

Progress Language Tutorial for Character

© 2001 Progress Software Corporation. All rights reserved.

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.

Progress, Progress Results, Provision and WebSpeed are registered trademarks of Progress Software Corporation in the United States and other countries. Apptivity, AppServer, ProVision Plus, SmartObjects, IntelliStream, and other Progress product names are trademarks of Progress Software Corporation.

SonicMQ is a trademark of Sonic Software Corporation in the United States and other countries.

Progress Software Corporation acknowledges the use of Raster Imaging Technology copyrighted by Snowbound Software 1993-1997 and the IBM XML Parser for Java Edition.

© IBM Corporation 1998-1999. All rights reserved. U.S. Government Users Restricted Rights — Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Progress is a registered trademark of Progress Software Corporation and is used by IBM Corporation in the mark Progress/400 under license. Progress/400 AND 400® are trademarks of IBM Corporation and are used by Progress Software Corporation under license.

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

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

.

May 2001



Product Code: 4530
Item Number: 81099U;9.1C

Contents

Preface . . . . . **xvii**
Purpose xvii
Audience xvii
Organization of This Manual xvii
How to Use This Manual xix
Typographical Conventions xx
Syntax Notation xxi
Example Procedures xxiv
Progress Messages xxvi
Other Useful Documentation xxviii
Getting Started xxviii
Development Tools xxix
Reporting Tools xxx
4GL xxxi
Database xxxii
DataServers xxxii
SQL-89/Open Access xxxii
SQL-92 xxxiii
Deployment xxxiv
WebSpeed xxxiv
Reference xxxv

1. Welcome to Progress **1–1**
1.1 Progress Solves Problems 1–2
1.1.1 A Classic Problem 1–2
1.2 The Progress Solution 1–4
1.3 About the Progress 4GL/RDBMS Product 1–5
1.4 About the Relational Database Management System 1–6
1.4.1 Database Basics 1–6

1.4.2 About the Data Dictionary 1–10
1.5 About the Application Development Language 1–10
1.5.1 About the Procedure Editor 1–12
1.6 Summary 1–12

2. Getting Started 2–1
2.1 Preparing to Use This Tutorial 2–2
2.2 Accessing the Language Tutorial Sample Procedures 2–2
2.3 Starting Progress 2–2
2.3.1 Windows NT 3.51 2–2
2.3.2 Windows 95 and Windows NT 4.0 2–2
2.3.3 UNIX 2–3
2.4 About the Procedure Editor 2–3
2.5 Using Object Interfaces 2–5
2.6 Using Key Functions and Menus 2–6
2.6.1 Basic Key Functions 2–6
2.6.2 Menus and Menu Options 2–8
2.7 Text Entry and Manipulation 2–13
2.7.1 Manipulating Blocks of Text 2–15
2.8 Edit Buffers and Procedure Files 2–17
2.9 Basic 4GL Constructs and Conventions 2–19
2.10 Using the Help Menu 2–20
2.10.1 Accessing the Help Menu 2–21
2.11 Leaving the Procedure Editor 2–21
2.12 Summary 2–22

3. Programming the Progress Way 3–1
3.1 The Progress Programming Model 3–2
3.1.1 User Interfaces 3–2
3.1.2 User Input 3–5
3.1.3 Responding to Events 3–8
3.1.4 Programming for Events 3–12
3.2 Widgets 3–14
3.2.1 Widget Categories 3–16
3.2.2 Defining Widgets 3–16
3.2.3 Widget Attributes 3–28
3.3 Event Types 3–30
3.4 Putting the Progress Programming Model to Work 3–33
3.5 Making Your Interface Responsive 3–41
3.5.1 Enabling and Disabling Widgets 3–41
3.5.2 Viewing and Hiding Widgets 3–44
3.5.3 Accessing Widget Attributes and Methods 3–49
3.6 Summary 3–53

4. Understanding the Database Environment 4–1
4.1 The Data Dictionary As a Programming Resource 4–2
4.2 Accessing the Data Dictionary 4–3
4.3 Exiting the Data Dictionary 4–3
4.4 Creating, Connecting, and Disconnecting Databases 4–4
4.4.1 Creating a Database 4–4
4.4.2 Connecting a Database 4–5
4.4.3 Disconnecting a Database 4–6
4.5 Using the Data Dictionary Interface 4–7
4.5.1 Using Menus 4–7
4.5.2 Working with Database Objects 4–8
4.6 Introduction to the sports Database Schema 4–9
4.7 Defining Database Objects 4–10
4.7.1 Defining Tables 4–10
4.7.2 Defining Fields 4–10
4.7.3 Defining Indexes 4–12
4.7.4 Defining Sequences 4–13
4.8 Establishing Application Defaults 4–15
4.8.1 Validation Expressions and Messages 4–15
4.8.2 Triggers and Database Events 4–17
4.8.3 Field Formats 4–18
4.8.4 VIEW-AS Phrases 4–20
4.8.5 Labels 4–21
4.9 Generating Reports 4–21
4.10 Summary 4–22

5. Working with Expressions 5–1
5.1 Understanding Expressions 5–2
5.2 Using Operands 5–2
5.2.1 Using Variables 5–3
5.2.2 Using Database Fields 5–8
5.2.3 Using Widget Attributes 5–9
5.2.4 Using Constants 5–10
5.3 Using Operators 5–10
5.4 Using Functions 5–14
5.4.1 Pre-defined Functions 5–14
5.4.2 User-defined Functions 5–15
5.5 Understanding Precedence in Expressions 5–15
5.6 Using Expressions 5–18
5.6.1 Calculating Values 5–18
5.6.2 Evaluating Conditions 5–19
5.6.3 Defining Selection Criteria 5–19
5.6.4 Manipulating Widget Attributes 5–20
5.7 Summary 5–25

6. Working with Control Structures 6–1
6.1 Modularizing Code with Procedures and Include Files 6–2
6.1.1 Using Procedures 6–2
6.1.2 Using User-defined Functions 6–6
6.1.3 Using Include Files 6–8
6.2 Sharing Information Between Procedures 6–10
6.2.1 Using Shared Variables 6–10
6.2.2 Using Parameters to Pass Values 6–11
6.2.3 Using Arguments to Pass Literals 6–17
6.3 About Blocks 6–21
6.3.1 Procedure Block 6–22
6.3.2 Trigger Blocks 6–23
6.4 Working with Control Blocks 6–24
6.4.1 Using the DO Block 6–24
6.4.2 Using the REPEAT Block 6–26
6.4.3 Using the FOR EACH Block 6–29
6.5 Block Property Summary 6–31
6.6 Conditional Processing 6–32
6.6.1 Using the IF Statement 6–32
6.6.2 Using the CASE Statement 6–33
6.7 Summary 6–37

7. Representing Data with Widgets 7–1
7.1 Programming Widgets 7–2
7.1.1 Using the VIEW-AS Phrase 7–3
7.1.2 Using the Format Phrase 7–4
7.1.3 Positioning Widgets 7–7
7.1.4 Using the Frame Phrase 7–8
7.1.5 Changing Widget Characteristics Programming Example 7–9
7.1.6 Referencing Widgets 7–12
7.1.7 Working with Widget Values 7–17
7.2 Working with Fill-in Fields 7–22
7.2.1 Fill-in Field Events 7–23
7.3 Working with Text Widgets 7–26
7.4 Working with Toggle Boxes 7–28
7.4.1 Toggle-box Events 7–29
7.4.2 Checked Attribute 7–29
7.4.3 Toggle Box Programming Example 7–30
7.5 Working with Radio Sets 7–32
7.5.1 Radio Set Events 7–33
7.5.2 Radio Set Programming Example 7–34
7.6 Working with Sliders 7–38
7.6.1 Slider Events 7–40
7.6.2 Slider Programming Example 7–41

7.7 Working with Selection Lists 7–44
7.7.1 Selection-list Events 7–47
7.7.2 LIST-ITEMS, NUM-ITEMS, and DELIMITER Attributes. 7–48
7.7.3 ADD-LAST( ) and LOOKUP( ) Methods. 7–48
7.7.4 Selection-list Programming Example 7–49
7.8 Working with Combo Boxes 7–52
7.8.1 Combo Box Events 7–55
7.8.2 Combo Box Programming Example 7–56
7.9 Working with Editors 7–58
7.9.1 Editor Events 7–60
7.9.2 Editor Programming Example 7–61
7.10 Summary 7–64

8. Using Database Records 8–1
8.1 How Progress Stores and Moves Data 8–2
8.2 Designing Displays for Interactive Database Access 8–4
8.3 Retrieving Data from a Database 8–7
8.3.1 Using the FIND and DISPLAY Statements 8–12
8.4 Using Defined Queries 8–18
8.5 Retrieving Data Through a Query with the GET Statement 8–20
8.6 Modifying Data 8–23
8.6.1 The DISPLAY, ENABLE, and ASSIGN Technique 8–24
8.6.2 The UPDATE Technique 8–32
8.7 Releasing Records 8–38
8.8 Creating Records 8–40
8.9 Deleting Records 8–44
8.10 Other Database Access Statements 8–49
8.10.1 The PROMPT–FOR Statement 8–49
8.10.2 The SET Statement 8–50
8.10.3 The INSERT Statement 8–51
8.10.4 Using the Browse Widget 8–52
8.11 Summary 8–65

9. Selecting, Sorting, and Relating Records 9–1
9.1 Uses of the Record Phrase 9–2
9.2 Selecting Records 9–2
9.2.1 Selecting with a WHERE Expression 9–4
9.2.2 Selecting with the USING Option. 9–5
9.2.3 Query and Selection Programming Example 9–6
9.3 Sorting Records 9–9
9.3.1 Sorting Records with the BY Phrase 9–9
9.3.2 Sorting Records with the USE-INDEX Option 9–12
9.3.3 Querying and Sorting Example 9–13
9.4 Relating Records 9–15

9.4.1 Understanding Table Relationships 9–16
9.4.2 Relating Record Options 9–18
9.4.3 Relating with OF 9–19
9.4.4 Relating with WHERE 9–22
9.4.5 Inner and Outer Table Joins 9–24
9.5 Summary 9–28

10. Creating Reports 10–1
10.1 Report Basics 10–2
10.1.1 Down Frames 10–2
10.1.2 Text Widgets 10–3
10.1.3 Control Blocks and Output Statements 10–4
10.1.4 Basic Report Demonstration 10–5
10.2 Designing an Interface for Viewing Report Data 10–7
10.3 Converting Widget Values to Report Data 10–11
10.3.1 Printing Reports and the STREAM-IO Option 10–11
10.3.2 Formatting Long Text Strings 10–12
10.4 Generating Reports with Control Breaks and Aggregates 10–15
10.5 Generating Reports from Multiple Tables 10–19
10.5.1 Reporting Information from One Table 10–19
10.5.2 Reporting Information from Two Tables 10–20
10.5.3 Reporting Information from Multiple Tables 10–22
10.6 Redirecting Output 10–24
10.6.1 OUTPUT TO and the Default Stream 10–24
10.6.2 Directing Output to a File 10–26
10.6.3 Directing Output to a Printer 10–27
10.6.4 Directing Output to Multiple Destinations 10–28
10.7 Designing Frames for Reports 10–29
10.7.1 Using Base Fields with the @ Option 10–29
10.7.2 Using the HEADER of a Frame for Running Page Heads 10–30
10.7.3 Using the HEADER of a Frame for Running Page Footers . . . 10–32
10.7.4 Programming Example 10–33
10.8 Using the PUT Statement 10–36
10.8.1 Using PUT for Printer Control 10–41
10.9 Summary 10–44

11. Building Menus 11–1
11.1 Menu Basics 11–2
11.2 Defining a Menu 11–3
11.2.1 Defining a Submenu 11–4
11.2.2 Defining a Menu Bar 11–6
11.2.3 Assigning a Menu Bar to a Window 11–8
11.2.4 Assigning Triggers to Menu Items 11–8
11.2.5 Menu Bar Example 11–10

11.3 Using Optional Menu Features 11–14
11.3.1 Disabled Menu Items 11–14
11.3.2 Toggle Boxes 11–15
11.3.3 Mnemonics 11–15
11.3.4 Accelerators 11–16
11.3.5 Programming Example 11–17
11.4 Menu Attributes 11–19
11.5 Design Conventions 11–20
11.6 Summary 11–21

12. Managing Your Application 12–1
12.1 Managing Procedures 12–2
12.1.1 Structuring an Event-driven Application 12–2
12.1.2 Using Persistent Procedures 12–3
12.2 Sharing Resources 12–4
12.2.1 Using SHARED Resources 12–4
12.2.2 Using Input and Output Parameters 12–5
12.2.3 Sharing with Persistent Procedures 12–5
12.3 Managing Interfaces 12–10
12.3.1 Using Single-frame Interfaces 12–11
12.3.2 Using Multi-frame Interfaces 12–11
12.3.3 Using Dialog Boxes 12–16
12.3.4 Using Alert Boxes 12–19
12.3.5 Using Multi-window Interfaces 12–22
12.4 Managing Application Control 12–27
12.4.1 Using Basic Control Mechanisms 12–27
12.4.2 Using Dialog Boxes with WAIT–FOR Statements 12–28
12.4.3 Using Windows with Persistent Procedures 12–28
12.5 Summary 12–29

13. Where to Go from Here 13–1
13.1 Progress Tools 13–2
13.2 Designing Databases 13–3
13.3 Learning the Language 13–3
13.4 Multi-user Programming 13–3
13.5 Data Integrity 13–4
13.6 Record Reading 13–4
13.7 Working with Large Data Sets 13–4
13.8 Multiple-window Applications 13–5
13.9 Menus 13–5
13.10 Dynamic Widgets and Direct Manipulation 13–5
13.11 Graphical Features 13–6
13.12 Internationalization and Localization 13–7

Glossary **Glossary–1**

Index **Index–1**

Figures

Figure 1–1: Paper-based Ordering Process 1–3
Figure 1–2: Components of the Progress 4GL/RDBMS Product 1–5
Figure 1–3: Relationship of Paper and Electronic Filing Systems 1–7
Figure 1–4: Structure of a Database 1–8
Figure 1–5: Simple and Compound Indexes 1–9
Figure 2–1: Procedure Editor Main Display 2–4
Figure 2–2: Input Focus and Navigation on a Screen 2–8
Figure 2–3: File Pull-down Menu 2–10
Figure 3–1: Basic Character Interface 3–3
Figure 3–2: Basic Graphical User Interface 3–4
Figure 3–3: How an Application Responds to Events 3–13
Figure 3–4: A Simple Interface 3–34
Figure 4–1: The Data Dictionary Main Window 4–3
Figure 4–2: Modify Table Dialog Box 4–10
Figure 4–3: Field Editor 4–11
Figure 4–4: Index Editor 4–12
Figure 4–5: Sequence Editor 4–14
Figure 4–6: Table Validation Dialog Box 4–16
Figure 4–7: Field Validation Dialog Box 4–16
Figure 4–8: Table Triggers Dialog Box 4–18
Figure 4–9: VIEW-AS Phrase Dialog Box 4–20
Figure 6–1: Main Procedure and Subprocedures 6–3
Figure 6–2: Main Procedure and Internal Procedures 6–5
Figure 6–3: Defining and Using a User-defined Function 6–7
Figure 6–4: Using an Include File 6–9
Figure 6–5: Shared Variables 6–11
Figure 7–1: Format Phrase Positioning Options 7–7
Figure 7–2: Parts of a Fill-in Field 7–22
Figure 7–3: Parts of a Text Widget 7–26
Figure 7–4: Parts of a Toggle Box 7–28
Figure 7–5: Parts of a Radio Set 7–32
Figure 7–6: Parts of a Slider 7–38
Figure 7–7: Parts of a Selection List 7–44
Figure 7–8: Parts of a Combo Box 7–53
Figure 7–9: Parts of an Editor 7–58
Figure 8–1: Data Locations 8–2
Figure 8–2: Data Movement in Progress 8–3
Figure 8–3: Basic Database Access Form 8–4
Figure 8–4: How a Database Access Form Works 8–6
Figure 8–5: Data Form with Navigation Buttons Enabled 8–12
Figure 8–6: Data Movement with the FIND Statement 8–13
Figure 8–7: Data Movement with the DISPLAY Statement 8–14
Figure 8–8: Data Movement with the GET Statement 8–20

Figure 8–9: Database Access Form with Update Enabled 8–24
Figure 8–10: Data Movement with the ENABLE Statement 8–26
Figure 8–11: Data Movement with the ASSIGN Statement 8–27
Figure 8–12: Data Movement with the UPDATE Statement 8–33
Figure 8–13: Data Movement with the RELEASE Statement 8–39
Figure 8–14: Data Movement with the CREATE Statement 8–40
Figure 8–15: Data Movement with the DELETE Statement 8–44
Figure 8–16: Data Movement with the PROMPT–FOR Statement 8–49
Figure 8–17: Data Movement with the SET Statement 8–50
Figure 8–18: Data Movement with the INSERT Statement 8–51
Figure 8–19: Parts of a Read-only Browse Widget 8–52
Figure 8–20: Parts of an Updatable Browse Widget 8–53
Figure 8–21: Database Access Form with a Browse Widget 8–54
Figure 9–1: Table Relations 9–16
Figure 9–2: One-to-one Relationship 9–17
Figure 9–3: One-to-many Relationship 9–17
Figure 9–4: Inner Join 9–26
Figure 9–5: Left Outer Join 9–27
Figure 11–1: Example of a Window with a Menu Bar 11–3
Figure 12–1: Flat Structure of an Event-driven Application 12–2
Figure 12–2: A Modeless Interface 12–17
Figure 12–3: A Modal Interface 12–18
Figure 12–4: QUESTION Alert Box 12–21

Tables

Table 2–1: Basic Key Functions in the Procedure Editor 2–7
Table 2–2: Procedure Editor Menus 2–8
Table 2–3: Progress Menu Symbols 2–12
Table 2–4: Basic Editing Keys 2–13
Table 2–5: Key Functions to Define Text-block Operations 2–16
Table 2–6: Buffer Tasks and Associated Menu Options 2–17
Table 2–7: Procedure Editor Help Menu 2–20
Table 3–1: Widget Types 3–14
Table 3–2: Commonly Used Attributes 3–29
Table 3–3: Important Event Functions 3–32
Table 3–4: Event Categories 3–32
Table 4–1: Data Dictionary Menus 4–7
Table 4–2: Tables in the sports Database 4–9
Table 4–3: Data Types 4–11
Table 4–4: Index Properties 4–13
Table 4–5: Sequence Properties 4–14
Table 4–6: Database Events 4–17
Table 4–7: Field Format Symbols 4–19
Table 4–8: Reports Menu Options 4–21
Table 5–1: Syntax Components of DEFINE VARIABLE 5–3
Table 5–2: Constant Data Types 5–10
Table 5–3: Numeric Operators 5–11
Table 5–4: Comparison Operators 5–12
Table 5–5: Date Operators 5–13
Table 5–6: Character Operator 5–13
Table 5–7: Logical Operators 5–14
Table 5–8: Operator Precedence 5–17
Table 6–1: Elements of the FUNCTION Statement 6–6
Table 6–2: Block Properties 6–31
Table 7–1: Ways to Define or Modify a Data Widget 7–2
Table 7–2: Format Phrase Options 7–5
Table 7–3: Frame Phrase Options 7–8
Table 7–4: Widget System Handles 7–14
Table 7–5: Radio Set Syntax 7–33
Table 7–6: Slider Syntax 7–39
Table 7–7: Selection List Syntax 7–45
Table 7–8: Selection List Attributes 7–48
Table 7–9: Selection List Methods 7–48
Table 7–10: Combo Box Syntax 7–54
Table 7–11: Editor Syntax 7–59
Table 8–1: DISPLAY Statement Syntax 8–15
Table 8–2: Locks 8–25
Table 8–3: DEFINE BROWSE Statement Syntax 8–55

Table 8–4:	Browse Control Keys	8–57
Table 9–1:	Record Selection Options	9–3
Table 9–2:	Record Sorting Options	9–9
Table 9–3:	Record Relation Options	9–19
Table 10–1:	Aggregate Phrase Options	10–15
Table 10–2:	OUTPUT Statement Options	10–25
Table 11–1:	MENU Element Options	11–4
Table 11–2:	MENU–ITEM Options	11–5
Table 11–3:	MENU Options	11–7
Table 11–4:	Menu Attributes	11–19
Table 11–5:	Submenu Attributes	11–19
Table 11–6:	Menu Item Attributes	11–20
Table 12–1:	Language Elements Used with Persistent Procedures	12–6
Table 12–2:	Frame Attributes	12–14
Table 12–3:	Alert Box Options	12–20

Procedures

lt-03-01.p 3–7
lt-03-02.p 3–10
lt-03-03.p 3–19
lt-03-04.p 3–21
lt-03-05.p 3–23
lt-03-06.p 3–25
lt-03-07.p 3–39
lt-03-08.p 3–43
lt-03-09.p 3–48
lt-03-10.p 3–51
lt-05-01.p 5–7
lt-05-02.p 5–22
lt-06-01.p 6–16
lt-06-02.p 6–16
lt-06-03.p 6–16
lt-06-04.p 6–28
lt-06-05.p 6–35
lt-07-01.p 7–11
lt-07-02.p 7–16
lt-07-03.p 7–21
lt-07-04.p 7–25
lt-07-05.p 7–31
lt-07-06.p 7–35
lt-07-07a.p 7–42
lt-07-08.p 7–51
lt-07-09.p 7–57
lt-07-10.p 7–62
lt-08-08.p 8–9
lt-08-f1.i 8–11
lt-08-01.p 8–17
lt-08-02.p 8–22
lt-08-03.p 8–29
lt-08-04.p 8–37
lt-08-05.p 8–43
lt-08-06.p 8–47
lt-08-07a.p 8–62
lt-08-07b.p 8–63
lt-09-01.p 9–7
lt-09-02.p 9–12
lt-09-03.p 9–14
lt-09-04.p 9–22
lt-09-05.p 9–24
lt-10-01.p 10–6

lt-10-02.p 10–9
lt-10-04.p 10–14
lt-10-05.p 10–18
lt-10-06.p 10–19
lt-10-07.p 10–21
lt-10-08.p 10–22
lt-10-09.p 10–26
lt-10-10.p 10–34
lt-10-11.p 10–38
lt-10-12.p 10–39
lt-10-13.p 10–42
lt-11-mn.i 11–11
lt-11-01.p 11–13
lt-11-02.p 11–18
lt-12-01.p 12–9
lt-12-04.p 12–26

Preface

Purpose

This book introduces the basic concepts of the Progress 4GL and programming model and provides opportunities for hands-on experience designing applications and user interfaces.

Audience

This book is intended for the person with some programming experience who wants to start developing applications in Progress.

Organization of This Manual

Chapter 1, "Welcome to Progress"

Introduces the Progress relational database management system and the application development language.

Chapter 2, "Getting Started"

Explains how to start and exit Progress, how to use the Procedure Editor, and how to access the help system and online example procedures described in this manual.

Chapter 3, "Programming the Progress Way"

Describes the Progress programming model and the basic syntax and structure of procedures.

Chapter 4, “Understanding the Database Environment”

Explains how to use the Data Dictionary tool to view the properties of database tables, fields, indexes, and sequences.

Chapter 5, “Working with Expressions”

Introduces the Progress conventions governing variables, data types, expressions, and operators.

Chapter 6, “Working with Control Structures”

Describes the structures for creating modularized code and controlling an application.

Chapter 7, “Representing Data with Widgets”

Explains the syntax, events, attributes, and programming techniques used with widgets.

Chapter 8, “Using Database Records”

Describes how to access, update, create, and delete database records.

Chapter 9, “Selecting, Sorting, and Relating Records”

Describes many important techniques used to define database queries.

Chapter 10, “Creating Reports”

Describes how to generate and format simple and complex reports.

Chapter 11, “Building Menus”

Explains how to integrate menu bars, submenus, and menu items into your application.

Chapter 12, “Managing Your Application”

Covers programming issues that arise with large-scale applications.

Chapter 13, “Where to Go from Here”

Suggests how to expand the development skills you acquired from this tutorial.

“Glossary”

Defines the terms and concepts the tutorial introduces.

