column.
- **initial-value-exp** — A string or expression evaluating to the initial value for the column.
- **label-exp** — A string or expression evaluating to the column label for the column.
- **pos** — The integer position of the new column within the browse display list. If you do specify *pos*, the column goes at the end of any columns already defined.

Notes on dynamic browses and browse columns

Here are some facts to consider when you create and use dynamic browses and browse columns:

- You can set attributes on any of the columns in a dynamic browse after you have added them.
- You can use all three of the ADD methods to add more columns to a static browse, as well as a dynamic browse. You can take advantage of this to add columns to a browse that already has some columns statically defined, or you can define a static browse with no columns at all as a placeholder for a browse whose query and columns are defined dynamically at run time. In this way, you can define some of the visual attributes of the browse statically and then fill in the definition at run time.
- You cannot specify the CAN-FIND function in the validation expression for a dynamic browse column. This is a principal reason why you normally want to set the NO-VALIDATE browse attribute to YES for a dynamic browse, in case there are any such expressions in the field-level validation inherited from the schema.
- You can set the VISIBLE attribute for a browse column as well as for the browse as a whole.
- If you use the ADD-CALC-COLUMN method to create one or more columns to hold calculated values, you must define the ROW-DISPLAY trigger where the value of the calculated column is set prior to adding the column, so that the trigger is already established before the column is added. This is to ensure that the initial viewport of the calculated column is correctly populated.
- A dynamic browse, as well as a static browse to which you add columns dynamically, automatically becomes a NO-ASSIGN browse. This means that you do not get any default update handling from Progress. You must take care of capturing changes to the browse data and applying those changes in procedure code.
- You can change the query of either a dynamic or a static browse, even if the underlying fields are not the same as those of the original query. However, if the underlying fields are not the same, all browse columns are removed and you have to specify the columns again using the ADD-COLUMNS-FROM, ADD-LIKE-COLUMN, and ADD-CALC-COLUMN methods. If you set the QUERY attribute of a browse to the Unknown value, all browse columns are removed.

Extending the sample procedure with a dynamic browse

If you start with h-CustOrderWin7.w, you can add a dynamic browse to it to illustrate some of the capabilities of this object.



To show all the OrderLines for the current Order in a browse:

1. In the **Definitions** section, add these definitions:

```
/* These variables are used just by the dynamic browse variation  
of this trigger. */  
DEFINE VARIABLE hBrowse    AS HANDLE      NO-UNDO.  
DEFINE VARIABLE hCalcCol  AS HANDLE      NO-UNDO.  
DEFINE QUERY qOrderLine FOR OrderLine SCROLLING.
```

The **hBrowse** variable will hold the handle of the dynamic browse you create, and **hCalcCol** will hold the handle of a dynamic calculated column for it. The query definition will be attached to the browse when you create it. These definitions need to be here at the top level of the procedure so that their values persist beyond the end of the trigger where they are assigned.

2. Add this statement to the end of the **initSelection** internal procedure, which sets the list of **OrderLine** fields in the selection list:

```
OlineFields:ADD-LAST("Price B4 Disc").
```

The new entry represents a calculated column that holds the **OrderLine** price before the discount is applied.

3. In the **LEAVE** trigger for the selection list called **OlineFields**, comment out all of the code.

This code created dynamic fields to display an **OrderLine**. Now you'll display a browse in the same place.

4. Because you can create a browse over and over again with different lists of columns, you should first check to see if there's already a dynamic browse and delete it:

```
/* This block of code is the second version of the trigger, which creates
   a dynamic browse to show the selected fields. */

IF VALID-HANDLE(hBrowse) THEN
  DELETE OBJECT hBrowse.
```

5. Add new code to create a dynamic browse. This complex statement defines the browse and its attributes:

```
CREATE BROWSE hBrowse
  ASSIGN FRAME = FRAME CustQuery:HANDLE
    WIDTH = 50
    DOWN = 6
    ROW = 8
    COL = 82
    ROW-MARKERS = NO
    SENSITIVE = TRUE
    SEPARATORS = TRUE
    READ-ONLY = FALSE
    NO-VALIDATE = YES
    VISIBLE = TRUE
    QUERY = QUERY qOrderLine:HANDLE.
```

6. If necessary, adjust the size and position of the browse according to the layout of your own window and **Order** browse.
7. Add code that walks through the selected fields from the **OlineFields** selection list as the dynamic fill-in code does:

```
cFields = OLineFields:SCREEN-VALUE.
DO iField = 1 TO NUM-ENTRIES(cFields):
  cField = ENTRY(iField, cFields).
```

8. If the selected field is the calculated field, you must add a calculated column to the browse to display it:

```
IF cField = "Price B4 Disc" THEN
    hCalcCol = hBrowse:ADD-CALC-COLUMN
    ("Decimal",
     ">,>>>,>>9.99",      /* Data type */
     "0",                      /* Format */
     "0",                      /* Initial value */
     "Price B4 Disc").        /* column label */
```

9. Otherwise, add a column like the selected field in the **OrderLine** table:

```
ELSE hBrowse:ADD-LIKE-COLUMN("OrderLine." + cField).
END.      /* END DO iField... */
```

10. Add this ROW-DISPLAY trigger, which executes each time a row is displayed in the browse:

```
ON ROW-DISPLAY OF hBrowse
  PERSISTENT RUN calcPriceB4Disc.
```

Because the code in the trigger block for LEAVE of **OlineFields** goes out of scope as soon as the trigger block ends, you must put the code for this nested trigger definition in a separate procedure, and use the special syntax PERSISTENT RUN calcPriceB4Disc to make Progress keep track of what to run when the ROW-DISPLAY event fires.

11. Add this LEAVE trigger to open the **OrderLine** query from the **Definitions** section:

```
OPEN QUERY qOrderLine FOR EACH OrderLine
  WHERE OrderLine.OrderNum = Order.OrderNum.
```

You attached this query to the browse in the CREATE BROWSE statement.

12. Add this code for the calcPriceB4Disc procedure:

```
/*
-----Procedure: calcPriceB4Disc
-----Purpose: Calculates the value of the calculated column PriceB4Disc.
-----Parameters: <none>
-----*/
IF VALID-HANDLE(hCalcCol) THEN
    hCalcCol:SCREEN-VALUE = STRING(OrderLine.Price *
OrderLine.Qty).

END PROCEDURE.
```

A trigger such as this should always check first that its column handle is valid in case it is fired once before the column is created. In this case, the column is optional, so it must always check first to see if it's there.

13. In the VALUE-CHANGED trigger for the **OrderBrowse**, comment out the existing code that blanks out the dynamic fill-ins and replace it with a statement that re-opens the **OrderLine** query:

```
/* This is the second version of this trigger, for the dynamic browse
example.*/
OPEN QUERY qOrderLine FOR EACH OrderLine WHERE OrderLine.OrderNum
= Order.OrderNum.
```

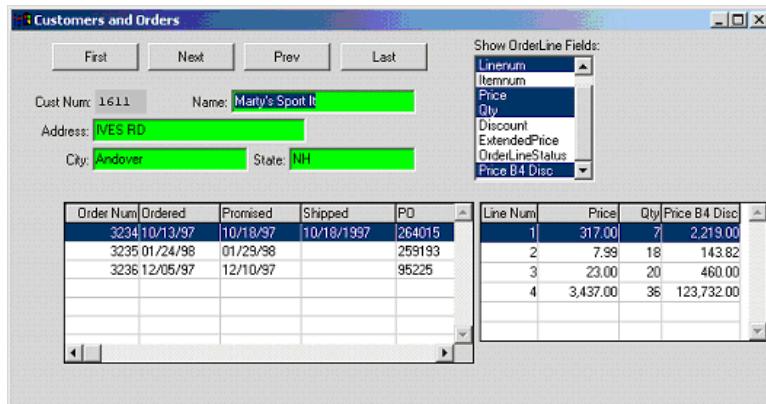
This code refreshes the **OrderLine** browse each time an **Order** is selected.

14. Add this statement to each of the four navigation button triggers to apply the **Order** browse's VALUE-CHANGED trigger:

```
DO:
    GET NEXT CustQuery.
    IF AVAILABLE Customer THEN
        DISPLAY Customer.CustNum Customer.Name Customer.Address Customer.City
            Customer.State
            WITH FRAME CustQuery IN WINDOW CustWin.
            {&OPEN-BROWSERS-IN-QUERY-CustQuery}
        APPLY "VALUE-CHANGED" TO BROWSE OrderBrowse.
    END.
```

The query opens when a new **Customer** is selected and its **Orders** first displayed.

15. Run h-CustOrderWin7.w. You can select some **OrderLine** fields along with the calculated field and see the results of your work:



Accessing the browse columns

Each of the browse columns is an object in its own right, with its own handle. If you want to set individual column attributes at run time (for example, to enable columns for update), you set those attributes through the handle. Static browse columns are objects with handles as well, but you can also access those by name. Dynamic browse columns, like other dynamic objects, have no name, so you must use their handle.

Note: To make a dynamic browse column updateable, set its READ-ONLY attribute to false.
Browse columns do not use the SENSITIVE attribute.

The browse FIRST-COLUMN attribute returns the handle to the first (leftmost) column in the browse. Each column's NEXT-COLUMN attribute returns the handle of the column next to it. For example, you can add this block of code to the OLineFields LEAVE trigger:

```
/* You can use code such as this to navigate through
   the columns of a browse dynamically. */
DEFINE VARIABLE hCol AS HANDLE      NO-UNDO.
DEFINE VARIABLE cCols AS CHARACTER NO-UNDO.
hCol = hBrowse:FIRST-COLUMN.
DO WHILE VALID-HANDLE(hCol):
   cCols = cCols + CHR(10) + hCol:LABEL. /* CHR(10) is a line feed. */
   hCol = hCol:NEXT-COLUMN.
END.
MESSAGE cCols.
```

When you run the procedure again and select fields, Figure 18–16 shows what you see.



Figure 18–16: Result of COLUMN example

Creating a dynamic menu

This final section shows you how to create a dynamic menu for a window. Dynamic menus share the same characteristics of static menus. As with other dynamic objects, you use CREATE statements rather than DEFINE statements to create a menu, submenu, or menu item. You parent these objects together in a hierarchy by setting their PARENT attribute. If you need to delete a dynamic menu before its widget pool is deleted, you use the DELETE OBJECT statement to delete the top-level menu object. Progress automatically deletes all its child submenus and menu items.

When you define a static menu, you need to define its elements in reverse order, so that you define the submenus before you reference them in the static DEFINE MENU statement. With dynamic menus you do the opposite. You first define the top-level menu, then its submenus, and then each submenu's menu items. As you create each one, you parent it to the next level up to establish the menu hierarchy.

Creating a menu

To create a menu, use this statement:

```
CREATE MENU menu-handle [ ASSIGN attribute = value [ . . . ] ] .
```

As with other dynamic objects, you can assign one or more attributes as part of the CREATE statement or in separate statements that use the menu handle. Typical menu attributes include:

- **POPUP-ONLY** — Set this to true to create a pop-up menu. The default value is false, which means that Progress creates the menu as a menu bar.
- **TITLE** — This is settable only for pop-up menus.
- **SENSITIVE** — Set this to false to disable the menu.

To associate a menu with a window, set the MENUBAR attribute of the window handle to the menu handle.

Creating a submenu

To create a submenu, use the CREATE SUB-MENU statement:

```
CREATE SUB-MENU submenu-handle [ ASSIGN attribute = value [ . . . ] ]  
[ trigger-block ] .
```

Typical attributes you can assign for a submenu include:

- **LABEL** — Every submenu must have a unique label.
- **PARENT** — Assign the PARENT attribute to the handle of the menu or parent submenu in a hierarchy.
- **SENSITIVE** — Set this to false to disable the submenu.

You can also define a trigger block (typically for a CHOOSE trigger) as part of the CREATE statement.

Creating menu items

In a dynamic menu, menu items are individually created objects with their own handles. Because you can delete an entire menu by deleting the top-level menu, or by deleting the procedure that controls the widget pool for the menu, you can normally use the same object handle for all your menu items. Each one must be parented to the submenu that contains it. Therefore, each menu item can be used in only one menu or submenu. Typical menu item attributes include:

- **LABEL** — The text label for the menu item.
- **PARENT** — The parent submenu handle.
- **SENSITIVE** — Set this to false to disable the menu item.
- **SUBTYPE** — The default SUBTYPE is NORMAL, for standard selectable menu items. Other valid values are READ-ONLY (for a read-only menu item), SKIP (for a blank line in the menu), or RULE (to create a line between menu items).

Navigating the hierarchy of menu handles

For both menus and submenus, your procedure code can traverse all the menu items and nested submenus. The FIRST-CHILD attribute of a menu or submenu returns the handle of its first menu item or child submenu. The NEXT-SIBLING attribute of each menu item or submenu returns the handle of the next object at that level. Object handles are returned in left-to-right order for sibling submenus, and top-to-bottom order for menu items. You can navigate in reverse by starting with the LAST-CHILD attribute and following the PREV-SIBLING chain. These attributes are readable only. Their values are established when you assign parents to each submenu and menu item.

You can also add dynamic submenus and menu items to an existing static menu. To do this, navigate the menu using these CHILD and SIBLING attributes to locate the handle of the object in the menu where you want to attach another submenu or menu item. Create the new objects and parent them to the existing object. You can parent a new submenu to the menu bar or to another submenu. You can parent a new menu item to an existing submenu. New objects are added to the end of the list of current objects at that level (that is, to the right of existing submenus in a parent menu or submenu) or to the bottom of the list of menu items for a submenu.

Adding a dynamic menu to the test window

In this section, you'll add a dynamic menu to the test window. The menu displays all **SalesReps** and lets the user filter the list of **Customers** by a selected **SalesRep**. This example demonstrates one typical use of dynamic menus—to create a list of menu items that are data-driven. In other cases, a list of dynamic menu items might be based on currently available windows or other elements of an application that can vary at run time.



To add a dynamic menu bar to the h-CustOrderWin7.w procedure.

1. Add a new fill-in field to the window to display the selected **SalesRep**. Name it **cSalesRep**.
2. Create a new internal procedure called **createMenu**. The procedure needs handle variables for the menu bar, its submenu, and one handle for all the menu items you create:

```
DEFINE VARIABLE hMenu      AS HANDLE      NO-UNDO.
DEFINE VARIABLE hSubMenu   AS HANDLE      NO-UNDO.
DEFINE VARIABLE hMenuItem AS HANDLE      NO-UNDO.
```

3. Create the menu bar and its submenu:

```
CREATE MENU hMenu.

CREATE SUB-MENU hSubMenu
  ASSIGN PARENT = hMenu
  LABEL = "SalesReps".
```

4. Create the list of **SalesReps** as menu items:

```
FOR EACH SalesRep:
  CREATE MENU-ITEM hMenuItem
    ASSIGN PARENT = hSubMenu
    LABEL = SalesRep.SalesRep + " " + SalesRep.RepName
  TRIGGERS:
    ON CHOOSE PERSISTENT RUN filterCust IN THIS-PROCEDURE.
  END TRIGGERS.
END.
```

Each one is parented to the submenu. You can reuse the same object handle for each one because you won't need to reference those objects individually. Each menu item has the **SalesRep** initials followed by the **SalesRep**'s full name as its label.

When the user chooses one of these items, you want to reopen the **Customer** query for just **Customers** of that **SalesRep**. Remember that when you define a trigger on an object inside an internal procedure or another trigger block, the trigger definition doesn't persist beyond the end of that procedure or block. Therefore, you have to use the special PERSISTENT RUN statement to tell Progress to run a separate procedure when the event occurs. You'll write this `filterCust` procedure in a moment.

5. Add a couple of special menu items to the end of the list. Create a RULE to separate the **SalesReps** from the final menu item:

```
CREATE MENU-ITEM hMenuItem
  ASSIGN SUBTYPE = "RULE"
  PARENT = hSubMenu.
```

6. Add a menu item that closes the window and its procedure:

```
CREATE MENU-ITEM hMenuItem
  ASSIGN PARENT = hSubMenu
  LABEL = "E&xit"
  TRIGGERS:
    ON CHOOSE PERSISTENT RUN leaveProc IN THIS-PROCEDURE.
  END TRIGGERS.
```

Once again, the trigger code has to be in a separate procedure.

7. Parent the menu bar to the window:

```
CURRENT-WINDOW:MENUBAR = hMenu.
```

8. Create the `filterCust` procedure for the **SalesRep** items:

```
/*
-----  
Procedure: filterCust  
Purpose: Filters customers by the selected SalesRep menu item.  
-----*/  
OPEN QUERY CustQuery FOR EACH Customer  
    WHERE Customer.SalesRep = ENTRY (1, SELF:LABEL, " " ).  
cSalesRep:SCREEN-VALUE IN FRAME CustQuery = SELF:LABEL.  
APPLY "CHOOSE" TO BtnFirst IN FRAME CustQuery.  
END PROCEDURE.
```

This code reopens the **Customer** query for just those **Customers** whose initials match the first part of the menu item label. Remember that the built-in handle **SELF** always evaluates to the object handle that triggered the event.

The procedure displays the selected **SalesRep** in the new fill-in field, and then resyncs the display to the first record in the new query by programmatically choosing the **First** button.

9. Define the internal procedure `leaveProc` to handle the **Exit** button:

```
/*
-----  
Procedure: leaveProc  
Purpose: Handles the Exit menu item  
-----*/  
APPLY "CLOSE" TO THIS-PROCEDURE.  
END PROCEDURE.
```

This code simply invokes the CLOSE event already defined as a standard part of the AppBuilder-generated code for the window.

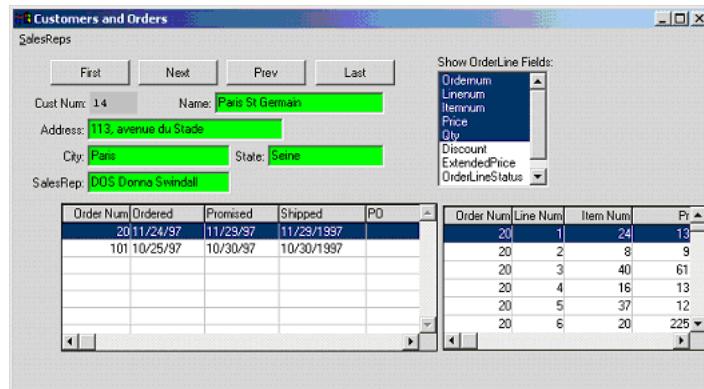
10. Add a statement to the procedure's main block to run `createMenu`:

```
RUN enable_UI.  
RUN createMenu.  
RUN h-StartSuper.p("h-dynsuper.p").  
RUN changeFields.
```

11. Run the procedure again. Now you can drop down a list of all the **SalesReps**:



12. Select one. Now **Customers** are selected for that **SalesRep** only:



Summary

In this chapter, you learned how to use the CREATE statement and handles to create and manage visual objects your application needs at run time. The next two chapters extend the discussion to cover dynamic forms of the data management objects as well: queries, buffers, and temp-tables. You'll also look at how to create a dynamic browse to display the contents of a dynamic temp-table.

Using Dynamic Queries and Buffers

In the previous chapter you learned how to create dynamic versions of visual objects to increase the flexibility of your application's user interface. You also learned how to use handles to access the attributes and methods of those objects as well as their static counterparts. In this chapter you'll learn how to do the same with data management objects such as queries and buffers. The next chapters continue with a discussion of temp-tables and the browse. The same principles apply to visual objects, so the concepts should already be familiar to you.

Using a dynamic query, you can postpone until run time the tasks of defining what table or tables a query should use, and what the selection criteria should be for the result set the query manages. Using a dynamic buffer, you can likewise allow the application to identify at run time what table a buffer should store data for.

Putting these dynamic objects together with dynamic temp-tables and browses gives you the ability to define almost any aspect of your application at run time, when this is necessary. Although you shouldn't make all your queries and buffers dynamic any more than you should make all your user interface objects dynamic, having the ability to do so allows you to build general purpose tools for certain parts of your application that might need to deal with a wide variety of user requirements in a consistent way. Being able to do this from a single procedure with dynamic objects is greatly preferable to writing and compiling many different procedures that do essentially the same job for different tables or fields.

It's important to note that these dynamic features are not meant to replace all the static statements in earlier versions of the Progress 4GL, or to be the standard way to write code for all new applications. There is a significant performance cost to using dynamic data management statements, which can be well worth the cost when the flexibility is needed but an unnecessary expense and complication, otherwise. If your tables, fields, and queries are known and fixed when you write the procedure, then there is no need to use dynamic statements to manage them. Likewise, if you have a static object, such as a query, on a specific table or tables but need to modify just the WHERE clause at run time or access some of its attributes and methods, you can easily do this through a handle attached to the static object. This keeps your code simpler and more efficient.

This chapter and the one that follows include straightforward but fairly comprehensive examples that show the power of dynamic data management objects, allowing you to browse data from any database table or temp-table you care to define. By the end of these two chapters, you'll have a solid understanding of some of the most powerful features in the Progress 4GL and how best to use them in your applications.

This chapter includes the following sections:

- [Using dynamic queries and query handles](#)
- [Using dynamic buffers and buffer handles](#)

Using dynamic queries and query handles

As with other dynamic objects, you can create a dynamic query with a form of the CREATE statement. But also as with other object types, you can attach a handle variable to a static query and use almost all of its methods and attributes to manipulate the query at run time.

The syntax for the CREATE QUERY statement is:

```
CREATE QUERY query-handle [ IN WIDGET-POOL pool-name ] .
```

There is no ASSIGN option on the CREATE QUERY statement, primarily because most of the operations you can perform on a dynamic query are methods, not attributes.

You can also attach a handle variable or field to a static query:

```
query-handle = QUERY query-name:HANDLE.
```

Query methods and attributes

Once you create a dynamic query or assign a handle to a static one, you can retrieve (and, in some cases, set) information about the query using its methods and attributes. This section provides a summary of those methods and attributes. Unless otherwise noted, they all apply to handles for both static and dynamic queries.

SET-BUFFERS method

This method takes a buffer handle or a comma-separated list of buffer handles and sets the query's buffer list to those buffers. It returns true if the operation succeeded and false otherwise.

You can pass a buffer reference in one of several ways:

- Using the BUFFER *buffer-name*:HANDLE syntax, which provides a handle for a static buffer on a known table.
- Using a handle variable or field that you associate with a static buffer in a separate statement, such as `hBuffer = BUFFER buffer-name:HANDLE`.
- Using the handle for a dynamic buffer, where you name the table only at run time. You'll learn about how to do this in the “[Extending the test window to use a buffer handle](#)” section on page 19–30.

For example, this sequence of statements creates a query and sets its buffer list to the **Order** and **Customer** buffers:

```
DEFINE VARIABLE hQuery AS HANDLE      NO-UNDO.  
  
CREATE QUERY hQuery.  
hQuery:SET-BUFFERS(BUFFER Order:HANDLE, BUFFER Customer:HANDLE).
```

This is the equivalent of defining a static query for those buffers, as you could do with this statement:

```
DEFINE QUERY OrderCust FOR Order, Customer.
```

You could then assign a handle to the static query in the same way, for example:

```
DEFINE VARIABLE hQuery AS HANDLE      NO-UNDO.  
DEFINE QUERY OrderCust FOR Order, Customer.  
  
hQuery = QUERY OrderCust:HANDLE.
```

This raises the basic question of when to use static and when to use dynamic queries. Typically, unless your procedure is of such general use that you do not know the tables or buffers it will use until run time, you can define a static query and then use its handle to modify it as needed at run time. This is the case if, for example, you need to modify the WHERE clause or sort order of the query in a variety of ways at run time. If you start with a static query, you can use any combination of the static statements you're familiar with to manipulate it, such as GET FIRST, OPEN QUERY, and so forth, or their dynamic equivalents, which are introduced in the following section. If you start with a dynamic query, you can use only dynamic methods to manipulate it.

The one aspect of a static query you cannot modify at run time is its buffer list, so the SET-BUFFERS method applies only to dynamic queries.

ADD-BUFFER method

This method takes a single buffer handle as an argument and adds it to the list of buffers for the query. In this way, you can have a programming loop that adds a sequence of buffers to a query one at a time, when that is appropriate and when your query needs to join two or more buffers. It returns true if the operation succeeded and false otherwise.

For example, you can use the ADD-BUFFER method to add a third buffer to the query that already has the **Order** and **Customer** buffers assigned to it:

```
DEFINE VARIABLE hQuery AS HANDLE      NO-UNDO.  
  
CREATE QUERY hQuery.  
hQuery:SET-BUFFERS(BUFFER Order:HANDLE, BUFFER Customer:HANDLE).  
hQuery:ADD-BUFFER(BUFFER SalesRep:HANDLE).
```

Like the SET-BUFFERS method, ADD-BUFFER is supported only for dynamic queries for the same reason that you cannot modify the buffer list of a static query.

NUM-BUFFERS attribute

This attribute returns the number of buffers for the query:

```
DEFINE VARIABLE hQuery AS HANDLE      NO-UNDO.

CREATE QUERY hQuery.
hQuery:SET-BUFFERS(BUFFER Order:HANDLE, BUFFER Customer:HANDLE).
hQuery:ADD-BUFFER(BUFFER SalesRep:HANDLE).
MESSAGE "This dynamic query has "
      hQuery:NUM-BUFFERS "buffers." VIEW-AS ALERT-BOX.
```

Figure 19–1 shows the result.

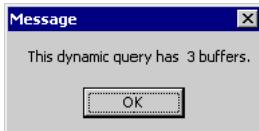


Figure 19–1: Number of buffers message

GET-BUFFER-HANDLE method

Given the number of buffers in the query, you can walk through them and retrieve each handle in turn, using the GET-BUFFER-HANDLE method, which takes the sequential buffer number as a parameter, as in this example:

```
DEFINE VARIABLE hQuery      AS HANDLE      NO-UNDO.
DEFINE VARIABLE iBufNum    AS INTEGER     NO-UNDO.
DEFINE VARIABLE cBufNames AS CHARACTER   NO-UNDO.

CREATE QUERY hQuery.
hQuery:SET-BUFFERS(BUFFER Order:HANDLE, BUFFER Customer:HANDLE).
hQuery:ADD-BUFFER(BUFFER SalesRep:HANDLE).
DO iBufNum = 1 TO hQuery:NUM-BUFFERS:
  cBufNames = cBufNames + hQuery:GET-BUFFER-HANDLE(iBufNum):NAME
  + " ".
END.
MESSAGE "This query uses buffers " cBufNames VIEW-AS ALERT-BOX.
```

Figure 19–2 shows the result.



Figure 19–2: Names of buffers message

NAME attribute

As you know, only static objects have names, so the NAME attribute only applies to static queries:

```
DEFINE VARIABLE hQuery AS HANDLE      NO-UNDO.  
DEFINE QUERY OrderCust FOR Order, Customer.  
  
hQuery = QUERY OrderCust:HANDLE.  
MESSAGE "This is a static query named " hQuery:NAME VIEW-AS ALERT-BOX.
```

Figure 19–3 shows the result.



Figure 19–3: Name of static query message

QUERY-PREPARE method

Once you've defined the buffers for a dynamic query, the next step is to provide it with a FOR EACH statement that it should use to retrieve and order data when you open the query. The QUERY-PREPARE method takes the FOR EACH statement as its argument. It returns true if the operation succeeded and false otherwise. You can use QUERY-PREPARE to define the record selection for a dynamic query or to change the selection for a static one. This sample code prepares the dynamic query for the **Order**, **Customer**, and **SalesRep** tables:

```

DEFINE VARIABLE hQuery      AS HANDLE      NO-UNDO.
DEFINE VARIABLE iBufNum    AS INTEGER      NO-UNDO.
DEFINE VARIABLE cBufNames AS CHARACTER    NO-UNDO.
DEFINE VARIABLE lSuccess   AS LOGICAL     NO-UNDO.

CREATE QUERY hQuery.
hQuery:SET-BUFFERS(BUFFER Order:HANDLE, BUFFER Customer:HANDLE).
hQuery:ADD-BUFFER(BUFFER SalesRep:HANDLE).
lSuccess =
  hQuery:QUERY-PREPARE("FOR EACH Order WHERE OrderStatus = 'Ordered', "
+ "FIRST Customer OF Order, "
+ "FIRST SalesRep OF Order "
+ "BY SalesRep").
IF NOT lSuccess OR ERROR-STATUS:NUM-MESSAGES NE 0 THEN
DO:
  /* Deal with possible errors in the Query Prepare. */
END.
```

There are a few important comments to make about this example:

1. Remember that when you use a one-to-one join in a FOR EACH statement for a query, you must include the FIRST or EACH keyword in each table phrase, as is shown in the following code example. Since there is just one **Customer** for each **Order**, and one **SalesRep** for each **Order**, there is no need to iterate through potentially multiple **Customers** or **SalesReps** for an **Order**, so the FIRST keyword suffices. In a FOR EACH statement in a block header, the FIRST keyword would be optional, as in this example:

```

FOR EACH Order WHERE OrderStatus = "Ordered",
  Customer OF Order, SalesRep OF Order:
  DISPLAY OrderNum Order.CustNum Customer.NAME SalesRep.RepName.
END.
```

2. If you know exactly what the WHERE clause and BY clause are when you write the procedure, you might not need to use dynamic constructs at all. More realistically, you use the QUERY-PREPARE method when you don't know the selection and sort criteria until run time. In the “Extending the sample window to filter dynamically” section on page 19–16, you'll build an extension to the **Customer and Orders** window that shows this flexibility more realistically.
3. Since you normally use the QUERY-PREPARE method in cases where the WHERE clause is truly variable, it is essential that you always assign the result of the method to a logical variable, and then check the value of that variable and the ERROR-STATUS:NUM-MESSAGES value to be sure that the prepare succeeded. If it doesn't and you continue without intercepting the error, your procedure generates a whole series of error statements as you attempt to open and use the query.

PREPARE-STRING attribute

Once you have prepared a query, you can verify what its current FOR EACH statement is using the PREPARE-STRING attribute, which returns the effect of the most recent QUERY-PREPARE method, as in this example:

```
MESSAGE hQuery:PREPARE-STRING VIEW-AS ALERT-BOX.
```

Figure 19–4 shows the result.



Figure 19–4: PREPARE-STRING message

The PREPARE-STRING attribute applies only to dynamic queries. It does not return the FOR EACH statement used in an OPEN QUERY statement for a static query. The example in the “Using the SELF handle to identify an object in a trigger” section on page 19–21 uses this attribute to save off the current FOR EACH statement for a query to restore it if the user's attempt to replace it with a different one fails.

QUERY-OPEN method

Once you have prepared a query you need to open it, using the QUERY-OPEN method:

```
hQuery:QUERY-OPEN()
```

The QUERY-PREPARE and QUERY-OPEN methods together accomplish what the OPEN QUERY statement does for a static query. The two methods are separated out to enable you to define the selection and sort criteria for a query separately from opening it. You might want to reopen the same query several times with the same FOR EACH statement, for example, to capture changes to the underlying data the FOR EACH statement retrieves. Having two separate methods gives you this flexibility. It also allows you to verify that the FOR EACH statement in a QUERY-PREPARE is valid before you try to open the query.

You can use the QUERY-PREPARE and QUERY-OPEN methods with static queries as well, even after the query has been opened one or more times with a static OPEN QUERY statement. Here's an example that extends the static query shown earlier:

```
DEFINE VARIABLE hQuery      AS HANDLE      NO-UNDO.
DEFINE VARIABLE cSalesRep AS CHARACTER   NO-UNDO.

DEFINE QUERY OrderCust FOR Order, Customer. /* Static definition */

hQuery = QUERY OrderCust:HANDLE.          /* Capture the handle */
OPEN QUERY OrderCust FOR EACH Order WHERE Order.SalesRep = "BBB",
    FIRST Customer OF Order. /* static OPEN */
MESSAGE "Static OPEN can't show PREPARE-STRING: " /* This is UNKNOWN */
hQuery:PREPARE-STRING VIEW-AS ALERT-BOX.
CLOSE QUERY OrderCust.
/* Code to ask the user for a SalesRep could go here... */
cSalesRep = "DKP".
/* Now use the dynamic methods to re-prepare and re-open the query. */
hQuery:QUERY-PREPARE("FOR EACH Order WHERE SalesRep = '" +
    cSalesRep + "'", FIRST Customer OF Order").

hQuery:QUERY-OPEN().
MESSAGE "QUERY-PREPARE sets PREPARE-STRING:" SKIP
hQuery:PREPARE-STRING VIEW-AS ALERT-BOX.
```

The first alert box, shown in [Figure 19–5](#), confirms that you can't use the PREPARE-STRING attribute on a static OPEN.

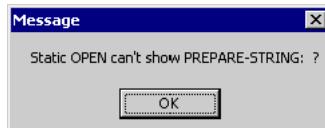


Figure 19–5: Static OPEN query message

But the one in [Figure 19–6](#) shows that you *can* use it when you prepare and open even a static query using the dynamic methods.

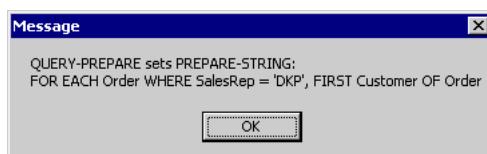


Figure 19–6: Dynamic methods with OPEN query message

Using the QUOTER function to assemble a query

Notice the use of the single quotation marks embedded in the strings that are assembled to build up the FOR EACH statement:

```
hQuery:QUERY-PREPARE("FOR EACH Order WHERE SalesRep = '" +
cSalesRep + "'", FIRST Customer OF Order").
```

The first literal ends with a single quote mark, which precedes the **SalesRep** initials. This, in turn, is followed by another literal that *begins* with a single quote mark to balance the first one. This kind of assemblage of strings with and without quote marks can get very tricky and lead to code that is hard to read and programming errors that are hard to identify. Here, for example, you need to remember not only to put the quote marks into the statement before and after the **cSalesRep** value, but also to use single quotes so that they don't interfere with the use of double-quotes around the literals themselves. You could reverse these and use double quotes inside single quotes, but either way you need to be sure to balance them and embed them properly.