How to Use This Manual

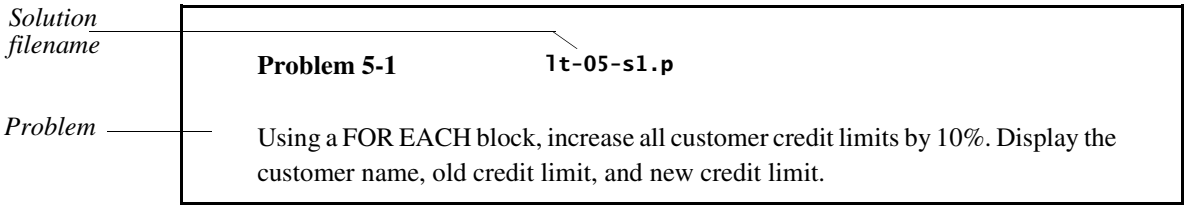
To help provide you with concrete examples, the tutorial follows the application development of All Around Sports, a fictitious business. In fact, you'll assume the role of All Around Sports's application developer. You won't have to start from scratch, however. All Around Sports has already hired a database developer, so you start with a complete sales and inventory database.

The sales and inventory database for All Around Sports is the sports database that comes with Progress. It contains data that you'll use as you work through the examples in the tutorial.

This tutorial uses two methods for making sure that you get plenty of hands-on experience. First, as the text of the tutorial moves along, you'll see plenty of code. The code included in the tutorial is also available on-line. Next to each printed code sample is the name of the file that contains the same code. The best way to use the tutorial is to open the file as you read about it. Running the code yourself and experimenting is the best way to become familiar with Progress quickly.

The second hands-on method is problem solving. From time to time, you'll see *practice problems* in the tutorial. These problems encourage you to try different solutions by modifying code or by creating your own code. Tackling the practice problems gives you the confidence that you've mastered the preceding techniques and concepts.

When a practice problem has a sample solution, the solution's filename is provided. For example, this a problem from Chapter 5, "Working with Expressions."



You can access the sample solutions on-line. Refer to the "Example Procedures" section later in this Preface for the specific details to access this information.

The best advice for success is to set a pace for yourself that makes the tutorial fun and gives you plenty of time to practice what you learn.

Typographical Conventions

This manual uses the following typographical conventions:

- **Bold typeface** indicates:
  - Commands or characters that the user types
  - That a word carries particular weight or emphasis
- *Italic typeface* indicates:
  - Progress variable information that the user supplies
  - New terms
  - Titles of complete publications
- `Monospaced typeface` indicates:
  - Code examples
  - System output
  - Operating system filenames and pathnames

The following typographical conventions are used to represent keystrokes:

- Small capitals are used for Progress key functions and generic keyboard keys.

  END-ERROR, GET, GO
  ALT, CTRL, SPACEBAR, TAB

- When you have to press a combination of keys, they are joined by a dash. You press and hold down the first key, then press the second key.

  CTRL–X

- When you have to press and release one key, then press another key, the key names are separated with a space.

  ESCAPE H
  ESCAPE CURSOR-LEFT

Syntax Notation

The syntax for each component follows a set of conventions:

- Uppercase words are keywords. Although they are always shown in uppercase, you can use either uppercase or lowercase when using them in a procedure.

  In this example, ACCUM is a keyword:

  SYNTAX

  ```
  ACCUM aggregate expression
  ```

- Italics identify options or arguments that you must supply. These options can be defined as part of the syntax or in a separate syntax identified by the name in italics. In the ACCUM function above, the *aggregate* and *expression* options are defined with the syntax for the ACCUM function in the *Progress Language Reference*.

- You must end all statements (except for DO, FOR, FUNCTION, PROCEDURE, and REPEAT) with a period. DO, FOR, FUNCTION, PROCEDURE, and REPEAT statements can end with either a period or a colon, as in this example:

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

- Square brackets (**[ ]**) around an item indicate that the item, or a choice of one of the enclosed items, is optional.

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

  SYNTAX

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

  In some instances, square brackets are not a syntax notation, but part of the language.

For example, this syntax for the INITIAL option uses brackets to bound an initial value list for an array variable definition. In these cases, normal text brackets ([ ]) are used:

SYNTAX

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

NOTE: The ellipsis (**...**) indicates repetition, as shown in a following description.

- Braces (**{ }**) around an item indicate that the item, or a choice of one of the enclosed items, is required.

  In this example, you must specify the items BY and *expression* and can optionally specify the item DESCENDING, in that order:

  SYNTAX

  ```
  { BY expression [ DESCENDING ] }
  ```

  In some cases, braces are not a syntax notation, but part of the language.

  For example, a called external procedure must use braces when referencing arguments passed by a calling procedure. In these cases, normal text braces ({ }) are used:

  SYNTAX

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

- A vertical bar (**|**) indicates a choice.

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

  SYNTAX

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

In this example, you must select one of *logical-name* or *alias*:

SYNTAX

```
CONNECTED ( { logical-name | alias } )
```

- Ellipses (**. . .**) indicate that you can choose one or more of the preceding items. If a group of items is enclosed in braces and followed by ellipses, you must choose one or more of those items. If a group of items is enclosed in brackets and followed by ellipses, you can optionally choose one or more of those items.

  In this example, you must include two expressions, but you can optionally include more. Note that each subsequent expression must be preceded by a comma:

  SYNTAX

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

  In this example, you must specify MESSAGE, then at least one of *expression* or SKIP, but any additional number of *expression* or SKIP is allowed:

  SYNTAX

  ```
  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 }:

  SYNTAX

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

- In some examples, the syntax is too long to place in one horizontal row. In such cases, **optional** items appear individually bracketed in multiple rows in order, left-to-right and top-to-bottom. This order generally applies, unless otherwise specified. **Required** items also appear on multiple rows in the required order, left-to-right and top-to-bottom. In cases where grouping and order might otherwise be ambiguous, braced (required) or bracketed (optional) groups clarify the groupings.

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

SYNTAX

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

In this example, ASSIGN requires one of two choices: either one or more of *field*, or one of *record*. Other options available with either *field* or *record* are grouped with braces and brackets. The open and close braces indicate the required order of options:

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 they are available online, the name of the procedure appears above the left corner of the box and starts with a prefix associated with the manual that references it, as follows:
  - `e-` — *Progress External Program Interfaces*, for example, `e-ddeex1.p`
  - `lt-` — *Progress Language Tutorial*, for example, `lt-05-s3.p`
  - `p-` — *Progress Programming Handbook*, for example, `p-br01.p`
  - `r-` — *Progress Language Reference*, for example, `r-dynbut.p`

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

- If they are not available online, they compile as shown, but might not execute for lack of completeness.

Accessing Files in Procedure Libraries

Documentation examples are stored in procedure libraries, `prodoc.pl` and `prohelp.pl`, in the `src` directory where Progress is installed.

You must first create all subdirectories required by a library before attempting 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 the *Progress Client Deployment Guide* for more details on the PROLIB utility.

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

Extracting Source Files from Procedure Libraries on UNIX Platforms

To extract `p-wrk1.p` from `prodoc.pl`, a procedure library, follow these steps at the UNIX system prompt:

1 ♦ Run the PROENV utility:

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

Running proenv sets the DLC environment variable to the directory where you installed Progress (by default, `/usr/dlc`). The proenv utility also adds the DLC environment variable to your PATH environment variable and adds the bin directory (`PATH=%DLC%;%DLC%\bin;%PATH%`).

2 ♦ At the proenv prompt, create the `prodoc` directory in your Progress working directory:

```
mkdir prodoc
```

3 ♦ Create the `proghand` directory under `prodoc`:

```
mkdir prodoc/proghand
```

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

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

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/proghand/p-wrk-1.p
```

PROLIB extracts `p-wrk-1.p` into `prodoc/proghand`.

Progress Messages

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

- Execution messages inform you of errors encountered while Progress is running a procedure (for example, if Progress cannot find a record with a specified index field value).
- Compile messages inform you of errors found while Progress is reading and analyzing a procedure prior to 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 Progress is getting ready to execute (for example, if you entered an invalid startup parameter).

After displaying a message, Progress 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 following execution messages.
- Returns to the Progress Procedure Editor so that you can correct an error in a procedure. This is the usual action taken following compiler messages.
- Halts processing of a procedure and returns immediately to the Procedure Editor. This does not happen often.
- Terminates the current session.

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

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

You can use the Progress PRO command to start a single-user mode character Progress client session and view a brief description of a message by providing its number. Follow these steps:

1 ♦ Start the Progress 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.

Other Useful Documentation

This section lists Progress Software Corporation documentation that you might find useful. Unless otherwise specified, these manuals support both Windows and Character platforms and are provided in electronic documentation format on CD-ROM.

Getting Started

Progress Electronic Documentation Installation and Configuration Guide (Hard copy only)

A booklet that describes how to install the Progress EDOC viewer and collection on UNIX and Windows.

Progress Installation and Configuration Guide Version 9 for UNIX

A manual that describes how to install and set up Progress Version 9.1 for the UNIX operating system.

Progress Installation and Configuration Guide Version 9 for Windows

A manual that describes how to install and set up Progress Version 9.1 for all supported Windows and Citrix MetaFrame operating systems.

Progress Version 9 Product Update Bulletin

A guide that provides a brief description of each new feature of the release. The booklet also explains where to find more detailed information in the documentation set about each new feature.

Progress Application Development Environment — Getting Started (Windows only)

A practical guide to graphical application development within the Progress Application Development Environment (ADE). This guide includes an overview of the ADE and its tools, an overview of Progress SmartObject technology, and tutorials and exercises that help you better understand SmartObject technology and how to use the ADE to develop applications.

Progress Master Glossary for Windows and *Progress Master Glossary for Character* (EDOC only)

Platform-specific master glossaries for the Progress documentation set. These books are in electronic format only.

Progress Master Index and Glossary for Windows and *Progress Master Index and Glossary for Character* (Hard copy only)

Platform-specific master indexes and glossaries for the Progress hard-copy documentation set.

Progress Startup Command and Parameter Reference

A reference manual that describes the Progress startup commands and parameters in alphabetical order.

Welcome to Progress (Hard copy only)

A booklet that explains how Progress software and media are packaged. An icon-based map groups the documentation by functionality, providing an overall view of the documentation set. *Welcome to Progress* also provides descriptions of the various services Progress Software Corporation offers.

Development Tools

Progress ADM 2 Guide

A guide to using the Application Development Model, Version 2 (ADM 2) application architecture to develop Progress applications. It includes instructions for building and using Progress SmartObjects.

Progress ADM 2 Reference

A reference for the Application Development Model, Version 2 (ADM 2) application. It includes descriptions of ADM 2 functions and procedures.

Progress AppBuilder Developer's Guide (Windows only)

A programmer's guide to using the Progress AppBuilder visual layout editor. AppBuilder is a Rapid Application Development (RAD) tool that can significantly reduce the time and effort required to create Progress applications.

Progress Basic Database Tools (Character only; information for Windows is in online help)

A guide for the Progress Database Administration tools, such as the Data Dictionary.

Progress Basic Development Tools (Character only; information for Windows is in online help)

A guide for the Progress development toolset, including the Progress Procedure Editor and the Application Compiler.

Progress Debugger Guide

A guide for the Progress Application Debugger. The Debugger helps you trace and correct programming errors by allowing you to monitor and modify procedure execution as it happens.

Progress Help Development Guide (Windows only)

A guide that describes how to develop and integrate an online help system for a Progress application.

Progress Translation Manager Guide (Windows only)

A guide that describes how to use the Progress Translation Manager tool to manage the entire process of translating the text phrases in Progress applications.

Progress Visual Translator Guide (Windows only)

A guide that describes how to use the Progress Visual Translator tool to translate text phrases from procedures into one or more spoken languages.

Reporting Tools

Progress Report Builder Deployment Guide (Windows only)

An administration and development guide for generating Report Builder reports using the Progress Report Engine.

Progress Report Builder Tutorial (Windows only)

A tutorial that provides step-by-step instructions for creating eight sample Report Builder reports.

Progress Report Builder User's Guide (Windows only)

A guide for generating reports with the Progress Report Builder.

Progress Results Administration and Development Guide (Windows only)

A guide for system administrators that describes how to set up and maintain the Results product in a graphical environment. This guide also describes how to program, customize, and package Results with your own products. In addition, it describes how to convert character-based Results applications to graphical Results applications.

Progress Results User's Guide for Windows and *Progress Results User's Guide for UNIX*

Platform-specific guides for users with little or no programming experience that explain how to query, report, and update information with Results. Each guide also helps advanced users and application developers customize and integrate Results into their own applications.

4GL

Building Distributed Applications Using the Progress AppServer

A guide that provides comprehensive information about building and implementing distributed applications using the Progress AppServer. Topics include basic product information and terminology, design options and issues, setup and maintenance considerations, 4GL programming details, and remote debugging.

Progress External Program Interfaces

A guide to accessing non-Progress applications from Progress. This guide describes how to use system clipboards, UNIX named pipes, Windows dynamic link libraries, Windows dynamic data exchange, Windows ActiveX controls, and the Progress Host Language Call Interface to communicate with non-Progress applications and extend Progress functionality.

Progress Internationalization Guide

A guide to developing Progress applications for markets worldwide. The guide covers both internationalization—writing an application so that it adapts readily to different locales (languages, cultures, or regions)—and localization—adapting an application to different locales.

Progress Language Reference

A three-volume reference set that contains extensive descriptions and examples for each statement, phrase, function, operator, widget, attribute, method, and event in the Progress language.

Progress Programming Handbook

A two-volume handbook that details advanced Progress programming techniques.

Database

Progress Database Design Guide

A guide that uses a sample database and the Progress Data Dictionary to illustrate the fundamental principles of relational database design. Topics include relationships, normalization, indexing, and database triggers.

Progress Database Administration Guide and Reference

This guide describes Progress database administration concepts and procedures. The procedures allow you to create and maintain your Progress databases and manage their performance.

DataServers

Progress DataServer Guides

These guides describe how to use the DataServers to access non-Progress databases. They provide instructions for building the DataServer modules, a discussion of programming considerations, and a tutorial. Each DataServer has its own guide, for example, the *Progress DataServer for ODBC Guide*, the *Progress DataServer for ORACLE Guide*, or the *Progress/400 Product Guide*.

MERANT ODBC Branded Driver Reference

The Enterprise DataServer for ODBC includes MERANT ODBC drivers for all the supported data sources. For configuration information, see the MERANT documentation, which is available as a PDF file in *installation-path*\odbc. To read this file you must have the Adobe Acrobat Reader Version 3.1 or higher installed on your system. If you do not have the Adobe Acrobat Reader, you can download it from the Adobe Web site at: http://www.adobe.com/prodindex/acrobat/readstep.html.

SQL-89/Open Access

Progress Embedded SQL-89 Guide and Reference

A guide to Progress Embedded SQL-89 for C, including step-by-step instructions on building ESQL-89 applications and reference information on all Embedded SQL-89 Preprocessor statements and supporting function calls. This guide also describes the relationship between ESQL-89 and the ANSI standards upon which it is based.

Progress Open Client Developer's Guide

A guide that describes how to write and deploy Java and ActiveX applications that run as clients of the Progress AppServer. The guide includes information about how to expose the AppServer as a set of Java classes or as an ActiveX server.

Progress SQL-89 Guide and Reference

A user guide and reference for programmers who use interactive Progress/SQL-89. It includes information on all supported SQL-89 statements, SQL-89 Data Manipulation Language components, SQL-89 Data Definition Language components, and supported Progress functions.

SQL-92

Progress Embedded SQL-92 Guide and Reference

A guide to Progress Embedded SQL-92 for C, including step-by-step instructions for building ESQL-92 applications and reference information about all Embedded SQL-92 Preprocessor statements and supporting function calls. This guide also describes the relationship between ESQL-92 and the ANSI standards upon which it is based.

Progress JDBC Driver Guide

A guide to the Java Database Connectivity (JDBC) interface and the Progress SQL-92 JDBC driver. It describes how to set up and use the driver and details the driver's support for the JDBC interface.

Progress ODBC Driver Guide

A guide to the ODBC interface and the Progress SQL-92 ODBC driver. It describes how to set up and use the driver and details the driver's support for the ODBC interface.

Progress SQL-92 Guide and Reference

A user guide and reference for programmers who use Progress SQL-92. It includes information on all supported SQL-92 statements, SQL-92 Data Manipulation Language components, SQL-92 Data Definition Language components, and Progress functions. The guide describes how to use the Progress SQL-92 Java classes and how to create and use Java stored procedures and triggers.

Deployment

Progress Client Deployment Guide

A guide that describes the client deployment process and application administration concepts and procedures.

Progress Developer's Toolkit

A guide to using the Developer's Toolkit. This guide describes the advantages and disadvantages of different strategies for deploying Progress applications and explains how you can use the Toolkit to deploy applications with your selected strategy.

Progress Portability Guide

A guide that explains how to use the Progress toolset to build applications that are portable across all supported operating systems, user interfaces, and databases, following the Progress programming model.

WebSpeed

Getting Started with WebSpeed

Provides an introduction to the WebSpeed Workshop tools for creating Web applications. It introduces you to all the components of the WebSpeed Workshop and takes you through the process of creating your own Intranet application.

WebSpeed Installation and Configuration Guide

Provides instructions for installing WebSpeed on Windows and UNIX systems. It also discusses designing WebSpeed environments, configuring WebSpeed Brokers, WebSpeed Agents, and the NameServer, and connecting to a variety of data sources.

WebSpeed Developer's Guide

Provides a complete overview of WebSpeed and the guidance necessary to develop and deploy WebSpeed applications on the Web.

WebSpeed Version 3 Product Update Bulletin

A booklet that provides a brief description of each new feature of the release. The booklet also explains where to find more detailed information in the documentation set about each new feature.

Welcome to WebSpeed! (Hard copy only)

A booklet that explains how WebSpeed software and media are packaged. *Welcome to WebSpeed!* also provides descriptions of the various services Progress Software Corporation offers.

Reference

Pocket Progress (Hard copy only)

A reference that lets you quickly look up information about the Progress language or programming environment.

Pocket WebSpeed (Hard copy only)

A reference that lets you quickly look up information about the SpeedScript language or the WebSpeed programming environment.

1

Welcome to Progress

Welcome, and congratulations for choosing Progress as your development product for database applications. Progress has a rich array of features and options, all contributing to an application development environment that allows you to deliver complete database applications quickly and easily. This tutorial couldn't possibly cover all the features of Progress, but it does cover those features that you need to begin coding right away.

The tutorial focuses on teaching you the Progress application development language. You'll use the demonstration database that comes with Progress to jump past database development and into code development. You're ready right now to discover why programming in the Progress 4GL is the best way to turn your ideas into robust applications.

Before you begin the tutorial, make sure that your hardware is set up correctly and that Progress is installed and ready to go. For complete installation and setup instructions, see the *Progress Installation and Configuration Guide Version 9 for Windows* or *Progress Installation and Configuration Guide Version 9 for UNIX*.

This chapter introduces you to some fundamental concepts and terminology. After you read this chapter, you'll be ready to begin working with the software in Chapter 2, "Getting Started."

This contains information on:

- Common business problems and how Progress can solve them
- The relational database management system
- The application development language

1.1 Progress Solves Problems

To help you gain experience solving business problems with Progress, the tutorial frequently checks in with All Around Sports, a small sporting goods supplier. All Around Sports has the problems typical of a small company that's growing quickly paper-based systems that were once effective now take too much time to keep current. All Around Sports wants to make the leap from paper-based business systems to computerized business systems.

1.1.1 A Classic Problem

Here's what happens during a typical sales order, using All Around Sports's paper-based sales and inventory system:

1. Second Skin Scuba calls All Around Sports to place orders for its annual "Deep Savings Sale." Before the sales representative (sales rep) can take the order, she needs to retrieve the file on Second Skin Scuba from the customer filing cabinets. Once she finds the folder, she copies the billing and shipping addresses from the folder onto an order form.

2. The sales rep writes down the name, quantity, and price of each item that Second Skin Scuba wants. Referring to a typed list, the sales rep quotes prices and calculates the grand total. After hanging up, the sales rep copies all the information neatly onto the order from.

3. The sales rep now needs to walk to the inventory filing cabinet, make sure the items are in stock, and subtract the newly sold items from the inventory sheets. Finally, the sales rep stores the order form where the Finance and Shipping departments can make copies.

Figure 1–1 shows the All Around Sports paper-based ordering process.

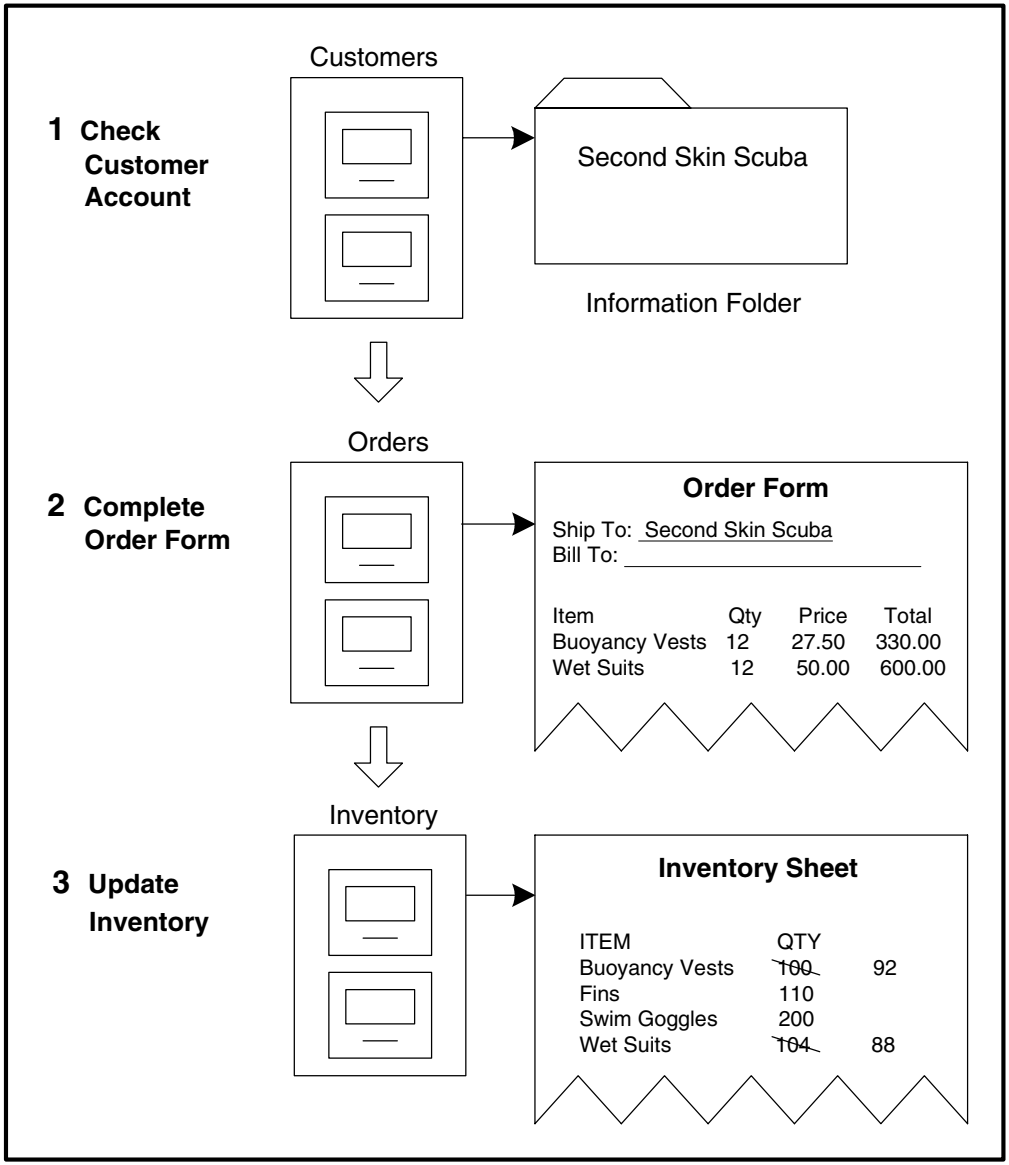


Figure 1–1: Paper-based Ordering Process

Although the sales rep carried out her job correctly and in quick order, she has a surprise waiting for her when she returns to her desk—two more customers called while she was completing the paperwork on the Second Skin Scuba order.

Both management and the sales reps agree that the sales reps need to spend more time with their customers and less time on paperwork. Management decides that the time is right to bring computer professionals into All Around Sports. Management envisions a system where the sales reps can use computer terminals on their desks to take orders and check inventory while they are still on the phone with their customers.

1.2 The Progress Solution

What's happening at All Around Sports is the kind of classic business problem that motivates an enterprise to computerize its business systems. When the computer professionals come to All Around Sports, they give the management team one strong recommendation: use Progress. As the computer professionals explain, successfully making the transition from paper systems to computer systems takes two steps.

1. Move the business information from paper to an electronic form that can be stored in a computer. This step involves:

   a. Gathering together the relevant business information, called *data*, and analyzing the relationships among the data. At All Around Sports, the relevant data is in the customer, order, and inventory filing cabinets.

   b. Designing and creating an electronic storage structure for the data. The electronic storage structure is called a *database*. A database that allows users to organize data so that it's easy to define relationships is called a *relational database.*

   Progress provides the tools that let you easily design, create, and maintain relational databases.

2. Program the computer to handle business tasks. This step involves:

   a. Breaking each business task into Progress programming language statements.

   b. Storing related statements in *procedures*. A *procedure* is a collection of Progress language statements that execute a specific task. The procedures that you create can work together to provide your end users with one tool to do many related tasks.

   c. Creating computer programs that run with an overall look and feel and are easy to learn and use.

Progress provides the tools that let you create and maintain procedures quickly. When you use Progress, you don't have to program the low-level data-access routines that move data around and take care of the database. This functionality is present and ready for you to tap into from the very first Progress 4GL statement that you write.Your computer instructions—also called *code*—can concentrate on displaying and manipulating the data in the database.

The database and procedures that you create with Progress form an *application*—a complete solution to a business problem. The first business problem that All Around Sports wants to tackle is speeding up the sales and inventory process. To do this, it needs a sales and inventory application. In other words, it needs a database for the sales and inventory data and procedures to perform the business tasks associated with the sales and inventory process.

The bottom line is that Progress allows you to create complete solutions to business problems quickly and easily. All Around Sports agrees and joins the fast growing list of Progress application developers.

1.3 About the Progress 4GL/RDBMS Product

At the most basic level, Progress is an application development language and a relational database management system. Together, the language and database are called the Progress Fourth Generation Language (4GL) and Relational Database Management System (RDBMS), or simply Progress 4GL/RDBMS. Figure 1–2 shows the components of the Progress 4GL/RDBMS product.

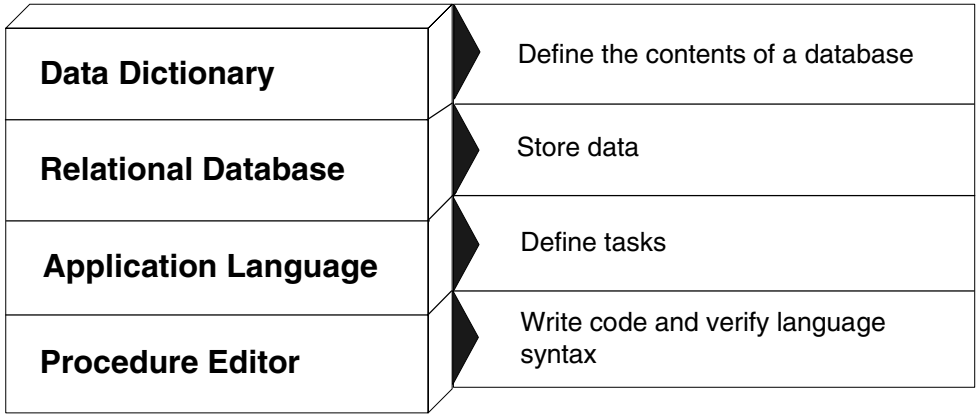


Figure 1–2: Components of the Progress 4GL/RDBMS Product

1.4 About the Relational Database Management System

Progress provides the means for you to store data in relational databases, and Progress also manages many of the low-level tasks of database access. That's what makes Progress a relational database management system. Progress reads and writes data in the database as your procedures direct, while protecting that data and maintaining the integrity of the database.

This tutorial doesn't cover the features of the RDBMS, but you do need a quick overview of database fundamentals before you can begin to create database applications effectively. This section provides a quick overview of basic database concepts and the Data Dictionary tool.

For complete information about developing databases, see the *Progress Database Design Guide*. For more information on the Data Dictionary, see the *Progress Basic Development Tools*.

1.4.1 Database Basics

The computer professionals that All Around Sports brings in to evaluate the sales and inventory system take note of how All Around Sports stores its files. All Around Sports keeps each type of file—such as customer information files, order files, and inventory files—in a separate filing cabinet. It also stores all these related filing cabinets together in one room. This strategy of grouping related information is also the basic strategy behind the structure of a database.

A database, which you can think of as an electronic filing system, stores all related information together in a structure called a *table*. A *table* is equivalent to a filing cabinet at All Around Sports. A collection of related tables makes up a database. A database is the equivalent of the room of filing cabinets at All Around Sports.

Figure 1–3 illustrates the relationship of the sales and inventory paper filing system to a database.

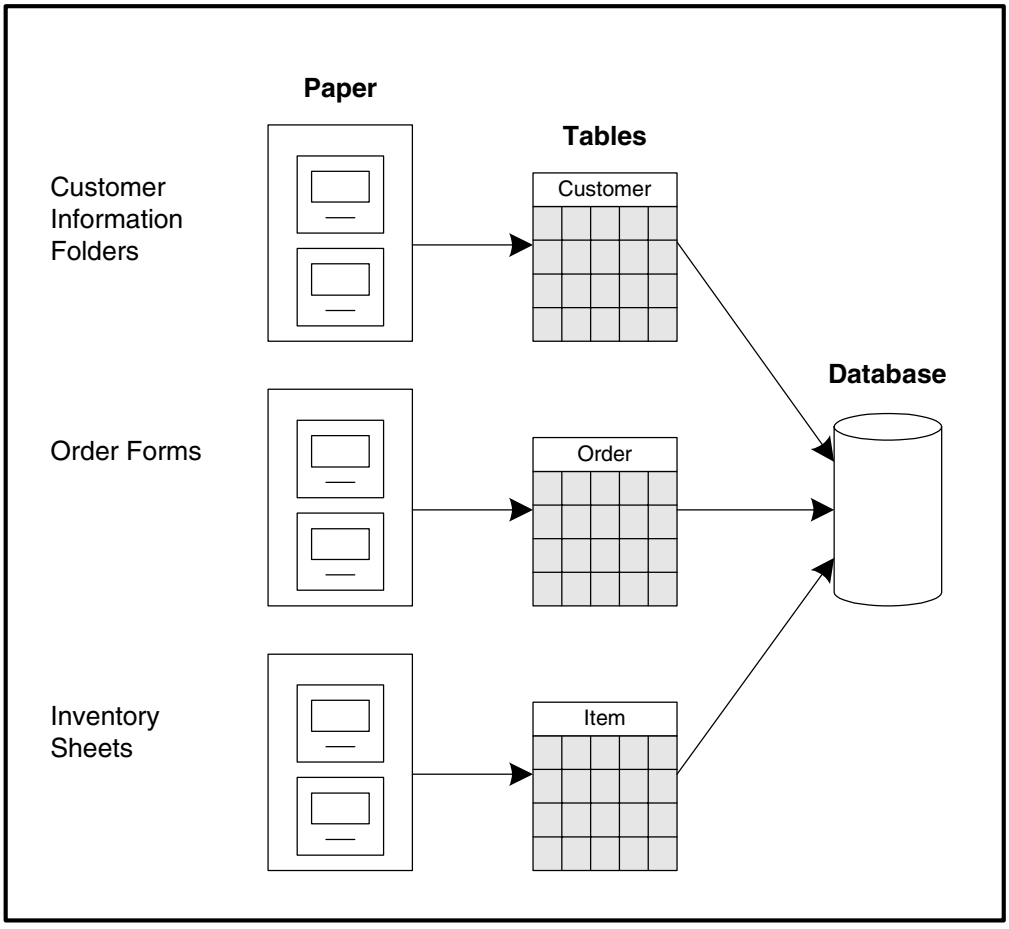


Figure 1–3: Relationship of Paper and Electronic Filing Systems

A database table is similar to a data table you might create with graph paper. A paper table consists of rows and columns. The intersection of each row and column contains one specific piece of data. In a database table, each row contains all the individual pieces of information about one member. A database row is called a *record*. In the paper filing system, each folder in a filing cabinet contains the equivalent of a row or record. For example, the customer filing cabinet contains one folder for Second Skin Scuba and another for Off the Wall Sports. When All Around Sports creates its database, the Customer table will contain one record for Second Skin Scuba and another for Off the Wall Sports.

In a database table, each discrete piece of information also represents the intersection of a row and column and is called a *field*. In addition to the data, each field in a record includes a descriptive name, just like the column heading in a paper table. For example, the customer's telephone number is one piece of information, and that information is stored in a field named Phone. Each record in the Customer table has a Phone field. Figure 1–4 illustrates the structure of a database.

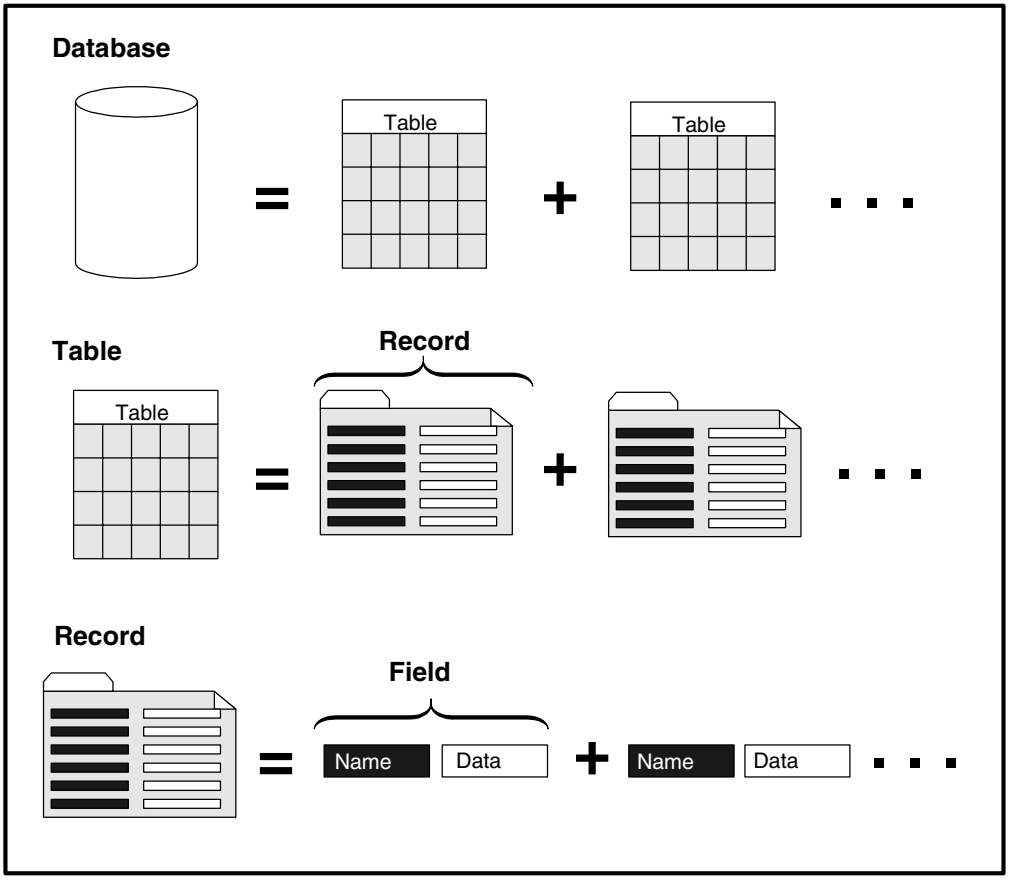


Figure 1–4: Structure of a Database

The last component of an electronic filing system is an *index*. When you open the drawer of a filing cabinet and thumb through the tabs on the folders, you're using a kind of index. What's on the paper tab is normally a copy of a piece of information found in the folder. For example, you'd expect the tabs in the customer filing cabinet to contain the name of the customer whose information is in that folder. Similarly, an electronic index is a component that serves as the basis for searching, sorting, or otherwise processing the records in a particular table.

In a database, an index is a list that contains a value for each record in the table. When you define an index, you choose the field or fields used to derive the index value for each record. For example, if you choose the Name field as an index, Progress creates an index for the Customer table that consists of a list of customer names, just like the list of tabs in the paper filing system.

A *simple index* is based on the value of one field, while a *compound index* is based on two or more fields. Figure 1–5 shows an example of a simple and a compound index.

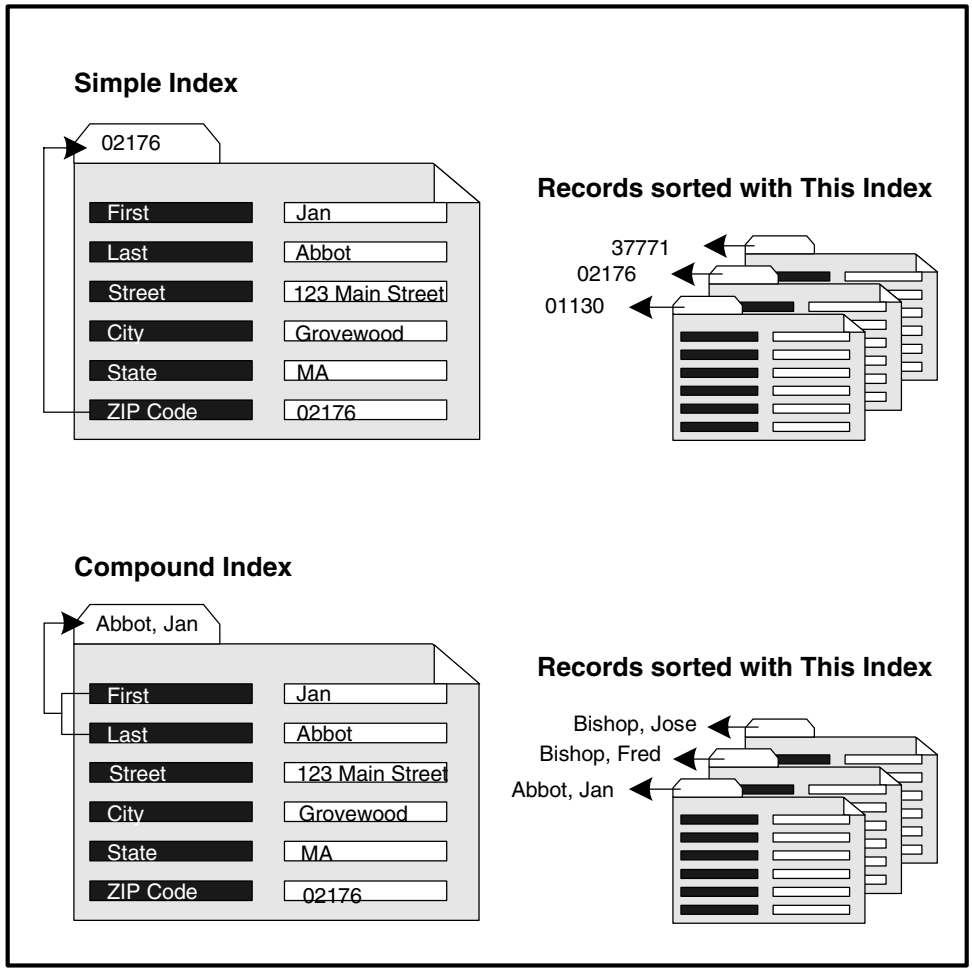


Figure 1–5: Simple and Compound Indexes

Depending on your needs, you can define more than one index per table. When you are coding applications, you'll be able to use indexes for faster processing.

1.4.2 About the Data Dictionary

The tool you use to create databases is the Data Dictionary. The Data Dictionary lets you move step by step through defining each table, field, and index that makes up your database. The Data Dictionary also provides an easy way to view and change the configuration of an existing database.

Each table, field, and index has a variety of properties that you can modify. Many of these properties affect Progress's behavior. When you begin programming Progress procedures, you'll also start exploring databases with the Data Dictionary to change properties and make Progress work more effectively for you.

Application Defaults

A major part of your application's work involves presenting database fields for end users to view and manipulate. Each time you access and display a field, you won't have to write lots of code—you can rely on Progress to use default information stored in the Data Dictionary to properly present the data. Progress stores these defaults in a Progress database:

- Default display information for fields, such as display formats and help messages
- Default data access behavior for both fields and tables, such as data input checking routines and default values

This ability to store defaults in the database saves time for developers. Using defaults makes it easier to enforce data-access rules and reduces error-checking code, default value generation, and display-formatting code in your procedures.

See Chapter 4, "Understanding the Database Environment," for more information about the Data Dictionary and Progress databases.

1.5 About the Application Development Language

Most general-use programming languages—such as COBOL, FORTRAN, and BASIC—are called *third generation languages (3GLs)*. These languages let you control a computer at a low level, while using an English-like syntax that makes the resulting code understandable.

The Progress application development language is a *fourth generation language (4GL)*. A 4GL makes programming much easier because it:

- Provides powerful statements that perform the work of multiple 3GL statements
- Uses intuitive default behavior
- Employs a highly readable syntax that is succinct and much closer to natural English

Besides giving you the complete flexibility of a general-use programming language, the Progress 4GL allows you to do two main tasks: access and manipulate data in a database, and present that data for user inspection and interaction.

Look at this Progress code:

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

This compact loop, where Customer is the name of a database table, performs these functions:

- Reads the contents of each Customer table record, beginning with the first record
- Accesses the default display format for each field in the record
- Creates an appropriate display
- Displays as many records as can fit in the display
- Pauses and waits for the user to signal when to display more information

Each of these tasks would require many lines of 3GL code. The Progress 4GL executes these tasks implicitly.

Progress also makes useful assumptions about what you want, relieving you from the tedious task of coding every tiny detail. For example, notice that the DISPLAY statement in the example does not offer any detail about how to format the output. Progress constructs a default output format by combining default formatting information for each database field with default output algorithms. This intuitive handling of defaults is called *default behavior*. This advantage does not come at the expense of customization. Every Progress statement includes options that allow you to override default behavior.

You'll find yourself relying on default behavior in the beginning of a project so that you can quickly create a prototype. Later, you'll get more involved with programming options to fine tune your application.

1.5.1 About the Procedure Editor

The Progress 4GL/RDBMS also provides you with the Procedure Editor. From the Procedure Editor, you create, compile, and run Progress procedures. The Procedure Editor also provides extensive facilities for checking syntax, managing multiple procedure files, and customizing your environment.

In the next chapter, you'll learn to use the Procedure Editor. It won't take you long to master the Procedure Editor and customize its environment. For the rest of the tutorial, the Procedure Editor will be your base for trying out example code and creating your own code.

1.6 Summary

First, you learned how Progress can solve tough business problems. A computerized solution to a business problem is called an application. An application consists of a database and Progress procedures. Here's a summary of important points about databases:

- A database is an electronic filing system for organizing and storing data that relates to a broad subject area, like sales and inventory.
- A database that lets you organize data so you can easily define and exploit relationships among data is called a relational database.
- At the top level, a database is made up of tables. A table is a collection of records about a specific subject, like customers.
- A record is a collection of pieces of information, called fields, about one thing, like a single customer.
- A field is a specific item of information, like a customer name.
- An index uses data from a field or fields as the basis for searching, sorting, or otherwise processing records.

Progress procedures are collections of statements that let you define tasks. Here are some important points about Progress procedures:

- You create procedures with the Progress fourth generation language (4GL).
- A 4GL is much easier to use than a third generation language (3GL) because 4GL statements perform the work of many 3GL statements and use default behavior to relieve you from programming every tiny detail.

You also learned about the Progress tools for creating and maintaining applications:

- The Data Dictionary lets you define and create databases.
- You can also use the Data Dictionary to store display and data-access defaults, which help cut down on redundant coding.
- The Procedure Editor lets you create, edit, and run procedures. The Procedure Editor also lets you check the syntax of your code.

2

Getting Started

This chapter teaches you the essentials of starting Progress and using the basic tools. Specifically, this chapter covers:

- Starting Progress
- Using the Procedure Editor
- Working with Progress interfaces
- Manipulating text and buffers
- Learning basic 4GL constructs and conventions
- Accessing Progress online help
- Accessing sample procedures
- Leaving the Procedure Editor

If you need more information about one of the Progress tools, see the *Progress Basic Development Tools* manual.

2.1 Preparing to Use This Tutorial

Before you start working through this tutorial, you need to:

1. Confirm that Progress is properly installed on your system.
2. Create a directory named `mytut`. The `mytut` directory will be your working directory for all tutorial exercises.
3. Create a subdirectory of `mytut` named `sports`. The `sports` subdirectory will contain procedures that are part of the Progress demonstration database. The tutorial uses the demonstration database, named sports, as the basis for programming examples and exercises.

If you run into problems with these tasks, consult the *Progress Installation Notes*, your operating system documentation, or your system administrator.

2.2 Accessing the Language Tutorial Sample Procedures

The remaining chapters of this tutorial use a set of sample procedures to illustrate Progress programming concepts. In fact, several of the manuals in the documentation set have sample procedures that come with Progress. You might want to access these sample procedures to help you better understand the information presented.

For information on extracting the sample files from their procedure libraries, see the "Example Procedures" section of the Preface.

2.3 Starting Progress

The procedure you use to start Progress in a character client mode depends on the operating system on which you are running Progress. This section describes how to access Progress from the platforms specified.

2.3.1 Windows NT 3.51

On your desktop, locate the program group box labeled Progress. Open it and look for the Progress Character Client icon. Double-click on the Progress Character Client icon to display the Procedure Editor.

2.3.2 Windows 95 and Windows NT 4.0

Click the Start button on the taskbar and choose Programs→ Progress→ Character Client. Click on the Progress Character Client program option to display the Procedure Editor.

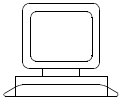
2.3.3 UNIX

To run Progress in character client mode on a UNIX operating system, execute this startup command from the UNIX system prompt: `pro [dbname] [parameters].`

NOTE: Keep the following points in mind when starting Progress:

- The *dbname* parameter is the pathname of a database that you want to connect when you start Progress. This parameter is optional for the startup command. However, you must connect to a database before you can access that database from a Progress application. (As you'll learn later, you can connect a database after you've started Progress.)
- You can also specify additional startup parameters with the startup command that tailor the Progress environment to your specific needs. For specifics on startup parameters, see the *Progress Startup Command and Parameter Reference*.

As necessary, refer to your specific operating system-related documentation.



Exercise

Follow these steps to start Progress:

1 ♦ Make the `mytut` directory your current working directory.

2 ♦ Use the appropriate startup activity as previously described in this section to start Progress.

2.4 About the Procedure Editor

When you start the Progress character client, the Procedure Editor is the first tool that you see. The Procedure Editor is a text editor that allows you to enter, edit, and save text in text files. The Procedure Editor is your hub for most programming tasks in the Progress toolset. You can use the Procedure Editor to create, compile, and run Progress procedures. A *procedure* is a series of Progress language statements that perform a desired data processing task.

Figure 2–1 shows the Procedure Editor main display.