Progress provides a helpful function to assist you in assembling these kinds of strings, the QUOTER function. If you insert values of any data type into a string, such as this QUERY-PREPARE as arguments to the QUOTER function, Progress assembles the string properly so that you don't need to worry about where the embedded quote marks go. For instance, here's an alternative version of the QUERY-PREPARE method that uses QUOTER:

```
hQuery:QUERY-PREPARE("FOR EACH Order WHERE SalesRep = " +  
    QUOTER(cSalesRep) + ", FIRST Customer OF Order").
```

In [Figure 19–7](#), Progress inserts quote marks around the **SalesRep** initials for you.

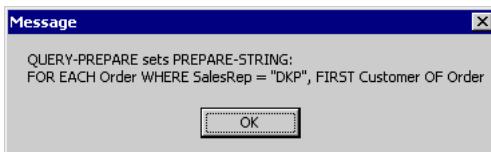


Figure 19–7: Example QUOTER function message

You can use QUOTER to avoid parsing errors for numeric values as well as string values. For example, DECIMAL formats in parts of the world outside the United States typically reverse the meaning of the decimal point or period/full stop (.) and comma (,) characters. If you use QUOTER on all such values, Progress assembles a query string that compiles properly without causing confusion to the syntax analyzer, which must interpret whether the “.” character is a decimal point, a separator between thousands in a numeric value, or the end of a statement.

QUERY-CLOSE method

There's a dynamic method to close a query as well:

```
hQuery:QUERY-CLOSE().
```

QUERY-CLOSE returns true if the operation succeeded and false otherwise. As with the CLOSE QUERY statement, you don't need to use QUERY-CLOSE if you are immediately again preparing and reopening a query, but you should close a query when you are done using it. You can freely mix and match static OPEN and CLOSE statements and QUERY-OPEN and QUERY-CLOSE methods for static queries but, as with all the query syntax, you can use the static statements only for static queries.

QUERY-OFF-END attribute

This attribute corresponds to the QUERY-OFF-END function for static queries. It returns true if the query is no longer positioned to a record in the result set, either because you have proceeded beyond the last row or moved backward beyond the first row.

IS-OPEN attribute

This LOGICAL attribute returns true if the query is open and false otherwise.

Navigation methods

There is a dynamic navigation method for each of the corresponding GET statements: GET-FIRST, GET-NEXT, GET-PREV, and GET-LAST. There is also a GET-CURRENT method that corresponds to the GET CURRENT statement, which again retrieves the current record from the database, normally to check to see whether it has been changed since you last read it.

These methods can take optional arguments that you can use to specify the lock mode (NO-LOCK, SHARE-LOCK, or EXCLUSIVE-LOCK) and wait mode (if it is NO-WAIT). The default lock mode is SHARE-LOCK. You will generally want to change this default to specify either NO-LOCK or EXCLUSIVE-LOCK, depending on whether you need to prepare for it to be changed and protect the record against changes by other users.

Here's a completion of the simple procedure used throughout this section, showing the QUERY-PREPARE, QUERY-OPEN, and GET-NEXT methods and the use of the QUERY-OFF-END attribute:

```

DEFINE VARIABLE hQuery      AS HANDLE      NO-UNDO.
DEFINE VARIABLE iBufNum     AS INTEGER      NO-UNDO.
DEFINE VARIABLE cBufNames   AS CHARACTER    NO-UNDO.
DEFINE VARIABLE lSuccess   AS LOGICAL     NO-UNDO.

CREATE QUERY hQuery.
hQuery:SET-BUFFERS(BUFFER Order:HANDLE, BUFFER Customer:HANDLE).
hQuery:ADD-BUFFER(BUFFER SalesRep:HANDLE).
lSuccess =
    hQuery:QUERY-PREPARE("FOR EACH Order WHERE OrderStatus = 'Ordered', "
        + "FIRST Customer OF Order, "
        + "FIRST SalesRep OF Order "
        + "BY SalesRep").
IF NOT lSuccess OR ERROR-STATUS:NUM-MESSAGES NE 0 THEN
DO:
    /* Deal with possible errors in the Query Prepare. */
END.
hQuery:QUERY-OPEN().
REPEAT WHILE NOT hQuery:QUERY-OFF-END:
    hQuery:GET-NEXT().
    DISPLAY Order.OrderNum Customer.NAME FORMAT "x(20)"
        SalesRep.RepName WITH FRAME OrderFrame 10 DOWN.
END.
hQuery:QUERY-CLOSE().
DELETE OBJECT hQuery.

```

Figure 19–8 shows the result.

| Order Num | Name | Rep Name |
|-----------|----------------------|------------------|
| 3991 | Finish Line | Brawn , Bubba B. |
| 3197 | Covered Bridge Sport | Brawn , Bubba B. |
| 2301 | Skater's Paradise | Brawn , Bubba B. |
| 1423 | Changes | Brawn , Bubba B. |
| 3867 | Back In The Game | Brawn , Bubba B. |
| 3198 | Covered Bridge Sport | Brawn , Bubba B. |
| 2300 | Skater's Paradise | Brawn , Bubba B. |
| 1696 | T & T Sports | Brawn , Bubba B. |
| 3866 | Back In The Game | Brawn , Bubba B. |
| 3148 | Inside Out | Brawn , Bubba B. |

Press space bar to continue.

Figure 19–8: Result of query methods and attributes example

Reposition methods

There are also reposition methods that match the corresponding REPOSITION statements you've seen earlier in the book. These include:

- **REPOSITION-FORWARD(*n*)** — Repositions the query *n* rows forward within the result set.
- **REPOSITION-BACKWARD(*n*)** — Repositions the query *n* rows backward within the result set.
- **REPOSITION-TO-ROW(*n*)** — Repositions the query to row *n* within the result set.
- **REPOSITION-TO-ROWID(*rowid* [, . . .])** — Repositions the query to the row whose buffers have the RowIDs passed to the method.

INDEX-INFORMATION attribute

This attribute allows you to retrieve information about the indexes Progress uses to retrieve the records that satisfy the selection criteria. There is also an INDEX-INFORMATION function for static queries, but the dynamic method is especially useful as part of a procedure that defines selection criteria at run time for either a statically defined or dynamic query. If there is variability in the WHERE clause your procedure accepts or generates at run time (perhaps based on a specific user request), the record retrieval might be highly inefficient if the query requires searching nonindexed field values for large tables. The INDEX-INFORMATION method takes a single INTEGER argument, which is the join level you want information for. For example, a value of 1 means that you want index information for the first table in the join.

If Progress is able to use one or more index brackets to satisfy the query so that it does not have to read all the records in a table, the method returns a comma-separated list of the indexes used. Progress uses more than one index to resolve a complex query if this results in the smallest number of nonindexed fields being searched.

If Progress is unable to use an index to reduce the number of records read because the selection involves nonindexed fields or fields that are not the primary components of a multi-component index, then it returns the "WHOLE-INDEX" string, followed by a comma, followed by the name of the index Progress uses to navigate the records. This is normally the primary index of the table.

You can use the INDEX-INFORMATION method to warn users of potentially inefficient queries or prevent them from executing queries that can't use an index.

You must prepare the query before you can query its INDEX-INFORMATION.

NUM-RESULTS and CURRENT-RESULT-ROW attributes

These attributes correspond to the static query functions of the same names that you’re already familiar with. They return the number of rows in the query’s current result list and the current row position within that list, respectively.

Cleaning up a dynamic query

Like any other dynamic objects, you should always delete a dynamic query when you’re done with it. If you create a query in a named widget pool, or if there is an unnamed widget pool associated with the procedure where you create the query, then you can use the widget pool mechanism to control when the query is deleted. Otherwise, it’s important to use the `DELETE OBJECT` statement to delete the query object when you’re done using it.

However, if you need another dynamic query, perhaps for a different buffer, then it is good practice to reuse an existing query object rather than deleting one query and then creating another one. Reusing an existing dynamic query object is much more efficient than creating a new one each time you need one. You can change the buffers of an existing query object with the `SET-BUFFERS` method and then use the other methods, such as `QUERY-PREPARE`, to change the use of the same query object entirely. You should still be sure to delete the object when you’re done with it.

Extending the sample window to filter dynamically

You already extended the `h-CustOrderWin4.w` version of the sample window once in [Chapter 10, “Using Queries,”](#) to filter the records for a particular state. In that version of the procedure, the `LEAVE` trigger on the **New State** field reopens the **Customer** query based on the state abbreviation entered:

```
OPEN QUERY CustQuery FOR EACH Customer WHERE Customer.State =  
cState:SCREEN-VALUE BY Customer.City.
```

A static `OPEN QUERY` statement suffices, because the `WHERE` clause is known. The only variable is the state abbreviation, and the procedure can plug that into the `OPEN QUERY` statement.

You should use the dynamic QUERY-PREPARE and QUERY-OPEN methods when your procedure needs more flexibility than this. To show how this can work, you can extend the filtering to apply to every field in the **Customer** table that the window displays. You can change the procedure to let the user filter the **Customers** based on a value you enter in any of the **Customer** fields. You can then add a filter button to the window that blanks out the **Customer** fields and enables them. When the user enters a value in any one of the fields, the code re-prepares and reopens the query to filter on the value for that field.



To make these changes:

1. Open h-CustOrderWin1.w and save it h-CustOrderWin8.w.
2. Disable all the **Customer** fields that are displayed.

Remember that you can select the fields, open the **Properties Window**, and set **Enabled** to **false** for all of them at once. Making them initially disabled allows you to enable them for input only when you want to filter on a value in a field.

3. Remove the phrases from the query that filter on State = "NH" and sort by **City**.

Because the user will be filtering on any field, it makes sense to remove the initial filtering and sorting that this version of the procedure from [Chapter 4, “Introducing the OpenEdge AppBuilder.”](#) If you double-click on the frame, its property sheet opens where you can access the **Query Builder** to make the changes. Select the **Where** and **Sort** radio set options in turn to make the changes.

4. Drop a button onto the frame next to the navigation buttons. Name it **btnFilter** and give it a **Label of Filter**.

5. Define a CHOOSE trigger for the **Filter** button. The trigger code needs to blank out all the **Customer** fields and enable them for input.

What code can you write to do this? The quickest way is to set the SCREEN-VALUE of each field to "" and to set the SENSITIVE attribute to yes. (Remember that the SCREEN-VALUE is the value displayed in the frame and that SENSITIVE is the attribute name for the **Enabled** property of an object.):

```
ASSIGN Customer.CustNum:SCREEN-VALUE = ""
Customer.Name:SCREEN-VALUE = ""
Customer.Address:SCREEN-VALUE = ""
Customer.City:SCREEN-VALUE = ""
Customer.State:SCREEN-VALUE = ""
Customer.CustNum:SENSITIVE = YES
Customer.Name:SENSITIVE = YES
Customer.Address:SENSITIVE = YES
Customer.City:SENSITIVE = YES
Customer.State:SENSITIVE = YES.
```

This code certainly works, but does it make you a little uncomfortable? What happens if you later add a field to the frame or remove one? Then you've got a maintenance problem on your hands. Try identifying the fill-ins dynamically instead, using what you learned in the previous chapter.

6. Define a variable scoped to the procedure to hold the field handles in the **Definitions** section:

```
/* Local Variable Definitions --- */  
DEFINE VARIABLE cFillIns AS CHARACTER NO-UNDO.
```

7. Add code so that the CHOOSE trigger builds up a list of all the handles of the objects in the frame that are fill-in fields, if the list hasn't already been built:

```

DO:
  DEFINE VARIABLE hContainer AS HANDLE      NO-UNDO.
  DEFINE VARIABLE hField       AS HANDLE      NO-UNDO.
  DEFINE VARIABLE iField        AS INTEGER     NO-UNDO.

  IF cFillIns = "" THEN
  DO:
    /* Identify all the fill-ins so they can be enabled, disabled,
       and cleared. */
    ASSIGN hContainer = FRAME CustQuery:HANDLE
           hContainer = hContainer:FIRST-CHILD. /* The field group */
    hField = hContainer:FIRST-CHILD.
    DO WHILE VALID-HANDLE(hField):
      IF hField:TYPE = "Fill-In" THEN
        cFillIns = cFillIns +
                    (IF cFillIns = "" THEN "" ELSE ",") + STRING(hField).
      hField = hField:NEXT-SIBLING.
    END.
  END.

```

8. Whether the field list has already been built or not, the procedure needs to walk through those fields and blank them and disable them:

```

DO iField = 1 TO NUM-ENTRIES(cFillIns):
  ASSIGN hField = WIDGET-HANDLE(ENTRY(iField, cFillIns))
         hField:SCREEN-VALUE = ""
         hField:SENSITIVE = YES.

END.

```

Now if you later change the fields on the screen, you won't have to change this code. That's part of the value of dynamic programming!

Defining a trigger block for multiple objects

Next, you need to define a LEAVE trigger for every field to filter the query based on what the user entered into that field.

This seems tedious, doesn't it? You shouldn't have to define the same trigger code for every **Customer** field, and in fact you don't. First, you need to learn a little trick about using the trigger code in the **Section Editor**. Although the field name the trigger applies to is displayed in a special drop-down list, you can add more fields to the top of the editable portion of the window, before the DO keyword, as shown in [Figure 19–9](#).

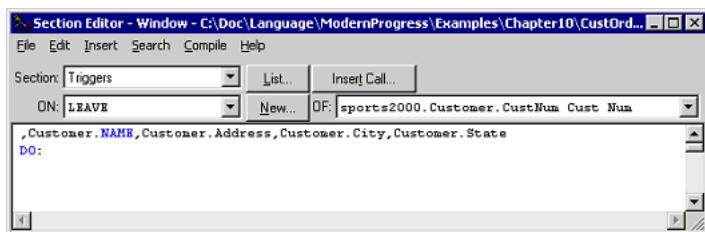


Figure 19–9: Example LEAVE trigger



To define a trigger block for multiple objects:

1. Add the code shown in [Figure 19–9](#).

This code defines the same trigger for all the fields. In the Progress 4GL syntax both the event name and the object name can be comma-separated lists.

2. Define a variable to hold the query handle and one to hold a prepare-string:

```
DEFINE VARIABLE hQuery      AS HANDLE      NO-UNDO.
DEFINE VARIABLE cPrepare AS CHARACTER    NO-UNDO.
```

3. Add the code that walks through the list of field handles, just as the **Filter** trigger does, in order to disable them. You need a variable for a field handle and a variable for a counter:

```
DEFINE VARIABLE iField     AS INTEGER     NO-UNDO.
DEFINE VARIABLE hField    AS HANDLE      NO-UNDO.
```

You may ask whether it would be simpler and more efficient to define these variables once, in the procedure's **Definitions** section, because they are used locally in both the **Filter** button trigger and the field LEAVE trigger. You should *not* do this. If you do, the leftover values from one procedure or trigger's use of the variables remain defined until the next trigger or procedure uses the variables, increasing the chances that a block of code will use a stale value by mistake, or that one procedure called from another will reset the variable that is being used for different purposes by each of them. It's good practice to scope variables and other objects as locally as possible, even if it means having the same definitions in more than one place.

Using the SELF handle to identify an object in a trigger

Now you need to check the SCREEN-VALUE of the field to see whether the user typed anything into it. The reason for this check is that the LEAVE trigger could fire under other circumstances, such as if the user clicked on a button while the cursor was positioned in the field. This action would fire the LEAVE event, but you want to disregard such events unless the user actually entered a value in the field.

But *which field*? You're defining the same trigger code for all the fields.

Progress defines another built-in handle that gives you what you need: the SELF handle. Within a trigger block, this handle resolves to the object the trigger has fired for.

► To identify a field in a trigger:

1. Check to see whether the user entered a value using this statement and start a block that is executed only if the user did:

```
IF SELF:SCREEN-VALUE NE "" THEN  
DO:
```

It's important to make this check because the LEAVE trigger fires in many different ways, including clicking on a button while the cursor is in the field. You need to check that the user really entered a value to filter on.

2. Inside this block, use the list of fill-in field handles to disable them again, now that the user has had a chance to enter a value into one of them:

```
DO iField = 1 TO NUM-ENTRIES(cFillIns):  
    ASSIGN hField = WIDGET-HANDLE(ENTRY(iField, cFillIns))  
    hField:SENSITIVE = NO.  
END.
```

Remember that you defined the `cFillIns` variable in the **Definitions** section specifically because you *do* want its value to be scoped to the whole procedure, so that its value persists between trigger or procedure calls. To remind yourself, and other readers of your code, that a variable like this is scoped to the entire procedure, you can use a naming convention such as putting a `g` for global on the front of the variable name. Even though such a variable is not actually global to the session, it is global to this procedure, so its scope is worth noting in some way.

3. Get the handle to the query, along with the current PREPARE-STRING:

```
ASSIGN hQuery = QUERY CustQuery:HANDLE  
cPrepare = hQuery:PREPARE-STRING.
```

You're saving off the current PREPARE-STRING before closing and re-preparing the query so that you can restore it later in the block in case the user chooses not to continue with the filtering. There's a special case you have to deal with here. If the query has not yet been re-prepared dynamically, then the PREPARE-STRING attribute is unable to return the FOR EACH statement used in the original static OPEN QUERY statement. In this case, you just have to set the variable to the default query string:

```
/* If the query hasn't been prepared dynamically just restore  
the original query definition. */  
IF cPrepare = ? THEN  
    cPrepare = "FOR EACH Customer".
```

4. Close the query and re-prepare it using its handle and the value the user entered into whichever one of the fill-in fields:

```

hQuery:QUERY-CLOSE().
hQuery:QUERY-PREPARE("FOR EACH Customer WHERE " + SELF:NAME +
                      (IF SELF:DATA-TYPE = "CHARACTER" THEN
                        " BEGINS " ELSE " = ")
                      + QUOTER(SELF:SCREEN-VALUE) +
                      " BY " + SELF:NAME).

```

The QUERY-CLOSE step is actually optional. The QUERY-PREPARE method closes the query automatically.

Using the DATA-TYPE attribute of the SELF handle, the code identifies the data type of the field the user is filtering on. If the field is CHARACTER, the code builds a query retrieving all **Customers** where the field value BEGINS with what the user typed in, which is the SELF:SCREEN-VALUE attribute. Otherwise, it retrieves records where the field value is equal to the filter value. You could, of course, set up such a query to filter records in any way that you want. Finally, the FOR EACH statement sorts the record by the filter field, using the SELF:NAME attribute.

Using INDEX-INFORMATION and a MESSAGE with a yes/no answer

Next, you need to check whether the field the user selected has an index. If not, you need to warn the user that the retrieval will not be indexed, and you need to provide the option of canceling the query.



To add this check and warning to your sample procedure:

1. Use the INDEX-INFORMATION attribute of the query and a MESSAGE statement that asks a question:

```

IF ENTRY(1, hQuery:INDEX-INFORMATION(1)) = "WHOLE-INDEX" THEN
DO:
  MESSAGE "This query can't use an index. Continue?"
  VIEW-AS ALERT-BOX BUTTONS YES-NO SET lContinue AS LOGICAL.
  IF NOT lContinue THEN
    hQuery:QUERY-PREPARE(cPrepare).
END.

```

If INDEX-INFORMATION returns "WHOLE-INDEX" for the first (and, in this case, the only) table in the query, then you know that there is no index Progress can use to retrieve the selected records, so it has to search the whole primary index.

To give the user the option of canceling the query, you can use an option on the MESSAGE statement that you haven't seen before. If you define a MESSAGE as VIEW-AS ALERT-BOX, you can define the choice buttons that appear in the alert box. As you have seen, OK is the default. Other choices are YES-NO, YES-NO-CANCEL, OK-CANCEL, and RETRY-CANCEL. You can then SET a variable of LOGICAL data type to record the answer. You can use a variable that you defined earlier or you can define it right in the MESSAGE statement using the AS LOGICAL phrase. You can then check the value of the variable to see which choice the user picked. The first choice within any set of buttons returns yes, the second no, and if there is a third choice (as in YES-NO-CANCEL), choosing that button sets the variable to the unknown value.

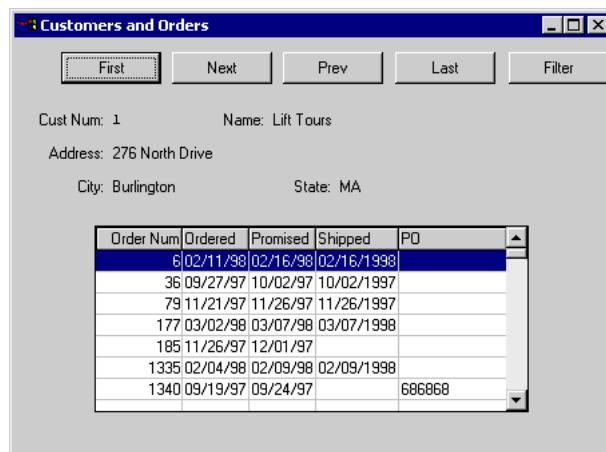
(You can set other kinds of variables and fields in messages that aren't defined as alert boxes, as well. See the online Help for details.)

In this case, if the user decides not to continue, the query is re-prepared using the PREPARE-STRING you saved off earlier.

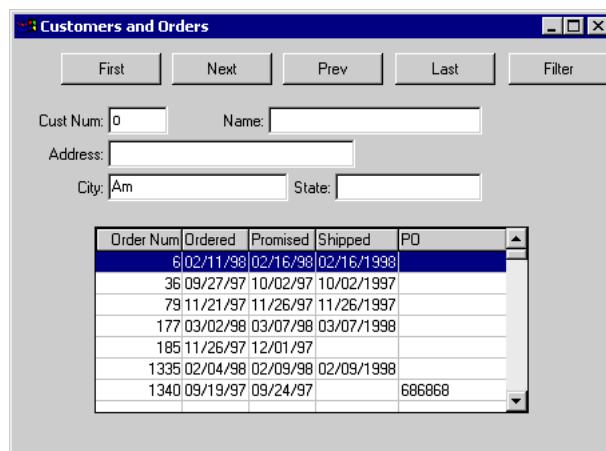
2. Reopen the query and APPLY CHOOSE to the **First** button to display the first matching record and its **Orders**:

```
hQuery:QUERY-OPEN() .
  APPLY "CHOOSE" TO btnFirst.
END. /* END DO IF SCREEN-VALUE Not "" */
```

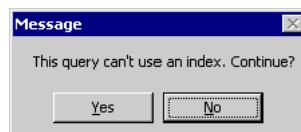
3. Run the window. The **Customer** fields initially come up disabled:



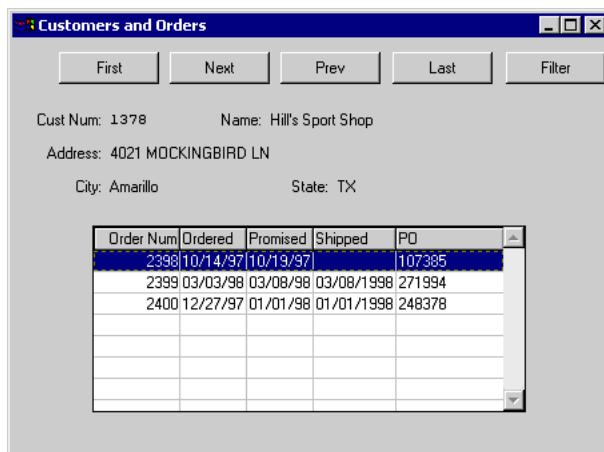
4. Choose the **Filter** button. The fields are enabled and blanked out, and you can enter a value into any one of them:



5. Enter a value into a nonindexed field, such as the **City**. You get a warning message:



6. If you answer **Yes**, the query is prepared and opened:



If you answer **No**, the procedure reverts to the previous query or to the default query if this is the first time the query has been re-prepared.

Using dynamic buffers and buffer handles

You can create a dynamic buffer at run time in much the same way that you create a dynamic query, using this syntax:

```
CREATE BUFFER buffer-handle FOR TABLE table-expression.
```

The *table-expression* can be either a literal database table or temp-table name (both in quotation marks) or a variable or other expression that evaluates to a table or temp-table at run time.

You can also access the handle of any static buffer at run time:

```
buffer-handle = BUFFER buffer-name:HANDLE.
```

Dynamic buffers are useful only when you really need to define the table name for a buffer at run time. Just as with dynamic queries, you will find that in many cases you can simply get the handle to a static buffer and use that handle's attributes and methods to manipulate the buffer dynamically, without creating a dynamic buffer at all.

Buffer handle attributes

This section describes some of the most useful attributes you can access through a buffer handle, for either a static or dynamic buffer. You can see a complete list and descriptions at the **Buffer Object Handle** entry in the online help.

NAME, TABLE, and DBNAME attributes

The NAME attribute is the name of the underlying buffer. For a dynamic buffer, it is the table name the buffer was created for.

The TABLE is the name of the database table or temp-table the buffer is defined or created for.

The DBNAME is the database name for a buffer defined or created for a database table. For a temp-table, this equals the string "PROGRESST".

ROWID attribute

The ROWID attribute is the value of the RowID of the record currently in the buffer. You can use this, for example, to reposition a query to the same record, as illustrated in the ["Extending the test window to use a buffer handle"](#) section on page 19–30.

NEW, LOCKED, and CURRENT-CHANGED attributes

The NEW attribute is a logical value that is true if the record in the buffer is newly created, and false if it is a record that has been read from the database table or temp-table the buffer is for.

The LOCKED attribute is a logical value that is true if the underlying record is locked and false if it is not.

The CURRENT-CHANGED attribute is a logical value that is true if the record in the buffer has any different field values from the underlying database table record, indicating that it has been changed by another user since it was read.

NUM-FIELDS attribute

The NUM-FIELDS attribute is the number of fields in the buffer.

AVAILABLE and AMBIGUOUS attributes

These attributes perform the same functions as their counterpart functions in static buffer references. If there is currently a record in the buffer, the logical AVAILABLE attribute is true, otherwise it is false. If a FIND statement or method executed on the buffer finds more than one matching record, then there is no record in the buffer (AVAILABLE is set to false) and the logical AMBIGUOUS attribute is set to true.

Buffer handle methods

There are many methods you can use to perform operations dynamically on a buffer handle, both for static and for dynamic buffers. These are all dynamic equivalents of static statements you're already familiar with. The principal value of these methods is that they let you define selection criteria for data retrieval and other values needed by the methods at run time. In many cases, you can use these methods effectively with queries and buffers that are defined as static objects, where the table names are known at compile time. The example that follows the method descriptions shows how to extend the **Customers** and **Orders** windows to use these dynamic methods on the query and buffer for the **Customer** table. These are summary method descriptions to make you aware of what is possible in working with buffer handles. As always, consult the online help or *OpenEdge Development: Progress 4GL Reference* for complete descriptions.

BUFFER-FIELD method

The BUFFER-FIELD method takes an argument, which can be either the name of a field or its ordinal position within the buffer, and returns the handle of that field object. You can then use the BUFFER-FIELD handle in turn to access various attributes of the field that are defined below. Note that BUFFER-FIELD is considered a method, rather than an attribute, only because it takes an argument to identify which field object you want. Beyond identifying the field, this method doesn't really do anything except return its handle.

BUFFER-COMPARE and BUFFER-COPY methods

These two methods on a target buffer handle take a source buffer handle (and other optional values) as an argument. BUFFER-COMPARE compares the field values in the two buffers and returns a report of their differences. BUFFER-COPY copies the field values in the source buffer to the target buffer.

BUFFER-CREATE, BUFFER-DELETE, and BUFFER-RELEASE methods

These methods perform the same function as the CREATE, DELETE, and RELEASE statements do.

Buffer FIND methods

There is a whole set of methods you can use to perform a FIND operation on a buffer dynamically. Some of these include:

- **FIND-BY-ROWID** — Takes a record RowID as an argument and reads that record into the record buffer. This method can be useful after a separate operation has recorded the RowID of a record using some other criteria and you wish to re-retrieve that record.
- **FIND-CURRENT** — Re-reads the current record from the database, replacing the contents of the record buffer. This method can be useful if you want to refresh the record in case it has been changed, for example by another user, since it was originally read.
- **FIND-FIRST** — Takes a WHERE clause (including the initial WHERE keyword) as an argument and retrieves the first record from the buffer's table that satisfies the WHERE clause into the buffer.
- **FIND-LAST** — Takes a WHERE clause as an argument and retrieves the *last* record from the buffer's table that satisfies the WHERE clause into the buffer.
- **FIND-UNIQUE** — Takes a WHERE clause as an argument that identifies a single record, and retrieves the one record from the buffer's table that satisfies the WHERE clause into the buffer. If more than one record satisfies the WHERE clause, then no record is retrieved and the AMBIGUOUS buffer attribute is set to true.

The FIND methods are a very useful and efficient way of identifying a single record without the overhead of preparing and opening a query with selection criteria that identify that one record, and then doing a GET-FIRST() to position the query cursor to that one record.

In addition to the RowID argument for FIND-BY-ROWID and the WHERE clause argument for the FIND-FIRST, FIND-LAST, and FIND-UNIQUE methods, all five of these methods take optional arguments that you can use to specify the lock mode (NO-LOCK, SHARE-LOCK, or EXCLUSIVE-LOCK) and wait mode (if it is NO-WAIT). The default lock mode is SHARE-LOCK. You will generally want to change this to specify either NO-LOCK or EXCLUSIVE-LOCK, depending on whether you need to prepare to allow it to be changed and protect the record against changes by other users.

This example changes the lock mode to NO-LOCK:

```
hBuffer:FIND-UNIQUE("WHERE CustNum = 125", NO-LOCK).
```

This example changes the lock mode to EXCLUSIVE-LOCK and the wait mode to NO-WAIT:

```
hBuffer:FIND-UNIQUE("WHERE CustNum = 125", EXCLUSIVE-LOCK, NO-WAIT).
```

Specifying the lock or wait option as a variable

NO-LOCK, EXCLUSIVE-LOCK, SHARE-LOCK, and NO-WAIT are Progress keywords that also represent constants that you can assign to an integer variable and pass in as an integer to the FIND- and GET- methods. For example:

```
DEFINE VARIABLE iLock AS INTEGER NO-UNDO.  
iLock = EXCLUSIVE-LOCK.  
hBuffer:FIND-FIRST("CustNum = 5", iLock).
```

This can be useful when the FIND or GET method is in a different procedure or function from the one that determines what the lock or wait mode should be, or when you want to enumerate these lock and wait options and pick one depending on some other value.

Extending the test window to use a buffer handle

In this section, you'll extend the h-CustOrderWin8.w procedure beyond the changes you already made in this chapter to illustrate the use and the value of some of the buffer handle attributes and methods. To preserve the first set of changes separately, this variant of the procedure is saved as h-CustOrderWin9.w. You'll add buttons to the window to reposition within the current query to a particular record, and also to use a dynamic FIND method to locate and display a single record without using the query at all.



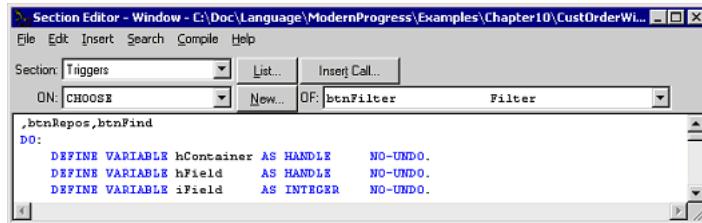
To extend the sample window to use a buffer handle:

1. Define another variable in the **Definitions** section to record which of several new buttons the user selected:

```
DEFINE VARIABLE cBtnChoice AS CHARACTER      NO-UNDO.
```

This variable keeps track of whether the user chose the **Filter** button or one of the new buttons labeled **Reposition** and **Find** that you'll define next.

2. Drop two new buttons onto the window beneath the **Filter** button. Call them **btnRepos** and **btnFind** and give them the labels **Reposition** and **Find**, respectively.
3. Extend the existing CHOOSE trigger on **btnFilter** to fire for the other buttons, just as you did for the **Customer** field's LEAVE trigger before:



4. At the end of the trigger, add a line to save off which button was chosen:

```
cBtnChoice = SELF:LABEL. /* Was this a Filter, Reposition, or Find? */
```

5. Go into the LEAVE trigger for **CustNum** (which of course applies to all the **Customer** fields). Define a variable to hold the buffer handle:

```
DEFINE VARIABLE hBuffer AS HANDLE      NO-UNDO.
```

6. After the statement that assigns the query handle and PREPARE-STRING, start a block based on the value of the new **cBtnChoice** variable. Execute the existing code only if the button chosen was **Filter**:

```

ASSIGN hQuery = QUERY CustQuery:HANDLE
      cPrepare = hQuery:PREPARE-STRING.
IF cBtnChoice = "Filter" THEN
DO:

```

7. End the DO block for the **Filter** choice and start an ELSE block for the **Reposition** choice. Save off the **Customer** buffer handle and execute a FIND-FIRST method on it that once again uses the BEGINS keyword for CHARACTER fields and the = comparison otherwise. Execute the FIND-FIRST method as NO-ERROR in case there's no such record:

```
END.      /* END DO IF "Filter" */
ELSE IF cBtnChoice = "Reposition" THEN
DO:
  hBuffer = BUFFER Customer:HANDLE.
  hBuffer:FIND-FIRST("WHERE " + SELF:NAME +
    (IF SELF:DATA-TYPE = "CHARACTER" THEN " BEGINS "
      ELSE " = ") +
    QUOTER(SELF:SCREEN-VALUE)) NO-ERROR.
```

8. Use the AVAILABLE attribute to check whether there was a matching record. If there was, use the REPOSITION-TO-ROWID method on the query handle to position the query to that record using its RowID. Do this operation NO-ERROR. Apply CHOOSE to the **Next** button to position onto the record itself, display it, and retrieve its **Orders**:

```
IF hBuffer:AVAILABLE THEN
  DO:
    hQuery:REPOSITION-TO-ROWID(hBuffer:ROWID) NO-ERROR.
    APPLY "CHOOSE" TO BtnNext.
  END. /* END DO IF AVAILABLE Customer */
```

There are a few things worth examining here. First, remember that because you are using the FIND-FIRST method, and not FIND-UNIQUE, Progress retrieves a record into the buffer even if there's more than one match. The AMBIGUOUS attribute is never true in this case.