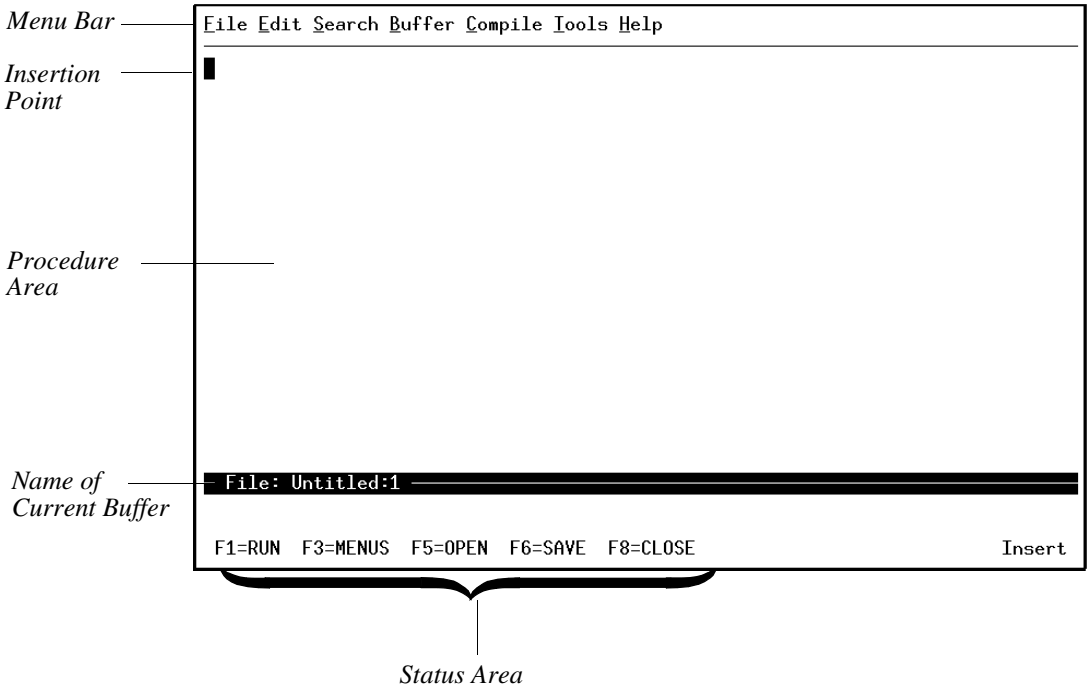


Figure 2–1: Procedure Editor Main Display

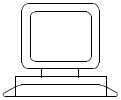
The main display of the Procedure Editor contains several features:

- **Menu bar** — Allows you to access and execute tasks. Each item on the menu bar is a menu title. A *menu* provides access to menu options. *Menu options* perform tasks.
- **Insertion point** — Marks the location where text appears when you start typing.
- **Procedure area** — The visible part of the current edit buffer. This is where you type and edit Progress procedures.
- **Name of current buffer** — Shows the name of the current edit buffer. An *edit buffer* is a temporary work area for procedures under construction in the Procedure Editor. The Procedure Editor allows you to open several buffers simultaneously. If a buffer has no name assigned, it appears as "Untitled" plus a number that makes it unique (Untitled:1).
- **Status area** — Where Progress displays helpful information for the user. By default, the Procedure Editor displays a list of keystrokes that perform basic tasks.

2.5 Using Object Interfaces

A large part of creating applications with Progress is providing your users with computer displays that give them workspace to accomplish their tasks. All the displays you create add up to the application *interface*. An individual display in your interface consists of interface objects that are controls, representations of data, decorations, or containers. In Progress, these interface objects are known as *widgets*.

To create successful Progress interfaces, you need to successfully program widgets. This tutorial will teach you everything you need to know about programming widgets. However, as a first step, you need to know how to use widgets. If you are already familiar with the kind of interfaces that are made up of objects, then you can skip this section and continue with the next.



Exercise

If you are new to object interfaces, access the on-line widget demonstration.

Type the complete, or absolute, pathname that references the exact location on your network where the on-line demonstration is installed. This location is typically the root program directory on which Progress is installed on your machine.

The following examples show how to enter these commands on the various operating systems that support the Progress character client. These examples assume that Progress is installed in the DLC directory. However, you might need to confirm exactly where Progress is installed.

1 ♦ To access the on-line demonstration in a character client mode on Windows, enter the following information in the Procedure Area of the Procedure Editor:

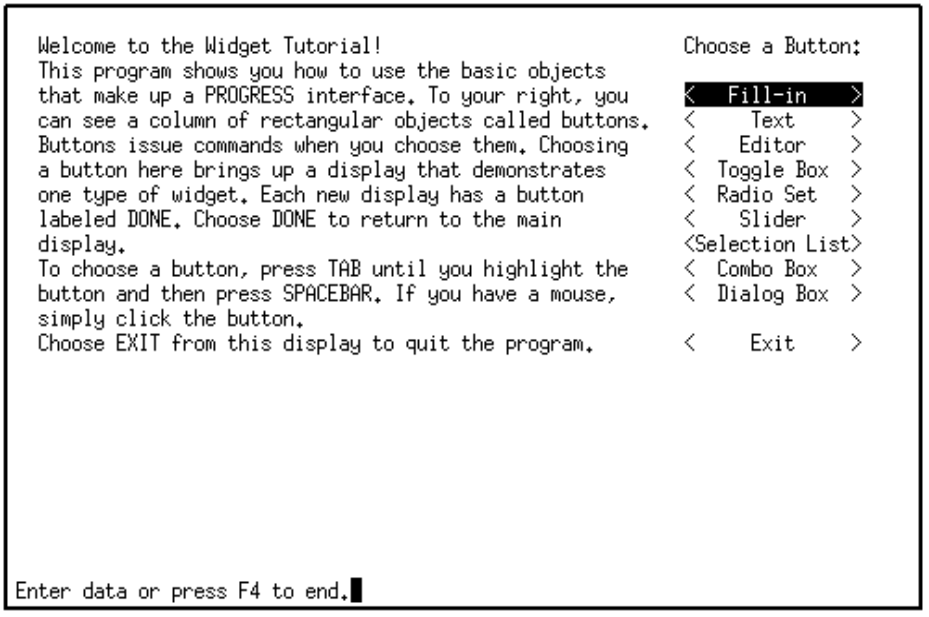
```
RUN d:\dlc\src\prodoc\langtut\widget.p
```

NOTE: The d: drive represents the exact drive location on which Progress is installed. This drive may be different on your machine.

2 ♦ To access the on-line demonstration in a character client mode on UNIX, enter the following information in the Procedure Area of the Procedure Editor:

```
RUN /dlc/src/prodoc/langtut/widget.p
```

3 ♦ Press F1. The on-line Widget Tutorial appears, as shown:



2.6 Using Key Functions and Menus

This section teaches you some basic skills you need to use the Procedure Editor, other Progress tools, and applications you create with Progress. In particular, this section describes:

- Basic key functions
- Menus and menu options

2.6.1 Basic Key Functions

The Procedure Editor responds to several keystrokes that perform editor functions. Tool functions that map to a keystroke are called *key functions* and are common throughout the Progress toolset. Progress uses a keyword like **HELP** or **GO** to represent a key function because the actual keystroke that performs the function may vary across operating systems. Many key functions in the Procedure Editor execute menu options available from the Procedure Editor menu bar. These keystrokes provide a quick and easy way to get things done.

Keep in mind that these default keyboard mappings can differ depending on the operating system on which Progress is running. For example, if you are running Progress in a character client mode on a UNIX platform, the default key mapping to **EXIT** a file is **ESC-Q.** However, if you are running Progress in a character client mode on a Windows platform, the default key mapping to **EXIT** a file is **CTRL-ALT-Q**. For a complete list of default keyboard mappings for key functions, refer to on-line Help.

Table 2–1 lists the most commonly used key functions.

Table 2–1: Basic Key Functions in the Procedure Editor

Key Function	Common Keyboard Mapping	Description
GO	F1	Run the code in the current buffer. This key function corresponds to Compile→ Run.
ENTER-MENUBAR	F3	Activate the menu bar for input.
END-ERROR	F4	Cancel an executing procedure, cancel a current dialog box, or deactivate the menu bar.
GET	F5	Open a specified text file and place it into a new buffer. This key function corresponds to File→ Open.
PUT	F6	Save the current buffer to a text file.

To interact with a widget using the keyboard, the widget must have input focus. The current widget contains the cursor or displays it in reverse video. You can move around the widgets on the display or in a dialog box and change the current widget using the **TAB** key or the **BACK-TAB** (**CTRL-U**) key.

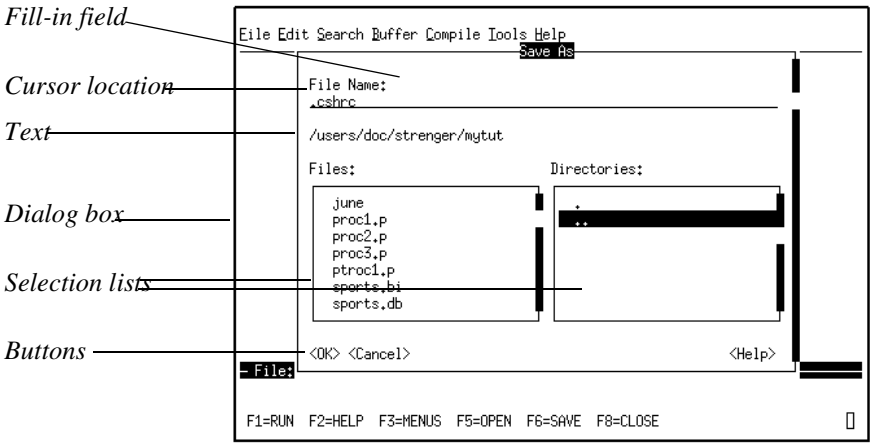


Figure 2–2: Input Focus and Navigation on a Screen

2.6.2 Menus and Menu Options

Most of the tools in the Progress toolset have a menu bar. The menu bar appears at the top of the tool display and provides access to most of the functionality in the tool. Table 2–2 provides a short description of the functionality available on each menu in the Procedure Editor.

Table 2–2: Procedure Editor Menus *(1 of 2)*

Menu	Description
File	Create new edit buffers; open text files that contain Progress language statements; save the contents of an edit buffer to a text file; print the current buffer; escape to the operating system; exit the Procedure Editor.
Edit	Manipulate blocks of text in edit buffers; insert a text file or a database field into the current edit buffer.
Search	Search and replace text in the current edit buffer; go to a specified line position in the current edit buffer.
Buffer	Navigate among open edit buffers.

Table 2–2: Procedure Editor Menus *(2 of 2)*

Menu	Description
Compiles	Run, compile, and check the syntax of the Progress code contained in the current buffer.
Tool	Access other tools in the Progress toolset.
Help	Get information about the current tool, system messages, the 4GL, and the Progress system.

The exercises in this chapter show you how to use many of the options on these menus. For a complete description of the Procedure Editor menu bar, see the *Progress Basic Development Tools* manual.

Navigating the Menu Bar and Choosing Menus Options

To choose a menu or menu option, the menu bar must be active. For example, when Progress is running in a character client mode on either a UNIX or Windows platform, you can use the **ENTER-MENUBAR (F3)** key to toggle in and out of the menu bar. That is, you can activate the menu bar and direct all keyboard input to it when you initially use the **ENTER-MENUBAR (F3)** key. Then, when your are done entering your data, you can deactivate the menu bar using the same **ENTER-MENUBAR (F3)** key.

When Progress is running in a character client mode on Windows, you can also use the mouse to activate the menu bar. Move the mouse pointer (the arrowhead cursor that displays) across the menu bar and click on the menu bar to display the associated pull-down menu options.

Figure 2–3 displays the File pull-down menu of the Procedure Editor.

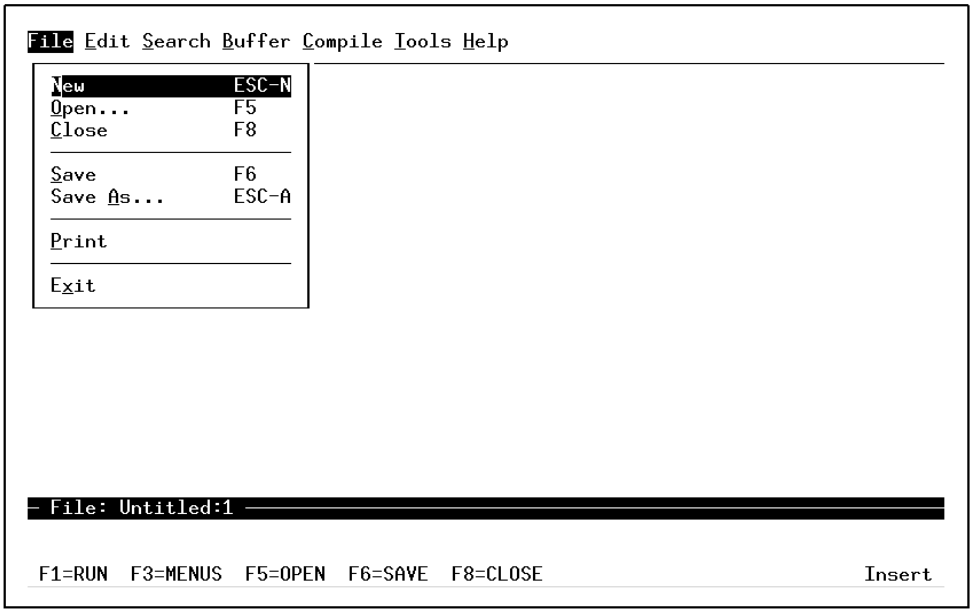


Figure 2–3: File Pull-down Menu

You can execute menu options using your cursor keys. For example, to exit the Procedure Editor, you can choose the Exit menu command from the File menu. The following information explains how to choose the Exit option with the cursor keys:

1 ♦ Press **ENTER-MENUBAR** (**F3**) to activate the menu bar. (Keep in mind that if you are running Progress in a character client mode on Windows NT, you can also use the mouse pointer to activate the menu bar.)

2 ♦ Use the ← and → keys to navigate through the menus on the menu bar and highlight the File menu.

3 ♦ Press the ↓ key to pull down the menu.

4 ♦ Use the ↓ and ↑ to navigate through the menu options on the File menu and highlight the Exit menu option. Then press **RETURN** or **ENTER** to choose the highlighted menu option.

Another way to execute a menu option is to use mnemonics. A *mnemonic* is a character associated with a menu or menu option. If the menu bar is active, you can type a mnemonic to display a menu or to execute a menu option. All menus and menu options in the Progress toolset have a mnemonic. The mnemonic is the underlined character from the label of the menu option and must be unique within a menu or the menu bar. For example, the mnemonic for the Help menu is the character "H".

Here is how to choose the Exit option from the File menu using mnemonics:

1 ♦ Press **ENTER-MENUBAR** (**F3**) to activate the menu bar. (Keep in mind that if you are running Progress in a character client mode on Windows, you can also use the mouse pointer to activate the menu bar.)

2 ♦ Type **f** to display the File menu.

3 ♦ Type **x** to choose the Exit menu option from the File menu.

The cursor keys are the best ways to explore menus in a tool and find out about available menu options. Once you become familiar with a tool, mnemonics offer a quicker way to get things done.

Progress Menu Conventions

All menu bars, menus, and menu options in the Progress toolset share several important characteristics:

- The menu option that exits a tool is always the last menu option on the first menu.
- A Help menu is available on all menu bars in the Progress toolset and is always the last menu on the menu bar.
- Some menu options have symbols that provide information about how they work. Menu options that do not use symbols are commands and simply execute the function that they represent.

Table 2–3 introduces Progress menu symbols.

Table 2–3: Progress Menu Symbols

Menu Symbol	Description
menu option →	Indicates that the menu option is a submenu. When the user chooses a submenu from a menu, additional menu options appear next to the original pull-down menu.
menu option . . .	Indicates that the menu option requires additional user input. When the user chooses a menu option with this symbol, a *dialog box* appears. Progress uses dialog boxes to prompt users for additional input before executing the function.
menu option → (on UNIX) √ menu option (on Windows)	Indicates that the menu option is a toggle for a tool setting. When the symbol is present, the setting is active or in the ON state.
√ menu option	Indicates that the menu option is a toggle for a tool setting. When the symbol is present, the setting is active or in the ON state.

- The Progress documentation set uses the following notation to represent a menu option:

  `File→ Exit`

  This notation is a shorthand representation of the Exit option on the File menu.

2.7 Text Entry and Manipulation

This section provides a basic overview of text entry and manipulation in the Procedure Editor. Much of the information in this section focuses on the keystrokes used to navigate through and edit text in an edit buffer.

The Procedure Editor is much like any other text editor. To enter text, make sure that the menu bar is not active and simply start typing. The characters you type appear at the current cursor location in the current edit buffer.

There are two modes of text entry in the Procedure Editor:

- **Insert mode —** This mode inserts a character that you type at the current cursor location, shifting existing characters to the right of the cursor by one space. This is the default text entry mode of the Procedure Editor.
- **Overtype mode** — This mode replaces any existing character at the current cursor location with the character that you type.

Press **INSERT** to toggle between the overtype and insert modes. The word "Insert" appears in the lower-right corner of the Procedure Editor display when you are in insert mode.

The procedure area is the visible portion of the current edit buffer. You can view it using the Procedure Editor. The Procedure Editor lets you scroll up and down through your data display, allowing you to expand your view of your data by using cursor keys.

Table 2–4 presents basic keys available in the Procedure Editor that help you move around and edit text in the current buffer. It highlights the similarities and differences between common keyboard mappings that you can use when you are running Progress in a character client mode either on Windows or UNIX.

Table 2–4: Basic Editing Keys *(1 of 3)*

Key Function	Common Keyboard Mapping on UNIX	Common Keyboard Mapping on Windows	Description
¦	CURSOR-UP	CURSOR-UP	Move the cursor up one line.
Ø	CURSOR-DOWN	CURSOR-DOWN	Move the cursor down one line.
¨	CURSOR-DOWN	CURSOR-LEFT	Move the cursor to the left one character.

Table 2–4: Basic Editing Keys *(2 of 3)*

Key Function	Common Keyboard Mapping on UNIX	Common Keyboard Mapping on Windows	Description
→	CURSOR-RIGHT	CURSOR-RIGHT	Move the cursor to the right one character.
BACKSPACE	BACKSPACE	BACKSPACE	Delete the character immediately to the left of the cursor.
BACK-TAB	CTRL-U	SHIFT-TAB	Move the cursor to the left one tab stop.
DEL	DEL	DEL	Delete the character the cursor is on.
DELETE-LINE	CTRL-D	CTRL-D	Delete the current line.
DELETE-WORD	ESC-D	CTRL-ALT-D	Delete the word the cursor is on.
END	ESC-.	–	Move the cursor to the end of the current buffer.
HOME	ESC-,	–	Move the cursor to the beginning of the current buffer.
INSERT	F9	F9	Toggle between insert mode and overtype mode.
LEFT-END	ESC-CURSOR-LEFT	HOME	Move the cursor to the beginning of the current line.
NEXT-WORD	CTRL-W	CTRL-CURSOR-RIGHT	Move the cursor right to the first character after the next space character.

Table 2–4: Basic Editing Keys *(3 of 3)*

Key Function	Common Keyboard Mapping on UNIX	Common Keyboard Mapping on Windows	Description
PAGE-DOWN	ESC-CURSOR-DOWN	PAGE-DOWN	Scroll down one screen in the current buffer.
PAGE-UP	ESC-CURSOR-UP	PAGE-UP	Scroll up one screen in the current buffer.
PREV-WORD	CTRL-P	CTRL-CURSOR-LEFT	Move the cursor left to the first character after the previous space character.
RETURN	RETURN	ENTER	Insert a new line at the cursor location.
RIGHT-END	ESC-CURSOR-RIGHT	END	Move the cursor to the end of the current line.
TAB	TAB	TAB	Move the cursor to the next tab stop.

The next section describes more advanced text editing techniques in the Procedure Editor.

2.7.1 Manipulating Blocks of Text

The Procedure Editor allows you to define and manipulate blocks of text. A block of text can be a set of contiguous characters, a whole line, several contiguous lines in a buffer, or an entire procedure.

Follow these basic steps to manipulate a block of text:

1 ♦ Position the cursor on the first character of the text block.

2 ♦ Press BLOCK (defined later in this section) to mark the beginning of the text block.

3 ♦ Position the cursor one character to the right of the last character in the text block.

4 ♦ Execute the desired text-block operation. You can cut a text block, copy a text block, or paste over a text block.

Text block manipulation centers around three operations: cut, copy, and paste. The Procedure Editor supports these operations with a special cut/paste buffer. This special cut/paste buffer is used by all platforms running in character mode. When you cut or copy a block of text, the text goes into the cut/paste buffer and remains there until you cut or copy another block of text or exit the Procedure Editor. A paste operation inserts the contents of the cut/paste buffer at the current cursor location or replaces a defined text block.

Table 2–5 lists the set of keys that allow you to define a block and execute a text-block operation on either a UNIX or Windows platform.

Table 2–5: Key Functions to Define Text-block Operations

Key Function	Common Keyboard Mapping on UNIX	Common Keyboard Mapping on Windows	Description
BLOCK	CTRL-V	CTRL-V	Mark the beginning of a text block.
COPY	ESC-C	F11	Copy the current text block into the cut/paste buffer. This key function is the same as Edit→ Copy.
CUT	ESC-X	F10	Cut the current text block into the cut/paste buffer. This key function is the same as Edit→ Cut.
PASTE	ESC-V	F12	Paste the contents of the cut/paste buffer at the current cursor location. This key function is the same as Edit→ Paste.

With this functionality, you can cut and paste code from one position to another in the current edit buffer. You can also cut and paste between edit buffers.

2.8 Edit Buffers and Procedure Files

An edit buffer is a temporary area in computer memory maintained by the Procedure Editor where you can create and edit Progress procedures. A buffer only lasts as long as a Progress session. When you are done with your session, you can use the Procedure Editor to save buffers to procedure files.

A *procedure file* is a text file that contains one or more Progress procedures. By convention, procedure files have a .p file extension. As you go through this tutorial, you will learn about specialized procedure files and other text files that contain Progress language statements but do not contain procedures. These Progress files have other file extensions.

It is important to understand the relationship between buffers and procedure files. When you open an existing procedure file using the File→ Open menu option, the editor creates a buffer whose title echoes the name of the file. Edits that you make in a buffer associated with an existing procedure file are temporary until you explicitly save the buffer. If you do not save the buffer, your edits are lost when the Procedure Editor session ends.

In addition to the editing tasks described in the last section, it is important to realize that you can cut or copy text from one buffer and paste into another buffer.

NOTE: Although you can open multiple buffers in the Procedure Editor, you can open only one buffer for each operating system file.

Table 2–6 summarizes the Procedure Editor features that help you create and manage buffers.

Table 2–6: Buffer Tasks and Associated Menu Options *(1 of 2)*

Task	Menu Option	Description
Create a new buffer.	File→ New	Create a new “Untitled” buffer and make it the current buffer.
Open a procedure file and create a buffer for it.	File→ Open	Open a specified procedure file into a new buffer named after the procedure filename. The newly opened buffer becomes the current buffer.
Close a buffer.	File→ Close	Close the current buffer. If you modified the contents of the buffer, Progress notifies you and asks you if you would like to save the modifications to a procedure file.

Table 2–6: Buffer Tasks and Associated Menu Options *(2 of 2)*

Task	Menu Option	Description
Save the contents of a buffer to a procedure file.	File→ Save	Save the current buffer to a procedure file. If the current buffer is "Untitled," the Save As dialog box prompts you for the name of a procedure file.
Save the contents of a buffer to a specified procedure file.	File→ Save As	Save the contents of the current buffer to a specified procedure file.
Print the contents of a buffer.	File→ Print	Print the contents of the current buffer.
List the open buffers.	Buffer→ List	Display a list of open buffers in the Buffer List dialog box. You can go to another buffer or save an open buffer from this dialog box.
Go to the next open buffer.	Buffer→ Next Buffer	Make the next buffer in the buffer list the current buffer.
Go to the previous open buffer.	Buffer→ Previous Buffer	Make the previous buffer in the buffer list the current buffer.
Display detailed information about a buffer.	Buffer→ Information	Display specific details about a buffer such as the name of the file, file access privileges, the number of lines, bytes, and columns that it contains, and whether it has been modified.
Compile and run the contents of the buffer.	Compile→ Run	Compile and attempt to run the contents of the current buffer as a Progress procedure. If there are syntax errors in the current buffer, Progress displays a dialog box that contains a list of error messages.
Check the contents of the current buffer for 4GL syntax errors.	Compile→ Check Syntax	Check the contents of the current buffer for 4GL syntax errors. If there are syntax errors in the current buffer, Progress displays a dialog box that contains a list of error messages.

2.9 Basic 4GL Constructs and Conventions

This section provides a brief overview of basic Progress language constructs and conventions. This information is by no means a complete description of the Progress language, but it does provide a starting point from which you can expand your knowledge of the Progress 4GL. To start, let's consider the following code:

```
DEFINE VARIABLE cnt AS INTEGER.
REPEAT cnt = 1 TO 15:
   DISPLAY "Welcome to Progress!"
      WITH CENTERED.
END.
```

This procedure contains the following Progress language constructs and elements:

- **Keywords** — A keyword is a word that instructs Progress to do something. All of the unquoted capitalized words in the procedure are Progress keywords. By convention, all 4GL code examples in this tutorial use uppercase letters to display Progress keywords. However, it is important to point out that Progress is case insensitive. You can enter Progress keywords into a procedure in uppercase, lowercase, or a mix of both.

- **Statements** — A statement is one complete instruction to Progress. Statements normally begin with a verb (a Progress keyword) and end with a period or colon. In the code above, the keywords DEFINE, REPEAT, DISPLAY, and END all begin statements and are called verbs. The REPEAT keyword begins a special kind of statement called a block header statement. Statements can extend for several physical lines in the Procedure Editor and can contain phrases, options, and values.

- **Phrases** — A phrase is a collection of keywords and values that modify the way Progress executes a statement. In the code above, the keyword WITH begins a frame phrase in the DISPLAY statement. Phrases can contain options and values.

- **Options** — An option is like a phrase, only smaller. Options usually consist of a single keyword and a possible accompanying value. Options also modify the way Progress executes a statement. In the code above, the keyword AS is an option of the DEFINE VARIABLE statement and the keyword CENTERED is an option in the frame phrase of the DISPLAY statement.

Progress is a block-structured language—that is, you can group statements together into blocks. Procedures are the largest block structures in Progress. Later in the tutorial, you'll learn about other types of blocks that can exist inside a procedure block. A repeat block is similar to the repeat loop, which is found in many 3GLs. The following code fragment shows a repeat block.

```
DEFINE VARIABLE cnt AS INTEGER.
REPEAT cnt = 1 TO 15
   DISPLAY "Welcome to Progress!"
      WITH CENTERED.
END.
```

Block structures within a procedure normally begin with a block-header statement and end with an END statement. You can use these block structures to apply processing services to a group of statements.

You will learn about more Progress language constructs and blocks as you proceed through this tutorial. To learn more about the syntax of the Progress language, see the *Progress Language Reference* or the Progress On-line Reference module in the Progress help system.

2.10 Using the Help Menu

Progress comes with an online Help menu that enables you to access information to help you understand online messages and how to map your keyboard to Progress keyboard functions. Table 2–7 lists and describes these specific Help menu options.

Table 2–7: Procedure Editor Help Menu

Help	
Messages...	Displays information about a specified system message in Progress. You enter the message number associated with an online message and press ENTER to view a detailed message description. Use the arrow keys to scroll through a lengthy message.
Recent Messages...	Displays message number and related text description information about recent system messages in Progress. Use the arrow keys to scroll through a lengthy message.
Keyboard...	Displays information about how your keyboard maps to Progress key functions.
About Procedure Editor	Displays the name, version number, control number, and serial number of the current tool.

2.10.1 Accessing the Help Menu

To access information associated with the Help menu, select the specific help option you want to display.

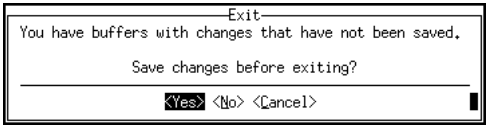
- Press the **F3** key to display the cursor on the Menu bar. Highlight the Help menu option. Then, depending on the operating system that you are using, press either the **ENTER** or the **RETURN** key to display all of the Help menu options and make your selection.
- Press the **F4** key to exit a displayed help topic.

2.11 Leaving the Procedure Editor

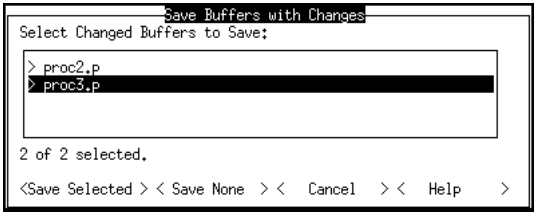
You can exit the Procedure Editor in one of the following ways:

- Choose File→ Exit
- Type **quit** in an empty buffer and press **GO**

When you exit the Procedure Editor, the editor checks the open buffers for unsaved changes. If there are buffers with unsaved changes, the following dialog box appears.



If you choose the Yes button to save modified buffers, the Save Buffers with Changes dialog box appears, as shown :



The right angle bracket next to a buffer name indicates that the buffer is selected. To toggle the selection of a buffer in the list, highlight the buffer name and press **RETURN**.

This dialog box presents three options:

- Save the selected buffers in the buffer list by choosing the Save Selected button.
- Opt not to save any buffers and exit the Procedure Editor by choosing the Save None button.
- Cancel the exit and save operations and return to the Procedure Editor display by choosing the Cancel button.

Exiting the Procedure Editor returns you to the operating system or to the Progress tool from which you invoked the editor.

2.12 Summary

Before you finish this chapter, take a moment to review your progress on these topics.

Starting Progress:

- The Progress command starts Progress from your operating system prompt.

The Procedure Editor and interface basics:

- The Procedure Editor is a text text editor that you can use to create, compile, and run Progress procedures. A procedure is a series of Progress language statements that performs a desired data processing task.
- A key function is an event in Progress that describes a behavior that occurs when you use a particular keystroke or key combination. Some common key functions are **HELP**, **ENTER-MENUBAR**, and **GO**. However, the keyboard mappings for specific keys may vary depending on the console you are using and the platform on which Progress is running.
- Many functions of the Procedure Editor are accessible from the menus and menu commands on the Procedure Editor menu bar. The **ENTER-MENUBAR** key activates the menu bar.
- The computer displays that enable a user to accomplish tasks are called the user interface. In Progress, interfaces are made up of objects, called widgets, that can be controls, representations of data, decorations, or containers. Widgets also serve as receptors for user input and application output.

Text entry and text manipulation in the Procedure Editor:

- To enter text in a buffer, deactivate the menu bar and start typing.
- The Procedure Editor supports the basic editing keystrokes that are standard to most text editors and word processors.
- You can define blocks of text to cut, copy, and paste in an edit buffer or between edit buffers.

Edit buffers and procedure files:

- An edit buffer is a temporary area in computer memory maintained by the Procedure Editor where you can create and edit Progress procedures. You can use the Procedure Editor to save buffers to procedure files.
- A procedure file is a text file that contains one or more Progress procedures. By convention, procedure files have a .p file extension.
- The Procedure Editor provides menu commands that allow you to edit and manage multiple edit buffers and open and save procedure files.

Basic 4GL constructs and conventions:

- A keyword is a word that instructs Progress to do something.
- A statement is one complete instruction to Progress. Statements normally begin with a Progress keyword (verb), and end with a period or colon.
- Phrases and options consist of keywords and values and alter the way Progress executes a statement.
- A block is a group of Progress language statements.

Accessing Language Tutorial Sample Procedures:

- You can access these sample procedures on line in the procedure libraries `prodoc.pl` and `prohelp.pl` in the `src` directory where Progress is installed.

Leaving the Procedure Editor:

- The File→ Exit menu option allows you to exit the Procedure Editor and return to the operating system or the tool that called the Editor.

For a complete description of the Procedure Editor, see the *Progress Basic Development Tools* manual.

3

Programming the Progress Way

So far you've learned what Progress is about and you've had a chance to work with the Procedure Editor. Now it's time to begin working with the 4GL. This chapter covers what is fundamental to every Progress application. First, the chapter presents the underlying concepts and techniques of the Progress 4GL. Second, the chapter examines the components of a Progress procedure. By the end of the chapter, you will be able to recognize a well-structured Progress procedure and understand the function of its constituent parts.

Specifically, you'll learn about:

- The Progress programming model
- The components of a user interface, called widgets
- User input, called events
- The structure of a typical Progress procedure
- Creating a user interface
- Manipulating widgets during run time

In this chapter, you'll see examples of complete Progress procedures. For now, your goal isn't to master all the language elements introduced. Instead, your goal is to become familiar with and comfortable with the major components of a Progress procedure.

3.1 The Progress Programming Model

As you learn more and more Progress features, you'll be able to see how they contribute to the major philosophies that Progress embodies. Progress strives to:

- Make the application developer more productive by minimizing the amount of code needed to create complete and powerful applications.
- Isolate the developer from as many portability issues as possible—let Progress handle platform-specific issues during compile time or run time.
- Provide flexible and intelligent features that allow the end user to work intuitively and productively with highly responsive Progress applications.

Chapter 1 discussed the nature of the 4GL and how it streamlines application development by providing more functionality with less code. The second philosophy, easy portability, allows you to focus on developing one application, using one programming language and one program structure. Then, when you move your application to other platforms, you'll see the benefits of Progress's portability philosophy:

- Progress ports with little or no modification.
- Progress provides the native look and feel of each platform.
- Progress maintains the functionality you coded.

The third philosophy, providing the end user with intuitive and responsive applications, takes a little more time to present. Creating responsive applications requires a certain style of coding—what Progress calls the *Progress programming model*. To use the programming model effectively requires looking at the process of user and application interaction in a certain way. The rest of this section discusses the concepts, terminology, and language elements that make up the basic Progress programming model.

3.1.1 User Interfaces

To begin, look at the application screen shown in Figure 3–1. It's simply three fields arranged on a computer screen. This simple screen helps illustrate some of the important concepts and terms of the Progress programming model:

- The screen is the mechanism for communicating with your user.
- The keyboard and the mouse are the user's mechanism for communicating with your application.

Together, these two devices make up the *user interface*.

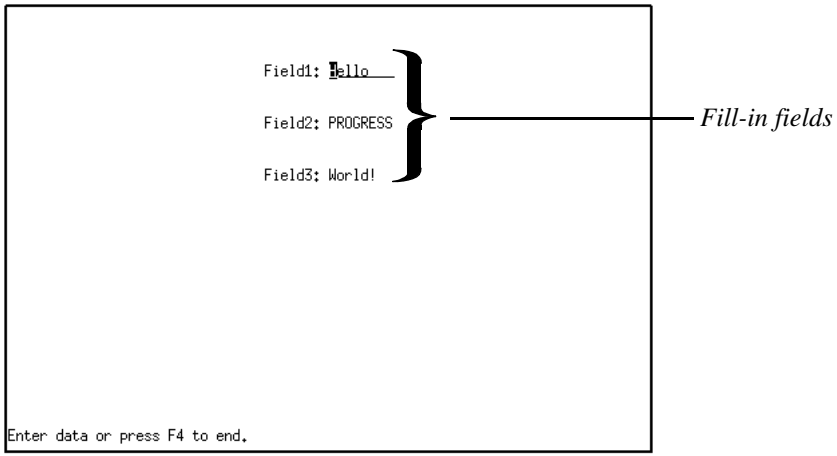


Figure 3–1: Basic Character Interface

Before continuing, it's important to distinguish between two types of user interfaces: *character interface* and *graphical user interface*, or *GUI*. These user interfaces are distinct from one another in appearance and behavior. The primary reason for their differences is the fact that these interfaces run on different operating systems that support different functionality.

For example, let's look at Figure 3–1, a character interface. A character interface supports a text-based data display that is presented in a single fixed-width font. The display occurs within a single, fixed-sized default window. A character interface accepts ASCII keyboard characters as input, and it can support limited mouse capabilities such as basic text selection and navigational activities. Some advantages associated with character interfaces are speed and portability.

In contrast, Figure 3–2 presents a GUI that is comparable to the graphical interface that Windows supports. This figure shows how this same simple Progress procedure looks when you run it on Windows 95, a GUI system. A GUI supports not only text-based presentations with multiple font types and sizes and fixed and proportional spacing, but it can also feature high-resolution graphics such as images and buttons. A programmer working in a graphical interface has the opportunity to create and control multiple windows, manipulating the properties associated with each unique window. Also, the graphical interface supports a wide range of mouse activities, expanding beyond simple selection and navigational activities into a broader range of editing capabilities. Due to its emphasis on visual presentation and the numerous mouse-supported "point and click" interactive opportunities, many users and programmers feel this interface is easier to learn and use.

NOTE: Progress supports a character client that runs in DOS only when DOS is running on Windows. However, when a character client is running in this manner, the interface that displays on both of these platforms is a character interface, not a graphical one.

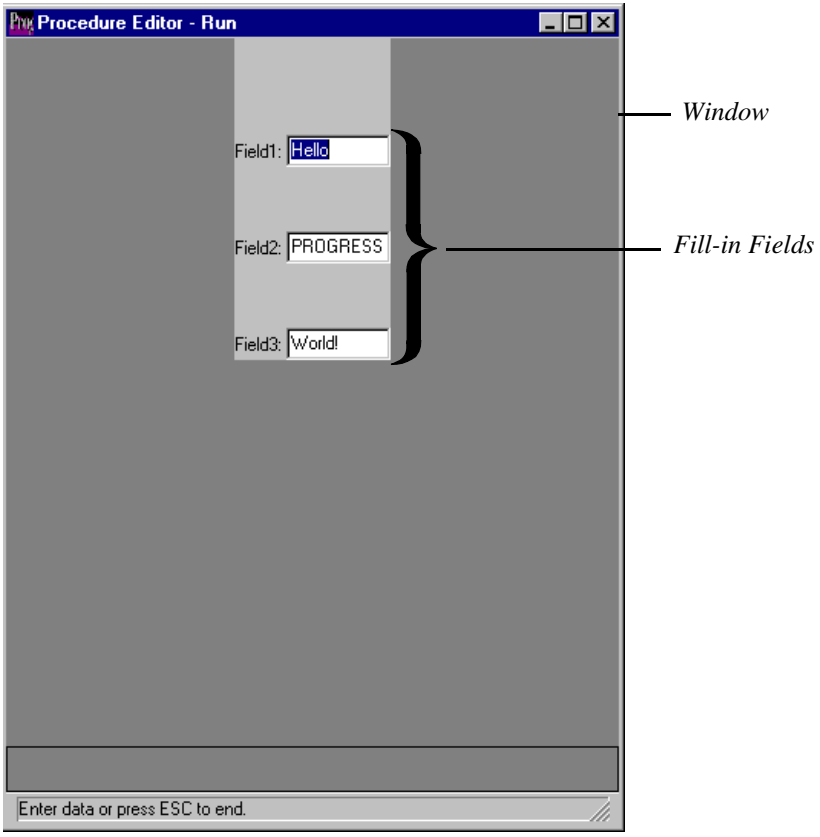


Figure 3–2: Basic Graphical User Interface

When you are working with a character interface, you will be working with what the Progress programming model refers to as the *default window*. (If you were to recompile your code on a GUI platform, Progress would run your application in the default GUI window.)

Now, look again at the contents of the window shown in Figure 3–2. The window contains three data fields. Instead of thinking of them as fields, however, think of them as a collection of individual objects. Each object has capabilities for communicating with and receiving input from the user. In this example, the field objects, called fill-in fields , display the current contents of three variables. (A *variable* is a temporary data location in memory.) The user can choose a fill-in field object and change the contents of that object, thereby changing the value of the underlying variable.

Every part of a Progress user interface, whether it is a character or a graphical interface, including a window, is an object. In Progress, user interface objects are called *widgets*. When you build an interface with Progress, you are essentially presenting the user with widgets that they can manipulate. As you'll see later in this chapter, Progress has an impressive inventory of widgets. For now, the fill-in field widget will illustrate how a user interacts with an interface and how Progress handles those interactions.

3.1.2 User Input

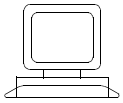
Any type of interaction a user can have with a Progress application is called an *event*. For example, typing new data into a fill-in field is actually a series of events—each individual keystroke is a separate event. Of course, there is a big difference between typing a character and pressing **RETURN**, and Progress understands those differences. On graphical interface platforms, where users have the added advantage of a mouse, each user action, like pressing a mouse button, is also a separate event.

When you program in Progress, think of the user interface as a tool for displaying data and receiving user events. Progress processes the user events and notifies your application so that your code can respond to the events.

Considering the number and combinations of ways that a user can interact with an interface, the task of receiving and responding to those events must seem daunting. However, the Progress programming model significantly simplifies this task for you. First, the low-level computer tasks of receiving events from a keyboard or mouse and notifying your application occur automatically with Progress. Second, sorting through the stream of events becomes easier once you understand that certain events go naturally with certain widgets. For example, while it makes perfect sense to type characters in a fill-in field widget, it does not make sense to type characters to a button widget. Therefore, fill-in fields accept keyboard character events and button widgets do not.

This widget-event pairing leads to the question: how do you know which widget is receiving events? The user can tell which widget receives events because of the cursor location or other visual cues that Progress and the operating system provide. Since you cannot predict the order in which the user will interact with your widgets, your code must be ready to respond to many events.

How does your code know which widget is receiving events? To begin with, a widget can receive events only after the application has *enabled* it to receive input. Of all the widgets that are enabled at any time, only one can receive events. When a widget is the current event receptor, it has *input focus*. The application or the user can designate a widget to have input focus. The application can assign input focus to an enabled widget programmatically. The user can assign input focus by selecting the desired widget. The concept of input focus is absolutely crucial to a Progress application.



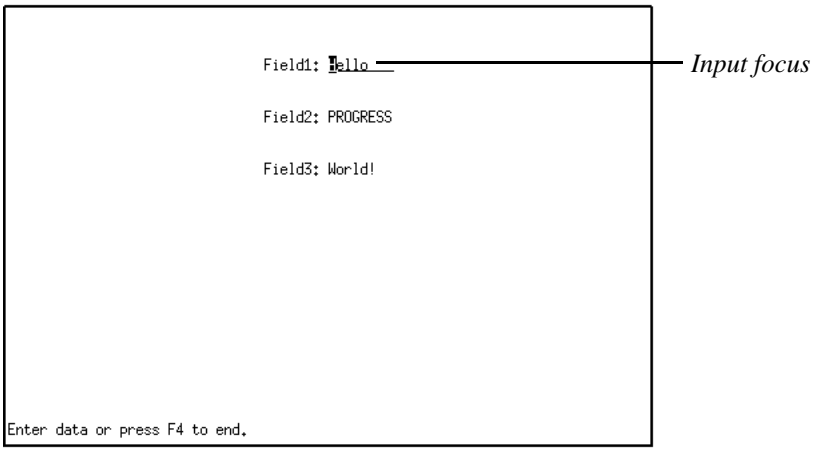
Exercise

User Controlled Input Focus

Follow these steps to observe how the user can control input focus:

1 ♦ From the Procedure Editor, open the `lt-03-01.p` procedure. (Use the Open option on the File menu.) Don't worry about trying to understand the code right now.

2 ♦ Choose Compile→ Run. You now see the basic interface you saw in Figure 3–1.



Notice that the cursor is positioned in Field1. This cursor illustrates two points:

- The user is normally aware of which widget has input focus by the visual cues provided automatically by Progress and the underlying operating system.
- On startup, your application normally assigns input focus to one widget.

3 ♦ Press **TAB** to move input focus among the fields. These actions illustrate the last important point about input focus: after startup, the location of input focus is normally controlled by the user. In other words, users can enter data into the fill-in fields in any order.

4 ♦ Move input focus to Field3 and press **RETURN** to end the procedure.

5 ♦ Press **SPACEBAR** to return to the Procedure Editor.

Because the user has more control of input focus, the user has more control of the application. The application's job is to respond to the user's events in a way that correctly interprets the user's desires. Because of this reliance on events, this type of programming model is called *event-driven programming*.

Parts of a Progress Procedure

Now, examine the code for the procedure you just ran:

lt-03-01.p

```
/*1*/  DEFINE VARIABLE Field1 AS CHARACTER INITIAL "Hello".
       DEFINE VARIABLE Field2 AS CHARACTER INITIAL "Progress".
       DEFINE VARIABLE Field3 AS CHARACTER INITIAL "World!".

/*2*/  DISPLAY SKIP(3) Field1 SKIP(2) Field2 SKIP(2) Field3
       WITH SIDE-LABELS CENTERED NO-BOX.

/*3*/  ENABLE Field1 Field2 Field3.

/*4*/  WAIT-FOR RETURN OF Field3.
```

You can break up this code into four sections:

1. **The first part of a Progress procedure defines widgets.** The three DEFINE VARIABLE statements create three new variables for text data (AS CHARACTER) and initialize them with a string (INITIAL " "). Progress displays variables or database fields as fill-in fields by default.

2. **The second part of a Progress procedure creates the user interface.** The DISPLAY statement creates an interface that consists of the default window, blank lines (SKIP), fill-in fields, and labels for the fields (SIDE-LABELS). This DISPLAY statement also centers the widgets within the default window (CENTERED) and suppresses the default border that surrounds the widgets (NO-BOX).

3. **The third part of a Progress procedure enables the widgets of the user interface.** The ENABLE statement turns on the widgets. By default, the first widget listed in an ENABLE statement gets initial input focus. Assigning input focus to Field1 means that Field1 receives events at startup.

4. **The fourth part of a Progress procedure blocks execution.** Now that you have presented a complete interface and enabled it for the user, you have to pause the execution of the procedure to allow the user to interact with the interface. This process is called *blocking*. If you did not block, Progress would execute the code to the end of the file and end the procedure.

   The WAIT-FOR statement is your primary tool for blocking execution. With it, you establish the condition that signals that the user has finished working with the current interface. In this case, if the user presses **RETURN** (WAIT-FOR RETURN) while input focus is in Field3 (OF Field3), then the procedure completes.

Every Progress procedure that interacts with the user follows this basic four-step process:

1. Define widgets.
2. Display widgets.
3. Enable widgets.
4. Block execution.

You explicitly code these four steps. Progress provides much of the additional functionality required to make this interface flexible and responsive to the user. For example:

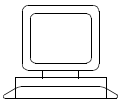
- Progress lets the user control input focus.
- Progress takes care of the normal events associated with a widget. For example, the user might want to type new text into the fields. Progress handles these events by default.

3.1.3 Responding to Events

For every widget, there is always a *default response* for each possible event. The default response for pressing **TAB** with most widgets is to move input focus to the next enabled widget. The default response for pressing a character key when a fill-in field has focus is to display that character in the field.

The default response for many events is **no response**. As mentioned earlier, some keystrokes with certain widgets don't make sense. Therefore, Progress ignores these events. Remember that for every widget-event pair, Progress executes the default response, even though that response may be no response.

Now you need a way to program new responses for widget-event pairs. For example, in the "Hello Progress World!" application, suppose you want to display a message as soon as input focus moves to Field3. The Progress programming structure for this task consists of an ON statement and a block of code called a trigger. The ON statement is the device that establishes the condition that executes a trigger. A *trigger* is the code that Progress executes only when a specified widget receives a specified event. A trigger is a *programmed response*. A programmed response occurs before the default response.



Exercise

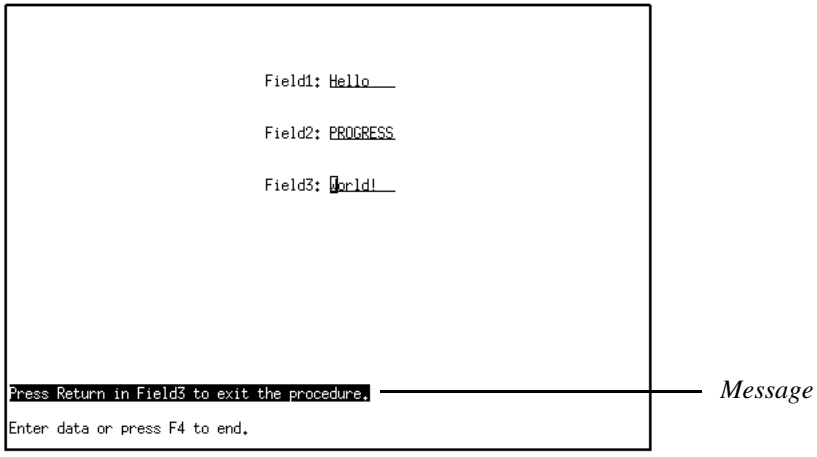
Using Triggers

Follow these steps to create a trigger:

1 ♦ Open `lt-03-02.p`.

2 ♦ Choose Compile→ Run.

3 ♦ Press **TAB** twice to position the cursor in Field3. Notice the message that now appears at the bottom of the window:



4 ♦ Press **RETURN** to end the procedure.

5 ♦ Press **SPACEBAR** to return to the Procedure Editor.

The following code points out the new language element:

lt-03-02.p

```
DEFINE VARIABLE Field1 AS CHARACTER INITIAL "Hello".
DEFINE VARIABLE Field2 AS CHARACTER INITIAL "Progress".
DEFINE VARIABLE Field3 AS CHARACTER INITIAL "World!".

DISPLAY SKIP(3) Field1 SKIP(2) Field2 SKIP(2) Field3
  WITH SIDE-LABELS CENTERED NO-BOX.

ENABLE Field1 Field2 Field3.

/* TRIGGER */
ON ENTRY OF Field3
DO:
  MESSAGE "Press Return in Field3 to exit the procedure.".
END.

WAIT-FOR RETURN OF Field3.
```

Notice the new language element in the example, the ON statement. The ON statement sets up the trigger. In the example, ENTRY is the event and Field3 is the widget. (ENTRY is an event that occurs when the user or the application gives input focus to a widget.) The MESSAGE statement is the trigger that occurs when Field3 receives the ENTRY event.