Second, you might be a little confused about why you have to reposition the query to the record you just retrieved in the buffer. Isn't it already there for you to see? Yes, it is. If you leave out the REPOSITION method on the query and just display the contents of the buffer, you see the record FIND-FIRST retrieved. *But* if you then choose the **Next** button, you don't see the next record following the one FIND-FIRST retrieved. You just see the next record in the query as it was before you ever did the FIND-FIRST. The reason for this is that the FIND-FIRST method on the buffer handle is effectively *reusing* the buffer for a purpose completely separate from the query. There is no connection between the FIND method and the records in the query. That's really the purpose of the FIND methods, that they allow you to fetch a specific record without using a query at all. To use the FIND method to reposition the query, you need code similar to what you just wrote, which uses the RowId to reposition the query to the same record.

Also, remember why the NEXT operation is required. If you use one of the FIND methods on a buffer, the record you want is placed in the buffer, and you can use it immediately. But if you use one of the query handle's REPOSITION methods, the query cursor is effectively placed immediately before the record you are repositioning to, so you need the GET NEXT statement to actually bring that record into the buffer.

And finally, consider the NO-ERROR qualifier on the FIND-FIRST method. It's entirely possible that the user might choose the **Filter** button and filter the query before choosing the **Reposition** button, and then enter a value to reposition to that's in the **Customer** table (and therefore found by the FIND-FIRST method on the buffer) but *not* in the query's result set as it has been filtered. For example, the user could filter on **Customer Names** beginning with **A**, and then reposition to the first **Customer Name** beginning with **B**. Progress successfully retrieves the first **Customer Name** starting with **B** from the database, but then the REPOSITION method on the *query* fails because that record is not in the query's result list. This is important to keep in mind as you use these objects and their methods.

9. Check whether there's a record available and display a message if there is not:

```
IF NOT hBuffer:AVAILABLE THEN
  MESSAGE "No record matches that value. Try again. ".
END.      /* END ELSE DO (IF "Reposition" ) */
```

In such a case, a record might not be available either because there was no matching record in the database or because that record was not in the query.

10. Define a block of code to support the **Find** button. Because the **Find** button is identifying a single record, the code uses the FIND-UNIQUE buffer method:

```
ELSE IF cBtnChoice = "Find" THEN
DO:
  hBuffer = BUFFER Customer:HANDLE.
  hBuffer:FIND-UNIQUE("WHERE " + SELF:NAME + " = " +
    QUOTER(SELF:SCREEN-VALUE)) NO-ERROR.
```

Because you're trying to find just one matching record, the comparison operator in the WHERE clause is simply "=".

As before, you need to invoke the method with the NO-ERROR qualifier, in case the selection either doesn't yield a record or yields more than one matching record. Next, you need to check for those conditions.

11. Add code to check the AMBIGUOUS and AVAILABLE attributes to make sure you got exactly one match:

```
IF hBuffer:AMBIGUOUS THEN
  MESSAGE "This choice returns more than one row. Try again.".
ELSE IF NOT hBuffer:AVAILABLE THEN
  MESSAGE "This choice does not match any row. Try again.".
```

12. Write the code to handle the successful case.
- Close the **Customer** query.
 - Disable the navigation buttons and the **Reposition** button because they don't apply if you've just got one record.
 - Display the record that the FIND-UNIQUE method retrieved.
 - Reopen the **Order** Browse:

```
ELSE DO:
  CLOSE QUERY CustQuery.
  DISABLE BtnFirst BtnNext btnPrev BtnLast BtnRepos
    WITH FRAME CustQuery.
  DISPLAY Customer.CustNum Customer.Name Customer.Address
    Customer.City Customer.State
    WITH FRAME CustQuery IN WINDOW CustWin.
    {&OPEN-BROWSERS-IN-QUERY-CustQuery}
  END.    /* END DO IF "Find" */
END.      /* END ELSE DO IF no errors */
END.      /* END DO IF SCREEN-VALUE NOT "" */
          /* END DO for the trigger block */
```

Why close the query? Because you're not using it. You found a record using the same buffer the query uses, but that record is not in any way related to the query.

Why disable the buttons? Because they expect to be able to navigate through the query or reposition within it. You'd get an error if you allowed the user to choose them.

You can copy the DISPLAY statement and the {&OPEN-BROWSERS-IN-QUERY-CustQuery} preprocessor from any of the navigation triggers. As you did in an earlier exercise, you could clean this up by factoring out the repeated code into an include file or internal procedure.

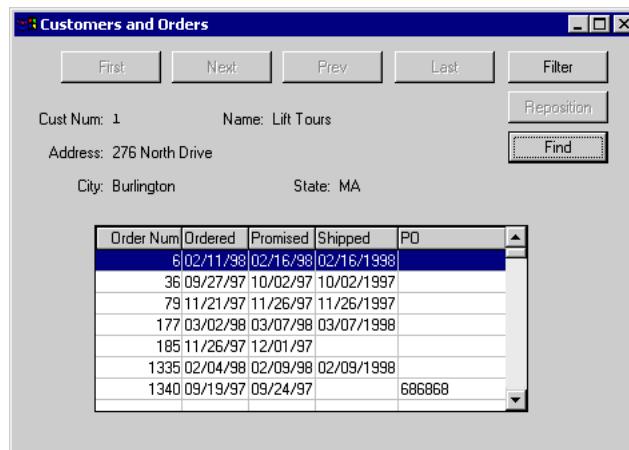
Don't forget to end all your blocks properly, with a comment on each END statement to help you verify the block structure as things get more complex.

Because the **Find** button disables the navigation and **Reposition** buttons, the block of code in the same trigger for the **Filter** button needs to re-enable them.

13. Add this ENABLE statement to that block, after it reopens the query:

```
hQuery:QUERY-OPEN().
ENABLE BtnFirst BtnNext BtnPrev BtnLast BtnRepos
      WITH FRAME CustQuery.
APPLY "CHOOSE" TO btnFirst.
```

14. Try out the procedure with various combinations of actions. When you choose the **Find** button, for instance, it should enable the fields, accept input, and then disable the fields along with the navigation and **Reposition** buttons and display the one matching record:



Using various other **Filter**, **Find**, and **Reposition** requests, you can test the various error conditions that you wrote code to handle.

Cleaning up dynamic buffers

This example uses a handle to a static buffer to illustrate how to use buffer attributes and methods, because the table name is known. You could also create a dynamic buffer for the table like this:

```
CREATE BUFFER hBuffer FOR TABLE "Customer".
```

Given that the table name is known and that it is, therefore, a literal in the CREATE BUFFER statement, there is no particular advantage to doing this. The CREATE BUFFER statement is better used when the buffer name is variable.

When you create dynamic buffers, rather than just using handles to static buffers, you need to delete them when you're done using them, just as with other objects. You use the same DELETE OBJECT statement to do this.

Often you will want to delete dynamic buffers along with the dynamic query that uses them. You can use the GET-BUFFER-HANDLE method to determine what buffers to delete when you have finished with a dynamic query with dynamic buffers, as in this example, saved as h-cleanup.p:

```
/* Procedure h-cleanup.p -- shows how to delete
   a dynamic query and its dynamic buffers. */
DEFINE VARIABLE hQuery      AS HANDLE      NO-UNDO.
DEFINE VARIABLE hBuffer     AS HANDLE      NO-UNDO.
/* A list of tables for the dynamic query comes in as a parameter
   or is determined at run time . This simulates the list of names. */
DEFINE VARIABLE cBufNames   AS CHARACTER   NO-UNDO
   INIT "Customer,Order,SalesRep".

DEFINE VARIABLE iBufNum     AS INTEGER     NO-UNDO.
DEFINE VARIABLE cBufHandles AS CHARACTER   NO-UNDO.

/* CREATE a dynamic query and dynamic buffers for each table.
   Add each buffer to the query. */
CREATE QUERY hQuery.
DO iBufNum = 1 TO NUM-ENTRIES(cBufNames):
   CREATE BUFFER hBuffer FOR TABLE ENTRY(iBufNum, cBufNames).
   hQuery:ADD-BUFFER(hBuffer).
END.

/* Verify that the query has the buffers we created. */
cBufNames = "".
DO iBufNum = 1 TO hQuery:NUM-BUFFERS:
   cBufNames = cBufNames + hQuery:GET-BUFFER-HANDLE(iBufNum):NAME
   + " ".
END.
MESSAGE "This query uses buffers " cBufNames VIEW-AS ALERT-BOX.

/* Application code here uses the dynamic query and its buffers... */
/* When you're done with the query, build up a list of its buffers. */
DO iBufNum = 1 TO hQuery:NUM-BUFFERS:
   cBufHandles = cBufHandles +
      (IF cBufHandles = "" THEN "" ELSE ",") +
      STRING(hQuery:GET-BUFFER-HANDLE(iBufNum)).
END.
MESSAGE "Buffer handles are: " cBufHandles VIEW-AS ALERT-BOX.
/* After building the list, delete the buffers and then the query. */
DO iBufNum = 1 TO hQuery:NUM-BUFFERS:
   DELETE OBJECT WIDGET-HANDLE(ENTRY(iBufNum, cBufHandles)).
END.
DELETE OBJECT hQuery.
```

Note that you cannot delete the buffers in the loop where the code uses the GET-BUFFER-HANDLE method at the end of the procedure because deleting the buffers would *un-prepare* the query and cause it to lose the list of buffers before you had retrieved them all. Thus, you need to build a comma-separated list of buffer handles and walk through the list afterwards to delete the buffers, and then delete the query.

Special dynamic buffer considerations

When you use dynamic buffers, there are a few special rules that you need to be aware of:

- You can define a delete validation expression for a table in the Data Dictionary when you define your database. Although this is not recommended in new applications, if you *have* done this, Progress is unable to verify whether the delete expression is true or not when you use a dynamic buffer for the table. It therefore disallows any BUFFER-DELETE operation on the table.
- It is important that you use the BUFFER-RELEASE method to release a dynamic buffer when you are done using it, especially if you have written to the buffer. This is true of static buffers as well, where you could use the static RELEASE statement. But because there is no inherent record scoping for dynamic buffers, a record release might be postponed indefinitely if you don't handle it yourself.
- A dynamic buffer has the same scope as the widget pool in which it was created. This means that Progress automatically deletes a dynamic buffer object only when it deletes the widget pool. To delete a dynamic buffer yourself, use the DELETE OBJECT statement, as the preceding example shows.
- If you place the phrase BUFFER *name* anywhere in a procedure file, where *name* represents the name of a table, not necessarily the name of a buffer you defined using a DEFINE BUFFER statement, Progress scopes the *name* as it would a free reference to the buffer.

Using BUFFER-FIELD objects

As you saw in the previous chapter, there is a BUFFER-FIELD object for each field in a buffer, whether the buffer is static or dynamic. You can access a field within a buffer through its handle, using one of two types of arguments to the BUFFER-FIELD method, either its ordinal position within the buffer or itsfieldname.

Buffer fields have all the same attributes that static field-level objects have. You're already familiar with these attributes from static syntax, including COLUMN-LABEL, DATA-TYPE, DBNAME, DECIMALS, EXTENT, FORMAT, HANDLE, HELP, INITIAL, LABEL, MANDATORY, NAME, TABLE, and WIDTH-CHARS. Buffer field objects also have several useful attributes distinctive to the object and accessible only through its handle. This section summarizes those objects.

You cannot create or delete a buffer field independent of its buffer. Buffer field objects are created when a buffer is created. They are deleted along with the buffer, if it is dynamic. In the case of static buffers, the buffer field is really just a handle through which you can access useful attributes.

BUFFER-HANDLE attribute

This attribute is the handle of the buffer the field belongs to. So, the BUFFER-FIELD handle points from the buffer to one of its fields, and the BUFFER-HANDLE points from any field back to its buffer.

BUFFER-NAME attribute

This CHARACTER attribute holds the name of the buffer the field belongs to.

BUFFER-VALUE and STRING-VALUE attributes

The BUFFER-VALUE is the value of the field in its native data type. The attribute therefore holds a value in whatever the data type of the field is. The STRING-VALUE, on the other hand, is a CHARACTER attribute that holds the field value as it is formatted for display. It is therefore in the same format as the SCREEN-VALUE of a displayed field. Note that this is not necessarily the same as what the expression STRING(hField:BUFFER-VALUE) would return, because the STRING-VALUE includes any characters that are part of the field format, whereas using the STRING function simply turns the value into a character string without applying any special formatting to it.

POSITION attribute

This INTEGER attribute is the ordinal position of the field within the buffer.

Note that there is a difference between the order of the fields you obtain using the `BUFFER-FIELD(n)` form to identify the field and the value of the field's POSITION attribute. The order using `BUFFER-FIELD(n)` is the display order of the fields, which is determined using the **Order #** as it is assigned in the Data Dictionary. The POSITION attribute is assigned internally as fields are created for a table, and does not change, even though the display **Order #** can be changed. In addition, the first field position is reserved so that the numbering of the POSITION begins with 2. For example, this code shows the POSITION attribute value for each **Customer** field:

```
DEFINE VARIABLE hCust AS HANDLE NO-UNDO.  
DEFINE VARIABLE hField AS HANDLE NO-UNDO.  
DEFINE VARIABLE cLabel AS CHARACTER FORMAT "X(20)" NO-UNDO.  
DEFINE VARIABLE cValue AS CHARACTER FORMAT "X(40)" NO-UNDO.  
DEFINE VARIABLE iCount AS INTEGER NO-UNDO.  
  
hCust = BUFFER Customer:HANDLE.  
hCust:FIND-FIRST("WHERE CustNum = 1", NO-LOCK).  
REPEAT iCount = 1 TO hCust:NUM-FIELDS:  
    ASSIGN hField = hCust:BUFFER-FIELD(iCount)  
        cLabel = hField:LABEL  
        cValue = hField:STRING-VALUE.  
    DISPLAY cLabel hField:POSITION FORMAT "Z9" LABEL "Position" cValue.  
END.
```

Figure 19–10 shows the result.

| cLabel | Position | cValue |
|--------------|----------|----------------------------------|
| Cust Num | 2 | 1 |
| Country | 8 | USA |
| Name | 3 | Lift Tours |
| Address | 4 | 276 North Drive |
| Address2 | 5 | |
| City | 6 | Burlington |
| State | 7 | MA |
| Postal Code | 17 | 01730 |
| Contact | 10 | Gloria Shepley |
| Phone | 9 | (617) 450-0086 |
| Sales Rep | 11 | HXM |
| Credit Limit | 13 | 66,700 |
| Balance | 14 | 903.64 |
| Terms | 15 | Net30 |
| Discount | 16 | 35% |
| Comments | 12 | This customer is on credit hold. |
| Fax | 18 | |
| Email | 19 | |

Figure 19–10: Result of using the POSITION attribute

The following chapter continues the discussion of dynamic data management objects with information on how to create and use dynamic temp-tables and dynamic browses.

20

Creating and Using Dynamic Temp-tables and Browses

This chapter continues the discussion of dynamic data management objects with information on how to create and use dynamic temp-tables and dynamic browses.

Using a dynamic temp-table, you can build up a table of any fields your application needs, some of which can be derived from database fields, and some or all of which can be completely independent of any database table. You can then define a dynamic buffer and a dynamic query to manage the temp-table data. Using a dynamic browse, you can define at run time not just what query a browse should display data for, but also what columns it should display and in what order.

This chapter includes the following sections:

- [Creating and using dynamic temp-tables](#)
- [Temp-table parameters](#)
- [Creating and using dynamic browses](#)
- [Building a comprehensive example](#)
- [Dynamic programming considerations](#)

Creating and using dynamic temp-tables

Temp-tables are one of the most important and useful constructs in an OpenEdge application. A temp-table lets you define in-memory storage for any number of rows of any combination of fields you require. A temp-table can be a mirror of a single database table, or it can be the result of a join of tables. It can contain a subset of selected fields from one or more tables, or it can contain fields that don't map to any database fields at all. And it can be any combination of these things. In general, temp-tables give you two basic capabilities:

- A temp-table lets you create business logic that is independent of the particular structure of the underlying data source.
- A temp-table is the mechanism you use for passing records from one OpenEdge session to another, in particular from an AppServer session where the database resides to a client session where the user interface for the application is.

This chapter expands on your knowledge of temp-tables by introducing you to *dynamic* temp-tables. It also introduces you to the attributes you can access and the methods you can invoke using a handle to either a static or dynamic temp-table.

Finally, this chapter details the various ways in which you can pass a temp-table from one procedure or one session to another.

As with other objects, you can get the handle to a static temp-table and use that handle to query its attributes, using this syntax:

```
tt-handle = TEMP-TABLE tt-name:HANDLE.
```

Having said this, however, a handle to a static temp-table is less useful than a handle to most other static objects, for two reasons:

1. There are only a few attributes you can access through a temp-table's handle, and none of these are settable for a static table.
2. None of the temp-table methods are usable for a static temp-table.

These methods define the fields and indexes for the temp-table. You cannot change or extend the fields or indexes for a static temp-table.

To create a dynamic temp-table, use this statement:

```
CREATE TEMP-TABLE tt-handle.
```

There are no other options on the CREATE TEMP-TABLE statement. You specify everything about the table after you create it. Thus, the CREATE statement really does nothing more than set up the *tt-handle* variable to be a handle for a temp-table structure to fill in later.

A dynamic temp-table can be in one of three states. When you first create it, using just the CREATE TEMP-TABLE statement, it is said to be in the *clear* state. That is, the temp-table handle has been allocated but there is no definition for the table yet. After you start to use the temp-table methods to define the table's fields and indexes, the table is said to be *unprepared*. This means that the table definition is not yet complete and you cannot start to use the table. After you complete the definition using its methods, you use a special TEMP-TABLE-PREPARE method to signal to Progress that the definition is complete. This effectively freezes the definition and allows you to start to use the table to store data. At this point, the temp-table is in the *prepared* state.

Temp-table methods

You define every aspect of a dynamic temp-table through its methods. Using these methods, you can define it to be like another table, to inherit individual field definitions from another table, or to create fields that have no relationship to fields in other tables. You can also define one or more indexes for the temp-table. You can execute a number of these methods in sequence to build up a temp-table that you then finalize and use. All the methods return a logical success flag. You should check this flag if it is possible that the arguments to the method might be invalid or that the method might fail for some other reason.

CREATE-LIKE method

If you want to create a temp-table LIKE a database table or another temp-table with all its fields and at least one of its indexes, use this method to copy all fields from the source table to the temp-table definition, along with index definitions:

```
tt-handle:CREATE-LIKE( { source-handle-exp | source-name-exp }  
[ , source-index-name-exp ] ).
```

You define the source table by either passing its handle or its name. The name expression can be either the table name as a quoted literal or a character expression that evaluates to the table name. A source table can be either a database table for a currently connected database or another temp-table whose scope makes it available to the procedure with the CREATE-LIKE method.

If you do not pass the optional second argument, then the temp-table inherits all the index definitions of the underlying table. If you want to specify only one index from the underlying table initially, you can pass its name as a character expression as the second argument. For example, this sequence of statements creates a dynamic temp-table with all the fields from the **Customer** table, along with just the **Name** index:

```
DEFINE VARIABLE hTT AS HANDLE      NO-UNDO.  
CREATE TEMP-TABLE hTT.  
hTT:CREATE-LIKE("Customer", "Name").
```

You can only use the CREATE-LIKE method once for a temp-table definition, when the temp-table is in the *clear* state. Thus, if you use this method, it must be the first method you invoke for the temp-table. If you want fields from additional tables in the temp-table, you use the ADD-FIELDS-FROM method to add them. If you want to add more indexes from the source table, you can use the ADD-LIKE-INDEX method to do this.

ADD-FIELDS-FROM method

If you want to add a subset of fields from another table to a temp-table, you use the ADD-FIELDS-FROM method:

```
tt-handle:ADD-FIELDS-FROM( { source-handle-exp | source-name-exp }  
[ , except-list-exp ] ).
```

You can also use this method as many times as you need to in order to add fields from one or more additional tables to a temp-table that you've already started to build using the CREATE-LIKE or another ADD-FIELDS-FROM method.

As with the CREATE-LIKE method, you pass either the source table handle or an expression representing its name. If you want to exclude some fields from being copied into the temp-table definition, pass a comma-separated list of these field names as the optional second argument. If you add a field whose name is already in the target temp-table, Progress ignores the duplicate field and does not add it to the temp-table.

This example adds all fields from the **SalesRep** table except the **MonthQuota** and **Region** fields to the fields from the **Customer** table already in the temp-table:

```
DEFINE VARIABLE hTT AS HANDLE      NO-UNDO.
CREATE TEMP-TABLE hTT.
hTT:CREATE-LIKE("Customer", "Name").
hTT:ADD-FIELDS-FROM("SalesRep", "MonthQuota, Region").
```

Both the **Customer** and **SalesRep** tables have a **SalesRep** field with the **SalesRep**'s initials. Because this field is already in the temp-table from the **Customer** table, it is not added from the **SalesRep** table, and no error results.

ADD-LIKE-FIELD method

If you want to add fields from another table to the temp-table individually, or you need to rename fields as you add them, use the ADD-LIKE-FIELD method once for each field:

```
tt-handle:ADD-LIKE-FIELD( tt-field-name-exp ,
{ source-buffer-field-handle-exp | source-db-field-name-exp } ).
```

This method takes two arguments:

- The name of the field in the temp-table, as a character expression.
- An identifier of the source field, which can be either its buffer-field handle or an expression evaluating to its name, with the table name qualifier (and, if necessary, its database name qualifier).

This example adds the **Region** field from the **SalesRep** table, which was not added in the ADD-FIELDS-FROM method, and renames it to **Area**:

```
DEFINE VARIABLE hTT AS HANDLE      NO-UNDO.
CREATE TEMP-TABLE hTT.
hTT:CREATE-LIKE("Customer", "Name").
hTT:ADD-FIELDS-FROM("SalesRep", "MonthQuota").
hTT:ADD-LIKE-FIELD("Area", "SalesRep.Region").
```

ADD-NEW-FIELD method

If you want to add one or more fields to a temp-table definition that are not derived from a specific other database or temp-table field name, use the ADD-NEW-FIELD method:

```
tt-handle:ADD-NEW-FIELD( tt-field-name-exp , data-type-exp [ , extent-exp
[ , format-exp [ , initial-exp [ , label-exp [ , column-label-exp ] ] ] ] ] ).
```

The first argument is the field name in the temp-table. The second is its data type. These two arguments are required. You can also define other attributes optionally, in this order:

1. The extent of the field, if it has one.
2. The format of the field.
3. The field's initial value.
4. The field's label.
5. The field's column-label.

Because these arguments are position-dependent, you must include values for any intervening arguments you don't specify. You don't need to include commas or values for optional arguments following the last one you specify. For example, this statement adds an integer field called **Sequence** to the temp-table, with a format of "9999" and an initial value of 1000:

```
DEFINE VARIABLE hTT AS HANDLE      NO-UNDO.
CREATE TEMP-TABLE hTT.
hTT:CREATE-LIKE("Customer","Name").
hTT:ADD-FIELDS-FROM("SalesRep","MonthQuota").
hTT:ADD-LIKE-FIELD("Area", "SalesRep.Region").
hTT:ADD-NEW-FIELD("Sequence", "INTEGER",0,"9999",1000).
```

ADD-LIKE-INDEX method

You've already seen how a temp-table can inherit either one or all of the indexes from an underlying table, in the CREATE-LIKE method. You can also add indexes one at a time using other methods. The first of these is ADD-LIKE-INDEX. This method adds a single index to the temp-table that is derived from an existing index on another table. You specify the name you want the index to have in the temp-table, the name of the index in the source table, and either the buffer-handle to the source table or an expression holding its name:

```
tt-handle:ADD-LIKE-INDEX( tt-index-name-exp , source-index-name-exp
{, source-buffer-handle-exp | source-db-table-name-exp } ).
```

This example adds the **CustNum** index to the temp-table, in addition to the **Name** index that was added in the CREATE-LIKE method:

```
DEFINE VARIABLE hTT AS HANDLE      NO-UNDO.
CREATE TEMP-TABLE hTT.
hTT:CREATE-LIKE("Customer", "Name").
hTT:ADD-FIELDS-FROM("SalesRep", "MonthQuota").
hTT:ADD-LIKE-FIELD("Area", "SalesRep.Region").
hTT:ADD-NEW-FIELD("Sequence", "INTEGER", 0, "9999", 1000).
hTT:ADD-LIKE-INDEX("CustNum", "CustNum", "Customer").
```

ADD-NEW-INDEX method

If you want to add an index to a temp-table that isn't derived from an index in another table, use the ADD-NEW-INDEX method:

```
tt-handle:ADD-NEW-INDEX( index-name-exp [ , is-unique [ , is-primary
[ , is-word-index ] ] ] ).
```

You supply the index name and up to three optional logical values that indicate whether the new index has enforcement of unique values, whether it is the primary index of the temp-table, and whether it is a word-index. (A *word index* is a special index type on a CHARACTER field that allows Progress to retrieve records based on any word the field contains.)

ADD-INDEX-FIELD method

After you use ADD-NEW-INDEX, you invoke the ADD-INDEX-FIELD method once for each field in the new index. You pass the name of the index, the name of the field to add, and an optional third argument that evaluates to `asc` for ascending (the default) or `desc` for descending:

```
tt-handle:ADD-INDEX-FIELD( index-name-exp , field-name-exp [ , mode-exp ] ).
```

If there are multiple fields in the index, the order in which you invoke ADD-INDEX-FIELD for the fields determines their order within the index.

This example adds a new unique index called SeqIndex to the temp-table and adds the **Sequence** field to it:

```
DEFINE VARIABLE hTT AS HANDLE      NO-UNDO.
CREATE TEMP-TABLE hTT.
hTT:CREATE-LIKE("Customer", "Name").
hTT:ADD-FIELDS-FROM("SalesRep", "MonthQuota").
hTT:ADD-LIKE-FIELD("Area", "SalesRep.Region").
hTT:ADD-NEW-FIELD("Sequence", "INTEGER", 0, "9999", 1000).
hTT:ADD-LIKE-INDEX("CustNum", "CustNum", "Customer").
hTT:ADD-NEW-INDEX("SeqIndex", YES).
hTT:ADD-INDEX-FIELD("SeqIndex", "Sequence").
```

TEMP-TABLE-PREPARE method

Once you have invoked all the methods that you need to fully define the temp-table fields and indexes, you must use the TEMP-TABLE-PREPARE method to finalize the definition before you use the temp-table. You must pass an argument to the method that gives the temp-table a name, as in this example:

```
DEFINE VARIABLE hTT AS HANDLE      NO-UNDO.
CREATE TEMP-TABLE hTT.
hTT:CREATE-LIKE("Customer", "Name").
hTT:ADD-FIELDS-FROM("SalesRep", "MonthQuota").
hTT:ADD-LIKE-FIELD("Area", "SalesRep.Region").
hTT:ADD-NEW-FIELD("Sequence", "INTEGER", 0, "9999", 1000).
hTT:ADD-LIKE-INDEX("CustNum", "CustNum", "Customer").
hTT:ADD-NEW-INDEX("SeqIndex", YES).
hTT:ADD-INDEX-FIELD("SeqIndex", "Sequence").
hTT:TEMP-TABLE-PREPARE("CustSequence").
```

Once you have invoked this method, the temp-table is in the *prepared* state. You can now begin to use it, creating and deleting records in the temp-table, opening queries on the table, and so forth. You cannot use the temp-table in any way until you prepare it. After you use the TEMP-TABLE-PREPARE method, you cannot invoke any of the other methods to change its definition except the CLEAR method discussed next.

CLEAR method

If you want to reuse a temp-table handle for a different temp-table definition, you can invoke the CLEAR method on the handle. The CLEAR method takes no arguments. It empties the temp-table if there are any records in it and completely erases its definition, returning the handle to the *clear* state. At this point, you can begin to build up a definition again from scratch. There is no way to undo parts of a dynamic temp-table definition, and there is no way to extend its list of fields and indexes once it has been prepared.

Temp-table attributes

This section describes the attributes you can access through a temp-table handle.

NAME attribute

This is the name of the temp-table. For a static temp-table, it is the name you gave it in the DEFINE TEMP-TABLE statement. For a dynamic temp-table, it is the name you specify in the TEMP-TABLE-PREPARE method. Thus, the NAME attribute is not defined for a dynamic temp-table until you have prepared it.

PREPARED attribute

This LOGICAL attribute returns true if the temp-table has been prepared and false otherwise. It always returns true for a static temp-table.

DEFAULT-BUFFER-HANDLE attribute

Every temp-table, whether static or dynamic, has a default buffer. This attribute holds the handle of that buffer. You use the buffer to create or delete records in the table. The default buffer has the same name as the temp-table, just as the default buffer for a database table has the same name as the table. The default buffer handle is not assigned until the temp-table is prepared.

PRIMARY attribute

This CHARACTER attribute holds the name of the primary index for the table. You can inherit the definition of the primary index from another table, in the CREATE-LIKE method, or you can define a temp-table index to be primary. This is the index the records in the table are sorted on, if there are no other sort criteria or if another index is not used to satisfy selection criteria of a WHERE clause on the table. You can set the value of the PRIMARY attribute for a dynamic temp-table only if the temp-table has not yet been prepared.

UNDO attribute

This LOGICAL attribute is true if the temp-table has undo properties and false if it is NO-UNDO. Because the static DEFINE TEMP-TABLE statement allows you to specify the NO-UNDO keyword on the definition, just as you can for variables, the default value of UNDO for a static temp-table is true. For a dynamic temp-table, the default value of UNDO is false. You can set the UNDO attribute for a dynamic temp-table only before it has been prepared.

Temp-table buffer methods and attributes

In addition to the default buffer handle that is an attribute of every temp-table, you can create one or more dynamic buffers for a dynamic temp-table. Use this syntax:

```
CREATE BUFFER buffer-handle FOR TABLE tt-handle:DEFAULT-BUFFER-HANDLE  
[ BUFFER-NAME buffer-name ] [ IN WIDGET-POOL pool-name ].
```

Note that it is necessary to include the reference to the DEFAULT-BUFFER-HANDLE. For example, you can create a second dynamic buffer for the temp-table in the current example with these statements:

```

DEFINE VARIABLE hTT AS HANDLE      NO-UNDO.
DEFINE VARIABLE hTTB AS HANDLE     NO-UNDO.
CREATE TEMP-TABLE hTT.
hTT:CREATE-LIKE("Customer", "Name").
hTT:ADD-FIELDS-FROM("SalesRep", "MonthQuota").
hTT:ADD-LIKE-FIELD("Area", "SalesRep.Region").
hTT:ADD-NEW-FIELD("Sequence", "INTEGER", 0, "9999", 1000).
hTT:ADD-LIKE-INDEX("CustNum", "CustNum", "Customer").
hTT:ADD-NEW-INDEX("SeqIndex", YES).
hTT:ADD-INDEX-FIELD("SeqIndex", "Sequence").
hTT:TEMP-TABLE-PREPARE("CustSequence").
CREATE BUFFER hTTB FOR TABLE hTT:DEFAULT-BUFFER-HANDLE
    BUFFER-NAME "CustSeq2".

```

Now you have two dynamic buffers for the temp-table, one called *CustSeq* that you get “for free” along with the temp-table definition and a second that you created yourself.

The dynamic buffer methods that you learned about earlier are essential to using dynamic temp-tables and their buffer. In particular, you will use the BUFFER-CREATE method on the buffer handle to create records in the dynamic temp-table, BUFFER-DELETE to delete them, BUFFER-RELEASE to release them, and BUFFER-COPY to copy fields from one or more database tables into newly created records in the temp-table.

There are a couple of additional buffer methods and attributes specific to their association with temp-tables, however.

TABLE-HANDLE buffer attribute

This attribute returns the handle of the temp-table the buffer is associated with. For a buffer not associated with a temp-table, it returns the unknown value. Therefore, the temp-table’s DEFAULT-BUFFER-HANDLE points to its default buffer, and that buffer’s TABLE-HANDLE attribute points back to the temp-table. In addition, the TABLE-HANDLE attribute of any other buffer defined or created for the temp-table, such as *CustSeq2*, also points back to the temp-table.

EMPTY-TEMP-TABLE buffer method

It is much more efficient to empty a temp-table in one statement rather than in a FOR EACH loop. To empty a static temp-table you can use the EMPTY TEMP-TABLE *tt-name* statement. With the handle of a buffer for the temp-table, you can also use the EMPTY-TEMP-TABLE method on the buffer. It takes no arguments. Remember that while you can have multiple buffers associated with a temp-table, just as you can with a database table, there is only one set of records in the temp-table. The multiple buffers simply allow you to have pointers to multiple different records in the temp-table at the same time. Thus, you can use the EMPTY-TEMP-TABLE method on any buffer for the temp-table and the result is the same.

Changing field attributes in a temp-table buffer

You can change certain field attributes in a dynamic temp-table after you have prepared it, including the label, format, help, and column-label. You do this by using the DEFAULT-BUFFER-HANDLE and setting one or more field attributes on a buffer-field. This is only possible with dynamic temp-tables.