NOTE: By default, window widgets reserve space at the bottom of the window to display messages from Progress and your application. The MESSAGE statement automatically outputs text enclosed in quotes to the message area.

This is a partial syntax for the ON statement. (For complete syntax descriptions, see the *Progress Language Reference*.)

SYNTAX

```
ON event-list  OF widget-list
  [ OR event-list OF widget-list ] ...
    trigger-block
```

The following table describes the elements of the ON statement:

Components	Description
`event-list`	A single event or a comma-separated list of events. One of the widgets specified must receive one of the events listed for the code in the trigger block to execute.
`widget-list`	A single widget or a comma-separated list of widgets. One of the widgets specified must receive one of the events listed for the code in the trigger block to execute.
`OR`	Use the OR phrase to create another widget-event pair that can execute the code in the trigger block.
`trigger-block`	A single 4GL statement or a block of statements that Progress executes when a specified widget receives a specified event. If the trigger block contains only one statement, then you don't have to enclose the statement in the DO. . .END syntax. For example: ON event OF widget MESSAGE "No DO syntax needed.".

Defining triggers is part of the basic Progress code template. Adding triggers to the basic template gives you this new coding template:

1. Define widgets.
2. Display widgets.
3. Enable widgets.
4. Define triggers.
5. Block execution.

3.1.4 Programming for Events

So far you've learned that the user interacts with an application through an interface made up of widgets. The user responds to the interface with events. A widget responds to an event by executing the programmed response, if one exists, followed by the Progress default response.

There is one more important development in the cycle of events and responses: you need a way to suppress default responses. Most of the time, you'll want to keep the default response, but you'll also encounter instances when you want to replace the default response with a programmed response.

To suppress a default response, you include the Progress statement RETURN NO-APPLY just before the END statement of your trigger. Later in the tutorial, you'll see examples of using RETURN NO-APPLY. The tutorial introduces it here because it is an important part of the event-response cycle, which Figure 3–3 illustrates.

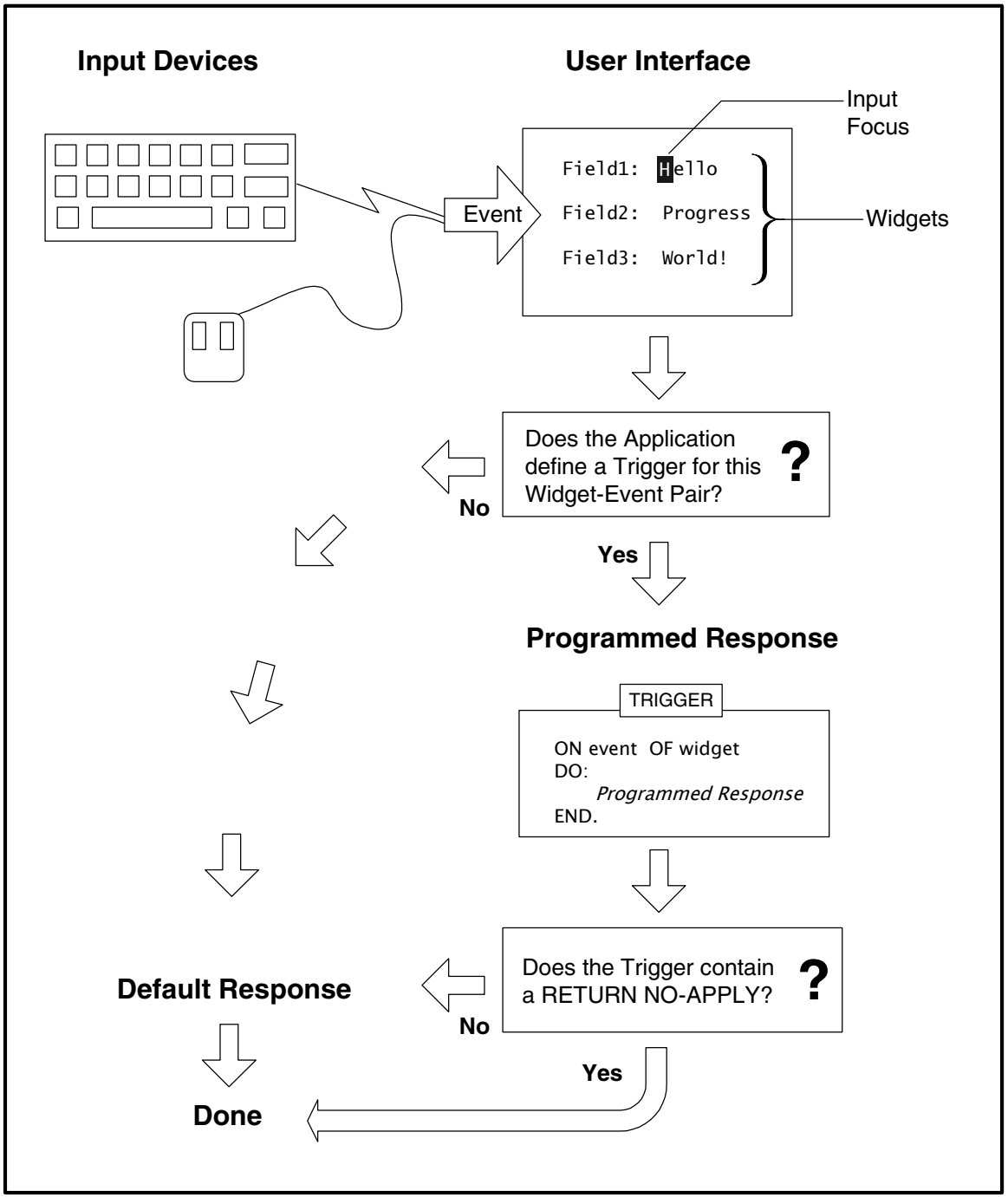


Figure 3–3: How an Application Responds to Events

The Progress programming model boils down to these main tasks:

1. Create your interface.

2. Code your triggers.

When you create the interface (define, display, and enable widgets), Progress automatically provides all the default behavior the user needs as they interact with that interface. The real work that your application does normally occurs as a response to a specific event from the user. That means that much of your functionality resides inside triggers. A good way to visualize this structure is to think of the code outside of the triggers as the code that creates and controls your interface. The code inside your triggers performs the tasks for which you created the application. The ON statement is the connection between the interface and the functionality inside the trigger.

3.2 Widgets

Since widgets make up the part of your application with which the end user interacts, it's important to thoroughly understand how to use each widget. Table 3–1 provides a brief description of each type of widget.

Table 3–1: Widget Types *(1 of 2)*

Widget	Use
Fill-in field	Displays and accepts any kind of data. When you display a variable or database field, the fill-in field is the default widget.
Text	Displays any kind of data as read-only text. The text widget is useful for creating lists, reports, or labels.
Selection list	Presents a list of value choices for a character variable or database field. A selection list can use scroll bars and allow multiple selections.
Combo box	Like a selection list, a combo box presents a list of value choices for a character variable or database field. The value list of a combo box is only visible after a user chooses the button next to the combo box. When the user selects a value, the combo box closes and displays only the selected value.
Editor	Displays and accepts as input character variables or database fields that can contain long strings. Use editor widgets where you want users to enter notes, descriptions, and so on.

Table 3–1: Widget Types *(2 of 2)*

Widget	Use
Radio set	Displays and accepts data that has a limited number of possible values. Use radio sets to display options where the user understands that only one option can be true at a time.
Toggle box	Displays and accepts data that has either a YES/NO or TRUE/FALSE value.
Browser	Displays and optionally updates key fields from a subset of database records. You define the subset of records and fields which the browse displays. The user interacts with a browse in the same way as a selection list.
Slider	Displays and manipulates numeric data. When the user moves the control inside the slider, the value changes proportionally to the move.
Rectangle	Creates a box. You might use a rectangle to group a set of widgets.
Image	Adds a high-resolution graphic to an interface. An image is a special widget available on GUI systems.
Button	Explicitly executes tasks. Use a button to allow the user to execute triggers or control the interface.
Frame	Creates a container for organizing and displaying other widgets.
Dialog box	Creates a frame that overlays the current interface. The main interface is disabled until the user is done working with the dialog box. A dialog box notifies the user of important information or requests more information. A dialog box is a container for other widgets.
Menu bar	Creates a collection of pull-down menus in a window. A menu bar can be attached only to a window widget.
Pull-down menu (submenu)	Organizes a related collection of commands and options for an application. A submenu can be attached to a menu bar, another widget, or a menu item.
Menu item	Creates an individual option on a submenu.

Since the goal of this section is to introduce widgets in general, you'll work with just four widgets: fill-in fields, text widgets, rectangle widgets, and button widgets. The tutorial covers other widgets in later chapters.

3.2.1 Widget Categories

All widgets fall into one of five categories that describe general function. These categories are:

- **Window Widgets** — Window widgets are workspace for your application. A window contains all other widgets. Character interfaces use one window, while graphical interfaces can use multiple windows.
- **Container Widgets** — Container widgets allow you to organize data, action, and graphic widgets. Container widgets include frames and dialog boxes.
- **Data Widgets** — Data widgets are the widgets you use to represent data from fields and variables. Data widgets include fill-in fields, text, editors, selection lists, combo boxes, radio sets, toggle boxes, sliders, and browse widgets.
- **Action Widgets** — Action widgets allow the user to direct the operation of your application. Menus provide groups of related processing options while buttons execute a single option. Buttons, menu bars, submenus, and menu items make up action widgets.
- **Graphic Widgets** — Graphic widgets help you decorate your interface. Use rectangles to make different kinds of borders. Images, which are available on graphical interface systems, allow you to include high-resolution graphics in your interface.

3.2.2 Defining Widgets

This section discusses the basic 4GL syntax that you use to define widgets. First, you'll learn how to represent a database field or a variable as a data widget. Next, you'll learn how to define buttons and rectangles. Finally, you'll see how to use a frame to arrange your widgets inside a window.

Defining Data Representations

Recall that Progress represents database fields and variables in an interface as a data widget. The widget and the field or variable are separate. You can think of a data widget as a suit of clothes that a database field or variable can put on or change out of as the need arises. In other words, in one part of your application you might represent a particular field or variable as a fill-in field, while in another part you might use an editor widget.

Before describing how you dress up a field or variable as a widget, you need to understand the syntax for defining variables. This is a partial syntax.

SYNTAX

```
DEFINE VARIABLE variable-name AS datatype
 [ INITIAL constant ]
 [ LABEL string ]
```

The following table describes the components:

Component	Description
variable-name	A name you supply to uniquely identify the variable. The variable name must begin with an alphabetic character (A–Z, a–z).
datatype	Describes the type of data stored in the variable. The tutorial thoroughly covers data types in later chapters. As you have already seen, the CHARACTER data type indicates text data, INTEGER indicates integer data, and LOGICAL indicates TRUE/FALSE or YES/NO data.
INITIAL constant	Specifies a constant value assigned to the variable at definition time.
LABEL string	Specifies a label that Progress uses when it displays the widget. If you do not specify a label, the variable name is the default label.

So, the DEFINE VARIABLE statement defines a variable and allows you to define the default data representation for that variable. The Progress language element for specifying data widgets is the VIEW-AS phrase. This is the general syntax.

SYNTAX

```
VIEW-AS widget-type [ options ]
```

The following table describes the elements of the VIEW-AS phrase:

Component	Description
`widget-type`	Specify the Progress keyword that indicates the appropriate data representation. The keywords are: FILL-IN TEXT SELECTION-LIST COMBO-BOX RADIO-SET TOGGLE-BOX EDITOR SLIDER
`options`	Specific widgets have options that let you define functional and display characteristics.

Note that VIEW-AS is a phrase, not a statement. This means that VIEW-AS can be attached to certain Progress statements to expand the function of the statement. When you attach the VIEW-AS phrase to a DEFINE VARIABLE statement, you are defining the default data representation of the variable as shown in this brief code example:

```
DEFINE VARIABLE Field1 AS CHARACTER VIEW-AS TEXT.
```

Whenever Progress displays Field1, it displays the variable as a text widget. (Text widgets are typically used for labels and reports.) If you do not use a VIEW-AS phrase in the DEFINE VARIABLE statement, Progress displays the variable as a fill-in field.

You don’t have to stick with the default data representation, however. You can override the default widget for a variable or a database field by attaching the VIEW-AS phrase to a run-time screen output statement, like this DISPLAY statement shows:

```
DISPLAY Field1 VIEW-AS FILL-IN.
```

Examine the code below, and load it into the Procedure Editor if you like:

lt-03-03.p

```
DEFINE VARIABLE Field1 AS CHARACTER INITIAL "Hello!" VIEW-AS TEXT.

DISPLAY SKIP(3) Field1 VIEW-AS FILL-IN
  WITH SIDE-LABELS NO-BOX CENTERED THREE-D.

ENABLE Field1.

WAIT-FOR RETURN OF Field1.
```

NOTE: The THREE-D option is relevant only on a Windows client; it is ignored by a character client.

When you run the `lt-03-03.p` procedure, you see a fill-in field because the VIEW-AS phrase on the DISPLAY statement overrides the default VIEW-AS phrase of the DEFINE VARIABLE statement.

Defining Buttons

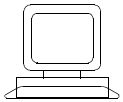
Button and rectangle widgets differ from data representations in that they are exclusively user interface components and are not related to database fields or variables. The button allows the user to execute a function directly and the rectangle is an interface decoration.

This is a partial syntax for the DEFINE BUTTON statement.

SYNTAX

```
DEFINE BUTTON button-name LABEL string
```

Typically, you associate a button with a trigger in your code. The button then becomes a tool that lets a user explicitly execute a trigger.

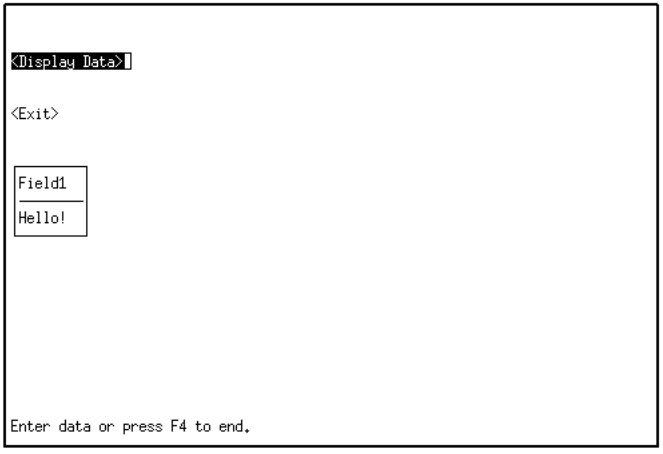


Exercise

Follow these steps for a demonstration of buttons and triggers:

1 ♦ Open `lt-03-04.p`.

2 ♦ Choose Compile→ Run. The interface, shown below, contains two buttons, Display Data and Exit:



3 ♦ Choose the Display Data button. The trigger executes on the CHOOSE event of the button widget, and displays Field1 as a text widget.

4 ♦ Press **TAB** to highlight the Exit button, then press **RETURN**.

5 ♦ Press **SPACEBAR** to return to the Procedure Editor.

Here is the code that created the display:

lt-03-04.p

```
    DEFINE VARIABLE Field1 AS CHARACTER INITIAL "Hello!" VIEW-AS TEXT.
    DEFINE BUTTON btn-Exit LABEL "Exit".
    DEFINE BUTTON btn-Data LABEL "Display Data".

    ENABLE SKIP(2) btn-Data SKIP(2) btn-Exit SKIP(2)
        WITH SIDE-LABELS NO-BOX THREE-D.

    ON CHOOSE OF btn-Data
    DO:
        DISPLAY Field1.
    END.

/*1*/  WAIT-FOR CHOOSE OF btn-Exit.
```

The Exit button does not have a trigger associated with it. That's because the procedure uses the CHOOSE event of btn-Exit to signal the unblocking response. Since this is a default response, the procedure doesn't require a trigger and a programmed response.

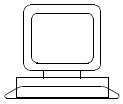
Defining Rectangles

The basic syntax for defining a rectangle lets you decide the size of the rectangle, whether it's filled or empty, and how wide the edge is.

This is a partial syntax for the DEFINE RECTANGLE statement.

SYNTAX

```
DEFINE RECTANGLE rectangle-name SIZE width BY height
  [ NO-FILL ]
  [ EDGE-CHARS width ]
```

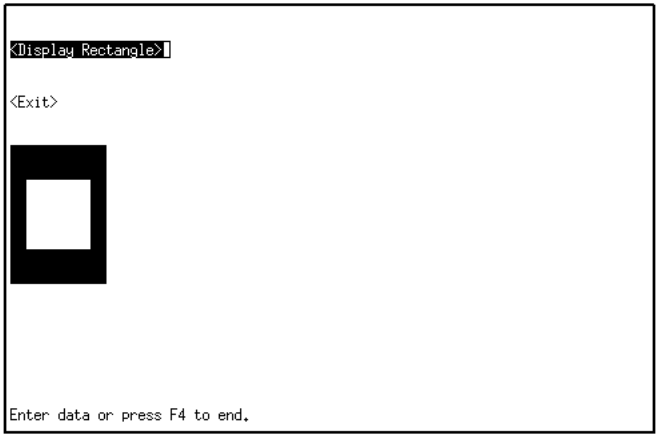


Exercise

The steps below use the same code as the last exercise, modified to use a DEFINE RECTANGLE statement:

1 ♦ Open `lt-03-05.p`.

2 ♦ Choose Compile→ Run. The interface, shown below, contains two buttons: Display Rectangle and Exit.



3 ♦ Choose the Display Rectangle button. The rectangle appears.

4 ♦ Press **TAB** to highlight the Exit button and press **RETURN** to exit the procedure.

5 ♦ Press **SPACEBAR** to return to the Procedure Editor.

Here is the code that created the display:

lt-03-05.p

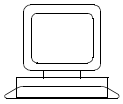
```
/*1*/  DEFINE RECTANGLE Rect1 SIZE-CHARS 12 BY 8 NO-FILL EDGE-CHARS 2.
    DEFINE BUTTON btn-Exit LABEL "Exit".
    DEFINE BUTTON btn-Rect LABEL "Display Rectangle".

    ENABLE SKIP(1) btn-Rect SKIP(1) btn-Exit SKIP(1)
        WITH SIDE-LABELS NO-BOX THREE-D.

    ON CHOOSE OF btn-Rect
    DO:
        DISPLAY Rect1 WITH NO-BOX.
    END.

    WAIT-FOR CHOOSE OF btn-Exit.
```

As shown at point 1, the size of the rectangle is 12 columns (width) by 8 rows (height). The NO-FILL option makes the rectangle empty. The EDGE-CHARS option specifies a border that's two characters wide. The trigger executes on the CHOOSE event of the btn-Rect widget, and displays a rectangle.

Defining Frames

A frame is a container for widgets. Frames allow you to organize data, action, and graphic widgets. Data, action, and graphic widgets that can be placed in a frame (or dialog box) are called *field-level widgets*.

You cannot place widgets in a window unless they are first contained in a frame widget. (One exception: menu bars go directly into the window.) If you don't specify a frame, Progress creates one for you. This frame is called the *default frame*.

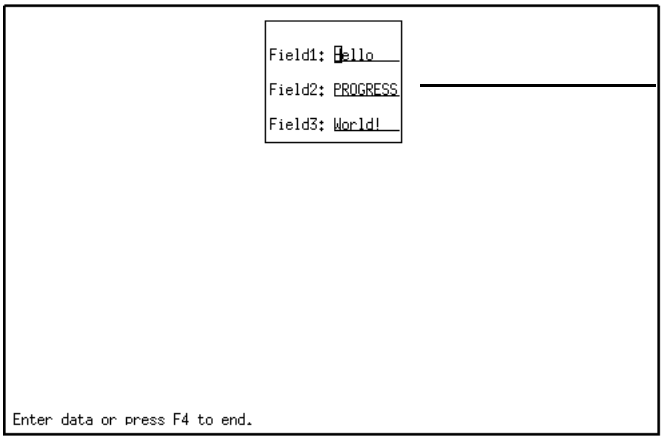


Exercise

Follow these steps for a demonstration of the default frame:

1 ♦ Open `lt-03-06.p`.

2 ♦ Choose Compile→ Run. The interface you see is the same "Hello Progress World!" interface, except that a graphic border surrounds the fill-in fields. The graphic border shows the area of the default frame follows:



Default frame graphic

3 ♦ Move input focus to Field3 and press **RETURN** to end the procedure.

4 ♦ Press **SPACEBAR** to return to the Procedure Editor.

Here is the code that created the display:

lt-03-06.p

```
    DEFINE VARIABLE Field1 AS CHARACTER INITIAL "Hello".
    DEFINE VARIABLE Field2 AS CHARACTER INITIAL "Progress".
    DEFINE VARIABLE Field3 AS CHARACTER INITIAL "World!".

    DISPLAY SKIP(1) Field1 SKIP(1) Field2 SKIP(1) Field3
/*1*/    WITH SIDE-LABELS CENTERED THREE-D.

    ENABLE Field1 Field2 Field3.

    ON ENTRY OF Field3
    DO:
      MESSAGE "Press Return in Field3 to exit the procedure.".
    END.

    WAIT-FOR RETURN OF Field3.
```

In the code you saw earlier, the DISPLAY statement included the NO-BOX option, which suppressed the default frame border. Since the option is not included here, as shown at point 1, the border appears.

The DEFINE FRAME statement allows you to name a frame and list the field-level widgets it contains. You also have several options for controlling the arrangement of the widgets within the frame, such as the SKIP option, which you have already seen.

Here is a partial syntax.

SYNTAX

```
DEFINE FRAME frame-name
  form-item ...
    [
      WITH
        [ SIDE-LABELS ]
        [ NO-BOX ]
        [ CENTERED ]
    ]
```

The following table describes the elements of the DEFINE FRAME statement:

Component	Description
form-item	Form items can include database fields, variables that have been previously defined, buttons, rectangles, and SKIP options.
Frame phrase (WITH)	The options shown at the bottom of the syntax diagram make up what is called the frame phrase. The frame phrase, which begins with WITH, lets you specify options that affect the frame itself or all the widgets in the frame as a group.

The frame phrase has other important applications outside of the DEFINE FRAME statement. The frame phrase can be attached to statements that output data to the screen to:

- Specify a particular frame to use
- Modify the characteristics of a frame

This is a partial syntax for the frame phrase.

SYNTAX

```
WITH FRAME frame-name
  [ SIDE-LABELS ]
  [ NO-BOX ]
  [ CENTERED ]
```

In the following example, the DISPLAY statement uses a frame phrase to alter the characteristics of a default frame:

```
DISPLAY Field1 WITH SIDE-LABELS NO-BOX CENTERED.
```

When you use a frame phrase without specifying a frame name, Progress applies the phrase to the default frame.

Defining Frame Families

A parent is a widget that contains or owns other widgets. In addition to parenting frames to windows, as described in the previous section, you can also parent frames to other frames. When you parent a frame to another frame, you create a *frame family*. A frame that is parented by another frame is called a *child frame* and a frame that parents another frame is called a *parent frame*. Frames that are parented to the same frame, in the same generation, are called *sibling frames*.

By defining a frame family, you create a relationship among the descendant frames that is different from that of frames parented to the same window. For example, you can:

- Tab among frames in the same family
- Control the hiding and viewing of child frames through the parent, or independently
- Use default or cancel buttons from any other frame in the same family
- Parent frames to a dialog box

Child frames behave like field-level widgets of the parent frame. Child frames must be positioned physically within the parent frame. When you specify a location of a child frame, the options you specify are with respect to the parent frame, not the window. However, the field-level widgets of child frames are not controlled by the parent frame. You must explicitly display, enable, and disable field-level widgets of child frames.

To define a frame family, set the FRAME attribute of the child frame to the widget handle of the parent frame. As shown in this example, frame child1 is a child frame of frame parent1 and button btn-Exit is a field-level widget of frame child1:

```
DEFINE BUTTON btn-Exit LABEL "Exit".
DEFINE FRAME parent1 WITH TITLE "parent1" SIZE 40 BY 10
  AT ROW 3 COLUMN 3.
DEFINE FRAME child1 btn-Exit WITH TITLE "child1" SIZE 20 BY 5
  AT ROW 3 COLUMN 3.

FRAME child1:FRAME = FRAME parent1:HANDLE.

DISPLAY WITH FRAME parent1.
ENABLE ALL WITH FRAME parent1.
ENABLE ALL WITH FRAME child1.

WAIT-FOR CHOOSE OF btn-Exit.
```

You can use frame families to create useful interactions among frames, but they do carry some restrictions that are important to remember. For example:

- Child frames cannot be down frames. (Down frames are frames that display multiple records, one per line, one after another.)
- Child frames do not inherit the attributes of their parent frames.
- Child frames must appear within the display area of their parent frame.

3.2.3 Widget Attributes

Each widget has a set of values that Progress uses to define the widget and change it during run time. These attributes define all the characteristics of a widget that you can read or set in your procedures. Attributes define the widget's:

- Location on the screen, or geometry
- Appearance
- Functionality
- Relationships to other widgets

When you first create a widget, Progress automatically determines many of the attributes. Progress defines or changes other attributes during run time.

An attribute has a keyword identifier, a data type, and in some cases, a default value. For example, every widget has the VISIBLE attribute. The VISIBLE attribute is a logical (TRUE/FALSE) value. When VISIBLE is TRUE, the widget is visible on screen. When VISIBLE is FALSE, it is not visible to the user. When you execute a DISPLAY statement, Progress sets the VISIBLE attribute to TRUE.

The remaining two properties of an attribute describe how you can access it. Some widget attributes are readable, some are settable, and some are both. *Readable* means that your application can read the current value and assign that value to a variable. *Settable* means that your application can assign a new value to the attribute.

Table 3–2 describes some attributes that you'll use frequently. The table also lists the data type of each widget and introduces a new data type: WIDGET-HANDLE. Basically, a widget handle is an internal identifier for a Progress widget. Most of the time, you can work with a widget by referencing the name of the variable or database field it is associated with. At other times, you'll need to provide the WIDGET-HANDLE. The tutorial describes how to use widget handles in later chapters.

Table 3–2: Commonly Used Attributes *(1 of 2)*

Attribute	Data Type	Description
HANDLE	WIDGET-HANDLE	Contains the WIDGET-HANDLE of the widget. You can read this attribute.
VISIBLE	LOGICAL	Defines whether the widget is visible to the user. You can read and set this attribute. The default value is FALSE.
HIDDEN	LOGICAL	Identifies whether or not Progress can display the widget through default behavior. In other words, when TRUE, the widget can only be made visible through an explicit 4GL statement. You can read and set this attribute. The default value is FALSE.
SENSITIVE	LOGICAL	Defines whether the widget can receive input focus. You can read and set this attribute. The default value is FALSE.
SCREEN-VALUE	CHARACTER	Contains the value that is currently displayed on the screen. You can read and set this attribute.

Table 3–2: Commonly Used Attributes *(2 of 2)*

Attribute	Data Type	Description
ROW	DECIMAL	Identifies the row location of the upper left corner of the widget. You can read and set this attribute.
COL	DECIMAL	Identifies the column location of the upper left corner of the widget. You can read and set this attribute.
FIRST-CHILD	WIDGET-HANDLE	Identifies the widget handle of the first widget contained in the window or container widget. You can read this attribute.
NEXT-SIBLING	WIDGET-HANDLE	Identifies the next widget in the chain of widgets in a window or container widget. You can read this attribute.

Typically, the default attribute values assigned implicitly by Progress statements provide the appearance and functionality you want in your widgets. Later, you'll learn how to read and set attributes directly.

3.3 Event Types

Earlier, the tutorial defined any user input as an event. All keystrokes are events. All mouse interactions are events. It's time to expand the definition of events. Progress recognizes two types of events: event actions and event functions. An *event action* is any simple user interaction, like a single keystroke or a simple mouse interaction. An *event function* is an abstraction of one or more event actions.

For example, on some platforms **F1** invokes the help system. On other platforms **F2** invokes the help system. **F1** and **F2** are both event actions, but both invoke the same function (in this example). Instead of writing code for the event action, you can write equivalent code using the Progress keyword that specifies the function. In this case, the keyword is HELP. HELP is an event function:

```
ON F1 OF widget-name
DO:
    /* Code */
END.

ON HELP OF widget-name
DO:
    /* Code */
END.
```

The first trigger relies on the event action, while the second trigger relies on the event function. For any particular widget, you can write a trigger for either the event action or the event function, but not for both. However, you should write your triggers for event functions wherever possible because:

- Your code is more readable.
- Your code is more portable. In the example above, writing a trigger for **F1** would not work on some platforms. Writing a trigger for HELP works everywhere.

For these reasons, this tutorial uses only event functions.

For every type of widget, Progress has a default response for every event that the widget can receive. This means that when you display widgets, Progress takes care of much of the normal activity the user would have with a widget. The events that you want to keep track of explicitly are those that:

- Execute a trigger
- Match a WAIT-FOR condition

You've already seen the syntax for the ON statement, which handles events that execute trigger code. Now, examine this partial syntax for the WAIT-FOR statement.

SYNTAX

```
WAIT-FOR event-list OF widget-list
  [ OR event-list OF widget-list ] ...
```

The WAIT-FOR statement translates to "on this widget-event pair, unblock execution." The result is that the user moves into another part of the procedure or completes the procedure.

Important Event Functions

Table 3–3 defines some important event functions that you'll use frequently.

Table 3–3: Important Event Functions

Function	Description
ENTRY	Any action by the user or the application that gives a widget input focus.
LEAVE	Any action by the user or the application that takes input focus from a widget.
GO	Any action that signals Progress to accept new data and continue processing.
CHOOSE	Any action that chooses a button or menu item.

Event Categories

There are literally hundreds of events, so the Progress documentation uses some terms to refer to groups of related events. Table 3–4 describes the categories.

Table 3–4: Event Categories

Category	Description
Universal key functions	These events are "universal" to all widgets, except menu widgets. They tend to be basic event functions, like **GO** or **HELP**.
Navigation key functions	These events are the keystrokes that let you move input focus in an interface, like **TAB** or **BACK-TAB**.
Field-editing key functions	These events are the keystrokes that let you manipulate text inside a widget, like **BACKSPACE** or **DELETE**.
High-level Progress event functions	These are the most important event functions in Progress, like LEAVE and ENTRY. They represent the easiest way to connect triggers with widgets. For the most part, the tutorial uses high-level events because they make readable, portable code.

3.4 Putting the Progress Programming Model to Work

Just when you were progressing nicely, your managers at All Around Sports stop by to see you. They're not too familiar with computers, and they have a request to make: "Can you show us an object interface? It doesn't have to work, we're just curious."

You go back and review the template that shows the parts of a Progress procedure. You've learned about frames since the last time you used the template, and know that you'll need to use a defined frame to organize your field-level widgets more precisely. So, the old first step, define widgets, breaks down into two new steps: define field-level widgets and define frames.

The new coding template has the following sections:

1. Define field-level widgets.
2. Define frames.
3. Define triggers.
4. Display widgets.
5. Enable widgets.
6. Block execution.
7. Disable user interface (end the procedure).

Sometimes traditional programming is called *top-down programming*, because you define larger structures first and then define smaller and smaller components. In some ways, you can think of event-driven programming as *bottom-up programming*, because you define the smaller components before you define the larger components. The template above illustrates this by placing field-level widgets before frames. If you reversed the order, the code would not compile—Progress needs to define the small components before it can construct a larger containing component.

Similarly, when you rely on default frames, Progress cannot interpret triggers that reference the frames until the main code block compiles. Therefore, when you first learned about triggers, they came after your DISPLAY and ENABLE statements. More properly, triggers should come before the main code block, as shown in the coding template above. Since you will almost always use defined frames, you won't run into any conflicts.

It's time to put this template to work.

Envisioning an Interface

You decide that creating the main interface of the sales and inventory application is enough to satisfy the curiosity of your bosses. What should that first screen consist of? You make a short list:

- A banner that identifies the application
- A field for users to enter their initials
- Buttons that launch the major sections of the application (for now, Customer, Order Entry, and Inventory)
- A button to exit the application

Figure 3–4 shows the interface you'll try to create.



Figure 3–4: A Simple Interface

Now that you've envisioned the interface, you can start coding the sections of the procedure to make it a reality.

Defining Field-level Widgets

You create the following code to define all of your field-level widgets. All of this code can be found in the file `lt-03-07.p`.:

```
/*1*/  DEFINE RECTANGLE rct-Border SIZE-CHARS 40 BY 7 NO-FILL
         EDGE-CHARS 1.

/*2*/  DEFINE VARIABLE Name1 AS CHARACTER FORMAT "x(19)"
         INITIAL "A L L   A R O U N D" VIEW-AS TEXT.
       DEFINE VARIABLE Name2 AS CHARACTER FORMAT "x(19)"
         INITIAL "    S P O R T S    " VIEW-AS TEXT.

/*3*/  DEFINE VARIABLE Username AS CHARACTER FORMAT "x(3)"
         LABEL "Initials".

/*4*/  DEFINE BUTTON btn-Cust LABEL "Customers".
       DEFINE BUTTON btn-Order LABEL "Order Entry".
       DEFINE BUTTON btn-Inv LABEL "Inventory".

/*5*/  DEFINE BUTTON btn-Exit LABEL "Exit".
```

These notes explain how field-level widgets serve various functions in this application:

1. Use the rectangle widget to create a large (SIZE-CHARS 40 BY 7) empty (NO-FILL) box with a one-character wide edge (EDGE-CHARS). The box makes the application title stand out.

2. Create two variables to hold the application title. The FORMAT "x(19)" option specifies how many characters of display space Progress should allow the text widgets to occupy. The default is 8. You define these variables as text widgets because you are using the values as a title. The variables are read-only text.

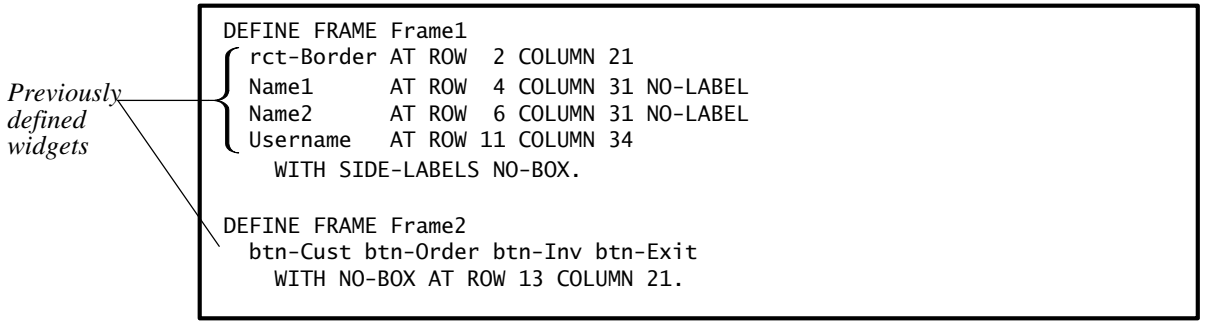
3. Create another variable to hold the initials of the user. Here, you need a fill-in field, but you don’t have to specify it as such because fill-in field is the default.

4. Create three buttons that launch various functions of a sales and inventory application.

5. Create the Exit button.

Define Frames

Your next step is to organize your widgets into frames. While you typically need only one frame per interface, here you use two because you don't want the buttons to appear until the user has entered valid initials in the fill-in field. Here is the code:



With the DEFINE FRAME statement, you list the widgets you want contained in each frame. In the first frame, you want precise placement of each widget. The AT phrase allows you to place a widget at a specific ROW and COLUMN within the frame, not the window. Here, you place the title variables inside the rectangle widget and you place the fill-in field below the rectangle. In the second frame, the default spacing works fine, so you don't need to use the AT phrase.

Define Triggers

You only need one trigger for now. When the user chooses any of the function buttons, you want to display the message "Sorry, that function is not yet implemented." Here is the code:

```
ON CHOOSE OF btn-Cust, btn-Order, btn-Inv
DO:
  HIDE MESSAGE NO-PAUSE.
  MESSAGE "Sorry, that function is not yet implemented.".
END.
```

As you can see, one ON statement can apply to many widgets. It can also apply to many events. The HIDE MESSAGE NO-PAUSE statement clears the default message area without pausing. It's a good habit to always use this statement before sending a new message.

Displaying, Enabling, and Blocking

Here is the code that displays, enables, and blocks the interface:

```
/*1*/  DISPLAY Name1 NO-LABEL Name2 NO-LABEL Username WITH FRAME Frame1.

/*2*/  ENABLE Username WITH FRAME Frame1.

/*3*/  WAIT-FOR RETURN OF Username.

/*4*/  DISABLE Username WITH FRAME Frame1.

/*5*/  ENABLE ALL WITH FRAME Frame2.

/*6*/  WAIT-FOR CHOOSE OF btn-exit.
```

Here are descriptions of each step:

1. The DISPLAY statement displays the title variables without labels (NO-LABEL). Next it displays the fill-in field. Finally, by using the frame phrase (WITH FRAME), you tell Progress to use a frame you've already defined.

2. Only the fill-in field in the frame is meant to have user interaction, so you enable it.

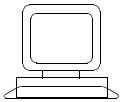
3. You block execution until the user presses **RETURN** in the fill-in field.

4. When the user presses **RETURN**, the application executes the code that follows the WAIT-FOR statement. Here, you disable the fill-in field from receiving further input.

5. This statement serves to more precisely illustrate the function of the DISPLAY statement. As you can see, you do not need a DISPLAY statement to make the buttons appear-enabling the buttons makes them appear. Basically, you use the DISPLAY statement with data widgets to display both the widget and the underlying value of the variable or database field that the widget represents. If you had skipped the DISPLAY statement for the first frame, the values of the text widgets, "All Around" and "Sports", would not appear.

   The ALL keyword tells Progress to enable all the widgets in the defined frame.

6. You establish the condition that signals the end of the application—when the user chooses the Exit button.



Exercise

It's time to put all your hard work together and view the results. Perform these steps:

1 ♦ Open `lt-03-07.p`.

2 ♦ Choose Compile→ Run.

3 ♦ Type your initials in the fill-in field and press **RETURN**. A row of buttons appears.

4 ♦ Choose one of the buttons. The "Sorry, that function is not yet implemented" message appears in the default message area.

5 ♦ Choose Exit to end the procedure.

6 ♦ Press **SPACEBAR** to return to the Procedure Editor.

The complete procedure is shown below:

lt-03-07.p

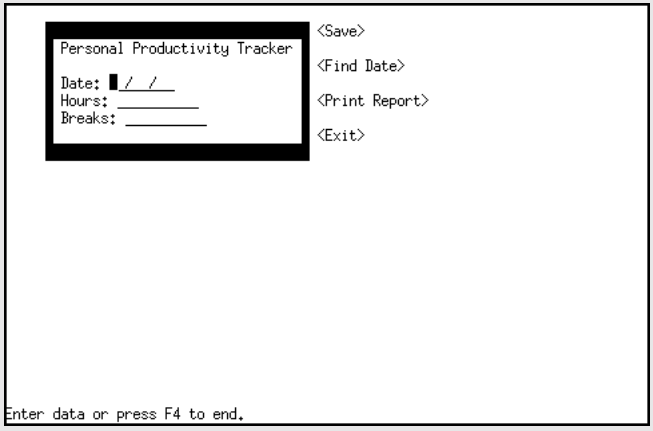
```
/**********  DEFINE FIELD-LEVEL WIDGETS  **********/
DEFINE RECTANGLE rct-Border SIZE-CHARS 40 BY 7 NO-FILL.
DEFINE VARIABLE Name1 AS CHARACTER FORMAT "x(19)"
    INITIAL "A L L   A R O U N D" VIEW-AS TEXT.
DEFINE VARIABLE Name2 AS CHARACTER FORMAT "x(19)"
    INITIAL "    S P O R T S    " VIEW-AS TEXT.
DEFINE VARIABLE Username AS CHARACTER FORMAT "x(3)" LABEL "Initials".
DEFINE BUTTON btn-Cust LABEL "Customers".
DEFINE BUTTON btn-Order LABEL "Order Entry".
DEFINE BUTTON btn-Inv LABEL "Inventory".
DEFINE BUTTON btn-Exit LABEL "Exit".

/**********  DEFINE FRAMES  **********/
DEFINE FRAME Frame1
    rct-Border AT ROW  2 COLUMN 21
    Name1      AT ROW  4 COLUMN 31
    Name2      AT ROW  6 COLUMN 31
    Username   AT ROW 11 COLUMN 34
        WITH SIDE-LABELS NO-BOX THREE-D.

DEFINE FRAME Frame2
    btn-Cust btn-Order btn-Inv btn-Exit
        WITH NO-BOX AT ROW 13 COLUMN 21 THREE-D.

/**********  DEFINE TRIGGERS  **********/
ON CHOOSE OF btn-Cust, btn-Order, btn-Inv
DO:
    HIDE MESSAGE NO-PAUSE.
    MESSAGE "Sorry, that function is not yet implemented.".
END.

/**********  MAIN LOGIC  **********/
DISPLAY Name1 NO-LABEL Name2 NO-LABEL Username WITH FRAME Frame1.
ENABLE Username WITH FRAME Frame1.
WAIT-FOR RETURN OF Username.
DISABLE Username WITH FRAME Frame1.ENABLE ALL WITH FRAME Frame2.
WAIT-FOR CHOOSE OF btn-Exit.
```

Practice Problems

If you feel like you need some practice creating interfaces, complete the exercise below. The filename listed next to the problem contains a sample solution. You can load the file into the Procedure Editor.

Problem 3-1: `lt-03-s1.p`

The procedure you just worked with, `lt-03-07.p`, creates a complete interface that follows the Progress programming model. Using the same techniques, you should be able to duplicate the interface shown below.



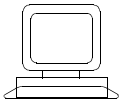
3.5 Making Your Interface Responsive

Defining and displaying an interface doesn't complete the task of making your application responsive. As you get input from the user, you'll want to change the interface in subtle or dramatic ways to guide your users and present them with attractive and intuitive ways to interact with your application. This section describes the most important ways you work with an interface during run time. You need to learn how to:

- Enable and disable widgets
- View and hide widgets
- Access widget attributes

3.5.1 Enabling and Disabling Widgets

As you've seen in previous examples, you use the ENABLE and DISABLE statements to turn widgets on and off. Typically, when you present an interface, you enable all the widgets in that interface. At other times, some widgets don't have to be enabled unless special conditions are met.

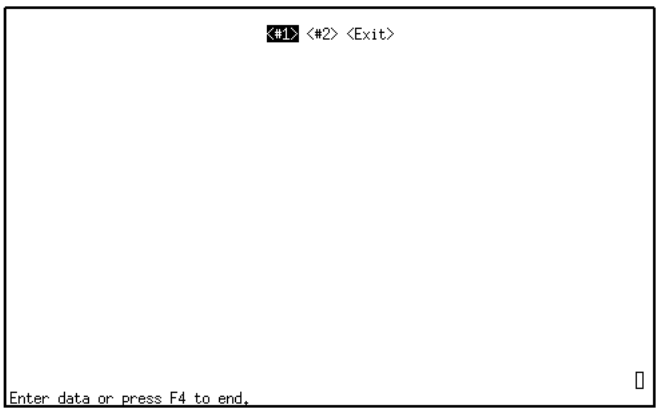


Exercise

Follow these steps for a demonstration of how you can enable and disable widgets during run time:

1 ♦ Open `lt-03-08.p`.

2 ♦ Choose Compile→ Run. The interface shown below appears. Note that buttons #1 and Exit are enabled, and #2 is disabled:



3 ♦ Choose the first button. The second button becomes enabled and the first becomes disabled.

4 ♦ Choose the second button. The first button becomes enabled again and the second button becomes disabled. You can repeat this process as many times as you like.

5 ♦ Choose Exit to end the procedure.

6 ♦ Press **SPACEBAR** to return to the Procedure Editor.

Here is the code that created the display:

lt-03-08.p

```
    /**********  DEFINE FIELD-LEVEL WIDGETS  **********/
    DEFINE BUTTON btn-One LABEL "#1".
    DEFINE BUTTON btn-Two LABEL "#2".
    DEFINE BUTTON btn-Exit LABEL "Exit".

    /**********  DEFINE FRAMES  **********/
    DEFINE FRAME Frame1
      SKIP(2) btn-One btn-Two btn-Exit
        WITH NO-BOX CENTERED THREE-D.

    /**********  DEFINE TRIGGERS  **********/
    ON CHOOSE OF btn-One
    DO:
/*1*/  ENABLE btn-Two WITH FRAME Frame1.
       DISABLE btn-One WITH FRAME Frame1.
    END. /* ON CHOOSE OF btn-One */
    ON CHOOSE OF btn-Two
    DO:
/*2*/  ENABLE btn-One WITH FRAME Frame1.
       DISABLE btn-Two WITH FRAME Frame1.
    END. /* ON CHOOSE OF btn-Two */

    /**********  MAIN LOGIC  **********/
    ENABLE btn-One btn-Exit WITH FRAME Frame1.
    WAIT-FOR CHOOSE OF btn-Exit.
```

The effect of the ENABLE and DISABLE statements at points 1 and 2 is that only one of the buttons can be active at a time. By selectively enabling and disabling parts of the interface, you help the user focus on the parts of the interface that are important at a given moment.

This is a partial syntax for the ENABLE statement.

SYNTAX

```
ENABLE { ALL | SKIP [ (n) ] | widget-list }
 { [ frame-phrase ] }
```

This is a partial syntax for the DISABLE statement.

SYNTAX

```
DISABLE { ALL | widget-list } { [ frame-phrase ] }
```

The following table describes the important elements of the ENABLE and DISABLE statements:

Element	Description
ALL	Applies the statement to every widget in the specified frame.
frame-phrase	Specifies the frame where the targeted widgets exist. Even when Progress does not require it, it is good programming practice to specify the frame.

3.5.2 Viewing and Hiding Widgets

There's some important default Progress behavior that determines when widgets are visible and when they are not. This section outlines Progress's default behavior so you know when to rely on it. This section also introduces the VIEW and HIDE statements, which allow you to control the display of widgets.

Default Viewing and Hiding

When you use DISPLAY and ENABLE statements, Progress takes care of making the appropriate widgets visible. When you DISPLAY any widget, all the widgets in the widget's frame become visible. When you ENABLE a widget, all the widgets in the frame become visible, although Progress only enables the widget you specified. Progress's default viewing and hiding behavior concentrates on displaying and hiding frames of widgets, not individual widgets.

Viewing and Hiding Widgets

You can explicitly view and hide frames or view and hide widgets within frames by using the VIEW and HIDE statements. This is a partial syntax for the VIEW statement.

SYNTAX

```
VIEW widget-list
```

This is a partial syntax for the HIDE statement.

SYNTAX

```
HIDE [ widget-list | MESSAGE | ALL ] [ NO-PAUSE ]
```

The following table describes the new language elements:

Element	Description
MESSAGE	You can also use the HIDE statement to hide the contents of the default message area.
NO-PAUSE	Use the NO-PAUSE option to prevent Progress from pausing before hiding data.

To hide or view a frame, substitute the keyword FRAME and frame name for `widget-list`:

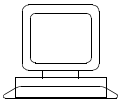
```
VIEW FRAME Frame1.
HIDE FRAME Frame2.
```

To hide or view one or more widgets in a frame, specify a list of widgets and, if necessary, append the IN FRAME `frame-name` syntax to individual widget references:

```
VIEW Widget1 IN FRAME Frame1 Widget2 IN FRAME Frame2.
HIDE Widget1 IN FRAME Frame1 Widget2 IN FRAME Frame2.
```

Up to now you used the frame phrase, which begins with WITH, to specify a frame. Here you use the IN FRAME syntax. The difference is that WITH is used to create or modify a frame and its attributes, while IN FRAME merely references an existing frame.

One very important thing to remember about using VIEW and HIDE is that they do not affect the enabled status of a widget. If a widget is enabled when you hide it, it will still be enabled when you view it again. Enabling and viewing are separate functions.