For example, adding this code to the previous example changes the label and format of the **CustNum** field:

```

DEFINE VARIABLE hTT      AS HANDLE      NO-UNDO.
DEFINE VARIABLE hCustNum AS HANDLE      NO-UNDO.
DEFINE VARIABLE hTTB     AS HANDLE      NO-UNDO.
CREATE TEMP-TABLE hTT.
hTT:CREATE-LIKE("Customer", "Name").
hTT:ADD-FIELDS-FROM("SalesRep", "MonthQuota").
hTT:ADD-LIKE-FIELD("Area", "SalesRep.Region").
hTT:ADD-NEW-FIELD("Sequence", "INTEGER", 0, "9999", 1000).
hTT:ADD-LIKE-INDEX("CustNum", "CustNum", "Customer").
hTT:ADD-NEW-INDEX("SeqIndex", YES).
hTT:ADD-INDEX-FIELD("SeqIndex", "Sequence").
hTT:TEMP-TABLE-PREPARE("CustSequence").
CREATE BUFFER hTTB FOR TABLE hTT:DEFAULT-BUFFER-HANDLE
    BUFFER-NAME "CustSeq2".

ASSIGN hCustNum = hTT:DEFAULT-BUFFER-HANDLE:BUFFER-FIELD("CustNum")
    hCustNum:LABEL = "Test"
    hCustNum:FORMAT = "999999".
MESSAGE "Label: " hCustNum:LABEL
    "Format: " hCustNum:FORMAT VIEW-AS ALERT-BOX.

```

The MESSAGE shown in [Figure 20–1](#) confirms that the field format and label have changed.



Figure 20–1: Changed field attributes message

If you look at the **CustNum** label and format through the second dynamic buffer, the **CustSeq2** buffer, you see that the format and label of the second buffer have the same value as in the default buffer even though the field in the default buffer was only changed after the second buffer was created.



To illustrate an important point about multiple temp-table buffers:

1. Change the message statement to display the attributes through the second buffer:

```
ASSIGN hCustNum = hTT:DEFAULT-BUFFER-HANDLE:BUFFER-FIELD("CustNum")
      hCustNum:LABEL = "Test"
      hCustNum:FORMAT = "999999".
MESSAGE "Label: " hTTB:BUFFER-FIELD("CustNum"):LABEL SKIP
        "Format: " hTTB:BUFFER-FIELD("CustNum"):FORMAT VIEW-AS ALERT-BOX.
```

The label and format are changed, even as seen through the second buffer:



2. Add a line of code that sets the label of the **CustNum** field in the *second* buffer:

```
hTTB:BUFFER-FIELD("CustNum"):LABEL = "Okay".
ASSIGN hCustNum = hTT:DEFAULT-BUFFER-HANDLE:BUFFER-FIELD("CustNum")
      hCustNum:LABEL = "Test"
      hCustNum:FORMAT = "999999".
MESSAGE "Label: " hTTB:BUFFER-FIELD("CustNum"):LABEL SKIP
        "Format: " hTTB:BUFFER-FIELD("CustNum"):FORMAT VIEW-AS ALERT-BOX.
```

None of the changes made to the default buffer are seen in that buffer:



So the rule is that any changes you make to fields through the default buffer are also seen in any additional dynamic buffers for the temp-table, *if* no other changes have been made through the other buffer. Otherwise, the alternate buffer's field attributes are initialized to the state of the fields at the time the alternate buffer is created. In that case, any changes made to fields through the alternate buffer are not seen in the default buffer, and any other changes made to the default buffer after that remain separate from the alternate buffer.

Cleaning up a dynamic temp-table

When you're done using a dynamic temp-table, you should delete it just as you should any other object. Use the `DELETE OBJECT tt-handle` statement to do this. The default buffer is deleted along with the temp-table. You cannot use the `DELETE OBJECT` statement on the temp-table's `DEFAULT-BUFFER-HANDLE`, but you must use it on any additional dynamic buffers created for the temp-table.

Extending the example to create and display records

This chapter concludes with a comprehensive example of using dynamic queries, buffers, temp-tables, and browses. The final example is saved as `h-testDynTT.p`.



To extend the current example to get some data into the temp-table and display it:

1. Add a HANDLE variable for a dynamic query and some variables to display data in:

```
/* Procedure h-testDynTT.p -- test dynamic temp-table
methods. */
DEFINE VARIABLE htt      AS HANDLE      NO-UNDO.
DEFINE VARIABLE htTBuf  AS HANDLE      NO-UNDO.
DEFINE VARIABLE hQuery  AS HANDLE      NO-UNDO.
DEFINE VARIABLE iSeq    AS INTEGER     NO-UNDO INIT 1000.
DEFINE VARIABLE cName   LIKE Customer.NAME    NO-UNDO.
DEFINE VARIABLE iCNum   LIKE Customer.CustNum  NO-UNDO.
DEFINE VARIABLE cRep    LIKE SalesRep.RepName NO-UNDO.
```

2. After the TEMP-TABLE-PREPARE method, capture the DEFAULT-BUFFER-HANDLE in a variable:

```
hTT:TEMP-TABLE-PREPARE("CustSequence").  
hTTBuf = hTT:DEFAULT-BUFFER-HANDLE.
```

3. Define a static FOR EACH block to populate the temp-table with **Customer** and **SalesRep** values for **Customers** in New Hampshire, as well as a unique sequence value:

```
/* Populate the temp-table with values from the database. */  
FOR EACH Customer WHERE State = "NH", SalesRep OF Customer:  
    hTTBuf:BUFFER-CREATE().  
    hTTBuf:BUFFER-COPY(BUFFER Customer:HANDLE).  
    hTTBuf:BUFFER-COPY(BUFFER SalesRep:HANDLE).  
    hTTBuf:BUFFER-FIELD("Sequence"):BUFFER-VALUE = iSeq.  
    iSeq = iSeq + 1.  
END.
```

4. Create a dynamic query for the temp-table buffer, prepare it to iterate through all the records in the temp-table, and open it:

```
/* Now create a query for the temp-table buffer and display values. */  
CREATE QUERY hQuery.  
hQuery:SET-BUFFERS(hTTBuf).  
hQuery:QUERY-PREPARE("FOR EACH CustSequence").  
hQuery:QUERY-OPEN().  
hQuery:GET-FIRST().
```

5. Walk through the query's result list and copy some fields from the temp-table into the variables you defined. Display the values in a frame:

```

REPEAT WHILE NOT hQuery:QUERY-OFF-END:
    ASSIGN iSeq = hTTBuf:BUFFER-FIELD("Sequence") : BUFFER-VALUE
    cName = hTTBuf:BUFFER-FIELD("Name") : BUFFER-VALUE
    iCNum = hTTBuf:BUFFER-FIELD("CustNum") : BUFFER-VALUE
    cRep = hTTBuf:BUFFER-FIELD("RepName") : BUFFER-VALUE.
    DISPLAY iSeq FORMAT "9999" LABEL "Sequence"
        iCNum
        cName FORMAT "X(20)"
        cRep FORMAT "X(20)" WITH FRAME CustSeqFrame 12 DOWN.
    hQuery:GET-NEXT() .
END.

```

In a more general-purpose procedure, you could create dynamic fill-ins for each of the fields you want to display and capture their format, label, and other attributes from the buffer fields of the temp-table.

6. Run the procedure:

| Sequence | Cust Num | Name | Rep Name |
|----------|----------|----------------------|----------------------|
| 1000 | 66 | First Down Football | Harry Munvig |
| 1001 | 1257 | Center Harbor Sport | Pitt , Dirk K. |
| 1002 | 1258 | Joe Jone's Ski Shop | Jan Loopsnel |
| 1003 | 1594 | Franconia Sport Shop | Donna Swindall |
| 1004 | 1595 | Tonto Sports | Robert Roller |
| 1005 | 1598 | Sports Stuff Inc | Robert Roller |
| 1006 | 1599 | True Sport Inc | Robert Roller |
| 1007 | 1600 | Waite Sports Special | Pitt , Dirk K. |
| 1008 | 1601 | Covered Bridge Sport | Brawn , Bubba B. |
| 1009 | 1602 | Exeter Sports Shop | Smith , Spike Louise |
| 1010 | 1603 | C2 Board Shop | Brawn , Bubba B. |
| 1011 | 1604 | Ducret's Sporting Go | Gilles Ehrer |

Press space bar to continue.

The primary index for the temp-table is the **Sequence** index, so the records come back in that order.

Temp-table parameters

There are several different ways you can pass a temp-table as a parameter between two procedures. It is important that you understand the differences between these, because what Progress does in the background to pass a temp-table from one place to another is complex and involves a considerable amount of overhead that is not visible to your application code directly. After all, a temp-table isn't just a single data value. It's a whole set of rows of data values, packaged up as if it were a database table. Not only that, but Progress can pass the entire *description* of the table along with the data, and it is critical to understand when you need to include the description and what the benefits are.

Using the TABLE form

When you pass a static temp-table between two procedures, you can use the INPUT TABLE, INPUT-OUTPUT TABLE, and OUTPUT TABLE forms you learned about earlier. To review them, in the RUN statement in the calling program, you define a temp-table parameter in this way:

```
[ INPUT ] TABLE temp-table-name
| { INPUT-OUTPUT | OUTPUT } TABLE temp-table-name [ APPEND ]
```

In the procedure that is called, you define the table as a parameter in this way for an INPUT or INPUT-OUTPUT parameter:

```
DEFINE [ INPUT ] | INPUT-OUTPUT
PARAMETER TABLE FOR temp-table-name [ APPEND ].
```

For an OUTPUT temp-table parameter returned from a called procedure, you use this syntax:

```
DEFINE OUTPUT PARAMETER TABLE FOR temp-table-name.
```

Remember that you must include a compatible static definition of the temp-table on both sides of the call. This means that at least the number and data types of fields must match in the two table definitions.

What happens when you pass a temp-table in this way? Progress packages up a description of the table and its fields and indexes, along with all the data in all the rows of the temp-table, and marshals it in a single stream from one procedure to the other. This means that the temp-table is copied into the procedure on the OUTPUT side of the call. To understand the implications of this, consider the two basic ways in which one procedure can call another.

Passing the TABLE within a session

Within a single run-time session, if one procedure calls another and passes a temp-table as a parameter, you wind up with two complete copies of the temp-table and all its data within that session. This is unnecessary in many cases and can be expensive. It takes a lot of memory to copy a large table, and (relatively speaking) a lot of time to copy it. In many cases, your application doesn't need two copies of the table. For this reason, you should generally avoid passing the table itself as a parameter when you can, and simply pass its HANDLE. Remember that for any kind of object, if you make the handle of the object available to another procedure within the session, the other procedure can access that object, its data, and its attributes regardless of where the object was defined or created. This holds true for a temp-table as well. Using the temp-table handle your procedure can get at its buffer, along with all the data in the table.

When should you consider passing the TABLE within a session? If both procedures use static temp-table definitions and static Progress 4GL language statements to access the table, then you need to pass the table itself instead of its handle. If you pass the handle, then the procedure that receives the temp-table can only manipulate it through its handle, using dynamic buffer and buffer field methods and attributes. If the two procedures have compatible but different definitions of the same table with, for example, different field names for the fields in the same relative positions within the temp-table buffer, you can successfully pass this table from one procedure to the other. In the course of copying it, Progress moves the data into the fields as they are defined in the procedure that receives the table.

If you need to pass the table itself because your procedures need to manage it with static 4GL statements, then do what you can to minimize the amount of data you pass. If the procedure is only going to look at a subset of the records in the temp-table (for example, those that have been changed since they were created or since the receiving procedure last saw them), then it might be more efficient to create a second smaller temp-table containing only those records that really need to be passed. If you simply keep in mind that there is an overhead to passing the table, this should influence your design and programming to minimize the overhead.

Passing the TABLE between sessions

If you need to pass a temp-table from one session to a completely separate session, and at least you have a static temp-table definition on the sending side, then you should use the TABLE form. Object handles aren't meaningful between sessions because they define memory locations within a single session. And when you pass a handle you are passing only a pointer to that memory location, not the contents. For this reason, if you need to pass a static temp-table from an AppServer session (that, for example, reads data out of the database and loads it into the table) to a client session that needs to use the data without a database connection, then use the TABLE form. You have no choice but to copy all the data from one session to the other. Again, the two temp-table definitions do not need to be identical. They need to have the same signature, the same number of fields, and the same data types in the same order.

Using the HANDLE form

Within a single session, you can simply pass a handle to the temp-table, and you should do this wherever possible. This is far faster than passing and copying the table. In deciding whether to use the TABLE or HANDLE parameter form, you need to decide whether it is important for the receiving procedure to operate on the temp-table as a static object. As with other object types, you can access and manipulate a temp-table using just its handle, but you have to do it with dynamic, handle-based statements. You can write a static FIND or FOR EACH statement against a temp-table handle. Think about how best to balance efficiency against ease of programming in each case where you need to pass temp-table parameters.

Using the TABLE-HANDLE form

The third parameter form is unique to temp-tables, and this is the TABLE-HANDLE. You can use a TABLE-HANDLE to either pass or receive a dynamic temp-table, just as you would use the TABLE parameter form to pass a static temp-table.

To run a procedure and pass a temp-table to or from the procedure using a table handle, use this syntax:

```
RUN procedure ( { [INPUT] | OUTPUT | INPUT-OUTPUT }
    TABLE-HANDLE tt-handle ).
```

In the called procedure, you define the parameter like this:

```
DEFINE { [INPUT] | OUTPUT | INPUT-OUTPUT }
PARAMETER TABLE-HANDLE tt-handle.
```

The `tt-handle` handle itself is exactly the same value you would pass as an ordinary HANDLE. However, the TABLE-HANDLE keyword tells Progress to pass not just the handle value but the entire definition and contents of the table as well, in exactly the same form as the TABLE parameter form uses.

You use the TABLE-HANDLE form to pass a dynamic temp-table and its description to another procedure, or to receive a dynamic temp-table, presumably from a procedure in another session on the other side of an AppServer connection, where you cannot simply pass the HANDLE.

The flexibility the TABLE-HANDLE form provides you is extremely valuable. For example, you might have procedures running on the server that represent business logic defined against static tables. On that side of the application, you can build static temp-tables that include database fields, calculated fields, and other elements. You can then pass the temp-table to the client using the TABLE parameter form, since you have a static temp-table definition locally.

On the client side of the application, you might have general purpose procedures to retrieve temp-tables from the server, perhaps to display data, allow updates, and do other client-side processing that might apply to many different tables received from the server. A general-purpose procedure that has no specific single temp-table definition can receive the table as a TABLE-HANDLE. It receives the entire table definition and data from the caller and can access it through the handle.

All combinations of TABLE and TABLE-HANDLE are valid. You can pass a static table using the TABLE form and receive it as a dynamic TABLE-HANDLE. You can pass a TABLE and receive it as a static TABLE of the same or compatible definition. You can pass a dynamic TABLE-HANDLE and receive it as a static TABLE, and you can pass a dynamic TABLE-HANDLE and receive it as a dynamic TABLE-HANDLE on the other side.

Use the TABLE-HANDLE form when the temp-table is not defined locally and you need to access it in a general way through its handle and buffer handle. Use the TABLE form when the temp-table *is* defined locally with a static DEFINE statement.

To review, Table 20–1 summarizes the possible combinations of temp-table parameter definitions in the procedure making the call and the procedure being called, along with their effects on the temp-table.

Table 20–1: Temp-table parameter definitions

(1 of 2)

| Caller RUN statement form | Callee parameter form | Parameter mode | Result |
|---------------------------|-----------------------|----------------|--|
| TABLE ttXYZ | TABLE ttXYZ | INPUT | Static temp-table ttXYZ in the caller is copied to the static definition of ttXYZ in the callee. The table definition is passed along with the data for validation only. |
| TABLE ttXYZ | TABLE-HANDLE hTT | INPUT | Definition and data of static temp-table ttXYZ in the caller are copied to the callee, which constructs a dynamic definition using handle hTT and loads the dynamic table with the data. |
| TABLE-HANDLE hTT | TABLE ttXYZ | INPUT | Definition and data of dynamic temp-table whose handle is hTT in the caller are copied to the callee, which receives the data into its static definition ttXYZ. |
| TABLE-HANDLE hTT | TABLE-HANDLE hTT | INPUT | Definition and data of dynamic temp-table whose handle is hTT in the caller are copied to the callee, which receives the definition and uses it to construct a dynamic temp-table using handle hTT, then loads the data into this dynamic table. |

Table 20–1: Temp-table parameter definitions

(2 of 2)

| Caller RUN statement form | Callee parameter form | Parameter mode | Result |
|--|--|-----------------------|--|
| TABLE ttXYZ TABLE ttXYZ TABLE-HANDLE hTT TABLE-HANDLE hTT | TABLE ttXYZ TABLE-HANDLE hTT TABLE ttXYZ TABLE-HANDLE hTT | OUTPUT | All the same combinations are supported. Nothing is passed in to the callee. The definition of the table and its data are passed back in the same form from callee to caller when callee returns. For the OUTPUT TABLE form, the definition is used to validate compatible temp-table definitions; for the OUTPUT TABLE-HANDLE form, it is used to construct the temp-table in the caller. |
| TABLE ttXYZ TABLE ttXYZ TABLE-HANDLE hTT TABLE-HANDLE hTT | TABLE ttXYZ TABLE-HANDLE hTT TABLE ttXYZ TABLE-HANDLE hTT | INPUT-OUTPUT | Once again, the same combinations are supported. The table definition and data are passed in from caller to callee. Callee can make changes to the data in the table, which is returned by being copied back to the caller. |

Creating and using dynamic browses

Especially when you start using dynamic temp-tables and defining their fields at run time, you will want to be able to define a dynamic browse objects to display their contents. You can also use a dynamic browse if you have a dynamic query on a database table that isn't known until run time, or anytime the column list of the browse needs to be definable at run time.

Much like a dynamic temp-table, you define a dynamic browse in stages, first defining the browse object itself with the CREATE BROWSE statement, and then adding columns to it in separate statements.

At run time, you can also add columns to a static browse control and define the query for a static browse.

You create the browse with this statement:

```
CREATE BROWSE browse-handle [ IN WIDGET-POOL pool-name ]  
[ ASSIGN attribute = value [ , . . . ] ].
```

As with other dynamic objects, if you don't assign a value to the attributes in the CREATE statement you can set them later on using the browse handle.

Attributes you can set for a dynamic browse include:

- **X and Y, or COLUMN and ROW** — The position of the browse.
- **WIDTH** — The width of the browse in characters.
- **DOWN** — The number of rows to display in the browse viewport.
- **TITLE** — A title for the browse object.
- **FRAME** — The handle of the frame to parent the browse to.
- **QUERY** — The handle of the associated query for the browse.
- **SENSITIVE** — Set to true to enable the browse. Note that this attribute enables only the browse object itself, allowing the user to scroll and otherwise manipulate the browse. You enable individual browse columns separately.
- **VISIBLE or HIDDEN** — Set VISIBLE to true to force the browse to be viewed. Set HIDDEN to false to make sure the browse is viewed when its container is viewed.

- **READ-ONLY** — Set to true to make all columns read-only. Set to false to allow individual columns to be enabled.
- **SEPARATORS** — Set to true to get lines between the browse columns.
- **ROW-MARKERS** — Set to true to have row markers at the beginning of each row, or false to remove them.

You must set a browse's query handle before you can add columns to the browse, and you must parent it to a frame before you can visualize it.

A dynamic browse always has the NO-ASSIGN quality, meaning that Progress is not able to automatically save changes you make to enabled columns in the browse. This is because the browse is completely defined at run time, and Progress can't anticipate and supply default behavior for column assignment at compile time. However, in a distributed application you would never make direct changes to a database table from a browse anyway, so the utility of the static browse that can do auto-assignment of changes is very limited.

Adding columns to the browse

After you create the browse you must add columns to it individually. There are three methods you can use to do this: ADD-COLUMNS-FROM, ADD-LIKE-COLUMNS, and ADD-CALC-COLUMN. These are much like the dynamic temp-table methods: ADD-FIELDS-FROM, ADD-LIKE-FIELD, and ADD-NEW-FIELD.

ADD-COLUMNS-FROM method

The ADD-COLUMNS-FROM method takes a table name (which can be a database table or a temp-table) and an optional list of fields to omit from the browse:

```
browse-handle:ADD-COLUMNS-FROM ( table-handle | table-name-exp  
[, except-list] ).
```

This method returns true if it succeeds and false otherwise.

As with the similar temp-table methods, you can specify the table using its handle or using a character expression that evaluates to its name.

The method creates a column in the browse for every field in the table you pass, except those in the *except-list*.

ADD-LIKE-COLUMN method

The ADD-LIKE-COLUMN method adds a single column to the browse whose attributes are taken from a field either in a database table that is connected when the procedure is compiled or in a temp-table that is in scope when the method is encountered:

```
buffer-handle:ADD-LIKE-COLUMN (fieldname-exp | buffer-field-handle  
[, pos]).
```

You can also pass an optional second argument that specifies the position of the column within the browse. If you don't specify the position, the column is added to the end of the column list for the browse.

This method returns the handle of the browse column created. You might want to save off the browse column handle, or you can access it later through the browse handle.

ADD-CALC-COLUMN method

The ADD-CALC-COLUMN method adds a single column to the browse that is not derived from a field in a database table or temp-table. You can use this method to add a field you use to display a calculated value:

```
buffer-handle:ADD-CALC-COLUMN ( data-type-exp , format-exp ,  
initial-value-exp , label-exp [, pos]).
```

This method also returns the handle of the browse column created. You must specify a data type, format, initial value, and label for the column. You can also optionally specify an ordinal position within the list of browse columns. If you don't specify the position, each new column is added to the end of the list.

Extending the test procedure with a dynamic browse

In this section, you'll apply a few of the things you just learned to add a dynamic browse to the procedure with the dynamic temp-table.



To extend the sample procedure with a dynamic browse:

1. Open the `h-testDynTT.p` procedure and save a new version of it as `h-testDynBrowse.p`.
2. Add a variable definition at the top for a browse handle and also a frame definition for a frame to hold the browse:

```
DEFINE VARIABLE hBrowse AS HANDLE      NO-UNDO.  
DEFINE FRAME BrowseFrame WITH SIZE 80 BY 10.
```

3. Remove the REPEAT block from the procedure that walks through the temp-table records using the dynamic query and displays them. Instead, you'll display them in a dynamic browse.
4. Add a CREATE BROWSE statement after the QUERY-PREPARE. It will be the only object in the frame, so it can be at ROW 1 and COLUMN 1. Its WIDTH needs to be slightly less than the frame to allow for the frame border. Parent it to the frame and assign the dynamic query to it. Make it SENSITIVE and VISIBLE, and give it SEPARATORS between columns but no ROW-MARKERS at the beginning of the row:

```
CREATE BROWSE hBrowse  
ASSIGN ROW = 1 COL = 1  
WIDTH = 79 DOWN = 10  
FRAME = FRAME BrowseFrame:HANDLE  
QUERY = hQuery  
SENSITIVE = YES  
SEPARATORS = YES  
ROW-MARKERS = NO  
VISIBLE = YES.
```

5. To get started, try adding all the columns from the temp-table buffer except for a handful that you exclude, using this statement:

```
hBrowse:ADD-COLUMNS-FROM(hTTBuf,  
"SalesRep,Country,address,Address2,State").
```

Note that you cannot pass the dynamic temp-table name as the table identifier for this method. Even though you do give the temp-table a name when you prepare it, that name is not available to the method. You must pass the handle to a buffer for the temp-table, either its DEFAULT-BUFFER-HANDLE or another buffer you've defined for it.

6. Enable everything in the frame and wait for the user to close the window so that the user can manipulate the browse when the window comes up:

```
ENABLE ALL WITH FRAME BrowseFrame.  
WAIT-FOR CLOSE OF CURRENT-WINDOW.
```

7. Run the procedure. You should see all the **Customers** in New Hampshire with all the fields except the ones you excluded:

| Cust Num | Name | City | Postal Code | Country |
|----------|------------------------------|---------------|-------------|---------|
| 66 | First Down Football | Loudon | 03301 | USA |
| 1257 | Center Harbor Sport Shop | Center Harbor | 03226 | USA |
| 1258 | Joe Jone's Ski Shop & Sports | Campton | 03223 | USA |
| 1594 | Franconia Sport Shop | Franconia | 03580 | USA |
| 1595 | Tonto Sports | Freedom | 03836 | USA |
| 1598 | Sports Stuff Inc | Concord | 03301 | USA |
| 1599 | True Sport Inc | Concord | 03301 | USA |
| 1600 | Waite Sports Specialists | Concord | 03301 | USA |
| 1601 | Covered Bridge Sport Shop | Dover | 03820 | USA |
| 1602 | Exeter Sports Shop | Exeter | 03833 | USA |

Enter data or press ESC to end.

**To try adding specific columns to the browse:**

1. Remove the ADD-COLUMNS-FROM method and instead add these four statements to add these four columns to the empty browse:

```
hBrowse:ADD-LIKE-COLUMN(hTTBuf:BUFFER-FIELD("Sequence")).  
hBrowse:ADD-LIKE-COLUMN(hTTBuf:BUFFER-FIELD("CustNum")).  
hBrowse:ADD-LIKE-COLUMN(hTTBuf:BUFFER-FIELD("Name")).  
hBrowse:ADD-LIKE-COLUMN(hTTBuf:BUFFER-FIELD("RepName")).
```

2. Run the procedure again. You should see just the four columns you specified:

| Sequence | Cust Num | Name | Rep Name |
|----------|----------|------------------------------|----------------------|
| 1000 | 66 | First Down Football | Harry Munvig |
| 1001 | 1257 | Center Harbor Sport Shop | Pitt , Dirk K. |
| 1002 | 1258 | Joe Jone's Ski Shop & Sports | Jan Loopsnel |
| 1003 | 1594 | Franconia Sport Shop | Donna Swindall |
| 1004 | 1595 | Tonto Sports | Robert Roller |
| 1005 | 1598 | Sports Stuff Inc | Robert Roller |
| 1006 | 1599 | True Sport Inc | Robert Roller |
| 1007 | 1600 | Waite Sports Specialists | Pitt , Dirk K. |
| 1008 | 1601 | Covered Bridge Sport Shop | Brawn , Bubba B. |
| 1009 | 1602 | Exeter Sports Shop | Smith , Spike Louise |

Browse columns and validation expressions

This section discusses the relationship between validation expressions and browse columns.



To see an example of this relationship:

1. Add the **SalesRep** field to the column list, with this statement:

```
hColumn = hBrowse:ADD-LIKE-COLUMN(hTTBuf:BUFFER-FIELD("SalesRep")).
```

2. Run the procedure. You should see this error message:



The reason for this error is that in the Sports2000 database there is a validation expression defined on the **Customer.SalesRep** field, which the temp-table column has inherited. This validation uses a CAN-FIND expression to check to make sure that the value in the **SalesRep** field matches the **SalesRep** field in a record in the **SalesRep** table. The field **SalesRep.SalesRep** (it's somewhat confusing that the table name and field name are the same) is the primary key for this value. **Customer.SalesRep** is the foreign key. Progress cannot process a CAN-FIND for a dynamic browse column, so if you want to include such a column, you need to exclude the column validation.

3. To do this, add a statement to blank out the validate expression before you add the column to the browse:

```
hTTBuf:BUFFER-FIELD("SalesRep") :VALIDATE-EXPRESSION = "".  
hColumn = hBrowse:ADD-LIKE-COLUMN(hTTBuf:BUFFER-FIELD("SalesRep")).
```

4. To see another variation, add a calculated column to the browse. This column will hold the difference between the **Customer CreditLimit** and **Balance** fields. It's a DECIMAL field with no extent and a label of **Available**. Place the new column in position 4 within the list of browse columns:

```
hColumn =  
hBrowse:ADD-CALC-COLUMN("DECIMAL","ZZ,ZZZ,ZZ9.99",0,"Available",4).
```

5. To populate the field for each row in the query, you need to define a ROW-DISPLAY trigger for the browse. You can add the trigger as a separate ON ROW-DISPLAY OF hBrowse statement, or in a TRIGGERS block at the end of the CREATE BROWSE statement:

```
CREATE BROWSE hBrowse
.
.
.
VISIBLE = YES
TRIGGERS:
  ON ROW-DISPLAY
    hColumn:SCREEN-VALUE =
      STRING(hTTBuf:BUFFER-FIELD("CreditLimit") : BUFFER-VALUE
      - hTTBuf:BUFFER-FIELD("Balance") : BUFFER-VALUE) .
END.
```

When the trigger executes, the handle of the column must be in the hColumn variable. You need to define the ROW-DISPLAY trigger before the calculated field is added to the browse, and before the query is opened. You can modify the SCREEN-VALUE and certain other attributes of any field within the trigger.

6. Move the QUERY-OPEN method after the statements that create the browse and its columns so that rows are initialized properly with the calculated value:

```
hColumn =
hBrowse:ADD-CALC-COLUMN("DECIMAL","ZZ,ZZZ,ZZ9.99",0,"Available",4).

hQuery:QUERY-OPEN().
```

If you open the query before you create the browse and its trigger, the first rows of the query appear in the browse viewport, but the calculated field isn't assigned to them. Only when you scroll down through the browse do the values for the calculated field appear. This is why it's important to define the trigger and add the column before you open the query.

7. Run the procedure:



The screenshot shows a static browse window titled 'P' with a grid of data. The columns are labeled 'Sequence', 'Cust Num', 'Name', 'Available', and 'Rep Name'. The data includes various customer entries such as 'First Down Football', 'Center Harbor Sport Shop', and 'Franconia Sport Shop'. A message at the bottom of the window says 'Enter data or press ESC to end.'

| Sequence | Cust Num | Name | Available | Rep Name |
|----------|----------|------------------------------|-----------|-------------|
| 1000 | 66 | First Down Football | 81,051.33 | Harry Mu |
| 1001 | 1257 | Center Harbor Sport Shop | 47,200.00 | Pitt , Dirk |
| 1002 | 1258 | Joe Jone's Ski Shop & Sports | 12,222.73 | Jan Loop |
| 1003 | 1594 | Franconia Sport Shop | 6,800.00 | Donna Si |
| 1004 | 1595 | Tonto Sports | 20,563.38 | Robert Ri |
| 1005 | 1598 | Sports Stuff Inc | 5,682.42 | Robert Ri |
| 1006 | 1599 | True Sport Inc | 48,200.00 | Robert Ri |
| 1007 | 1600 | Waite Sports Specialists | 4,300.00 | Pitt , Dirk |
| 1008 | 1601 | Covered Bridge Sport Shop | 25,500.00 | Brawn , B |
| 1009 | 1602 | Exeter Sports Shop | 4,000.00 | Smith , S |

Adding columns to a static browse

You can use the ADD-COLUMNS-FROM, ADD-LIKE-COLUMN, and ADD-CALC-COLUMN methods to add columns to a static browse as well. Such a browse automatically becomes a NO-ASSIGN browse, just as a dynamic browse is.

You can also modify the query of a browse at run time, including the query of a static browse. If the query navigates a table with the same field list as the original query, you can change the query without any visible effect, other than to substitute one set of records for another. There is no need to rebuild the browse. However, if the field list in the table managed by the query is different from the previous query, Progress clears the browse column list and you have to rebuild it using the dynamic methods.

Query methods for use with browses

This section describes the two query methods you can use with browses.

CREATE-RESULT-LIST-ENTRY method

This method creates an entry in the result list for the current row. You use the CREATE-RESULT-LIST-ENTRY method in conjunction with new browse rows or new query rows to synchronize the data with the query.

DELETE-RESULT-LIST-ENTRY method

This method deletes the current row of a query's result list.

The dynamic CALL object

Sometimes you need to be able to run a number of different procedures from the same place in your application. These might be, for example, procedures that handle different values for a parameter or a field in specialized ways.

You can make the name of a procedure to run a variable by using the VALUE function in a RUN statement:

```
RUN VALUE(cProcName) (parameters).
```

The one significant limitation of this dynamic name is that you must fully specify the parameter list in the source procedure. Therefore, you must use the same dynamic name in all the possible procedures you might run. In some limited cases, you might need to build a general purpose procedure that is prepared to run any procedure with any parameter list. This would certainly not be for ordinary situations, but only for a specialized tool that acts as a gateway for a number of different procedure calls that all must be handled in a consistent way.

Progress provides a dynamic CALL object to handle this need. Using the CALL object, you can specify the procedure name, the number of parameters, their types and values, and other information as run-time attributes of this dynamic object, and then invoke it to do the RUN. You can also use a dynamic CALL object to access object attributes dynamically.

Because your use of this object is likely to be limited to very special situations, it is not described here. The complete syntax for dynamic CALL, along with some simple examples, is in *OpenEdge Development: Progress 4GL Reference* and online Help.

Building a comprehensive example

This section examines a more comprehensive example of a useful set of procedures that include dynamic objects to display data from any database table or temp-table. It serves as a summary and review of all the material in this and the previous chapters on dynamic objects. The code is somewhat too substantial to describe in full, so the material refers to selected parts of the procedures. You can then examine the final procedures in the examples directory.

The main procedure is `h-dbbrowser.w`, which puts up a window where you can select a database (if there's more than one connected), a table, and one or more fields from that table. The procedure displays those fields in a dynamic browse. [Figure 20–2](#) shows an example using the **Customer** table from the Sports2000 database.

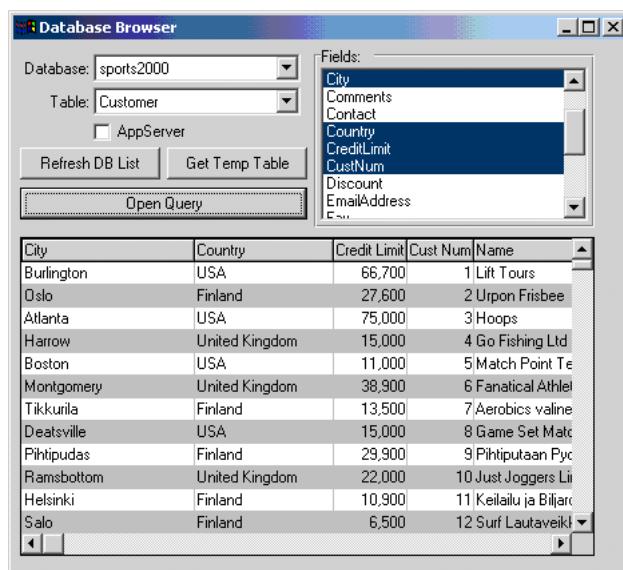


Figure 20–2: Result of running `h-dbbrowser.w`

Look at some of the code that makes this happen.

The main block of `h-dbbrowser.w` runs an `initializeObject` procedure and then a custom `enableUI`:

```
DO ON ERROR UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
  ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:
    RUN initializeObject.
    RUN enableUI.
    IF NOT THIS-PROCEDURE:PERSISTENT THEN
      WAIT-FOR CLOSE OF THIS-PROCEDURE.
  END.
```

Creating a dynamic browse and customizing its display

The `initializeObject` procedure creates a dynamic browse whose columns are filled in when you select fields for a table. It also defines a `ROW-DISPLAY` trigger that fires as each row in the browse is displayed:

```
/* Create the dynamic browser here */
CREATE BROWSE hBrowse
ASSIGN
  ROW = 8
  COLUMN = 2
  WIDTH = 86
  DOWN = 12
  VISIBLE = NO
  ROW-MARKERS = NO
  SEPARATORS = YES
  COLUMN-RESIZABLE = YES
  COLUMN-MOVABLE = YES
  NO-VALIDATE = YES
  FRAME = FRAME {&FRAME-NAME}:HANDLE

/* The trigger clause is here to do the alternate line color in the broswe */
TRIGGERS:
  ON ROW-DISPLAY PERSISTENT RUN rowDisplay IN THIS-PROCEDURE.
END TRIGGERS
```

The ROW-DISPLAY event lets you intercept the display of each row in the browse to change colors, formats, or calculated values. If you look at the trigger procedure that handles that ROW-DISPLAY event, you see that it is responsible for alternating the browse rows between white and gray. The `lRow` logical variable is defined in the **Definitions** section for the main procedure block so that its value is maintained between calls to `rowDisplay`:

```
/* Static toggle switch for the line color */
DEFINE VARIABLE lRow AS LOGICAL NO-UNDO.
```

The `rowDisplay` procedure sets foreground and background colors depending on the setting of `lRow`:

```
/* If the toggle is yes */
IF lRow THEN
  ASSIGN
    iBGCColor = 8 /* Set the background color grey */
    iFGColor = 0 /* and foreground color black */

ELSE
  ASSIGN
    iBGCColor = 15 /* else background color white */
    iFGColor = 0 /* and foreground color black */
```

It then assigns these colors for each cell in the row. There's no attribute that lets you do this for the entire row at once:

```
/* Iterate through the list of browse columns */
DO iCount = 1 TO NUM-ENTRIES(cBrwsCols):
  /* Convert the string value to a handle */
  hBrwsCol = WIDGET-HANDLE(ENTRY(iCount,cBrwsCols)).

  IF VALID-HANDLE(hBrwsCol) THEN
    DO:
      hBrwsCol:BGCOLOR = iBGCColor. /* Set the cell's background color */
      hBrwsCol:FGCOLOR = iFGColor. /* and it's foreground color */
    END.
  END.
```

Finally, it reverses the value of `lRow` in preparation for the next call:

```
lRow = NOT lRow. /* Set the toggle opposite */
```

Reading the database metaschema data

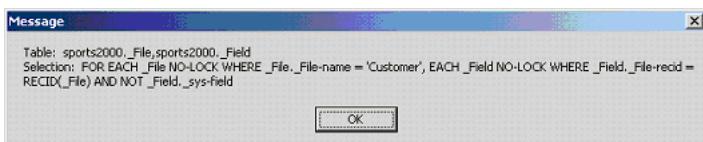
Next, take a look at the `getListItems` procedure, which populates the combo boxes and the selection list in the window.

This procedure is run, for example, to populate the list of fields for a selected database table. It receives the filename or filenames and a query definition as input parameters.

This is an opportunity to share a bit of information about how Progress defines tables and fields in its database. All these definitions are stored in a set of database tables called the *metaschema*. These tables hold all the information about the database tables that make up your application. All these special table names begin with an underscore (_), which is not valid for your application tables and helps distinguish them from ordinary database tables:

- Information about database tables is stored in the table named `_File`.
- Information about fields in those tables is stored in the table named `_Field`.
- Information about indexes is stored in `_Index`, and the cross-references that associate fields with indexes is in `_Index-Field`.

For example, when you select the **Customer** table in the window, `getListItems` is run with this table list and query selection:



The `_File` and `_Field` tables are joined using a field in the `_Field` table called `_File-Recid` and the `RECID` of the `_File` record. The `RECID` is an integer Progress record identifier that was used in earlier times as the standard way to uniquely identify a database record. It has since been superceded by the `RowID` that you've been introduced to in this book, which is more portable to other databases and has other advantages. In any case, the `RECID` still exists and is used as the mechanism for joining tables in the metaschema. You should avoid using it in application code because the `RowID` is now the preferred row identifier. The `_sys-field` field tells you if this is flagged as a system field that isn't intended to be seen as part of the application information.

There's just one more piece of information to share about the metaschema tables. Each `_File` record has a `_File-Num` field. For tables that are part of your application schema, the `_File-Num` is greater than 0 and less than 32K. For metaschema tables, the `_File-Num` is less than 0. For virtual system tables, which you can learn more about in the database administration documentation, the `_File-Num` is greater than 32K. This also helps identify which tables are specific to your application and which are the support tables Progress uses to manage your application tables.

Knowing about the metaschema tables can help you understand the relational database structure better, and can be useful in cases where you want to browse or display all the tables in your database. However, using this information is only for advanced users and for special situations like this example.

Populating a selection list dynamically

`GetListItems` uses the table list and query to ask the `getQuery` procedure to generate a dynamic query for it. It then gets the field values for each record in the query (all the fields in the **Customer** table in this case), and uses them to populate the selection list:

```
/* Add this row to the list box */
ihListBox:ADD-LAST(cLabel,cValue).
```

Finally, it makes the first item in the list the selected item:

```
/* Set the value of the list-box to be the first entry in the list */
IF ihListBox:NUM-ITEMS > 0 THEN
    ihListBox:SCREEN-VALUE = ihListBox:ENTRY(1).
```

Creating a dynamic query

Now take a look at `getQuery`. It first creates a dynamic query, to pass back as an `OUTPUT` parameter (hence the naming convention `ohQuery`):

```
/* Create the query and assign the handle to the output parameter variable */
CREATE QUERY ohQuery.
```

Then it walks through the list of buffer names passed in, creates a dynamic buffer for each one, and adds each buffer to the query:

```
/* Create a buffer for this table */
CREATE BUFFER hTable FOR TABLE cTableName.

/* Add this buffer to the query */
ohQuery:ADD-BUFFER(hTable).
```

Finally, it prepares the dynamic query using the FOR EACH statement passed in:

```
/* Prepare the query */
ohQuery:QUERY-PREPARE(icForEach).
```

This is a general-purpose mechanism for creating not only any dynamic query, but also all the buffers it uses, without defining anything in advance. It's prepared to work against any list of tables for any database.

It's also a good idea to look at the cleanupQuery function, which reminds you to always delete dynamic objects when you're done with them. It deletes the query and then all the dynamic buffers the query used:

```
/* Now delete the query object */
deleteObject(hQuery).

/* Now iterate through the string */
DO iCount = 1 TO NUM-ENTRIES(cBuffers):
/* convert each entry to a handle */
hBuffer = WIDGET-HANDLE(ENTRY(iCount,cBuffers)).
/* and pass the handle to deleteObject */
deleteObject(hBuffer).
END.
```

Creating a dialog box procedure

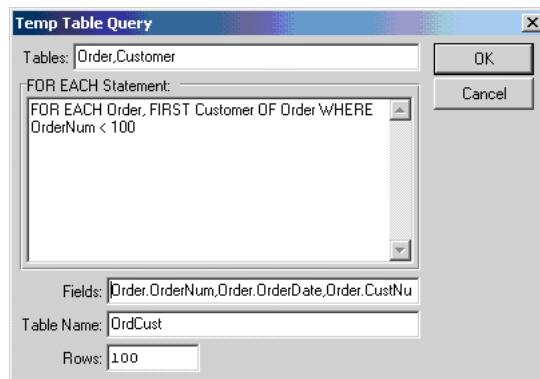
You can also use the h-dbbrowser window to define a temp-table with any fields you want, from one or multiple database tables, and then populate that temp-table with a query.



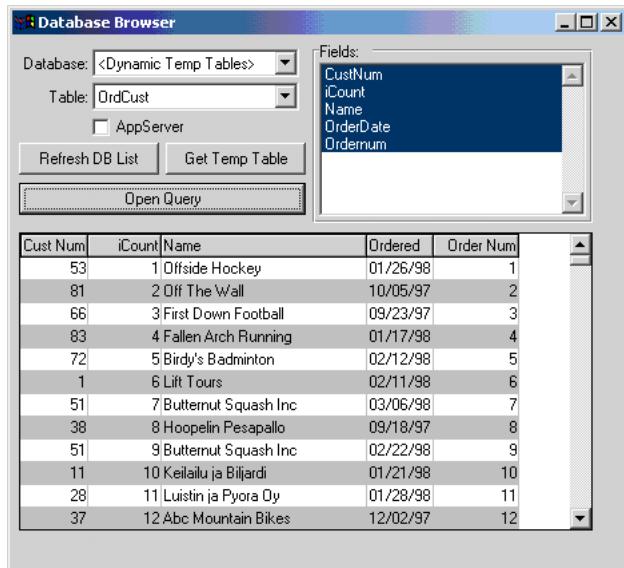
To create a dialog box procedure:

1. Choose the **Get Temp-Table** button. A dialog box appears where you enter the following information:
 - A list of database tables to base your temp-table on.
 - A **FOR EACH** statement to use to populate the temp-table.
 - A list of fields to add to the temp-table. You should qualify these fields with their database field names, but the temp-table fields will just use the base field names. The temp-table also gets a row counter field called **iCount**.
 - A name for the temp-table.
 - A maximum number of rows to retrieve into the temp-table.

Here's an example that joins **Order** to **Customer** and selects the **OrderNum**, **OrderDate**, **CustNum**, and **Customer.Name** fields for **Orders** less than 100:



2. Choose **OK** in this dialog box and select all its fields in the main window. You see the contents of the temp-table in the dynamic browse:



Here are a few interesting parts of this code:

The `getTempTable` procedure runs a separate dialog box procedure called `h-dlgtt.w`, built in the AppBuilder from the Dialog template (that is, by selecting **New→Dialog** in the AppBuilder). This is the **Temp-Table Query** dialog box used to define the `OrdCust` temp-table above. It is a fairly standard dialog box procedure, which prompts for some field values and returns them as OUTPUT parameters ON CHOOSE of the **OK** button. You can use a dialog box such as this anytime it is necessary to block other input in your application until the user has provided information the dialog box is prompting for.

Making a call to an AppServer

The code next checks the toggle box in the window that signals whether to retrieve the temp-table data from a separate AppServer session. A discussion of the details of how to use the OpenEdge AppServer is beyond the scope of this book, but this one example gives you a flavor of how straightforward it is to separate your client session from an AppServer session on another machine where the database is connected.

If the AppServer toggle box is set, the procedure creates an AppServer object, which has a handle like any other Progress object, and uses it to connect to the AppServer:

```
CREATE SERVER hAppServer.  
1Ans = hAppServer:CONNECT("-H localhost -S 5162 -AppService asbroker1").
```

The local hostname, server port number, and AppService name that are the parameters to the CONNECT method are the defaults for connecting to a local AppServer session for testing purposes.

If the AppServer flag was not set, then the procedure sets the hAppServer variable to be the handle of the local session:

```
/* Otherwise set the AppServer handle to be the SESSION handle */  
ELSE  
    hAppServer = SESSION:HANDLE.
```

This signals OpenEdge that there is no AppServer running, but that all requests to this AppServer handle should just be run within the local session.

Then any RUN statement, such as the following, can run a procedure ON SERVER hAppServer and have it run identically, whether there is really a separate AppServer session or not:

```
RUN h-gettemptable.p ON SERVER hAppServer  
(cTables, cForEach, cFields, cTableName, iRows, OUTPUT TABLE-HANDLE hTable).
```

As you can see, running a procedure on an AppServer is just like running it within your session, except for the ON SERVER syntax. Even with this syntax, you can run the same procedure locally with no change to your code whatsoever, just by using SESSION:HANDLE as the alternative value for the server handle.

Creating a dynamic temp-table

Next, look at a few parts of procedure `h-gettemptable.p`.

The main block creates a dynamic query for the database tables requested (such as **Order** and **Customer**). The `createBuffers` procedure creates a dynamic buffer for each of those tables.

The `getFieldHandles` procedure assembles a list of the handles to the fields requested (such as **Order.OrderNum**).

The `createTempTable` procedure creates a dynamic temp-table called `hTT`, and adds `iCount` to it as its first field:

```
/* Create the temp table */
CREATE TEMP-TABLE hTT.

/* Add a count field - we need this for the sequence so that we get the records
back in the order that they were created */
hTT:ADD-NEW-FIELD("iCount","INTEGER").
```

It then walks through the list of database field handles and adds a field to the temp-table like each field:

```
/* Iterate through the list of field handles in the cTableFields */
DO iCount = 1 TO NUM-ENTRIES(cTableFields):
  /* Convert the current entry to a handle */
  hField = WIDGET-HANDLE(ENTRY(iCount,cTableFields)).
  /* Add a field to the temp table like this field with the same name as this
   field */
  hTT:ADD-LIKE-FIELD(hField:NAME,hField).
END.
```

It then adds an index on `iCount`, so that the records remain in the order they were created:

```
/* Add a primary unique index to the temp-table */
hTT:ADD-NEW-INDEX("pudx",True,True).

/* Add the counter field to the index */
hTT:ADD-INDEX-FIELD("pudx","iCount").
```

Finally, it prepares the temp-table, which freezes its definition and enables you to add records to it:

```
/* Prepare the temp-table with the name the user chose */
hTT:TEMP-TABLE-PREPARE(icTableName).
```

Next, the h-gettemptable.p procedure populates the temp-table in the populateTempTable internal procedure. Handle hTTBuf holds the handle to the temp-table's default buffer:

```
/* Store the handle to the buffer for the temp-table */
hTTBuff = hTT:DEFAULT-BUFFER-HANDLE.
```

The procedure prepares the database query with the FOR EACH statement entered in the dialog box, opens it, and retrieves the first record:

```
/* Prepare a query using the string the user provided */
hQuery:QUERY-PREPARE(icForEach).
/* Open the query */
hQuery:QUERY-OPEN().
/* Get the first result in the result set */
hQuery:GET-FIRST().
```

For as many rows as there are in the query, or as many rows as you asked for, whichever is less, the procedure creates a temp-table row using the buffer handle:

```
/* Iterate for as many rows as the user has chosen, or until the query is
   off-end */
REPEAT iCount = 1 TO iiRows WHILE NOT hQuery:QUERY-OFF-END:
   /* Create a temp-table record */
   hTTBuff:BUFFER-CREATE().
```

Because you selected individual fields from possibly multiple tables in the database to add to the temp-table, it doesn't do a BUFFER-COPY to move fields from the database records to the newly created temp-table row. Instead, the next block of code uses the BUFFER-FIELD and BUFFER-VALUE attributes on both the database buffer fields and the temp-table buffer fields to move values from one to the other.

The procedure then releases each temp-table row, iterates through the query until the REPEAT condition fails, and then closes the query:

```
/* Release the temp-table record */
hTBBuff:BUFFER-RELEASE().
/* Get the next result in the query */
hQuery:GET-NEXT().
END.
/* Close the query */
hQuery:QUERY-CLOSE().
```

Using the SESSION handle to identify dynamic objects

One final procedure to look at in h-dbbrowser.w is listTempTables. This procedure populates the drop-down list of available temp-tables after you have built one or more of them using the dialog box. It illustrates the usefulness of the SESSION handle as a way to get at various kinds of objects that you've created since your session started. In this case, the session keeps track of a list of all dynamic buffers created in the session. The FIRST-BUFFER attribute gives you the first one. The NEXT-SIBLING attribute walks through the list. If a buffer has a TABLE-HANDLE, then it is a buffer for a temp-table, and the procedure adds it to the drop-down list:

```
hBuffer = SESSION:FIRST-BUFFER.

/* Walk through the list of buffers that belong to Dynamic Temp Tables*/
DO WHILE VALID-HANDLE(hBuffer):
  IF VALID-HANDLE(hBuffer:TABLE-HANDLE) THEN
    DO:
      /* Convert the handle to a string */
      cHandle = STRING(hBuffer:TABLE-HANDLE).
      /* If the handle is not in cList, add it */
      IF NOT CAN-DO(cList,cHandle) THEN
        cList = cList + (IF cList = "" THEN "" ELSE ",") +
               cHandle.
    END.
    /* Go on to the next Buffer */
    hBuffer = hBuffer:NEXT-SIBLING.
  END.

RETURN cList.    /* Function return value. */
```

This comprehensive example has used just about every aspect of programming with both dynamic visual objects and dynamic database objects. Remember that dynamic programming like this is valuable when you need to create a single procedure to use in a variety of situations, just as this window can display fields from any table.

Dynamic programming considerations

Using a dynamic object lets you make everything about the data run-time defined, including the table name, the WHERE clause for the records you select, and so on. It's important to note here some basic principles of dynamic programming that pertain specifically to transaction management.

The basic rule of dynamic programming is that, because you are defining a procedure's behavior at run time, it is not possible for Progress to provide as much default behavior as it does for a statically defined procedure or to give you compile errors for constructs that are incorrect. You have much more responsibility to make sure that your program functions properly at run time with the entire range of possible data input that can drive its dynamic behavior. Because of this, there are a few basic rules you need to strictly adhere to when you're doing dynamic programming. These rules apply to static programming as well, but they are more essential in a dynamic procedure. Partly, this is because a dynamic procedure provides less predictable default behavior and has more flexibility in what it might be expected to do at run time. Also, it is because you cannot use tools such as the LISTING file to confirm the record scope and transaction scope for your procedure, because Progress cannot determine them with certainty until run time.

Here are the rules:

- **It is essential that you explicitly define every transaction using the TRANSACTION keyword in a block header.** Because dynamic programming is largely independent of the normal block structure of Progress, your transaction scope is likely to be very unpredictable (and possibly larger than you expect).
- **Make sure you do not find a record with an EXCLUSIVE-LOCK or otherwise reference any buffer outside your defined transaction in such a way that you force the transaction scope to be larger than you expected.** For example, if you find a record, then start a transaction block with a DO TRANSACTION header, and then update that record, Progress forces the scope of the transaction at run time to be larger than the DO TRANSACTION block. The scope will likely encompass the entire procedure, and your procedure will hold records and locks longer than you want.

- **Always use the dynamic BUFFER-RELEASE method on each buffer handle to release the record when you are finished with it.** Again, this is because the default release handling is block-oriented and dynamic procedures are not tied to blocks as firmly as static procedures are.
- **Use the dynamic BUFFER-COPY method to copy multiple fields from one buffer to another, rather than assigning values to fields individually.** This is also true for a static buffer in that using an ASSIGN statement for more than a few fields in a buffer is more expensive than copying the entire buffer. It's especially true for dynamic programming, however, because in a static program you can at least ASSIGN multiple fields in a single statement, whereas in dynamic programming you can only assign a single field at a time.
- **When you run one procedure from another in a dynamic program, be especially careful that you do not reference record buffers or other objects in such a way as to adversely affect your transaction scope.** This mistake is very easy to make because you can reference an object that is defined anywhere in your application if you have access to its handle. This kind of access can drastically change transaction scope at run time in ways you cannot easily predict just by casually looking at the block structure of your procedures.
- **There is little reason to use SHARE-LOCKS in any case, but it is especially unwise to ever use a SHARE-LOCK in a dynamic procedure.** The transition from SHARE-LOCK to EXCLUSIVE-LOCK and back is obscure in the best of cases and very much so in a dynamic procedure. Using SHARE-LOCKS can lead to locks being held much longer than you expect.

Once again, these are valuable guidelines for any procedure, but especially so for one that uses dynamic statements.

The next and final chapter provides more guidelines on how to best develop applications with the Progress 4GL.

Progress Programming Best Practices

Here you have arrived at the final chapter of this guide to programming in Progress. Along the way you have learned a number of factors that can improve the performance and the effectiveness of the procedures in your Progress 4GL applications. This chapter repeats and reinforces some of those points and introduces you to additional language constructs and programming techniques to help you build the best-performing applications you can.

This chapter includes the following sections:

- [Writing efficient procedures](#)
- [Memory management in Progress](#)
- [Conclusion to the book](#)

Writing efficient procedures

There are many things you can do to maximize the efficiency of your Progress procedures and to avoid unnecessary overhead in your applications. This section discusses some of these.

Using NO-UNDO variables and temp-tables

When variables without the NO-UNDO qualifier are updated, a before-image of the previous value is generated in what amounts to a separate record buffer for all NO-UNDO variables. This is useful when you want the original value restored when you UNDO a block or transaction. When you do not need this capability, the before-image generation is needless overhead. Use of NO-UNDO also causes the Progress local before-image (.lbi) file, which is maintained for each user connected to a database, to be smaller. As noted in [Chapter 2, “Using Basic 4GL Constructs,”](#) when this guide first introduced the DEFINE VARIABLE statement, it is unusual for variables to require the UNDO support that they receive by default. So, it is a good practice to make the NO-UNDO keyword a standard part of your variable definitions unless a particular variable really needs the support.

With temp-tables the situation is not always as clear. There might be cases where you are adding records to a temp-table or changing records in a temp-table within a transaction, and you want to be able to undo those changes. If this is not the case, however, you’ll benefit significantly from defining the temp-table with the same NO-UNDO qualifier as for variables. This spares you from having Progress create a before-image of every temp-table change made within a transaction.

Grouping assignments with the ASSIGN statement

If you are assigning two or more values in a row, it is significantly faster to assign them all in a single statement using the ASSIGN keyword. Even if the values being assigned have nothing to do with one another, it is faster to do it in a single statement, as in this example:

```
ASSIGN THIS-PROCEDURE:PRIVATE-DATA = STRING(OrderNum)
   hSource = SOURCE-PROCEDURE
   shipDate:BGCOLOR = dateColor(PromiseDate, ShipDate).
```

Here the code is setting an attribute on a procedure handle, a variable, and a field attribute. Even though these are unrelated, it is still best to ASSIGN them together.

If you are assigning multiple field values within a single record buffer, the ASSIGN statement can be even more important. Progress adjusts index entries and does other work as part of each statement. If you assign two fields that participate in the same index in two separate statements, the index block is rebuilt once after each statement—a much greater overhead than doing it in one statement. In fact, because Progress assigns index entries at the end of each statement, you might even cause a temporary (but fatal) unique index violation if you assign part of a composite key in one statement and the other part in another. Don't ever do this.

Because of the efficiency of an ASSIGN statement without multiple assignments, some Progress developers always use the ASSIGN keyword even when the statement only does one assignment. There is no advantage to this. That is, these two statements are comparably fast:

```
Variable-a = Variable-b + 1.  
ASSIGN Variable-a = Variable-b + 1.
```

Using BUFFER-COPY and BUFFER-COMPARE

When copying fields from one record buffer to another, BUFFER-COPY does it more efficiently than ASSIGN. Use the EXCEPT option when some fields are not to be copied, and the ASSIGN option when some fields are renamed during the copy operation. Even doing a BUFFER-COPY of a table with 100 fields is faster than an ASSIGN statement that copies a half dozen of those fields.

In particular, the static BUFFER-COPY statement allows Progress to resolve the exact field list to copy and to identify the field mappings at compile time so that the operation can be made as fast as possible. Using the dynamic BUFFER-COPY method on a buffer handle requires Progress to evaluate the arguments during program execution, which of course is slower. As with all dynamic language, use the BUFFER-COPY method only when you truly don't know the buffers or the fields to copy until run time.

All this applies equally well to the BUFFER-COMPARE statement or method which compares two buffers and returns a list of differences. BUFFER-COMPARE is much faster than using a series of explicit comparison statements on individual fields.

Block-related tips

This section offers some advice on creating blocks of code.

Using DO instead of REPEAT

Back in [Chapter 6, “Procedure Blocks and Data Access,”](#) you learned about the various properties of different kinds of blocks. Remember that the DO block, by default, does not provide you with many of the default services that the REPEAT block does (for example, transaction management and default frame management). The flip side of this is that a DO block is much faster than a REPEAT block if you don’t need these services. So, use a simple DO block whenever possible for any kind of iterating block that doesn’t need to manage a transaction or iterate through a DOWN frame.

Minimizing block nesting

All blocks in Progress incur some overhead. Because Progress is a 4GL that provides many services to make programming easier and to make each statement do more than you’re used to from other languages, there is a cost to all blocks, even simple DO blocks. For this reason, you should avoid unnecessary block nesting wherever possible. Therefore, always use the form:

```
IF expression THEN statement.
```

Instead of this form:

```
IF expression THEN
  DO:
    statement.
  END.
```

The AppBuilder always gives you a DO-END block as a starting point for triggers, for instance. You should feel free to remove the block if your trigger requires only a single statement. And remember that there can be an additional benefit to combining multiple assignments into a single statement. If you turn several assignments into a single ASSIGN statement, a DO block that would otherwise have several statements can be reduced to just one statement with no block header.

Minimizing nesting of procedure calls

Procedures, whether internal or external, are blocks as well, and relatively expensive ones. Obviously, you should use procedures as necessary to structure your application properly. However, if you run a relatively small procedure or invoke a function many times in a performance-sensitive loop, you should consider moving the code directly into the procedure that calls it to execute it inline. If it's executed many times, this can make a significant difference in performance. If a procedure of this type is invoked from multiple places, you can make it into an include file and include it each place where it would otherwise be run. In this way, the code remains reusable but it is compiled directly in place wherever it is used. This can make the code much faster.

The CASE statement

If you have a sequence of IF, THEN, ELSE clauses that all operate on different values of the same variable or expression, you can combine these into a single block using the CASE statement, which has this syntax:

```
CASE expression :
  { WHEN value [ OR WHEN value ] ... THEN
    { block | statement }
  }
  ...
  [ OTHERWISE
    { block | statement }
  ]
END [ CASE ].
```

The *expression* can be a simple field or variable or any other expression involving multiple fields or values. Part of the optimization of the CASE statement is that it evaluates the expression only once, when the CASE statement is entered. By contrast, nested IF statements evaluate the expression in each IF clause, even if the expression is the same each time.

Following the block header are a number of WHEN clauses, each of which starts with a *value* for the *expression*, followed by THEN, followed by a statement or block to execute if the expression has that value. You can combine multiple WHEN clauses with OR if the same block or statement executes on multiple values of the expression.

Finally, the CASE block can conclude with an optional OTHERWISE clause with a statement or block to execute if the expression matches none of the values in the WHEN clauses.

The CASE statement is most useful when a variable or field can have one of a small number of possible values, and the procedure needs to react differently to each value.

Here's a simple example that uses the CASE statement to count the number of **Orders** of different types. There are four valid values for the **OrderStatus** field. The OTHERWISE clause tallies any that don't match any of the valid values (as it turns out, there aren't any):

```

DEFINE VARIABLE iOrderStatus AS INTEGER EXTENT 5      NO-UNDO.
FOR EACH Order:
  CASE Order.OrderStatus:
    WHEN "Shipped" THEN
      iOrderStatus[1] = iOrderStatus[1] + 1.
    WHEN "Ordered" THEN
      iOrderStatus[2] = iOrderStatus[2] + 1.
    WHEN "Back Ordered" THEN
      iOrderStatus[3] = iOrderStatus[3] + 1.
    WHEN "Partially Shipped" THEN
      iOrderStatus[4] = iOrderStatus[4] + 1.
    OTHERWISE
      iOrderStatus[5] = iOrderStatus[5] + 1.
  END CASE.
END. /* END FOR EACH Order */
DISPLAY iOrderStatus[1] LABEL "Shipped Orders" SKIP
      iOrderStatus[2] LABEL "Ordered Orders" SKIP
      iOrderStatus[3] LABEL "Back Ordered Orders" SKIP
      iOrderStatus[4] LABEL "Partially Shipped Orders" SKIP
      iOrderStatus[5] LABEL "Invalid Orders"
WITH FRAME DisplayFrame SIDE-LABELS.

```

Figure 21–1 shows the result.

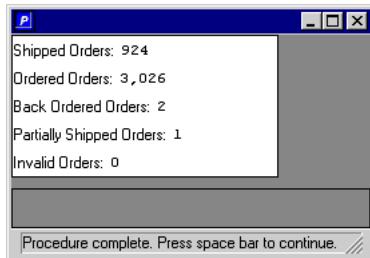


Figure 21–1: Result of CASE statement example

Using arrays instead of lists

This book cautions you against defining array fields with extents as part of your database definition, for a number of reasons. However, array *variables*, as well as array fields in temp-tables, can be very useful. One thing to keep in mind is that array element access is much faster than accessing an element of a list of comma-delimited strings. The CASE example above shows an example of this. The totals could have been accumulated within a single delimited list, but this would have involved scanning the list for the delimiters with each operation. The memory location of each element of an array is distinct and, therefore, quickly accessed.

Using FOR FIRST versus FIND

The FIND statement retrieves a single record in a single statement. It has its limitations, however, when compared to a FOR block. A FIND statement always uses only one index and does not allow use of field lists or word indexes. By contrast, a FOR block can use multiple indexes to evaluate a complex WHERE clause that involves fields not found in a single index. A FOR block can also use a word index and the CONTAINS keyword to locate records, and supports the use of a field list to retrieve only a subset of the fields from the selected records. Both word indexes and field lists are briefly described in the “[Using indexes properly](#)” section on page 21–14 and the “[Defining efficient queries and FOR EACH statements](#)” section on page 21–16. You can use the form FOR FIRST when fetching just a single record, if any of the conditions apply that would give it a performance advantage. Of course, the FOR FIRST construct involves creating a block, which has an overhead of its own that the FIND statement doesn’t have. Therefore, it’s good practice not to use FOR FIRST in all cases unless you have a specific reason.

The ETIME function

You might be wondering (in fact, you *should* be wondering) how to tell which of two constructs is faster in your situation or how to measure the performance of a part of your application. Progress has a performance profiling tool (called **Profiler**) that can show you exactly how much of your processing time is going to what routines, but there is also a much simpler way to do a test of a specific part of your code: the ETIME function (for elapsed time). ETIME returns an integer value representing the number of milliseconds since the function was reset to zero. Because a millisecond is a substantial amount of time on a modern processor, you often have to repeat an action many times inside a loop to measure accurately just what its cost is.

To reset the counter that ETIME uses, you invoke the function with an argument value of yes or true. Otherwise, you invoke it with no argument and no parentheses. If you forget to reset it to zero before you start, ETIME returns some enormous integer representing the number of milliseconds since your session started.

This example shows you whether it is faster to use the FIND statement or a FOR FIRST block to find the **Order** with a **Customer Number** of **24** and an **Order Date** of **1/31/98**.

If you look at the indexes for the **Order** table in the Data Dictionary, you see that there is an index on the **Order Date** field, and another index that has the **CustNum** field as its primary component, followed by the **OrderNum** field. The FOR FIRST construct takes advantage of both of these indexes to resolve the request in the most efficient way possible. The FIND statement can use only one of the indexes.

The code first resets ETIME, then does the same FIND in a loop 10000 times. It then saves the ETIME counter for this operation. Then it resets it again, does a FOR FIRST 10000 times, saves that value, and finally displays both values:

```
DEFINE VARIABLE iCount      AS INTEGER      NO-UNDO.
DEFINE VARIABLE iFindTime  AS INTEGER      NO-UNDO.
DEFINE VARIABLE iForTime   AS INTEGER      NO-UNDO.
ETIME(TRUE).
DO iCount = 1 TO 10000:
  FIND Order WHERE CustNum = 24 AND OrderDate = 1/31/98.
END.
iFindTime = ETIME.
ETIME(TRUE).
DO iCount = 1 TO 10000:
  FOR FIRST Order WHERE CustNum = 24 AND Orderdate = 1/31/98:
    END.
END.
iForTime = ETIME.
MESSAGE "The FIND took " iFindTime " milliseconds " SKIP
      "The FOR FIRST took " iForTime VIEW-AS ALERT-BOX.
```

Figure 21–2 shows the result.

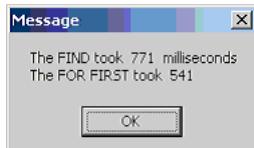


Figure 21–2: Result of ETIME function example

So, the FOR FIRST was significantly faster. The actual difference is very dependent on the actual indexes and the number of records to search.

Using the CAN-FIND function

The CAN-FIND function lets you determine whether a record with certain field values exists, for example as part of field validation. CAN-FIND takes a buffer name and a WHERE clause just as you would write for a FIND statement. It returns true if the record exists and false otherwise. Just as with the FIND statement, the CAN-FIND function can identify a unique record satisfying the WHERE clause, or you can include the FIRST (or even LAST) keyword before the buffer name to determine whether at least one record exists that satisfies the WHERE clause. The CAN-FIND function returns false if more than one record satisfies the selection criteria and you do not include the qualifier FIRST or LAST.

This simple procedure shows two uses of CAN-FIND:

```

FOR EACH Order:
    IF NOT CAN-FIND(Customer OF Order) THEN
        MESSAGE "Order " Order.OrderNum " has no Customer" Order.CustNum.
    END.

FOR EACH Customer WHERE NOT CAN-FIND (FIRST Order OF Customer):
    DISPLAY CustNum NAME.
END.

```

In the first case, the code looks for **Orders** whose **CustNum** field doesn't match any **Customer** record. This code would cause an error in the database's referential integrity.

The second block looks for **Customers** that have no **Orders**. This code would probably be a valid condition, since you need to add a **Customer** before you start adding **Orders** for it.

As it turns out, the Sports2000 database doesn't have any invalid **Orders** without **Customers**. If you add a few extra **Customers** to the database that aren't in the standard database and that don't have any **Orders**, the DISPLAY statement will show those, as you can see in [Figure 21–3](#).

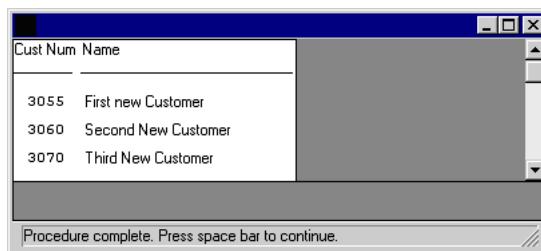


Figure 21–3: Result of CAN-FIND function example

Why does the Progress 4GL have this function in addition to the FIND statement? CAN-FIND can return true or false simply by looking at the index entries if the selection criteria can be resolved strictly through a single index, without having to retrieve record values. After all, the statement is only asking whether a record exists (that is, if it can be found). It is not retrieving any particular field values.

Thus, CAN-FIND is beneficial and efficient only when you use it to identify records through one or more fields in a single index. If CAN-FIND has to retrieve the database records themselves, then you have lost its performance advantage over the FIND statement.

Dynamic programming tips

This section provides some advice on programming with dynamic objects.

Dynamic versus static programming

The most basic thing to consider about programming with dynamic objects is whether and when to do it at all. Since support for dynamic objects was added to the Progress 4GL in more recent releases, some developers think that dynamic constructs are inherently an improvement over their static counterparts and are tempted to program *everything* using dynamic objects. This is a big mistake. Dynamic objects are intended to give you more choices when you develop an application, not to replace static references to tables and fields. There are several important considerations:

- Generally, dynamic constructs execute more slowly than their static equivalents. This is logical because, at compile time, Progress cannot anticipate what the procedure will do at run time and so cannot set up the structures to support the procedure. More of the work is done at run time, when the interpreter looks at the values of the dynamic procedure elements, just in time to prepare and execute them.
- Dynamic programming is more difficult than static programming. Code that is full of dynamic references is usually more difficult to understand at first glance. Since important values are stored in variables rather than hard-coded into the source procedures, it is less clear what is going on when you first look at the source code. Also, the use of handles and other devices that provide you with a level of indirection makes it harder to follow what a procedure is doing without studying it carefully. It is especially important here that you provide good internal documentation in the form of comments.
- You will usually want to write your business logic in static form, with specific references to the tables and fields (or their temp-table equivalents). Your business logic is likely to be unique to a single situation, rather than something repeated many times throughout your application. You can't write static business logic against dynamically defined objects.

So consider the power of dynamic programming for those situations where your application *is* doing the same thing in many places. Rather than writing many variations on the same source procedure or creating many compiled versions of the same procedure using include files or preprocessors, you can create a single procedure that does the same general job for every part of your application that needs it. This is why dynamic constructs exist. Consider their cost and the responsibility of having to clean up after yourself when you use them. Where they are the right thing to use, they can add tremendous flexibility to your application and dramatically reduce the number of different procedures you have to maintain.

Reusing dynamic objects

The major section of this chapter discusses Progress memory management and emphasizes that you must delete any dynamic objects you create. It's important to keep in mind, however, that if you are going to create another dynamic object of the same type, especially inside a loop that is executed many times, it is much more efficient to reuse the same dynamic object rather than deleting it and re-creating it. This is true even if you change all the attributes of the object. Then you must simply remember to delete it after you are completely done with it.

For example, the following procedure needs to generate a dynamic buffer and a dynamic query for every table in the Sports2000 database. The information about tables, fields, and indexes is actually stored as schema information in the database itself, and you can access it the same way you do any other information. There are just a few bits of information you need to know to understand this example:

- A record for each table is stored in a table called `_file`.
- The name of the file is in a field called `_file-name`.
- Each table has a number called the `_file-num`.
- There are special tables in the database that have file numbers either less than zero or greater than 32K, so the procedure skips those.

```
/* h-DeleteObject.p */
DEFINE VARIABLE hQuery AS HANDLE      NO-UNDO.
DEFINE VARIABLE hBuffer AS HANDLE      NO-UNDO.
DEFINE VARIABLE iCount AS INTEGER     NO-UNDO.

ETIME(TRUE).
DO iCount = 1 TO 100:
  FOR EACH _file WHERE _file-num > 0 AND _file-num 32000:
    CREATE BUFFER hBuffer FOR TABLE _file._file-name.
    CREATE QUERY hQuery.
    hQuery:SET-BUFFERS(hBuffer).
    hQuery:QUERY-PREPARE("FOR EACH " + _file._file-name).
    hQuery:QUERY-OPEN().
    hQuery:GET-FIRST().
    hQuery:QUERY-CLOSE().
    DELETE OBJECT hQuery.
    DELETE OBJECT hBuffer.
  END.
END.
MESSAGE "100 iterations took " ETIME "milliseconds" VIEW-AS ALERT-BOX.
```

For each database table, the code creates a dynamic query and a dynamic buffer, and sets the query's buffer to the dynamic buffer handle. Then the code prepares a default query for the table, opens it, gets the first record, and closes it. It then deletes the dynamic query and the dynamic buffer. To show the effect of the performance more dramatically, this is all done inside a loop 100 times. [Figure 21–4](#) shows how long it took for one sample run.

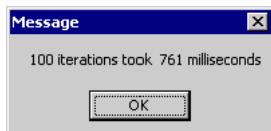


Figure 21–4: Result of DELETE OBJECT example

There's a more efficient way to do this. If you move the CREATE QUERY hQuery statement before the DO block and the FOR EACH block, and you move the DELETE OBJECT hQuery statement after the end of those blocks, the procedure runs somewhat faster. Even though you are resetting the query's buffer and re-preparing it for a completely different buffer, it is still faster to reuse the same dynamic object:

```
/* h-DeleteObject.p */
DEFINE VARIABLE hQuery AS HANDLE      NO-UNDO.
DEFINE VARIABLE hBuffer AS HANDLE     NO-UNDO.
DEFINE VARIABLE iCount AS INTEGER    NO-UNDO.

CREATE QUERY hQuery.
ETIME(TRUE).
DO iCount = 1 TO 100:
  FOR EACH _file WHERE _file-num > 0 AND _file-num < 32000:
    CREATE BUFFER hBuffer FOR TABLE _file._file-name.
    hQuery:SET-BUFFERS(hBuffer).
    hQuery:QUERY-PREPARE("FOR EACH " + _file._file-name).
    hQuery:QUERY-OPEN().
    hQuery:GET-FIRST().
    hQuery:QUERY-CLOSE().
    DELETE OBJECT hBuffer.
  END.
END.
MESSAGE "100 iterations took " ETIME "milliseconds" VIEW-AS ALERT-BOX.
DELETE OBJECT hQuery.
```

Figure 21–5 shows the result.

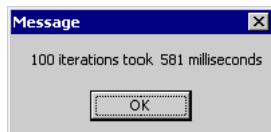


Figure 21–5: Result of more efficient DELETE OBJECT example

Just be sure you don't forget to delete the object when you're done with it! Unless you're reusing the object a large number of times, the difference in performance won't be dramatic, so it isn't worth the risk of forgetting to delete a dynamic object unless you will be reusing it many times in succession.

Note that you can't reuse the dynamic buffer in the same way. When you use the CREATE BUFFER statement, you must name the table the buffer will be for. This name can be an expression, as it is here, so that the buffer name can be assigned dynamically at run time, but you can't then reuse that same dynamic buffer object for a different buffer. So you have to create it and delete it inside the loop.

Using indexes properly

A detailed discussion of index design is beyond the scope of this book. But this section offers a few guidelines that relate directly to how proper use of indexes can contribute to the performance of your application.

Using word indexes for status indicators

A *word index* is a special index type that you can define for a character field. It indexes not the entire field value, as a normal index would, but every individual word in the index. There are delimiters you can define to tell the database manager just what you would like to see treated as a word, what the delimiters between words are, and so forth. You can learn about word indexes in *OpenEdge Development: Programming Interfaces*. Word indexes can be a tremendously powerful mechanism for identifying database fields that contain particular words. In fact, there is a special CONTAINS operator for a WHERE clause, similar to BEGINS and MATCHES, which is reserved for use with word indexes.

One powerful use of word indexes is not just to provide an index on all the words in a free text field, such as a status message or customer comments, but to create special character fields in which you store strings that identify other aspects of the record. For example, you can create a character field for a table in which you store various attributes of the record that otherwise would be individual logical fields with true/false values. It's much more efficient to use the CONTAINS operator on a word-indexed field than to evaluate a number of different indexed fields. You can also store some combination of field names and field values in a word-indexed field to make it easier and faster to find a record based on a number of different search criteria, such as customers where you have some particular bits and pieces of name and address information.

Avoiding indexes on logical values

If your application needs to identify records that satisfy some Boolean condition (such as Active vs. Inactive, Male vs. Female, or Domestic vs. Foreign), it is not a good idea to do this by means of indexes on Logical fields that represent the two conditions. The same is true of other fields that have only a handful of values, whether they are character values, such as Foreign and Domestic, or integer values representing those meanings. An index bracket is the portion of the index that the OpenEdge RDBMS must search through to identify all the records that match your selection criteria. If this is half or a large fraction of all the records, then the index is not serving its purpose and data retrieval is not efficient. Instead, you should consider encoding these kinds of values in a word-indexed character field. Under very special circumstances it might be beneficial to define an index on a logical value when, for example, 99 percent of the records are true for that value and you frequently need to identify the one percent that are false.

Using multi-component indexes

You can define an index on one or more fields in a table. Defining a multi-component index can be much more effective than defining multiple indexes on the same individual fields, but *only* when your application needs to access that combination of fields in the order in which they appear in the index. For example, if your application sometimes needs to select data based on the value of field A, and sometimes on A and B together, and sometimes on A, B, and C, then it makes sense to define a multi-component index with fields A, B, and C in that order.

However, if your application sometimes needs to select data based just on field B, or on C, or on B and C together, without knowing the value of A, then this index will do you no good any more than you can easily locate a word in the dictionary by knowing the second or third letter in the word.

Always evaluate the selection requirements of your application carefully as you design your database indexes.

Avoiding unneeded indexes

Another danger is simply defining too many indexes on a table. You should define an index for a table when you have most or all of these requirements:

- The index greatly reduces the amount of data to search to locate needed records. You should avoid indexes on small numbers of distinct values.
- The index is needed frequently. If only one occasionally used procedure needs some unusual selection, it is probably not worth defining an index just for that case.
- Fast performance is essential for the procedure that uses the index. If you have a batch report that runs once a month that needs some special selection criteria, it probably isn't worth defining an index just for that purpose.

Maintaining an index every time you create or update a record is relatively expensive. Maintaining many indexes on the same table can be *very* expensive. Avoid defining indexes you don't really need.

Hiding screen contents when making changes

Various types of operations make a series of changes to the user interface of an application before the user interface arrives at a final state. This might involve calculating a succession of values that are visible on the screen or adjusting the state of objects, such as combo boxes and selection lists. If you set the objects' (or where appropriate the frame's) HIDDEN attribute to true while you are making the adjustments, not only will you avoid flashing on the screen as things are changing, but the changes generally execute faster because the screen does not have to refresh after each adjustment.

Remember that the browse has a special REFRESHABLE attribute that you can set to false while you are making changes to the data displayed in the browse, to also prevent flashing for that object. This avoids having the browse hidden during the calculations, which might be distracting, but effectively freezes it so that the user sees only the final result of the changes.

Defining efficient queries and FOR EACH statements

This section provides some tips for defining queries and FOR EACH statements.

Using field lists

Progress allows you to specify a reduced list of fields to retrieve when you define a query or start a FOR EACH block. This is the syntax for the DEFINE QUERY field list:

```
DEFINE QUERY query-name FOR buffer-name
{ FIELDS field ... | EXCEPT field ... } [, buffer-name ... ].
```

This is the syntax for the FOR EACH field list:

```
FOR EACH buffer-name
{ FIELDS field... | EXCEPT field ... } [, buffer-name ... ].
```

If you specify a list of FIELDS for a buffer, only those fields are retrieved. If you specify an EXCEPT list, only those fields are *not* retrieved.

Under some circumstances, using a field list can reduce the amount of data transferred across the network in a client/server environment. However, there are serious limitations to the field list that mean that you should have limited use for it in most modern OpenEdge applications:

- The field list option was primarily designed for use with OpenEdge DataServers, which provide a connection to non-OpenEdge databases, such as Oracle and SQL Server. These kinds of databases typically have fixed-length data values that can be much larger than their OpenEdge counterparts, where all data is stored in an efficient, variable-length form.
- The field list has an effect only in a client/server environment, where your client application session has a direct connection to a database server on another system. This is not the recommended architecture for any new applications. A truly distributed application uses an AppServer to run an OpenEdge session that accesses the database and returns data to the client using temp-tables, as has been extensively discussed in this book. In this environment, you are completely in control of what fields you pass between client and server through your temp-table definitions. The field list mechanism plays no role in this.
- Even if you have a database connection in a client/server environment, Progress always retrieves the entire record if you lock the record with an EXCLUSIVE-LOCK.
- Progress retrieves additional fields beyond those in the field list for its own purposes, including evaluating some of the selection criteria of the query or FOR EACH.
- You must remember that the field list is not the same as a display list for a browse or a field list for the fields in a frame. You define the DISPLAY list of columns in a browse independently of the query that the browse uses. If you inadvertently leave out a field in the FIELDS list of a query definition that is needed by any part of the application that uses the query, your application will generate an error at run time. This can cause serious maintenance problems if your query definitions must explicitly name every field that is used from that query anywhere on the client.

The bottom line here is that in a distributed application, you control the field list through the definition of temp-tables that pass data from server to client, and the FIELDS phrase on a query definition is not needed as part of that definition.

Structuring your selection criteria in a join

When you need to retrieve data from multiple joined tables in a single query or FOR EACH statement, it is important to put the tables into the proper sequence and to specify your selection criteria as early in the retrieval process as possible.

OpenEdge does not optimize complex joins in the same way that some other database managers do, rearranging the order of tables and fields. There is a very good reason for this. Because Progress is designed to make it easy and effective to deal with individual records and multiple levels of selection, rearranging a join in a single statement is not typically an issue. For example, this kind of nesting of data retrieval blocks is very typical in Progress business logic:

```
FOR EACH Customer WHERE condition>:  
    /* Customer processing */  
    FOR EACH Order OF Customer:  
        /* Order processing */  
        FOR EACH OrderLine OF Order:  
            /* OrderLine processing */  
        END.  
        /* More Order processing after all OrderLines have been handled. */  
    END.  
    /* Final Customer processing. */  
END.
```

In this kind of code, the developer understands the order in which data is retrieved and is relying on that order to structure the business logic for related tables.

If you join tables in a single statement, Progress retrieves the data in the order you specify. Progress does not second-guess your selection and rearrange the retrieval for you. This means that you have to take responsibility for structuring your selection efficiently. For example, if you want to retrieve orders processed today for **Customers** with a **CreditLimit**, this kind of statement is very inefficient in Progress:

```
/* Less efficient selection: */  
FOR EACH Customer WHERE Customer.CreditLimit NE 0,  
    EACH Order OF Customer WHERE OrderDate = TODAY:
```

Hardly any **Customers** have a **CreditLimit** of 0, so the **Customer** selection is going to return nearly all **Customers**. On the other hand, only a few **Customers** have placed **Orders** today. It would be much more efficient to identify the **Orders** first, and then get the **Customer** for each of those **Orders**:

```
/* More efficient selection: */  
FOR EACH Order WHERE OrderDate = TODAY,  
    FIRST Customer OF Order WHERE CreditLimit EQ 0:
```

It's especially important to place the selection criteria for each table as high up in the statement (that is, as close to the front) as possible. Always define the selection for each table as part of the phrase for that table's buffer. That is, don't write a statement such as this:

```
/* Inefficient selection: */  
FOR EACH Customer,  
    EACH Order OF Customer WHERE Customer.CreditLimit NE 0 AND OrderDate =  
        TODAY:
```

In this case, Progress does just what you ask it to do:

1. Retrieves each **Customer** in the **Customer** table in turn, into the **Customer** buffer, regardless of its **CreditLimit** or anything else.
2. Retrieves each **Order** for each **Customer** in turn, into the **Order** buffer.
3. Examines the **CreditLimit** value in the **Customer** buffer to see if it equals 0.
4. If the **CreditLimit** does not equal zero, examines the **OrderDate** in the **Order** buffer to see if it's equal to today's date.

This is clearly a very inefficient way to go through the data, especially because there is an index on the **OrderDate** field and another index on the **CustNum** field in both the **Order** table and the **Customer** table that allows Progress to identify those **Orders** and their **Customers** immediately.

If you are used to working with other data retrieval languages, you might miss the optimization of complex queries that they do, but Progress gives you control over your application behavior by presenting you with the data you ask for in the way that you ask for it. When you are writing real business logic this is much more useful than having the DBMS evaluate some complex set of expressions on a **WHERE** clause that joins multiple tables and return a single processed result.

Copying temp-tables as parameters

Chapter 11, “Defining and Using Temp-tables,” describes how you can pass temp-tables from one procedure to another using the static TABLE parameter form, the dynamic TABLE-HANDLE parameter form, or simply by passing the HANDLE of the temp-table. It is *extremely important* that you pass temp-tables using the simple HANDLE parameter whenever possible, if your procedure call is within the same session. When you pass the handle to a temp-table, as with any object, you are simply passing a pointer to its location elsewhere in the session. If you pass the temp-table using either the TABLE or TABLE-HANDLE form, you are forcing Progress to perform a complete copy of the structure of the temp-table and the data in the table. You need to do this only if the procedure receiving the temp-table needs a static definition of the table. Especially when the procedure receiving the temp-table is only an intermediary, which simply passes the temp-table on to some other procedure, there is never a need to pass the table itself, only its handle. Observing this guideline whenever you use temp-tables can *greatly* improve your application’s performance.

Setting your Propath correctly

The Propath is the set of directories Progress looks in to locate procedure files to run. Configuring your Propath correctly is essential to good application performance. The more directories there are on the Propath, the longer it takes to search for files. Larger directories may take longer to search than smaller ones, depending on your operating system.

You should order directories by their frequency of use. Put the most frequently used directories first.

Database and AppServer-related issues

This section describes issues related to the OpenEdge database and the AppServer.

Avoiding unnecessary database connections

Connecting to a database is a relatively slow operation that should be done only once. Do not connect and disconnect over and over. Specify the databases to be connected on the command line rather than using the CONNECT statement in the 4GL. If necessary, you can specify database connections when you enter significant modules that use that database, but be aware of the cost.

Minimizing creation of subprocesses

Creating a new process is an expensive operation that can be very time consuming if done often and especially when done by multiple users on the same system. Use statements like INPUT THROUGH, OUTPUT THROUGH, and UNIX with care.

Using database sequences

[Chapter 16, “Updating Your Database and Writing Triggers,”](#) you saw how you can use database sequences to generate a sequence of unique integer values, for example, as key values for a database field such as a **Customer Number** or **Order Number**. Sequences are much faster than using an integer field in a control record stored in the database, and they do not cause lock conflicts as a control record does. To obtain unique values from a field in a database record, each user needs to get an EXCLUSIVE-LOCK on the record within a transaction, retrieve its value, increment the value, and then end the transaction and release the record. This is very time consuming, especially when you consider that it is likely that many users will be trying to do this at the same time. This is exactly the purpose that database sequences were designed for. When you execute the NEXT-VALUE function on a sequence, the OpenEdge RDBMS increments the sequence and returns the value to you in a single operation, so that there is no chance that another user can see the same value, but without any need for a transaction.

Minimizing the size and number of network messages

The two slowest operations in most systems are transmitting data over a network and accessing data on a disk. When using AppServers, sockets, or client/server database access, take care to minimize the number of network calls you make. Whenever possible, send large amounts of data at once rather than many small transmissions.

Always remember to test on slow networks. A dial-up connection is many times slower than a fast LAN connection. Network messages take considerably longer to transmit on slow connections. Sometimes an application works well during development because a fast network is used and fails miserably when deployed in the real world.

Network latency is limited and is generally beyond your control. The speed of light limits network transmission time. For example, it is 3266 miles from Boston to London. Via the Internet, it might be 5000 miles or more. A sample routing chart shows 18 hops from a PC in a Boston office to a server in a London office. It takes at least 40 milliseconds to send a message, assuming best possible conditions. Typical conditions might require 80 or 100 milliseconds. So, sending a message and getting a response takes around 160 milliseconds if the server responds quickly. At that rate, you can send and receive 6 (typical) to 12 (best case) messages per second. Nothing will ever make this turnaround time faster. Technical improvements in the Internet will only allow you to send more data in the same time.

Configuring your session using startup options

Progress supports a large number of startup options that you can use, among other things, to tune the configuration of your session, controlling memory allocation for various purposes and other important factors that can have a large effect on performance. A detailed discussion of these is beyond the scope of this book, but this section mentions a few key ones.

Using the -q startup option

The Quick Request (-q) option avoids constant file lookups to determine if a previously loaded .r file needs to be replaced in memory by a newer one. If you specify -q, then the Propath search is done only once, on the initial load of the .r and not on every invocation of it. You should always use the -q option except when you are in development mode and regularly changing and compiling application procedures.

Increasing the -Bt startup option size

The -Bt startup option sets the buffer size for temporary tables. The default value is *very* small, only 10 buffers. Increasing the buffer size to 100 or more can be useful.

Increasing the -Mm startup option size

In client/server environments, increasing the maximum size of the buffers used for network messages allows more data per message to be sent. This is especially beneficial when reading groups of records with NO-LOCK. The size is a maximum and smaller messages are sent when there is not enough to completely fill the buffer. The default size is 1024 bytes. Increase it to 16384. Note that the same size must be specified on both the client and server side, and that the server allocates message buffers for each client. Large buffers consume more server memory. The increase on the client side is not significant.

Memory management in Progress

The chapters on dynamic objects taught you that even in a high-level language like the Progress 4GL, you have to do your own memory management when you use dynamic objects. The basic rule is very simple: *You create it, you delete it!*

This section re-enforces this basic concept and gives you some examples of things you need to be aware of and techniques to use to make sure that your application doesn't sprout memory leaks that bring it to a halt when you put it into production.

You shouldn't treat the need for memory management as a reason to avoid programming using dynamic objects. Remember that the Progress 4GL started out as a strictly static language, where everything was defined in the procedural source code and the compiler resolved every table and field reference. Dynamic objects have been added to the language precisely because they can make your development much more effective, and allow you to reuse procedures and logic, when you need a single operation on different tables, fields, or other objects at different times. You should take full advantage of dynamic objects. You just need to remember to clean up after yourself.

The need for careful programming is especially important because many memory management problems only show up when you run your application over long periods of time in production, and the effects of even small memory leaks cause drastic problems. That is definitely not the time for you to be discovering the problems in your application! A little discipline up front protects you from this.

Cleaning up dynamically allocated memory

When you run a procedure with only static objects, these objects come and go with the procedure that defines them. Because Progress can see the individual `DEFINE` statements for the objects, it knows when to create them, when they go out of scope, and when to delete them. You don't have to worry about deleting them because Progress does it automatically for you.

The same is true, incidentally, of handles for static objects. When you store the handle of a static object in a `HANDLE` variable, that value is just a pointer to the object. It doesn't extend or affect the scope of the static object in any way. The variable itself, because it's defined statically, is also deleted by Progress when its procedure goes out of scope.

When you create dynamic objects, however, Progress cannot control their scope or their lifetime. If you create a dynamic object inside a loop, Progress might have no way of knowing at compile time how many of those objects the procedure will create. If you pass the handle of an object as a parameter to another procedure, Progress has no way of knowing at compile time where the handle came from, what it will be used for, or when the object it represents can safely be deleted. This is why you are responsible for taking care of this.



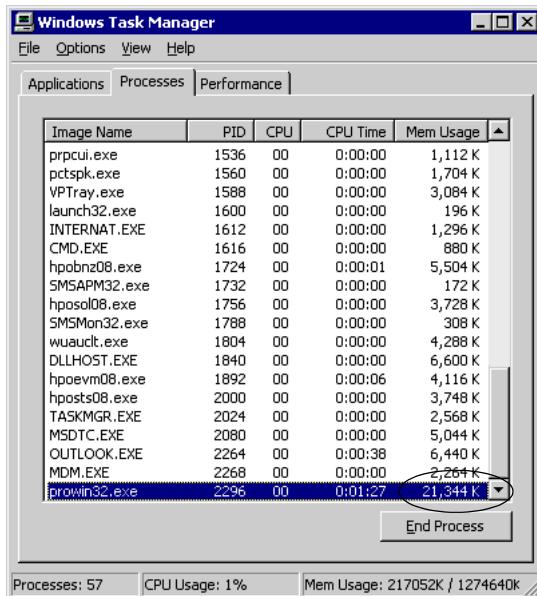
To see a dramatic example of the importance of memory management:

1. Open the h-DeleteObject.p procedure used earlier and comment out the statements that delete the dynamic query and buffer objects at the end of the loop:

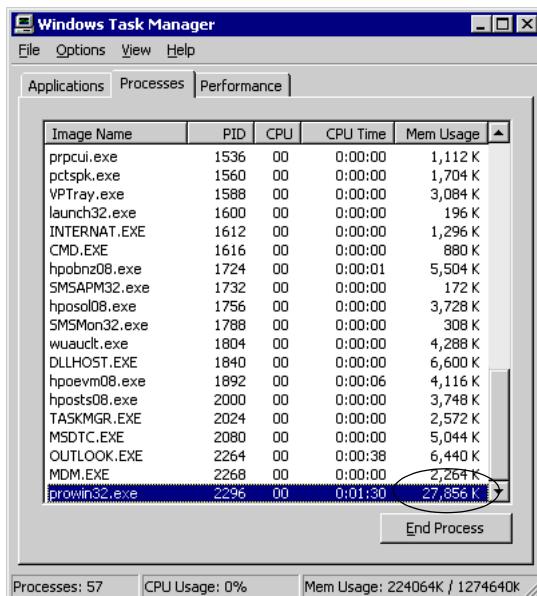
```
/* h-DeleteObject.p */
DEFINE VARIABLE hQuery AS HANDLE      NO-UNDO.
DEFINE VARIABLE hBuffer AS HANDLE     NO-UNDO.
DEFINE VARIABLE iCount AS INTEGER    NO-UNDO.

ETIME(TRUE).
DO iCount = 1 TO 100:
  FOR EACH _file WHERE _file-num > 0 AND _file-num  32000:
    CREATE QUERY hQuery.
    CREATE BUFFER hBuffer FOR TABLE _file._file-name.
    hQuery:SET-BUFFERS(hBuffer).
    hQuery:QUERY-PREPARE("FOR EACH " + _file._file-name).
    hQuery:QUERY-OPEN().
    hQuery:GET-FIRST().
    hQuery:QUERY-CLOSE().
/*
  DELETE OBJECT hQuery. */
/*
  DELETE OBJECT hBuffer. */
END.
MESSAGE "100 iterations took " ETIME "milliseconds" VIEW-AS ALERT-BOX.
```

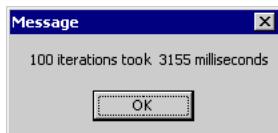
2. Bring up the **Windows Task Manager**, select the **Processes** tab, and find the running executable called **prowin32.exe**:



3. Run the procedure, and watch the memory go:



After just 100 iterations of this loop, which only created two dynamic objects, you've lost *six megabytes* of memory! Not only that, but even though you *removed* two key statements from the procedure, it ran about four times slower:



The reason for this is that all that extra memory allocation is getting in the way of your procedure running efficiently. So you pay a heavy price all the way around.

Use the `DELETE OBJECT` statement to get rid of the memory for any dynamic object, regardless of its type. To summarize, you use a `CREATE QUERY` statement to create a dynamic query, a `CREATE BUTTON` statement to create a dynamic button, and a `CREATE BROWSE` statement to create a dynamic browse, but you use the `DELETE OBJECT` statement to clean up each of them.

You must also remember to clean up persistent procedures when you're done with them. Every time you execute a statement of the `RUN procedure-name PERSISTENT SET proc-handl[e]` form, you are also allocating memory for the procedure and all its contents. You need to delete the procedure, when you're done with it, with the `DELETE PROCEDURE` statement.

Using widget pools

Deleting individual dynamic objects is a big responsibility, and a serious nuisance as well. Widget pools are designed to help you make your use of dynamic objects much simpler and more reliable.

A widget pool provides a means of treating objects you create as a set. When you delete the pool, all the objects you created in it go away together. The simplest way to group objects using widget pools is to associate all the objects in a single procedure with a widget pool. In fact, the template procedures used by the AppBuilder for windows (and for visual SmartObjects, such as SmartWindows and SmartDataViewers) help you do this by having a `CREATE WIDGET-POOL` statement in the template's definitions section:

```
/* Create an unnamed pool to store all the widgets created
   by this procedure. This is a good default which assures
   that this procedure's triggers and internal procedures
   will execute in this procedure's storage, and that proper
   cleanup will occur on deletion of the procedure. */

CREATE WIDGET-POOL.
```

This means that, by default, all the dynamic objects you create that go into the window are deleted when the window is closed and its procedure deleted. This includes not just visual objects but dynamic buffers, queries, and so forth.

Remember that a simple `CREATE WIDGET-POOL` statement creates an *unnamed* widget pool. This pool automatically goes away when its procedure terminates. This might not always be the behavior you want. In fact, by passing the handles to dynamic objects around, you can easily wind up with a handle whose scope exceeds the lifetime of the dynamic object it points to. In this case, the value of the handle can become invalid.

Here's a simple example of how this can happen. The `h-MakeBuffer.p` procedure has an internal procedure called `getBuffer` that creates a dynamic buffer and returns it to the caller. The `CREATE WIDGET-POOL` statement puts all such buffers into an unnamed widget pool for that procedure:

```
/* h-MakeBuffer.p */
CREATE WIDGET-POOL. /* Unnamed widget pool! */

PROCEDURE getBuffer:
  DEFINE INPUT PARAMETER cTable AS CHARACTER NO-UNDO.
  DEFINE OUTPUT PARAMETER hBuffer AS HANDLE NO-UNDO.
  /* The buffer will be allocated to the unnamed widget pool. */
  CREATE BUFFER hBuffer FOR TABLE cTable.
END PROCEDURE.
```

Another procedure, `h-RunMakeBuff.p`, runs `h-MakeBuffer.p` PERSISTENT and then runs `getBuffer`:

```
/* h-RunMakeBuff.p */
DEFINE VARIABLE hProc AS HANDLE NO-UNDO.
DEFINE VARIABLE hMyBuf AS HANDLE NO-UNDO.

RUN h-MakeBuffer.p PERSISTENT SET hProc.
RUN getBuffer IN hProc (INPUT "Customer", OUTPUT hMyBuf).

MESSAGE "While MakeBuffer is alive, my buffer is "
       hMyBuf:NAME VIEW-AS ALERT-BOX.
DELETE PROCEDURE hProc.
MESSAGE " After I delete it, I get nasty errors"
       hMyBuf:NAME VIEW-AS ALERT-BOX.
```

The first message statement, as expected, correctly displays the name of the dynamic buffer that was returned, as shown in [Figure 21–6](#).

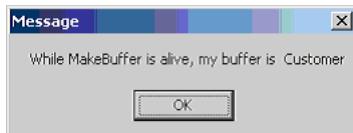


Figure 21–6: Result of widget pool example

What happens, though, when the procedure has deleted its instance of `h-MakeBuffer.p`?

The unnamed widget pool in that instance of `h-MakeBuffer.p` goes away, and the dynamic buffer that `hMyBuf` points to goes away with it. But the `hMyBuf` variable is still very much alive, and in fact the *value* of the handle that it holds hasn't changed. However, that handle value doesn't point to anything anymore, so you get a string of errors, as shown in Figure 21–7.

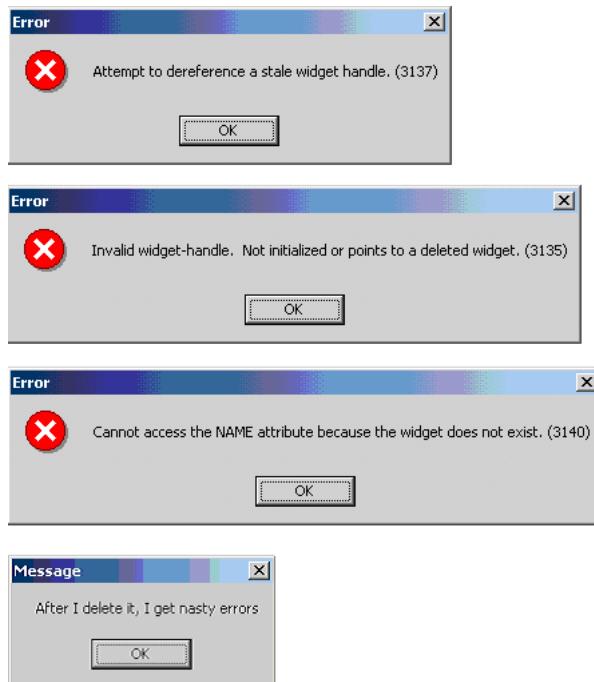


Figure 21–7: Errors for unnamed widget pool example

This sequence of errors is the bane of any dynamic programmer's existence. It is the most likely consequence of failing to make sure that all your dynamic objects live exactly as long as they need to, but no longer.

Using named widget pools

The solution to this is to give names to widget pools that hold objects that might outlive the procedure that created them. In addition to giving the pool a name, you must also define it to be **PERSISTENT**. You can then create dynamic objects explicitly in that pool:

```
/* h-MakeBuffer.p */

CREATE WIDGET-POOL "MakeBuffPool" PERSISTENT.

PROCEDURE getBuffer:
  DEFINE INPUT  PARAMETER cTable AS CHARACTER  NO-UNDO.
  DEFINE OUTPUT PARAMETER hBuffer AS HANDLE    NO-UNDO.
  /* The buffer will be allocated to the unnamed widget pool. */
  CREATE BUFFER hBuffer FOR TABLE cTable IN WIDGET-POOL "MakeBuffPool".
END PROCEDURE.
```

The persistent widget pool becomes another dynamic object that your application has to take responsibility for managing. You must delete it when you're finally done with, or all the objects in it will sit there in memory until your session ends. Here the calling procedure cleans up after it is done using the buffer handle:

```
/* ... end of h-RunMakeBuff.p ... */
DELETE PROCEDURE hProc.
MESSAGE " No more errors, the buffer is still"
      hMyBuf:NAME VIEW-AS ALERT-BOX.
DELETE WIDGET-POOL "MakeBuffPool".
```

When you run the procedure, the errors go away as shown in [Figure 21–8](#), because the handle is valid until you delete the pool.

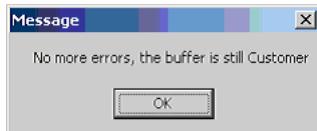


Figure 21–8: Result of named widget pool example

Remember that all these steps are important:

1. You must give the widget pool a name if you want to allocate objects to it specifically or if you want it to outlive the procedure that creates it.
2. The widget pool name is a character expression. So, if you are using a literal string rather than a variable for the name, you must remember to put it in quotes both where you create it and wherever you reference it. This is different from other CREATE statements, where you normally specify a handle variable as a target for the create and where the dynamic object you create doesn't really have a name the same way that static objects do.
3. You must *also* make a named pool PERSISTENT if you want it to outlive the procedure that creates it.
4. You must remember to delete the widget pool when you are done using it, just as you delete individual dynamic objects that aren't in a specific widget pool when you are done with them.

You might create a named widget pool that was *not* persistent simply to put different dynamic objects in different pools within a single procedure, so that you could delete one widget pool within the procedure without deleting objects in some other widget pool. In this case, all of the pools that haven't been specifically deleted during the execution of the procedure are deleted when the procedure is deleted.

A widget pool cannot be created as PERSISTENT without giving it a name.

When you create a persistent named pool, its name effectively becomes global to the session. Any procedure running anywhere in the session can delete it. You cannot, however, access named widget pools through the SESSION handle as you can with some other kinds of objects, such as windows and procedures.

Making the best use of dynamic objects

There are a few simple rules for making the best use of dynamic objects without incurring memory leaks:

- Keep your interactions short. The longer the span between when you create an object and when your application is done using it, the greater the likelihood that you will forget to delete it and that you will never detect this until your application dies in production because of a memory leak. This span really has less to do with time than with the organization of your procedures. If the architecture of your application is clear about how dynamic objects are created and when they are deleted, then you will do well. If your code is inconsistent about this, then you will have a very difficult time identifying whether you've cleaned up after yourself or not.
- Always use widget pools. Remember that the `CREATE WIDGET-POOL` statement at the top of a procedure is not a Progress default, or even a default for all procedures you build in the AppBuilder. It is simply a convention observed by a few template files for some kinds of procedures the AppBuilder creates. Create your own templates and your own convention and stick to it.
- Make a practice of always deleting every object as soon as you are done with it, even if you are using widget pools. A widget pool in a procedure can accumulate an enormous number of unused objects if you wait until the procedure goes away or until the widget pool is explicitly deleted. The widget pool mostly serves as a backup mechanism to get rid of objects you somehow forgot to delete explicitly or to help you organize the deletion of related groups of objects. If you have a procedure that creates a large number of objects and then deletes them all at once, then go ahead and use the `DELETE WIDGET-POOL` statement for that purpose. In that case, be as conscientious about deleting every widget pool as soon as you are done with it as you should be with individual objects.
- If you create *named persistent* widget pools, be very clear in your application architecture about making sure that some procedure deletes them, unless they are used only for objects that always must live for the duration of the session. Also, make sure that you never delete a pool while its objects are still being used.
- If your procedure is *immediately* going to create another object of the same type as one you are done with, then go ahead and reuse the same object without deleting it and creating a new one. This is generally faster, even if you are changing all the characteristics of the object. But *don't* leave piles of objects lying around in your code on the off chance that you might want to reuse them.
- Test your application rigorously with an eye to memory usage before you ship it to your customers.

Avoiding stale handles

You saw the errors that you get if you inadvertently try to reference (or *dereference*, as the error message says) a stale handle. A stale handle is one that holds a pointer to an object that no longer exists. If you make sure that your objects always live just as long as they should, then you shouldn't get this type of error. But it's still very good practice to check the validity of a handle when there's any chance that it might be invalid, which almost certainly includes whenever it comes from another procedure that might be maintained independently of the one that needs to use the handle.

You can use the VALID-HANDLE function to check whether a handle is valid. If you do use this check, then always determine how your procedure should react if a handle that ought to be valid turns out not to be. Simply substituting your own "Invalid handle!" message for the default one that Progress gives you is not the right approach. There are many things you can try to blame the end user of your application for, but supplying a procedure with an invalid object handle is not one of them.

Remember, too, that once you have encountered an invalid handle, whether it is a stale handle for a deleted object or simply a handle variable that was never properly initialized, your end user will get another ugly message for every statement in your procedure that vainly tries to reference it. In the Runmake.p example, the procedure generated three internal system errors because the buffer handle was not valid. It could easily have been a *hundred* errors, if the procedure had gone on to continue to try to work with that invalid handle. Never risk subjecting your end users to this abuse.

The VALID-HANDLE function is also very useful and necessary anytime your code is walking through a chain of handles, such as in this procedure that displays all the windows currently running in the session:

```
DEFINE VARIABLE hChild AS HANDLE      NO-UNDO.
hChild = SESSION:FIRST-CHILD.
REPEAT WHILE VALID-HANDLE(hChild):
  DISPLAY hChild:NAME FORMAT "X(30)"
    hChild:TYPE WITH FRAME F 10 DOWN.
  hChild = hChild:NEXT-SIBLING.
END.
```

You could also use the VALID-HANDLE function as a signal that tells your procedure whether an attempt to locate a particular object was successful or not, somewhat like the AVAILABLE function after a FIND statement on a buffer.

In early versions of Progress, it was possible (and quite likely) that if you deleted an object and then created another object, even an object of a different type, the new object would have the same handle value as the object you just deleted. Therefore, the VALID-HANDLE function could return true because the handle was pointing to a valid object, even though it was not the object you expected it to be. The TYPE and UNIQUE-ID object attributes exist largely to help you identify whether an object is the same one you referred to earlier. As of Progress Version 9.1C, this is no longer the case. Handles are allocated in such a way that you cannot have a handle variable that inadvertently points to a different object residing in the same memory location as one that was deleted earlier in the session. Therefore, you can be confident that when you use the VALID-HANDLE function, it is not only telling whether the object the handle points to is *some* valid object, but assuring you that it is the *same* object it always was before. Progress does this using a mechanism called *opaque handles*. You do not need to do anything special to get the benefit of them.

Deleting persistent procedures

You must clean up persistent procedure instances you have created just as with other objects. Remember that the RUN prog.p PERSISTENT statement is much the same as CREATE *object*. Progress creates a running instance of the procedure and all its contents, and leaves it in memory until you specifically delete it. The simplest way to delete a procedure when you're done with it is to use this statement:

```
DELETE PROCEDURE procedure-handle.
```

Note that you use the PROCEDURE keyword in this statement, not OBJECT.

The AppBuilder templates observe a convention that is a very useful alternative to simply deleting a procedure. If you delete a procedure from the outside, it might not be finished doing what it needs to do and might not have a chance to clean up the objects inside it properly. It is much more reliable, and much more object-oriented, to tell the procedure to delete itself rather than to simply kill it. You've seen the way the AppBuilder-generated procedure handled this earlier, but it's worth repeating in the context of this discussion about memory management.

The procedure's main block runs a standard internal procedure called `enable_UI` and then waits for a CLOSE event if the procedure is not being run persistent. If it *is* being run persistent, then it stays in memory without the need for any WAIT-FOR statement:

```
/* Now enable the interface and wait for the exit condition.      */
/* (NOTE: handle ERROR and END-KEY so cleanup code will always fire.   */
MAIN-BLOCK:
DO ON ERROR    UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
    ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:
        RUN enable_UI.
        IF NOT THIS-PROCEDURE:PERSISTENT THEN
            WAIT-FOR CLOSE OF THIS-PROCEDURE.
END.
```

The WINDOW-CLOSE event on the window applies the procedural CLOSE event to the procedure:

```
DO:
    /* This event will close the window and terminate the procedure. */
    APPLY "CLOSE":U TO THIS-PROCEDURE.
    RETURN NO-APPLY.
END.
```

The main block also defines this CLOSE event, which runs the cleanup code in the `disable_UI` internal procedure, which again is part of the AppBuilder's own standard mechanism for starting and stopping procedures:

```
/* The CLOSE event can be used from inside or outside the procedure to */
/* terminate it.                                                 */
ON CLOSE OF THIS-PROCEDURE
    RUN disable_UI.
```

And finally, `disable_UI` deletes any dynamic objects. Then, if the procedure was run persistent, it explicitly deletes itself. If it wasn't run persistent, then Progress deletes it for you:

```
/* Delete the WINDOW we created */
IF SESSION:DISPLAY-TYPE = "GUI":U AND VALID-HANDLE(C-Win)
THEN DELETE WIDGET C-Win.
    IF THIS-PROCEDURE:PERSISTENT THEN DELETE PROCEDURE THIS-PROCEDURE.
END PROCEDURE.
```

Whether you use the AppBuilder templates or not, use some similar convention that makes sure your procedures clean up after themselves and deletes them when your application is done with them.

Deleting temp-table copies

When you pass a static temp-table from one procedure to another using the static TABLE parameter form, either locally or remotely, Progress takes care of deleting the temp-table and its contents when the procedure that defines it is deleted, just as it does for other static objects.

However, if you pass a temp-table through a TABLE-HANDLE, then you are creating a dynamic copy of the temp-table. Whether there is a static temp-table definition for it on one side of the call or the other, the TABLE-HANDLE parameter creates a dynamic temp-table. If a procedure passes a temp-table to another procedure using the INPUT TABLE-HANDLE form, then the *calling* procedure must take responsibility for deleting the temp-table when it's done with it. If a procedure receives a temp-table from another procedure using the OUTPUT TABLE-HANDLE form, then the *receiving* procedure must delete its copy of the temp-table after it is done with it.

As with other dynamic objects, Progress does not know when you are done with a dynamic copy of a temp-table and therefore cannot delete it for you. You cannot allocate a temp-table to a widget pool, so this mechanism does not help you out. The temp-table is created in the SESSION widget pool and therefore stays there until it's deleted or until the session ends. Because a temp-table is normally a much larger object than a dynamic button, query, or buffer, it is critically important that you delete temp-tables when you're done with them. If you're done with the data in the temp-table but need to use the table again, then use the EMPTY-TEMP-TABLE method on the table handle to remove the data. Delete the table itself using the DELETE OBJECT statement when you are done with it altogether.

Because dynamic temp-tables are allocated in the SESSION pool, you can locate them through the SESSION handle's buffer chain. If the buffer has a valid TABLE-HANDLE attribute, then it's a temp-table buffer and its TABLE-HANDLE points to the temp-table itself:

```
DEFINE VARIABLE hBuffer AS HANDLE      NO-UNDO.  
  
hBuffer = SESSION:FIRST-BUFFER.  
REPEAT WHILE VALID-HANDLE (hBuffer):  
  IF VALID-HANDLE (hBuffer:TABLE-HANDLE) THEN  
    DISPLAY hBuffer:NAME WITH FRAME F 10 DOWN.  
  hBuffer = hBuffer:NEXT-SIBLING.  
END.
```

Other object types

Memory management is also an issue for other object types that aren't discussed in this book, but which this section mentions briefly:

- **MEMPTR** — A MEMPTR is a pointer to an area of your computer's memory that you allocate and use independently of any Progress objects. You are fully responsible for controlling this memory. You allocate memory using the SET-SIZE function. You deallocate it by using SET-SIZE to set the size to zero. There are other functions that manipulate memory allocated with a MEMPTR, but Progress does not recognize the form of the contents in any way.
- **ActiveX control** — You can extend the visual content of your user interface and other aspects of an application that runs on MS Windows using ActiveX controls. You access an ActiveX control through a special handle called a COM-HANDLE. You are responsible for using a special RELEASE OBJECT statement to release memory associated with the object. Because ActiveX controls operate outside the bounds of objects known to Progress, and because they can do so much on their own, they can be a major source of memory leaks. You need to use and test them very carefully.
- **Socket** — A socket is a connection to a process running outside the OpenEdge session, possibly on another machine. OpenEdge provides access to other procedures through its support for sockets. A SOCKET is another kind of dynamic object you can create and use from the 4GL. You must clean up sockets just like any other objects. There is a special SESSION:FIRST-SOCKET attribute that lets you access the head of the chain of all the sockets allocated in your session.
- **Asynchronous request** — You can run a procedure in another OpenEdge session with a special ASYNCHRONOUS keyword to allow your session to continue to run until the other session responds. A procedure run in this way is kept in memory even after the request is complete so that you check the completion status. You must take special care to delete these procedures when you are done with them. There is also a SESSION:FIRST-ASYNC-REQUEST attribute that identifies the chain of all outstanding asynchronous requests in your session.

There is a certain amount of information on these special types in *OpenEdge Development: Progress 4GL Reference* and online help, but you will find detailed information in *OpenEdge Development: Programming Interfaces*.

Conclusion

Memory management in Progress is not that difficult if you establish a standard set of templates for your procedures and an architecture that makes it clear what procedures are responsible for creating dynamic objects and deleting them, and when your session is done using them. If you are organized in your design and test your application carefully, you should have no trouble creating a reliable application that makes appropriate use of the power and flexibility of dynamic objects. Just remember the basic rule:

You create it, you delete it!

Conclusion to the book

This book has introduced you to the essentials of the Progress 4GL. Its purpose is to give you the skills you need to start to build successful business applications using the OpenEdge platform. Even in a book of this scope, it is not possible to discuss all the elements of the Progress 4GL. Language statements that support interfaces to other programming environments, such as socket support, support for generating and consuming XML documents, and others, are left to other documentation. Also, language statements used in constructing Progress-based reports and statements directly in support of the database and AppServer are discussed elsewhere. In addition, the many new features of OpenEdge Release 10 are covered in other documentation.

In any case, the language statements alone are never sufficient to build a complex enterprise application. It's very important that you adopt an overall architecture for your application to give you the consistency and flexibility you need to complete your application and to maintain and extend it in a fast-changing world. OpenEdge provides components such as Progress SmartObjects, as well as the Progress Dynamics development framework to give you a basis for creating powerful and sophisticated applications quickly. Other OpenEdge 10 documentation also provides guidelines and principles for an overall application architecture. Whether you use OpenEdge-provided components or develop your own framework, you should plan the structure of your application carefully before you begin writing large numbers of Progress 4GL procedures. A thorough up-front design will save you tremendous effort in the long run and give you a basis for an application that will ensure your success long into the future.

Index

Symbols

- ! Format character 2–11
- & Ampersand 4–20
- & Menu mnemonic 9–30
- * Format character 2–11
- * Multiplication sign 2–24
- + Plus sign 2–24
- Hyphen
 - versus minus sign 2–24
- Minus sign 2–24
- / Division sign 2–24
- /* Character (comment) 3–14
- ? (Question mark) 2–16
- ? (Unknown value) 2–9
- @ At sign 12–16
- @ Symbol 6–36
- ~ Tilde 1–14

Numbers

- 9 format character 2–11
- A
- A format character 2–11
- Abbreviations
 - data types 2–28
- Absolute keyword 2–26
- Accelerator keyword
 - menus 9–28
- Accessing
 - data 6–1, 6–28
 - online help 2–32
 - PRO*Tools 3–3
 - triggers 16–28
- Accessing data
 - queries 10–2
- Active-window system handle 18–44
- ActiveX controls 21–37
- ADD-BUFFER method
 - dynamic queries 19–5

- ADD-CALC-COLUMN method
 - dynamic browses 18–52, 20–25
- ADD-COLUMNS-FROM method
 - dynamic browses 18–51, 20–24
- ADD-FIELDS-FROM method
 - dynamic temp-tables 20–4
- ADD-INDEX-FIELD method
 - dynamic temp-tables 20–8
- Adding
 - browse to a window 4–13
 - buffer to query 19–5
 - buttons 4–24
 - combo boxes 12–40
 - comments 3–14, 5–15
 - dynamic objects 18–26
 - fields 5–20
 - selection lists 18–26
 - toggle boxes 9–9
 - warning messages 19–23
- Add-like-column method
 - dynamic browses 18–52, 20–25
- Add-like-field method
 - dynamic temp-tables 20–5
- Add-like-index method
 - dynamic temp-tables 20–7
- Add-new-field method
 - dynamic temp-tables 20–6
- Add-super-procedure method 14–15
- Advanced Properties dialog box 8–15
- Alert boxes
 - creating 10–28
- Alias Editor dialog box 2–30
- Aliases
 - CMT 3–14
 - editing 2–29
 - p4gl.als file 2–30
- Aligning objects 4–31
- Alternate layout option AppBuilder 4–32
- Ambiguous attribute
 - dynamic buffer handles 19–28
- Ambiguous field error 6–18
- Ampersand (&) 4–20
 - menu mnemonic 9–30
- Analyze preprocessor 5–2
- ANY-KEY event 8–24
- ANY-PRINTABLE event 8–24
- ANYWHERE option
 - ON statement 8–33
- AppBuilder
 - advantages 5–20
 - defining functions 14–9
 - description 1–3
 - Section Editor 4–18
 - starting 4–2
 - template files (.cst) 18–13
 - using 4–1
- AppBuilder palette 4–3
- Appearance attributes 8–10
- APPEND option
 - temp-tables 11–13
- Application Development Tools (ADE) 3–5
- Application flow
 - call stack 13–4
- Application programming interface (API)
 - 13–34
- APPLY statement 5–12, 7–25
- Applying events 8–33

AppServers
calling 20–40

Arithmetic expressions 2–24

Array variables 21–7

Arrays
vs lists 21–7

ASCENDING keyword 6–16

ASSIGN event 16–24

ASSIGN header 16–27

ASSIGN statement 7–19
grouping assignments 21–2
vs BUFFER-COPY 21–3

Assigning
object attributes 18–3
values to variables 3–10

Asterisk (*) 2–11, 3–14
as wildcard in strings 2–3
multiplication sign 2–24

Asynchronous keyword 21–37

Asynchronous requests 21–37

At-sign (@) 6–36, 12–16

Attribute values
changing 8–7

Attributes
appearance 8–10
data management 8–13
dynamic objects 5–10, 18–3
dynamic queries 19–3
dynamic temp-tables 20–9
frames 2–6
geometry 8–10
identifying 8–14
inheriting 2–13
listing 18–25
modifying 9–2
multiple fields 9–6
relationship 8–13

setting 8–6
windows 18–44

AUTO-RESIZE attribute 12–50

AVAILABLE attribute
dynamic buffer handles 19–28

AVAILABLE function
queries 10–9

AVAILABLE keyword 6–31

Avoiding
flashing 21–16

B

-Bt startup parameter 11–3

Back slash (/)
division sign 2–24

Back-tab key 12–57

Basic objects 8–2

before 17–26

Before-image (BI) file 17–26

BEGINS keyword 2–3

bin directory 3–5

Blanks
unknown value 2–10

Block header
ON phrase 17–27

Blocks
defining 6–7
FOR 6–15
guidelines 21–4
identifying 6–24
leaving 6–23
looping 6–10
naming 6–24
nested 2–4

procedure 6–1
REPEAT 6–26
scoping 6–3
types 6–3

Breaking
infinite loop 6–12
infinite loops 18–23

Breaking, lines 1–14

Browse columns
accessing 12–36, 18–58
enabling 12–10
locking 12–36
moving 12–38
validation expressions 20–28

Browse columns. *See also* columns.

Browse icon 4–3

Browse rows
creating 12–28
deleting 12–31
refreshing 12–25
repositioning focus 12–25
updating 12–27

Browses
adding to a window 4–13
character-mode 12–51
defined 12–1
defining 5–6
defining queries 12–3
events 12–19
handles 12–13
moving 12–48
multiple-select 12–12
options 12–5
overlaying objects 12–39
properties 4–16
queries 10–4
query interaction 12–15
read-only 12–47
repositioning 12–26
resizing 12–47
row height 12–9, 12–49

triggers 7–22
user interface 12–33
when to use 9–14

Browses. *See also* dynamic browses, static browses.

Buffer attributes
dynamic temp-tables 20–10

Buffer field handles
using 18–28

Buffer FIND methods 19–29

Buffer handle methods 19–28

Buffer handles
using 18–28

Buffer lists
setting 19–4

Buffer methods
dynamic temp-tables 20–10

Buffer scope
raising 7–16, 7–33

Buffer size (-bt) startup parameter 21–22

BUFFER-COMPARE method
dynamic buffer handles 19–28

BUFFER-COMPARE statement
vs ASSIGN 21–3

BUFFER-COPY method
dynamic buffer handles 19–28

BUFFER-COPY statement 11–16
vs ASSIGN 21–3

BUFFER-CREATE method
dynamic buffer handles 19–28

BUFFER-DELETE method
dynamic buffer handles 19–28

BUFFER-FIELD attribute
field objects 18–29

-
- BUFFER-FIELD method
dynamic buffer handles 19–28
- Buffer-field objects
using 19–39
- BUFFER-HANDLE attribute
buffer fields 19–39
- BUFFER-NAME attribute
buffer fields 19–39
- BUFFER-RELEASE method
dynamic buffer handles 19–28
- Buffers
creating 7–3
defined 7–2
index cursors 7–18
naming 7–2
screen 8–13
temp-tables 11–8
- BUFFER-VALUE attribute
buffer fields 19–39
- Building
menu bars 9–29
queries 4–10
- Built-in functions
CAN-FIND 6–36
- Button icon 4–3
AppBuilder palette 4–24
- Button triggers
editing 9–7
- Buttons
adding 12–28
adding to a window 4–24
aligning 4–31
defining 5–5
events 4–28
triggers 4–25
- BY keyword 4–11
- BY phrase 6–16
- C
- .cst files 18–13
- Caching data 10–6
- calcdays procedure 3–9
- Calculating
columns 12–15
totals 7–18
- Calculations
procedures 2–24
- Call stack
defined 13–3
- Calling
AppServers 20–40
- Canceling
subscribe 14–40
triggers 8–33
- CAN-FIND function 6–36
vs FIND 21–9
- CAN-FIND keyword 10–6
- CAN-QUERY function 18–24
- CAN-SET function 18–24
- Capitalization
keywords 1–14
- CASE statement 21–5
- Case-sensitivity
4GL 1–14
- Changing
attribute values 8–7
indexes 6–32
object formats 2–8
object labels 2–8
row height 12–49
- Changing in temp-table buffer 20–12
- Character data type 2–9

- Character string functions 2–22
- Characteristics
 - Progress 4GL 1–13
- Character-mode
 - menus 9–41
- Character-mode browses 12–51
- Checked attribute
 - menus 9–36
- Checking
 - queries 10–26
 - syntax 2–35, 7–19
- Child window family
 - defined 18–47
- CHOOSE event 4–28
- Choose Event dialog box 4–28
- CHOOSE trigger 4–25
- Cleaning up
 - dynamic buffers 19–36
 - dynamic queries 19–16
 - dynamic temp-tables 20–14
 - memory 21–23
 - procedures 13–27
- Clear method
 - dynamic temp-tables 20–9
- Clear state
 - dynamic temp-tables 20–3
- CLOSE QUERY statement 10–10
- Closing
 - queries 10–10
- Clusters 17–26
- CMT alias 3–14
- Code
 - AppBuilder-generated 5–1
 - copying 4–25
 - duplicating 11–17, 13–18
 - pasting 4–28
 - printing 5–2
 - viewing 4–18
- Code generation 1–3
- Code Preview dialog box 4–20
- Code References dialog box 9–8
- Colon syntax 12–45
- Colons
 - aligning 18–39
- Color Coding Setup dialog box 2–31
- Coloring
 - cells 12–46
 - columns 12–46
 - text 14–10
- Colors
 - character mode browse 12–56
- Column attribute 8–10
- Column editor 4–14
- Column events 12–22
- COLUMN-DCOLOR attribute 12–56
- COLUMN-LABEL keyword 2–12
- COLUMN-PFCOLOR attribute 12–56

-
- Columns
 - adding to dynamic browse 20–24
 - adding to static browse 20–31
 - calculated 12–15
 - coloring 12–46
 - enabling 12–10
 - moving 12–38, 12–49
 - resizing 12–49
 - searching 12–24
 - sizing 12–18
 - stacked labels 12–45
 - Columns. *See* fields.
 - Columns. *See also* browse columns.
 - Combo boxes
 - adding 12–40
 - defined 9–14
 - triggers 12–41
 - Comma (,) format character 2–11
 - Comments
 - adding 3–14, 5–15
 - Comments field 9–5
 - Commit units
 - defined 16–2
 - Comparison operators 2–3
 - COMPILE statement
 - LISTING option 7–8
 - Compiled files 3–2
 - location 3–5
 - Compiling
 - multiple procedure at once 2–36
 - procedures 1–12, 2–35
 - Conflicting tables 7–14
 - Conflicts
 - locking 15–9
 - Connecting
 - databases 1–8
 - Contains keyword 2–3
 - Control keys
 - character mode 12–53
 - Controls. *See* objects.
 - Conventions
 - naming variables 2–13
 - Converting
 - data 2–20
 - Copying
 - code 4–25
 - multiple fields 11–16
 - sports2000 database 1–7
 - CREATE event 16–24
 - CREATE header 16–25
 - CREATE triggers
 - database sequences 16–31
 - CREATE WIDGET-POOL statement 5–3
 - dynamic objects 18–12
 - CREATE-LIKE method
 - dynamic temp-tables 20–3
 - CREATE-RESULT-LIST-ENTRY method
 - browses 20–31
 - Creating
 - browse rows 12–28
 - dialog boxes 20–39
 - dynamic browses 20–23
 - dynamic objects 18–3
 - dynamic queries 19–11, 20–37
 - dynamic temp-tables 20–2, 20–42
 - editor objects 9–4
 - listing files 7–8
 - queries 4–10
 - static objects 8–3
 - visual objects 18–3
 - warning messages 10–28
 - windows 4–4, 5–8
 - Current row
 - identifying 10–16

Current working directory
default 3–5

CURRENT-CHANGED attribute
dynamic buffer handles 19–27

CURRENT-CHANGED function 15–19

CURRENT-COLUMN attribute 12–36

CURRENT-RESULT-ROW attribute
dynamic queries 19–16

CURRENT-RESULT-ROW function
10–16

CURRENT-WINDOW statement 5–13

CURRENT-WINDOW system handle
18–18

Cursors
index 6–30, 7–18

Customer table 1–5

Customizing
dynamic browses 20–34
PRO*Tools palette 3–3

Cyclic Redundancy Check (CRC) identifier
16–29

D

Data
accessing without looping 6–28
converting 2–20
passing between sessions 11–2
retrieving 6–18, 6–22, 10–6
retrieving in advance 10–13
summarizing 11–8
temporary storage 11–2

Data access 6–1
queries 10–2

Data administration tool 15–9

Data Dictionary
closing 1–9
indexes 6–19
starting 1–6

Data handling 18–33

Data handling statements 15–4

Data representation objects
using 9–2

Data types
abbreviations 2–28
basic 2–9
formatting 2–11

Data typing 18–34

Database events 16–24

Database fields. *See also* fields.

Database indexes
defined 6–19

Database sequences 21–21
CREATE triggers 16–31

Database transactions
managing 17–1

Database trigger procedures 16–22

Database triggers
defining 16–22
guidelines 16–23

Databases
connecting 1–8
copying 1–7
logic procedures 16–12
sample 1–2
sports2000 1–2
updating 16–1

Databases connections 21–20

DATE data type 2–9

Date functions 2–20

- DAY keyword 2–20
DB fields icon 4–3
DBNAME attribute
 dynamic buffer handle 19–27
DCOLOR attribute 12–56
Deadly embrace 15–13
Decimal data type 2–9
Decimal point format character 2–11
Decimals keyword 2–12
Default event processing
 suppressing 8–35
DEFAULT-ACTION event 12–19
DEFAULT-ACTION trigger
 browse 12–20
DEFAULT-BUFFER-HANDLE attribute
 dynamic temp-tables 20–9
DEFAULT-WINDOW system handle
 18–18
DEFINE BROWSE statement 12–4
DEFINE BUFFER statement 7–3
DEFINE BUTTON statement 5–5
DEFINE FRAME statement 9–9
DEFINE VARIABLE statement 8–5
 aliases 2–28, 2–30
DEFINE work-table statement 11–3
Defining
 blocks 6–7
 database triggers 16–22
 objects 5–4
 strings 2–4
 triggers 16–28
 variables 2–14
Definitional triggers
 defined 8–27
Definitions section 5–3
DELETE event 16–24
DELETE header 16–25
DELETE WIDGET-POOL statement 18–13
DELETE-RESULT-LIST-ENTRY method
 browses 20–31
DELETE-SELECTED-ROWS method
 12–31
Deleting
 browse rows 12–31
 dynamic objects 18–10
 dynamic queries 19–16
 persistent procedures 13–24
 temp-table copies 21–36
DESCENDING keyword 6–16
Design window 4–4
Desktop. *See* Application Development Environment (ADE) desktop
Destroying
 objects 8–23
Developer events 8–26
Dialog boxes
 creating 20–39
 parenting 5–13
Direct manipulation events 8–25
Directory
 compiled files 3–5
Disabling
 fill-in fields 9–2
 menu items 9–27
DISPLAY statement 1–12, 2–19

- DISPLAY type
 - changing 9–6
- Displayed-fields preprocessor 9–7
- Displaying
 - rows 12–21
- Displaying objects 8–10
- Display-type keyword 5–10
- Distributed applications
 - defined 1–4, 15–1
- Division sign (/) 2–24
- DO block
 - vs. REPEAT block 6–28
- DO blocks 6–7
 - vs REPEAT block 21–4
- DO FOR block 7–14
- DO WHILE statement 6–12
- Double quotation marks 2–4
- Down frames
 - defined 2–6, 10–12
- DOWN WITH statement 10–12
- Dragging, browses 12–48
- Duplicating
 - code 11–17, 13–18
- Dynamic applications
 - defined 1–4
- Dynamic browse columns
 - guidelines 18–53
 - identifying 18–22
- Dynamic browses
 - adding columns 20–24
 - creating 20–23
 - customizing 20–34
 - guidelines 18–53
- Dynamic buffer handle attributes 19–27
- Dynamic buffer handles 19–26
 - using 19–30
- Dynamic buffers
 - creating 19–26
 - deleting 19–36
 - guidelines 19–38
- Dynamic fields
 - adding 18–34
 - creating 18–32
- Dynamic graphical objects
 - using 18–1
- Dynamic menu items
 - creating 18–61
- Dynamic menus
 - creating 18–59
- Dynamic objects
 - adding 18–26
 - attributes 5–10
 - creating 18–3
 - defined 5–9
 - deleting 18–10
 - handles 18–5
 - identifying 20–44
 - managing 18–9
 - positioning 18–32
 - reusing 21–11
 - sizing 18–32
 - triggers 18–5
 - when to use 19–8
- Dynamic programming
 - considerations 20–45
 - vs static programming 21–10
- Dynamic queries
 - assembling 19–11
 - attributes 19–3
 - creating 20–37
 - deleting 19–16
 - handles 19–3
 - methods 19–3

-
- Dynamic temp-table
 - changing field attributes 20–12
 - Dynamic temp-tables
 - attributes 20–9
 - buffers 20–10
 - creating 20–2, 20–42
 - deleting 20–14
 - methods 20–3
 - parameters 20–17
 - states 20–3
 - DYNAMIC-FUNCTION function 14–12
 - E**
 - EACH keyword 6–15
 - alternatives 6–19
 - Editing
 - code 4–18, 4–19
 - Propath 3–5
 - triggers 9–7
 - Editing options 2–29
 - Editing shortcuts 2–29
 - Editor objects
 - creating 9–4
 - ELSE keyword 2–25
 - EMPTY-TEMP-TABLE-BUFFER method
 - dynamic temp-tables 20–12
 - ENABLE attribute
 - unique identity
 - frames 8–19
 - ENABLE phrase
 - browses 12–4
 - enable_ui procedure 4–19
 - ENABLED option
 - fill-in fields 7–21
 - Enabling
 - fill-in fields 9–2
 - menu items 9–27
 - END event 12–19
 - END PROCEDURE statement 3–12, 4–23
 - ENDKEY condition 17–31
 - END-SEARCH keyword 12–24
 - Enter Constant dialog box 4–11
 - ENTRY function 7–29
 - ENTRY keyword 2–17, 2–21
 - Entry triggers 12–42
 - EROOR-STATUS system handle 17–28
 - Error condition
 - handling 17–27
 - Error messages 7–14
 - avoiding 19–12
 - buffer handle 18–7
 - DATA-TYPE 18–24
 - FIND 17–27
 - FIND processing 7–11
 - illegal nested block 7–10
 - mismatched parameters 7–19, 7–34
 - ROW-MARKERS 18–51
 - stale widget handle 21–29
 - static handle 18–11
 - unknown table 18–11
 - validation expression 20–29
 - Errors
 - syntax 2–36
 - ESCAPE key 6–26
 - ETIME function 21–7
 - European (-e) startup parameter 2–10
 - Event name option
 - Section Editor 5–20

Event processing
 suppressing 8–35

Event-driven applications
 defined 5–13

Event-driven code
 triggers 5–12

Event-driven programs
 vs. procedural 5–17

Events
 applying 8–33
 browse 12–19
 database 16–24
 defined 4–28
 developer 8–26
 direct manipulation 8–25
 keyboard 8–26
 MENU-DROP 9–36
 nonstandard 8–37
 potable mouse 8–25
 publishing 14–39
 subscribing 14–38
 user interface 8–24
 user manipulation 12–50
 windows 18–44

Exclamation point (!) 2–11

EXCLUSIVE_LOCK keyword 15–10

Executing
 vs. processing procedures 5–12

EXP keyword 2–26

EXPANDABLE attribute 12–50

Expressions
 arithmetic 2–24

EXTENT keyword 2–12

External procedures
 defined 3–8

F

FETCH-SELECTED-ROW method 12–13,
 12–32

Field attributes 20–12

Field formats
 versus width 18–42

Field names
 qualifying 2–4

Field Properties dialog box 8–4

Fields
 adding 5–20
 adding to a window 4–5
 arranging in a window 4–6
 changing formats 2–8
 changing labels 2–8
 copying multiple 11–16
 identifying in trigger 19–21
 naming conventions 4–22
 setting multiple attributes 9–6
 sorting 4–11
 sports2000 database 1–9
 visualizing 8–4

File extensions 4–8

File-name attribute 13–13

Filenames
 inserting into a procedure 5–20
 temporary 13–16

Files
 inserting contents of 5–20

Fill keyword 2–22

Fill-in fields
 adding 7–20
 attributes 7–21, 9–2
 changing to editors 9–4
 disabling 9–2
 property sheet 8–14
 vs. text fields 9–25

- Filtering
dynamically 19–16
- FIND CURRENT function 15–19
- FIND event 16–25
- FIND header 16–25
- Find processing error 7–11
- FIND statement 6–28
vs CAN-FIND 21–9
vs FOR FIRST 21–7
- Finding
procedures 3–7
running procedures 13–11
- FIRST keyword 6–19
- FIRST-CHILD attribute
frame 18–19
- FIRST-CHILD keyword 9–37
- FIRST-COLUMN attribute 12–36
- Flashing
avoiding 21–16
- FONT-TABLE system handle 18–39
- FOR blocks
vs. DO blocks 6–15
- FOR EACH statements
guidelines 21–16
- FOR FIRST construct
vs FIND 21–7
- FOR phrase 6–16
- FORMAT keyword 2–12
- Formats
data types 2–11
- Formatting
fields 2–8
- Forward declarations
function 14–4
- FRAME attribute 8–13
- FRAME-NAME attribute 8–13
- Frames
attributes 2–6
centering objects 4–31
defined 2–6
defining 5–7
one down 5–8
parenting 5–13, 18–32
referencing 6–25
resizing 4–13
scoping 6–13
sizing 8–22
- Free reference
defined 7–7
- Free references
FIND statement 7–12
- Functions
4GL 2–20
AppBuilder-generated 14–9
arithmetic 2–26
built-in 2–17
declaring IN SUPER 14–8
defining 14–3
externally defined 14–6
forward declarations 14–4
user-defined 14–2

G

- Generating
listing files 7–8
- Geometry attributes 8–10
- GET statements 10–8
- GET-BUFFER-HANDLE method
dynamic queries 19–6
- GET-SIGNATURE method 13–13

GLOBAL keyword
 temp-tables 11–7

Global shared objects
 defined 13–40

Grid lines 4–13

Grouping assignments 21–2

gui directory 3–7

Guidelines
 database triggers 16–23
 dynamic buffers 19–38
 dynamic programming 20–45
 memory management 21–23
 record buffers 7–7
 record locking 15–16
 super procedure 14–18

H

HANDLE data type 2–9

HANDLE form
 dynamic temp-tables 20–19

HANDLE keyword 5–5

Handle variables
 defining 5–4

Handles
 browse 12–13
 defined 8–3
 dynamic objects 18–5
 dynamic queries 19–3
 persistent procedures 13–10
 queries 10–4
 SELF 19–21
 SESSION 20–44
 stale 21–33
 static objects 18–7
 storing 18–34
 temp-tables 11–14, 20–2

Handling
 labels 18–33

Headers
 block 6–24
 trigger procedures 16–25

Height attributes 8–10

Help
 online 2–32

HELP attribute 8–11

HIDDEN attribute 8–10, 9–11

HIDDEN keyword 5–10

Hiding
 screen content 21–16
 windows 18–47

Hiding objects 8–10

Highlighting
 fill-in fields 9–3

History
 Progress 4GL 1–2

HOME event 12–19

Hyphens (-)
 naming conventions 1–14
 versus minus sign 2–24

I

Identifying
 dynamic browse columns 18–22
 dynamic objects 19–21
 rows 10–20

IF AVAILABLE phrase 4–22

IF keyword 2–25

IF-THEN-ELSE phrase 2–15

Images
 adding 9–20

Improving performance
 queries 10–18

-
- IN FRAME phrase 8–22
 - IN WINDOW phrase 4–22
 - Include files
 - button trigger 11–22
 - defined 11–17
 - Indent style option 2–29
 - Indentation 1–14
 - Indenting
 - nested blocks 2–4
 - Index cursors
 - buffers 7–18
 - defined 6–30
 - Index keyword 2–22
 - Index Properties dialog box 6–20
 - Indexed-reposition keyword 10–18
 - Indexes
 - logical values 21–14
 - primary 6–29
 - sorting data 6–19
 - specifying 6–34
 - status indicators 21–14
 - switching 6–32
 - temp-tables 11–6
 - unnecessary 21–15
 - Index-information attribute
 - dynamic queries 19–15, 19–23
 - Index-information function
 - static queries 19–15
 - Indexing
 - guidelines 21–14
 - Infinite loop
 - breaking 6–12
 - Infinite loops
 - breaking 18–23
 - Inheriting, attributes 2–13
 - INITIAL attribute 8–13
 - INITIAL keyword 2–12
 - INITIAL phrase 6–11
 - Initialization file 3–3
 - Inner joins
 - defined 6–18
 - INPUT parameter 3–8
 - INPUT-OUTPUT parameters 3–8
 - INSERT statement 15–4
 - Insert statement 6–26
 - Inserting
 - browse rows 12–28
 - fields 5–20
 - menu lines 9–27
 - INSERT-ROW method 12–28
 - Instantiating
 - objects 8–18
 - persistent procedures 13–8
 - INTEGER data type 2–9
 - INTEGER function 7–29
 - Intelligent Edit control 2–28
 - Internal procedures
 - AppBuilder-generated 5–17
 - defined 3–8
 - disable_UI 5–14
 - enable_UI 4–19
 - list of 5–20
 - writing 3–9
 - Internal-entries attribute 13–13
 - Interpreter 1–10, 2–35
 - IS-OPEN attribute
 - dynamic queries 19–13
 - Iterating, through records 7–31

J

Joining
tables 2–4, 6–16, 10–6

Joins
inner vs. outer 6–18
one-to-one 19–8
selection criteria 21–18

Justifying, labels 12–45

K

KBLABEL function 9–41

KEEP-TAB-ORDER attribute 5–8

Keyboard Event dialog box 8–26

keys
F1 2–32
F3 9–41
F5 9–9
F6 2–19, 2–35

Keywords
capitalization 1–14
Progress 4GL 1–11

L

-L startup parameter 15–17

LABEL attribute 8–12

LABEL keyword 2–12

LABEL-DCOLOR attribute 12–56

Labels
colon-aligning 18–39
handling 18–33
justifying 12–45
side 18–33
stacked 12–45

LAST keyword 6–19

Layout
adjusting 4–31

Layout grid lines 4–13

LEAVE statement 6–23

LEAVE trigger
multiple objects 19–20

LEAVE triggers 12–42

Leaving
rows 12–21

LEFT-TRIM keyword 2–22

LENGTH keyword 2–22

Libraries
run-time 13–10

LIKE keyword 2–13

LIKE option
DEFINE TEMP-TABLE statement 11–4

Line break 1–14

Lines
menus 9–27

List functions 2–21
using 7–28

List Sections dialog box 4–29

Listing
objects 9–14
valid attributes 18–25

Listing files 7–8

LISTING option
COMPILE statement 7–8

LIST-QUERY-ATTRS function 18–25

Lists
populating at run-time 18–30
vs arrays 21–7

-
- LIST-SET-ATTRS function 18–25
 - Local forward declaration 14–5
 - Local-before-image (LBI) file 17–26
 - Locating
 - running procedures 13–11
 - Lock table entries (-l) startup parameter 15–17
 - Locked attribute
 - dynamic buffer handles 19–27
 - Locked columns
 - defined 12–36
 - Locking
 - browse columns 12–36
 - Locking conflicts
 - testing 15–9
 - LOG keyword 2–26
 - Logic procedures
 - database updates 16–12
 - Logical data type 2–9
 - Logical values
 - indexes 21–14
 - LOOKUP function 7–29
 - LOOKUP keyword 2–21
 - Looping
 - DO blocks 6–10
 - infinite 6–12
- M**
- Main block
 - defined 4–18, 5–13
 - Main window, appbuilder 4–2
- Managing
 - dynamic objects 18–9
 - memory 5–3
 - MATCHES keyword 2–3
 - MAXIMUM keyword 2–26
 - Maximum size of buffers (-Mm) startup parameter 21–22
 - Memory
 - excessive usage 3–12
 - Memory leaks
 - preventing 18–36
 - Memory management 21–11
 - guidelines 21–23
 - widget pools 5–3
 - MEMPTR
 - defined 21–37
 - Menu bars
 - assigning to windows 9–29
 - defined 9–26
 - defining 9–27
 - property sheet 9–31
 - Menu hierarchy
 - defined 9–26
 - Menu items
 - enabling 9–27
 - Menu mnemonics 9–30
 - Menu triggers 9–33
 - Menu-drop event 9–36
 - Menus
 - character-mode 9–41
 - code 9–38
 - Menus. *See also* dynamic menus.
 - MESSAGE statement 10–28
 - dynamic queries 19–23

Messages
network 21–21

Methods
buffer FIND 19–29
defined 8–3, 8–16
dynamic buffer handles 19–28
dynamic queries 19–3
dynamic temp-tables 20–3
invoking 8–16
windows 18–44

Minimizing
windows 18–48

Minimum keyword 2–26

Minus sign (-) 2–24

Minus sign (-) format character 2–12

Mnemonics
menus 9–30

MOBVABLE attribute 8–12

MODULO keyword 2–26

MONTH keyword 2–20

MOVE-COLUMN method 12–38

Moving
browse columns 12–38, 12–49
browses 12–48
objects 4–31

Multi-component indexes
Indexes
multi-component 21–15

Multi-field Selector dialog box 4–5

Multiple rows, selecting 12–12

Multiple windows
using 18–43

Multiplication sign (*) 2–24

N

n format character 2–11
NAME attribute 8–14
dynamic buffer handle 19–27
dynamic temp-tables 20–9
static queries 19–7
Named widget pools 21–30
dynamic objects 18–14

Naming
buffers 7–2
objects 2–8, 4–7
queries 10–5
variables 2–13

Naming conventions 1–14
procedures 3–12

Navigating
menu handles 18–61

Navigation
character mode 12–57

Navigation methods
dynamic queries 19–13

Nested blocks
creating 2–4
error message 7–10

Nesting
DO blocks 6–9
procedure calls 21–5
weak-scoped references 7–10

Nesting blocks
minimizing 21–4

Network messages 21–21

NEW attribute
dynamic buffer handles 19–27

New Function dialog box 14–9

New procedures, creating 4–4

-
- NEXT statement 6–25
- NEXT-COLUMN attribute 12–36
- NEXT-SIBLING keyword 9–37
- NO-APPLY keyword 8–35
- NO-ASSIGN keyword
 DEFINE BROWSE statement 12–27
- NO-ASSIGN option
 browses 12–5
- NO-ERROR qualifier
 FIND statement 6–31
- Nonindexed fields
 dynamic queries 19–15
- Nonstandard events 8–37
- NO-ROW-MARKERS keyword
 browses 12–5
- NO-UNDO keyword 2–13
 temp-tables 17–9
- NO-UNDO variables
 using 21–2
- NO-WAIT option
 AVAILABLE function 15–14
 LOCKED function 15–14
- NUM-BUFFERS attribute
 dynamic queries 19–6
- NUM-COLUMNS attribute 12–36
- NUM-ENTRIES function 7–28
- NUM-FIELDS attribute
 dynamic buffer handles 19–27
- NUM-RESULTS attribute
 dynamic queries 19–16
- NUM-RESULTS function 10–11
- NUM-SELECTED-ROWS attribute 12–13
- O**
- Queries
 as objects 10–4
- Object handles 18–5
 defined 8–3
- Object properties 4–16
- Objects
 aligning 4–31
 defined 6–3
 defining 5–4
 destroying 8–23
 dynamic 18–1
 formats 2–8
 initializing 8–18
 labels 2–8
 list of 5–20
 naming 4–7
 realizing 8–18, 8–23
 relationships between 18–21
 renaming 4–7, 4–17
 selecting multiple 4–24
 types 8–2
- OF keyword 2–4
- OFF-END event 12–19
- OFF-HOME event 12–19
- ON phrase
 block header 17–27
- ON RETURN phrase 17–30
- ON statement
 ANYWHERE option 8–33
- One down frames 5–8
 defined 2–6
- One record, retrieving 6–35
- Online Help
 accessing 2–32
 keywords 8–8
- OPEN QUERY statement 10–6, 15–4

OpenEdge AppBuilder
using 4–1

OpenEdge DataServers 21–17

Opening
queries 10–6

Operands
arithmetic 2–24

Operators
comparison 2–3

Optimistic locking strategy
defined 15–18

Options
Procedure Editor 2–29

Ordering
data retrieval 6–22

Outer joins
defined 6–18

OUTER-JOIN keyword 10–6

OUTPUT parameter 3–8

Overlaying objects, on browse cells 12–39

OWNER attribute
menus 9–26

OWNER keyword 9–37

P

Temp-tables
as parameters 11–13

Palette
PRO*Tools 3–3

Palettes
AppBuilder 4–3

Parameters
dynamic temp-tables 20–17
INPUT 3–8
OUTPUT 3–8
passing 14–39
persistent procedures 13–9
temp-tables 11–13

PARENT keyword 9–37

Parenting
dialog boxes 5–13

Passing
temp-tables between sessions 20–19
temp-tables within a session 20–18

Pasting
code 4–28

Performance
improving 21–21
improving in queries 10–18

Period (.)
as wildcard in strings 2–3

PERSISTENT keyword 13–5

Persistent procedures 13–3
as run-time libraries 13–10
deleting 13–24
instantiating 13–8
parameters 13–9
sharing 13–17

Persistent triggers 8–31

Pessimistic locking strategy
defined 15–18

PFCOLOR attribute 12–56

Photographs
adding 9–20

Physical transactions
defined 16–2

Pictures
adding 9–20

-
- Placeholder queries 12–33
 - Plus sign (+) 2–24
 - Plus sign (+) format character 2–12
 - Populating
 - selection lists dynamically 20–37
 - Pop-up menus
 - defined 9–26
 - defining 9–39
 - Portable mouse events 8–25
 - POSITION attribute
 - buffer fields 19–40
 - Positioning
 - objects 18–32
 - queries 10–8
 - Precaching data 10–6
 - PREPARED attribute
 - dynamic temp-tables 20–9
 - Prepared state
 - dynamic temp-tables 20–3
 - Prepare-string attribute
 - dynamic queries 19–9
 - Preparing
 - triggers 8–27
 - Preprocessor values
 - defined 4–20
 - Preprocessors
 - DISPLAYED-FIELDS 9–7
 - list of 5–20
 - query 6–8
 - PRESLECT phrase 7–30
 - queries 10–6
 - PREV-COLUMN attribute 12–36
 - Previewing code 4–20
 - PRIMARY attribute
 - dynamic temp-tables 20–10
 - Primary index 6–29, 6–32
 - Printing
 - code preview 5–2
 - Private-data attribute 8–14, 13–13
 - PRO*Tools
 - accessing 3–3
 - PRO*Tools palette 13–11
 - customizing 3–3
 - Procedural programs
 - vs. event-driven 5–17
 - Procedure blocks 6–1
 - transactions 17–8
 - Procedure call stack 13–3
 - Procedure Editor 2–29
 - color coding 2–31
 - intelligent editor 2–28
 - options 2–29
 - starting 1–10
 - Procedure handles
 - persistent 13–10
 - PROCEDURE keyword 3–9
 - Procedure libraries
 - defined 16–35
 - Procedure object viewer 13–26
 - Procedure settings section 5–8
 - Procedure signature
 - defined 13–14
 - PROCEDURE statement 3–12
 - Procedure windows
 - multiple 2–36

Procedures

calculations 2–24
compiling 1–12, 2–35
finding 3–7, 13–11
internal vs. external 3–12
naming conventions 3–12
running 1–12, 3–1
search order 3–12
stopping 4–9
writing efficiently 21–2

Processing

vs. executing procedures 5–12

Program variables 2–9**Progress 4GL**
compared to SQL 1–15**Progress 4GL Options dialog box** 2–31**Progress 4GL, history** 1–2**Progress Advisor** 16–7**Progress Dynamics** 1–4, 7–26, 8–2, 14–24,
15–21**progress.ini file** 3–3**PROMPT-FOR statement** 15–6**Propath**

editing 3–5
setting 21–20
using 3–3

Properties Window 9–2**Properties.** *See also* attributes.**Property sheets**

fill-in field 8–14
opening 9–6
using 4–16

ProtoGen tool 14–30**Prototypes** 14–4**PUBLISH statement** 14–37, 14–39**Q****Qualifying**
variables 2–12**Queries**

browse 12–3
browse interaction 12–15
characteristics 10–4
checking validity 10–26
closing 10–10
current row 10–16
defining 5–6, 10–5
dynamic vs static 19–5
empty example 6–8
guidelines 21–16
handles 10–4
naming 10–5
number of rows 10–11
opening 10–6
placeholder 12–33
populating browses 10–4
positioning 4–22, 10–8
repositioning 10–20, 12–26
temp-tables 11–20

Queries. *See also* dynamic queries.

Query Builder 16–8
opening 5–20
using 4–10**Query handles**
dynamic 19–3**QUERY-CLOSE method**
dynamic queries 19–12**QUERY-OFF-END attribute**
dynamic queries 19–13**QUERY-OFF-END function** 10–9**QUERY-OPEN method**
dynamic queries 19–10**QUERY-PREPARE method**
dynamic queries 19–8**Question mark (?)** 2–16

-
- Quick request (-q) startup parameter 21–22
- QUIT condition 17–33
- QUIT statement 6–25
- Quotation marks 2–4
- QUOTER function
dynamic queries 19–11
- R**
- .r extension 4–8
- Radio sets
adding 9–15
code 9–18
defined 9–15
- Raising
buffer scope 7–16, 7–33
- Random (-rand) startup parameter 2–27
- RANDOM keyword 2–27
- RAW keyword 2–22
- r-code 2–35
- READ-FILE method 8–17
- Reading
database metaschema data 20–36
dynamic object attributes 18–21
- Reading records
NO-LOCK 15–15
- READ-ONLY attribute 8–11
- Read-only browses 12–47
- Read-only flag
Section Editor 4–19
- Read-only option
fill-in fields 7–21, 9–3
- Realizing
objects 8–18, 8–23
- RECID keyword 10–23
- Record buffers
defined 7–2
- Record locking
guidelines 15–16
locking
records 15–7
strategies 15–18
- Record locks
deadly embrace 15–13
releasing 15–16
- Record scope 7–6
- Records
filtering dynamically 19–16
iterating through 7–31
retrieving one 6–35
scoping 6–13
sorting 6–16
- Rectangles
adding 9–18
characteristics 9–18
- Referencing
frames 6–25
- REFRESHABLE attribute
browses 21–16
- Releasing
record locks 15–16
- Remove-super-procedure method 14–15
- Removing
browse rows 12–31
- Renaming
objects 2–8, 4–7, 4–17
- REPEAT blocks 6–26
vs DO 21–4, 6–28

- REPEAT PRESELECT blocks 7–30
- REPLACE keyword 2–22
- Reposition methods
 - dynamic queries 19–15
- REPOSITION statement 10–23
- Repositioning
 - browses 12–26
 - queries 10–8, 10–20
- RESIZEABLE attribute 8–12
- Resizing
 - browse columns 12–49
 - browses 12–47
 - fields 6–32
 - frames 4–13
- Restoring
 - windows 18–48
- Result sets
 - defined 10–2
 - retrieving in advance 10–13
 - size 12–3
- Results list
 - defined 10–11
- RETAIN-SHAPE option
 - images 9–20
- Retrieving
 - data 6–22
 - single record 6–35
- Retrieving data
 - queries 10–6
- RETURN NO-APPLY statement 5–12
- RETURN statement 13–2, 17–30
- RETURN-VALUE function 13–2
- RETURN-VALUE statement 14–40
- REVERT statement 8–33
- RIGHT-TRIM function 7–28
- RIGHT-TRIM keyword 2–22
- R-INDEX keyword 2–22
- Root window
 - defined 18–47
- ROUND keyword 2–27
- ROW attribute 8–10
- Row events
 - events
 - rows 12–21
- Row height, browses 12–9
- RowID
 - defined 10–20
- ROWID attribute
 - dynamic buffer handles 19–27
- ROW-LEAVE event 12–27
- Rows
 - current 10–16
 - height 12–49
 - identifying 10–20
 - queries 10–11
 - selecting multiple 12–12
- Rows. *See also* browse rows.
- RULE keyword
 - menus 9–27
- Rules
 - menus 9–27
 - record buffers 7–7
- RUN PERSISTENT statement 13–5
- RUN statement 3–2
- RUN SUPER statement 14–17
- Running
 - procedures 1–12, 2–35, 3–1
 - procedures in AppBuilder 4–9

-
- Run-time
 - populating lists 18–30
 - Run-time interpreter 2–35
 - Run-time libraries 13–10
 - Run-time references 14–12
 - Run-time triggers
 - defined 8–27
 - S**
 - Sample databases
 - location 1–8, 3–5
 - sports2000 1–2
 - Sample procedures 11–22
 - calcdays 3–9
 - childproc.p 13–6
 - customer.p 1–16
 - FOR EACH Customer 1–5
 - h-BinCheck.p 7–26
 - h-buttontrig1.i 11–22
 - h-cleanup.p 19–36
 - h-ConvTemp1.p 14–5
 - h>DeleteObject.p 21–13
 - h-findCustUser1.p 15–12
 - h-finduseful.p 13–12
 - h-mainsig.p 13–15
 - h-MakeBuffer.p 21–28
 - h-ordercalcs.p 7–18
 - h-OrderLogic.p 16–17
 - h-RunMakeBuff.p 21–28
 - h-StartSuper.p 14–33
 - h-testsig.p 13–15
 - h-UsefulProc.p 13–12
 - h-WinSuper.p 14–33
 - mainproc.p 6–4, 13–37
 - parentproc.p 13–6
 - subproc 6–4
 - subproc.p 13–37
 - viewing code 5–2
 - Sample windows
 - Customers and Orders 4–7
 - C-WIn 4–7
 - Saving
 - PRO*Tools palette position 3–4
 - procedures 2–19, 2–35, 4–8
 - Schema trigger procedures 16–22
 - Scope
 - defined 6–3
 - Scoping
 - blocks 6–3
 - buffers 7–16
 - dynamic objects 18–9
 - frames 6–13
 - free reference 7–7
 - records 6–13, 7–6
 - strong 7–6
 - temp-tables 11–7
 - weak 7–6
 - Screen buffers 8–13
 - SCREEN-VALUE attribute 8–13, 9–10
 - SCROLLABLE keyword 12–3
 - queries 10–5
 - Scrollbars
 - adding 9–7
 - SCROLLBAR-VERTICAL keyword 8–5
 - SCROLL-NOTIFY event 12–19, 12–20, 12–39
 - Search order
 - procedures 3–12
 - Searching
 - columns 12–24
 - Searching for
 - procedures 3–7
 - Section Editor
 - inserting text 9–7
 - using 4–18
 - Select Related Table dialog box 16–8
 - SELECTABLE attribute 8–12, 12–47

- SELECT-FOCUSED_ROW method 12–29
- Selection criteria
 - joins 21–18
- Selection lists
 - adding 18–26
 - defined 9–14
 - populating dynamically 20–37
- SELF handle
 - triggers 19–21
- SELF keyword 12–23
- SENSITIVE attribute 8–11
- SEPARATOR-FGCOLOR attribute 12–50
- SEPARATORS attribute 12–50
- SEPARATORS keyword
 - browses 12–5
- Sequences, database 16–31
- Session Attributes tool 13–11
- Session handle 13–11
 - identifying dynamic objects 20–44
- Session pool
 - defined 18–12
- Session super procedures 14–30
- Session triggers
 - defined 16–30
 - scope 16–33
- Sessions
 - configuring 21–22
 - multiple 15–9
 - passing data 11–2
 - starting 1–6
- SET phrase 13–5
- SET statement 15–4
- SET-BUFFERS method
 - dynamic queries 19–4
- SET-REPOSITIONED-ROW method 12–25
- Setting
 - attributes 8–6
 - buffer lists 19–4
- SHARED keyword
 - temp-tables 11–7
- Shared objects
 - avoiding 13–38
 - defined 13–35
- SHARE-LOCK keyword 15–12
- Sharing code 13–17
- Shortcut keys
 - menus 9–28
- Shortcuts. *See* aliases.
- Side-labels 18–33
- Signature
 - defined 13–14
- Simple objects 8–2
- Single quotation marks 2–4
- Single record, retrieving 6–35
- SIZE keyword 5–5
 - editors 8–5
- Sizing
 - browses 12–18
 - columns 12–18
 - objects 18–32
- SKIP keyword
 - menus 9–27
- Slider objects
 - code 9–14
 - defining 9–12
- SmartDataObject (SDO) icon 4–3

-
- SmartObjects 15–21
 - defined 8–2
 - purpose 5–21
 - Snowplow metaphor 1–3
 - Sockets 21–37
 - Software failures
 - transactions 17–32
 - SORT keyword 2–27
 - Sorting
 - fields 4–11
 - records 6–16
 - Source-procedure function 13–30
 - Spacing
 - code 1–14
 - objects 4–31
 - Specifying
 - indexes 6–34
 - Sports2000 database
 - location 3–5
 - SQL language
 - compared to Progress 4GL 1–15, 2–5, 6–18
 - src directory 3–7
 - Stack execution
 - super procedures 14–20
 - Stacked labels
 - columns 12–45
 - Stale handles
 - avoiding 21–33
 - Starting
 - AppBuilder 4–2
 - Procedure Editor 1–10
 - sessions 1–6
 - START-SEARCH keyword 12–24
 - Startup parameters
 - European (-E) 2–10
 - rand 2–27
 - recommended 21–22
 - Startup procedure (-p) startup parameter 17–31
 - Statements
 - 4GL 1–14
 - transaction 15–6
 - types 1–15
 - Static browses
 - adding columns 20–31
 - Static browses. *See also* browses.
 - Static object handles 18–7
 - Static objects
 - creating 8–3
 - defined 5–8
 - Static programming
 - vs dynamic programming 21–10
 - Static temp-tables
 - handles 20–2
 - Status indicators
 - indexes as 21–14
 - STOP condition 17–31
 - STOP statement 6–25
 - Stopping
 - procedures 4–9
 - Storing
 - data temporarily 11–2
 - list of handles 18–34
 - STRETCH-TO-FIT option
 - images 9–20
 - STRING function 2–20
 - String functions 2–22
 - using 7–28

- Strings
defining 2–4
manipulating 2–22
translating 5–11
- STRING-VALUE attribute
buffer fields 19–39
- Strong-scoped
defined 7–6
- Strong-scoped references
DO-FOR block 7–14
- Submenus 9–26
creating 18–60
defining 9–27
- Subprocedures
running 3–2
- Subprocesses
minimizing 21–21
- SUBSCRIBE
canceling 14–40
- SUBSCRIBE statement 14–37
- SUBSTR function 7–4
- SUBSTRING keyword 2–23
- Subtransactions 17–23
- Summarizing, data 11–8
- Super procedures
defined 14–14
guidelines 14–18
order 14–16
session 14–30
stack execution 14–20
- Switching
indexes 6–32
- Symbols
formatting 2–11
- Syntax
checking 2–35
validating 2–16
- Syntax check 7–19
- Syntax completion
automatic 2–28
- Syntax errors 2–36
ambiguous field 6–18
- Syntax expansion option
- System failures
transactions 17–32
- System handles
ERROR-STATUS 17–28
window 18–44
- T**
- T startup parameter 11–3
- Trig startup parameter 16–35
- Tab key 12–57
- TABLE attribute
dynamic buffer handle 19–27
- TABLE form
dynamic temp-tables 20–17
- Table joins
selection criteria 21–18
- Table Properties dialog box 16–28
- Table Selector dialog box 4–5, 16–4
- Table Triggers dialog box 16–28
- TABLE-HANDLE buffer attribute
dynamic temp-tables 20–11
- TABLE-HANDLE form
dynamic temp-tables 20–19

-
- Tables
 - joining 2–4, 6–16
 - one-to-one join 19–8
 - Tables. *See also* records.
 - Tabs 1–14
 - Procedure Editor 2–29
 - TARGET-PROCEDURE function 14–23
 - TARGET-PROCEDURE handle 14–32
 - Template procedure files (.cst) 18–13
 - Temporary directory 11–3
 - Temporary filenames 13–16
 - Temporary tables. *See* temp-tables.
 - Temp-table Maintenance dialog box 16–4
 - TEMP-TABLE-PREPARE method
 - dynamic temp-tables 20–8
 - Temp-tables
 - buffers 11–8
 - copying as parameters 21–20
 - creating 11–4
 - defined 11–1
 - defining 16–3
 - handles 11–14, 20–2
 - indexes 11–6
 - NO-UNDO 21–2
 - passing between sessions 20–19
 - passing in a session 20–18
 - queries 11–20
 - scope 11–7
 - temp-tables. *See also* dynamic temp-tables.
 - Test procedures. *See* sample procedures.
 - Testing
 - run-time 5–16
 - Text
 - coloring 14–10
 - Text fields
 - adding 9–23
 - code 9–24
 - vs. fill-in fields 9–25
 - THIS-PROCEDURE function 13–6
 - Tilde (~) 1–14
 - TIME keyword 2–20
 - Titles
 - objects 4–7
 - TODAY keyword 2–20
 - Toggle boxes
 - adding 9–9
 - code 9–11
 - TOOLTIP attribute 8–11
 - ToolTips
 - defined 12–46
 - Totals
 - calculating 7–18
 - Transaction blocks 16–2
 - Transaction size
 - controlling 17–2
 - decreasing 17–6
 - increasing 17–4
 - Transaction statements 15–6
 - Transactions
 - defined 16–2
 - managing 17–1
 - scope 17–16
 - triggers 17–8
 - Translating strings 5–11
 - Trigger procedures

Triggers
accessing 16–28
browses 7–22
button 4–25
canceling 8–33
combo boxes 12–41
defined 4–18, 8–23
defining 5–11, 8–27
dynamic objects 18–5
editing 9–7
event-driven code 5–12
menus 9–33
multiple objects 19–20
persistent 8–31
preparing 8–27
transactions 17–8
writing 16–1

TRIM keyword 2–23

TRUNCATE keyword 2–27

TYPE attribute 9–4

U

U tag 5–11

UDF. *See* User-defined functions.

UNDO attribute
dynamic temp-tables 20–10

UNDO statement
syntax 17–11

Unique find 6–35

Unique row identifier 10–20

Unknown value 3–10
blank 2–10
using 2–16

Unknown value (?) 2–9

Unnamed widget pools 21–27
dynamic objects 18–16

Unprepared state
dynamic temp-tables 20–3

UPDATE statement 15–4

Updateable browses
creating 18–58

Updateable browses 12–5

Updating
browses 12–27

USE-INDEX phrase 6–22, 6–34

User interface events 8–24
defining 4–28

User manipulation events 12–50

User-defined functions 14–2
defined 4–18

V

Valid attributes
listing 18–25

Validating
syntax 2–16

Validation expressions
browse columns 20–28

VALID-HANDLE keyword 5–10

VALUE-CHANGED event 12–19

VALUE-CHANGED trigger
fill-in fields 9–10

VALUE-CHANGED triggers
browses 7–22
radio sets 9–17
toggle boxes 9–36

Values
list 7–28
variables 3–10

VAR abbreviation 5–4

-
- Variables
 - assigning values 3–10
 - defining 2–14
 - naming 2–13
 - program 2–9
 - qualifying 2–12
 - scoping 6–3
 - VIEW-AS** phrase 7–23, 8–4
 - DEFINE VARIABLE statement 8–5
 - VIEW-AS TEXT** attribute 9–25
 - Viewing
 - code 4–18, 4–20
 - sample procedures 5–2
 - windows 18–47
 - VISIBLE** attribute 8–10
 - Visualizing
 - fields 8–4
 - W**
 - .w extension 4–8
 - WAIT-FOR** statement 5–15
 - Warning messages
 - adding 19–23
 - creating 10–28
 - Weak-scoped
 - defined 7–6
 - Weak-scoped references
 - multiple 7–31
 - WEEKDAY** keyword 2–20
 - WHERE** clause 4–10
 - WIDGET** keyword
 - dynamic objects 18–10
 - Widget pools
 - defined 5–3, 18–12, 18–16
 - dynamic objects 18–12
 - guidelines 21–27
 - WIDGET-HANDLE** keyword 5–5
 - Widgets
 - defined 5–3
 - Widgets. *See also* objects
 - WIDTH** attributes 8–10
 - Wild cards
 - strings 2–3
 - WINDOW** attribute 8–13
 - Window families
 - defined 18–47
 - Window system handles 18–44
 - WINDOW-CLOSE** event 5–11, 18–47
 - Windows
 - adding buttons 4–24
 - adding fields 4–5
 - arranging objects 4–6
 - attributes 18–44
 - creating 4–4, 5–8
 - default for session 18–18
 - design 4–4
 - events 18–44
 - methods 18–44
 - minimizing 18–48
 - multiple 18–43
 - restoring 18–48
 - WITH FRAME** option 6–13
 - WITH FRAME** qualifier
 - DISPLAY** statement 4–22
 - WITH** keyword 5–6

Word index
defined 20–7, 21–14

Word-index keyword 2–3

Word-wrap option
editors 9–7

Work-tables 11–3

WRITE event 16–24

WRITE header 16–26

Writing
dynamic object attributes 18–21
efficient procedures 21–2

X

X format character 2–11

Y

YEAR keyword 2–20

Z

Z format character 2–11