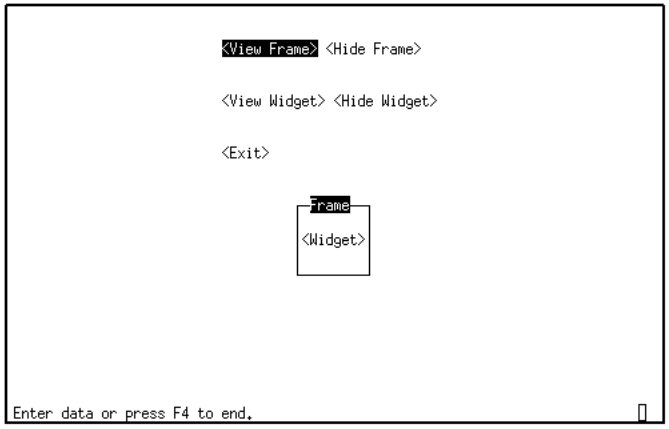


Exercise

Follow these steps to demonstrate both uses of HIDE and VIEW.

1 ♦ Open `lt-03-09.p`.

2 ♦ Choose Compile→ Run. The visible interface consists of buttons and a small frame with its border showing and a widget within the frame. All the HIDE and VIEW statements will operate on this small frame and the widget within it:



3 ♦ Choose Hide Frame. The small frame disappears.

4 ♦ Choose View Frame. The frame reappears.

5 ♦ Choose Hide Widget. The widget inside the small frame disappears.

6 ♦ Choose View Widget. The widget inside reappears.

7 ♦ Choose Hide Widget again.

8 ♦ Choose Hide Frame and then View Frame. Notice that the widget inside the small frame is not visible. When you explicitly hide a widget within a frame, viewing and hiding the containing frame does not affect the hidden widget.

9 ♦ Choose View Widget. Now the hidden widget is visible.

10 ♦ Choose Exit to end the procedure.

11 ♦ Press **SPACEBAR** to return to the Procedure Editor.

Here is the code that created the display:

lt-03-09.p

```
    /**********  DEFINE FIELD-LEVEL WIDGETS  **********/
    DEFINE BUTTON btn-View-F LABEL "View Frame".
    DEFINE BUTTON btn-Hide-F LABEL "Hide Frame".
    DEFINE BUTTON btn-View-W LABEL "View Widget".
    DEFINE BUTTON btn-Hide-W LABEL "Hide Widget".
    DEFINE BUTTON btn-Widget LABEL "Widget".
    DEFINE BUTTON btn-Exit LABEL "Exit".

    /**********  DEFINE FRAMES  **********/
    DEFINE FRAME Frame1
      SKIP(1) btn-View-F btn-Hide-F SKIP(1)
      btn-View-W btn-Hide-W SKIP(1)
      btn-Exit SKIP(1)
         WITH NO-BOX CENTERED THREE-D.
    DEFINE FRAME Frame2
      SKIP(1) btn-Widget SKIP(1)
/*1*/    WITH TITLE "Frame" CENTERED THREE-D.

    /**********  DEFINE TRIGGERS  **********/
    ON CHOOSE OF btn-View-F
    DO:
/*2*/    VIEW FRAME Frame2.
    END. /* ON CHOOSE OF btn-View-F */
    ON CHOOSE OF btn-Hide-F
    DO:
/*3*/    HIDE FRAME Frame2.
    END. /* ON CHOOSE OF btn-Hide-F */
    ON CHOOSE OF btn-View-W
    DO:
/*4*/    VIEW btn-Widget IN FRAME Frame2.
    END. /* ON CHOOSE OF btn-View-W */
    ON CHOOSE OF btn-Hide-W
    DO:
/*5*/    HIDE btn-Widget IN FRAME Frame2.
    END. /* ON CHOOSE OF btn-Hide-W */

    /**********  MAIN LOGIC  **********/
    ENABLE btn-View-F btn-Hide-F btn-View-W btn-Hide-W btn-Exit
        WITH FRAME Frame1.
    ENABLE btn-Widget WITH FRAME Frame2.
    WAIT-FOR CHOOSE OF btn-Exit.
```

These notes help explain the code:

1. When you display the default border of a frame, you can use the TITLE option to give the whole frame a label.

2. The first trigger views the frame. You must include the keyword FRAME or Progress will attempt to find a field-level widget with the specified name.

3. As with VIEW, the HIDE statement only requires a valid frame reference.

4. Here, you are viewing a widget within a frame. You need to specify the widget and the frame reference with IN FRAME.

5. Hiding a specific widget also requires both widget and frame references.

3.5.3 Accessing Widget Attributes and Methods

Usually, the default attribute values assigned implicitly by Progress statements provide the appearance and functionality you want in your widgets. However, Progress also supports a syntax that lets you access attribute information directly. This syntax is shown below.

SYNTAX

```
ASSIGN widget-name:attribute-name
```

To display an attribute's value, all you have to do is reference it in a DISPLAY statement, as shown below:

```
DISPLAY widget-name:attribute-name WITH Frame1.
```

You can also assign widget attribute values to variables, as long as the variable is of the same data type, using the following syntax.

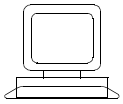
SYNTAX

```
ASSIGN var-name = widget-name:attribute-name
```

To set an attribute, you assign the new value to the attribute, but you must also specify the frame in which the widget resides. You use the IN FRAME phrase to identify the containing frame. Here is the syntax.

SYNTAX

```
ASSIGN widget-name:attribute-name IN FRAME frame-name = value
```



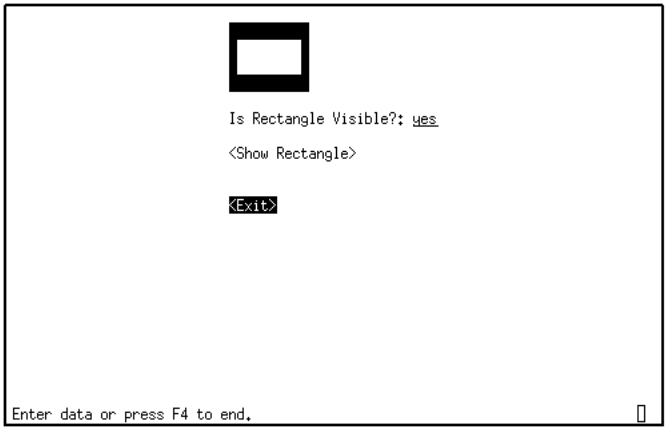
Exercise

Follow these steps for a demonstration of widget attributes:

1 ♦ Open `lt-03-10.p`.

2 ♦ Choose Compile→ Run. You see a fill-in field with the label “Is Rectangle Visible?” and the value no.

3 ♦ Choose the Show Rectangle button. A rectangle becomes visible and the value of the fill-in field becomes yes, as shown below:



4 ♦ Choose Exit to end the procedure.

5 ♦ Press **SPACEBAR** to return to the Procedure Editor.

This is the code that created the display:

lt-03-10.p

```
    /**********  DEFINE FIELD-LEVEL WIDGETS  **********/
    DEFINE RECTANGLE Rect1 SIZE-CHARS 10 BY 4 EDGE-CHARS 1.
    DEFINE VARIABLE Field1 AS LOGICAL LABEL "Is Rectangle Visible?".
    DEFINE BUTTON btn-Rect LABEL "Show Rectangle".
    DEFINE BUTTON btn-Exit LABEL "Exit".

    /**********  DEFINE FRAMES  **********/
    DEFINE FRAME Frame1
        SKIP(1) Rect1 SKIP(1) Field1 SKIP(1) btn-Rect SKIP(2) btn-Exit
            WITH SIDE-LABELS NO-BOX CENTERED THREE-D.

    /**********  DEFINE TRIGGERS  **********/
    ON CHOOSE OF btn-Rect
    DO:
/*3*/   ASSIGN Rect1:VISIBLE = YES
/*4*/          Field1 = Rect1:VISIBLE.
/*5*/   DISPLAY Field1 WITH FRAME Frame1.
        DISABLE btn-Rect WITH FRAME Frame1.
    END. /* ON CHOOSE OF btn-Rect */

    /**********  MAIN LOGIC  **********/
/*1*/  ASSIGN Rect1:VISIBLE = NO
/*2*/      Field1 = Rect1:VISIBLE.
      DISPLAY Field1 WITH FRAME Frame1.
      ENABLE Field1 btn-Rect btn-Exit WITH FRAME Frame1.
      WAIT-FOR CHOOSE OF btn-Exit.
```

These notes help explain the code:

1. When you first create a widget, the VISIBLE attribute is set to YES by default. However, a field-level widget is not visible until its frame is visible. Therefore, when a frame becomes visible, all the widgets in it are visible by default. In this statement, you set the VISIBLE attribute to NO. When the frame displays, you won't see the rectangle.

2. You assign the value of the rectangle's VISIBLE attribute to Field1.

3. When the user chooses the button, the trigger changes the value of the VISIBLE attribute to YES, making the widget visible, since its frame is currently visible.

4. You assign the new value of the VISIBLE attribute to Field1.

5. The DISPLAY statement outputs the new value of Field1 to the screen.

This example also illustrates an important rule about accessing attributes for field-level widgets: You cannot access a widget's attributes until that widget is referenced in a frame.

Methods

A *method* is a specialized function that modifies how a widget works. For example, the READ-FILE method allows you to read an operating system file into an editor widget. Methods are relatives of attributes and you access them in the same way you access attributes.

To use a method, follow these basic steps:

1. Reference the widget using the standard attribute syntax.
2. Supply any input parameters required by the method.
3. Assign the result of the method call to a variable of the same data type.

Here's an example:

```
ASSIGN variable-name = widget-name:method-name(parameter-name).
```

The later chapters of the tutorial discuss some of the most useful methods.

3.6 Summary

The Progress programming model is a way of programming that creates highly responsive interfaces. A user interface is made up of a keyboard, or other input device such as a mouse, and whatever the user sees on the screen.

A window is an object that contains your application's workspace. Character interfaces use only one window. Graphical interfaces, such as Windows 95 or Windows NT, can have multiple overlapping windows on the screen.

Progress interfaces are made of objects called widgets. An interface is a tool for displaying data and receiving events. A particular interface can receive events when it is enabled for input and the procedure or the user designates that widget as the event receiver. A widget so designated is said to have input focus.

Each widget responds to each possible user event (keyboard or mouse action) with a default response. You can program a widget to have a response in addition to the default response using triggers. A trigger is a block of code following an ON statement. The code executes when the widget specified in the ON statement receives the user event specified in the ON statement.

The parts of a Progress procedure include:

- Widget definition
- Frame definition
- Trigger definition
- Interface creation
- Interface enabling
- Execution blocking
- Interface disabling

Progress has the following widgets:

- **Window Widgets** — Window widgets are workspace for your application. A window contains all other widgets. Character interfaces use one window, while graphical interfaces can use multiple windows.
- **Container Widgets** — Container widgets allow you to organize data, action, and graphic widgets. Container widgets include frames and dialog boxes.

- **Data Widgets** — Data widgets are the widgets you use to represent data from fields and variables. Data widgets include fill-in fields, text, editors, selection lists, combo boxes, radio sets, toggle boxes, sliders, and browse widgets.

- **Action Widgets** — Action widgets allow the user to direct the operation of your application. Menus provide groups of related processing options while buttons execute a single option. Buttons, menu bars, submenus, and menu items make up action widgets.

- **Graphic Widgets** — Graphic widgets help you decorate your interface. Use rectangles to make different kinds of borders. Images, which are available on graphical interface systems, allow you to include high-resolution graphics in your interface.

You learned about these Progress statements:

- DEFINE VARIABLE creates a variable for use in your procedure.
- DISPLAY displays widgets and the data contained in the associated fields.
- ENABLE turns on widgets for interaction with the user.
- DISABLE turns off widgets.
- WAIT-FOR blocks execution and establishes the condition that unblocks execution.
- ON sets up a widget-event pair that executes a trigger.
- DEFINE BUTTON creates a button.
- DEFINE RECTANGLE creates a rectangle.
- DEFINE FRAME creates a frame, including the field-level widgets that it contains.
- VIEW makes visible frames or widgets within frames.
- HIDE makes invisible frames or widgets within frames.
- ASSIGN explicitly assigns attribute values to widgets.

4

Understanding the Database Environment

This chapter introduces you to the Data Dictionary, a tool that you use to create, modify, and view the properties of a database. The chapter covers:

- The Data Dictionary as a programming resource
- Accessing and exiting the Data Dictionary
- Creating, connecting, and disconnecting databases
- Using the Data Dictionary interface
- The sports database schema
- Defining database objects
- Establishing application defaults
- Generating reports

For additional information about the Data Dictionary, see the *Progress Basic Development Tools* on-line help. For a discussion of designing and creating effective databases, see the *Progress Database Design Guide*.

4.1 The Data Dictionary As a Programming Resource

The Data Dictionary is the most important tool for database developers. The Data Dictionary allows developers to create a database and specify the properties that define the objects, or components, of that database. Database objects include tables, fields, indexes, and sequences. Collectively, these defining properties are called *schema*. Once you establish your schema, you can create procedures to create records and populate your database.

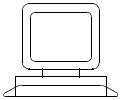
The Data Dictionary is also a vital tool for 4GL programmers. In addition to object definitions, the schema includes properties that affect the default behavior of Progress procedures as they interact with database objects. These properties are know as *application defaults*. For example, suppose a particular database field needs upper and lower numeric limits. To keep values within the limits, you would have to code checks into every procedure that accesses the field. Checking data to make sure it meets criteria is known as *validation*. Instead of coding your validation routines, you can take advantage of Progress database validation properties to automatically check the data.

The purpose of this chapter is to teach you how to use the Data Dictionary to view the schema of a Progress database and manage important application defaults. Wisely using application defaults:

- Reduces development time by moving functionality from procedures to the database
- Reduces redundant coding by providing a central location for some functionality
- Simplifies application maintenance of this functionality

To illustrate how the Data Dictionary and application defaults work, you'll create and use your own copy of the sports database.

4.2 Accessing the Data Dictionary



Exercise

Follow these steps to access the Data Dictionary directly from the Procedure Editor.

1 ♦ Activate the Procedure Editor menu bar.

2 ♦ Choose Tools→ Data Dictionary. Progress exits the Procedure Editor and starts the Data Dictionary. Figure 4–1 shows the Data Dictionary main screen.

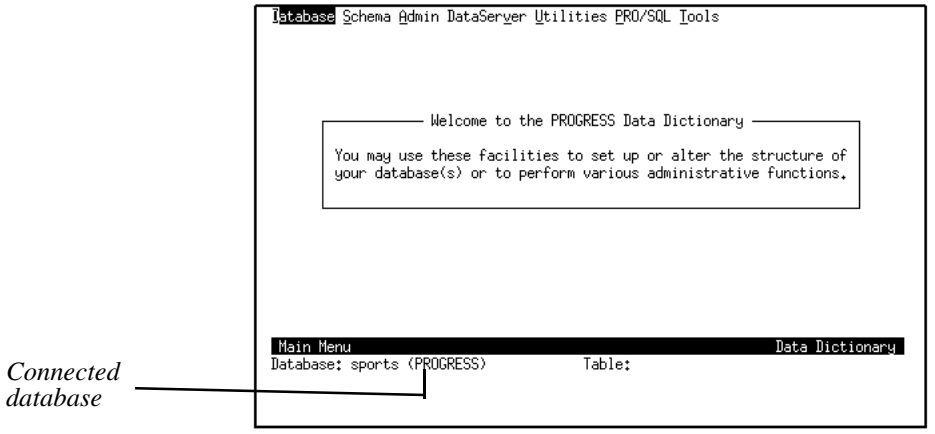


Figure 4–1: The Data Dictionary Main Window

Before any Progress tool can work with a database, you must *connect* to that database. Connecting a database means that Progress opens the database file and has access to the schema and data from the file.

Note that while you work with the Data Dictionary, you cannot work with the tool you started the Data Dictionary from (the Desktop or the Procedure Editor), although it remains open.

4.3 Exiting the Data Dictionary

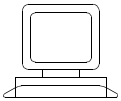
Choose the Database→ Exit. Progress exits the Data Dictionary and returns to the Procedure Editor. As you can see, it's easy to move back and forth between the two tools. If you had open buffers in the Procedure Editor when you started the Data Dictionary, they are still there when you exit the Data Dictionary and return to the Procedure Editor.

4.4 Creating, Connecting, and Disconnecting Databases

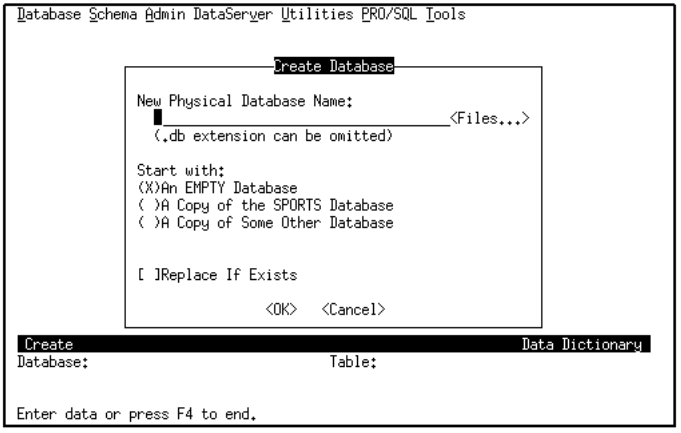
In this section, you learn how to create, connect, and disconnect a database. While you're learning these important tasks, you'll also be preparing for tutorial exercises in other chapters. The other chapters assume that you have your own copy of the sports database and that you have connected it.

4.4.1 Creating a Database

Follow these steps to access the Data Dictionary and create a new database.


Exercise

1 ♦ Choose Database→ Create from the Data Dictionary menu bar. The Create Database dialog box appears as shown:



2 ♦ Type **sports** in the New Physical Database Name field.

3 ♦ The Start With radio-button set displays the following copy options:

- **An EMPTY Database** — Progress supplies an empty database template. This is the database you use to create and design a Progress database from scratch.
- **A Copy of the SPORTS Database** — Progress supplies a sports inventory demonstration database to help illustrate database design and application programming concepts in the Progress documentation set. The examples and exercises in this tutorial use a copy of the sports database.
- **A Copy of Some Other Database** — You can specify another Progress database to copy.

4 ♦ Select the SPORTS database radio button.

5 ♦ Activate the Replace If Exists toggle box. In this case, the Replace If Exists toggle box represents a YES/NO question. (Do you want the new database to replace any existing database of the same name in the current directory?)

6 ♦ Choose OK to create a copy of the sports database.

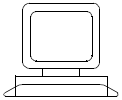
7 ♦ Once you create a Progress database, you must connect the database to access the data in it. When you choose OK in the Create Database dialog box, the Database Connect dialog box appears to help you connect the sports database. Choose Cancel for now. The next section describes how to access this dialog box directly.

The main Data Dictionary display appears.

4.4.2 Connecting a Database

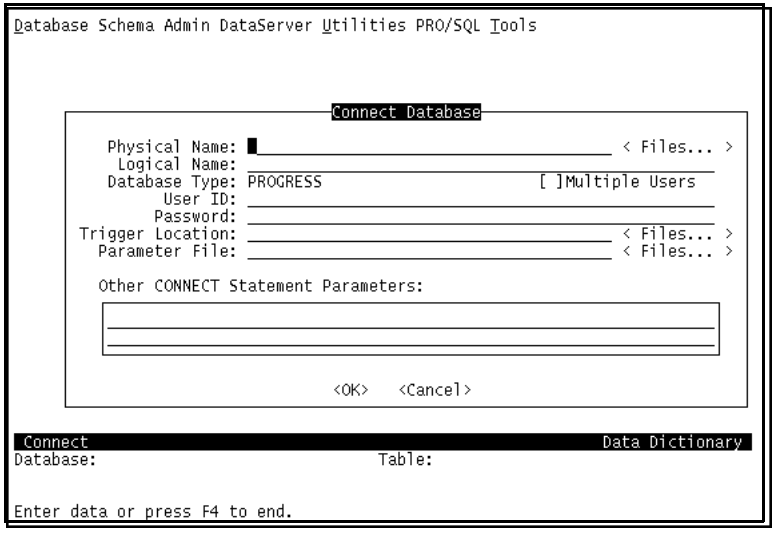
When you connect a database with the Data Dictionary, you make the database available to all Progress tools and applications. In other words, if you exit the Data Dictionary without explicitly disconnecting a connected database, the database is still available to other Progress tools and applications. From within the Data Dictionary, connecting a database also means that you have access to view or modify the database schema.

Follow these steps, to connect a database:



Exercise

1 ♦ In several cases, the Data Dictionary displays the Database Connect dialog box automatically. To access it directly, choose Database→ Connect from the Data Dictionary main display. The dialog box appears, as shown:



The Database Connect dialog box allows you to specify how you want to connect to a database. For specifics on database connection parameters, see the *Progress Startup Command and Parameter Reference*.

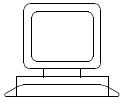
2 ♦ Type **sports** in the Physical Name field.

The *physical name* is the name of the operating system file that contains the database. The physical name for the sports database is `sports.db`. You don't need to type the .db extension because Progress looks for files with that extension.

The *logical name* of a database is the database's internal name. The default logical name is the same as the physical name. In this case, it's **sports**. The logical name is the name you use to reference a database in a Progress procedure.

3 ♦ Choose OK to connect to the new sports database and return to the Data Dictionary display. The Data Dictionary display confirms the database connection.

You can connect several databases, but you can only work with one at a time in the Data Dictionary. If you have several connected, choose the one you want to work with the command Database→ Select Working Database.



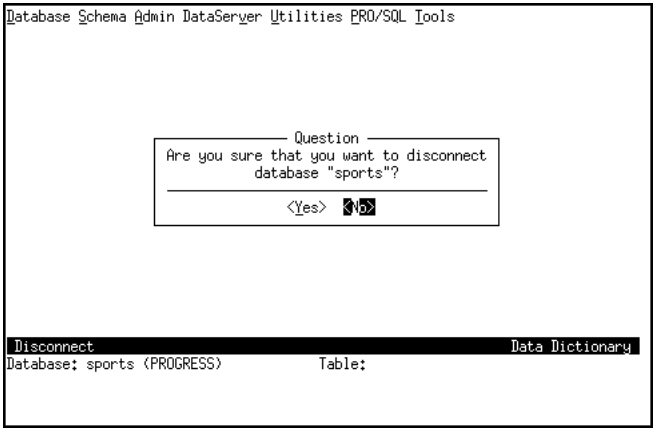
Exercise

4.4.3 Disconnecting a Database

When you disconnect a database, that database is no longer available to other Progress tools and applications and the schema is no longer available in the Data Dictionary

Follow these steps to disconnect a database:

1 ♦ Choose Database→ Disconnect. Progress displays a dialog box and prompts you to verify that you want to disconnect as shown:



2 ♦ Choose Yes. The status area no longer lists the sports database.

3 ♦ Reconnect the sports database so that it is available for the next exercise.

4.5 Using the Data Dictionary Interface

The Data Dictionary is an easy-to-use application that lets you quickly select a database object and view the definition and application defaults associated with the object. The rest of this section provides information on using the different interface components.

4.5.1 Using Menus

The menus on the menu bar allow you to control the application and work with database objects. Table 4–1 provides a brief description of the menus available in the Data Dictionary.

Table 4–1: Data Dictionary Menus

Menu	Description
Database	Connect and disconnect databases; choose a working database; create new databases; and run reports.
Schema	Define and view tables, fields, indexes, and other database components. This menu is not available until you connect to a database.
Admin	Protect and maintain your definitions and data. This menu is not available until you connect to a database.
DataServer	View and manipulate non-Progress databases. This menu is not available until you connect to a database.
Utilities	Access utilities for advanced users.
PRO/SQL	Perform SQL-related tasks. This menu is not available until you connect to a database.
Tools	Connect to other Progress tools.

4.5.2 Working with Database Objects

The Data Dictionary interface allows you to quickly access the schema of a database object. As mentioned before, the schema consists of properties that define a database object or properties that establish an application default for a database object. A *property*, then, is an individual schema element. The paradigm of the Data Dictionary is to provide a simple interface that lets you quickly select an individual database object and bring up a dialog box that organizes and displays all the properties associated with that type of object. These dialog boxes are known as *editors*. For example, there is a field editor, an index editor, and so on.

Here is the basic procedure for accessing an editor:

1. To work with tables, select the appropriate command from the Schema menu.
2. Select an individual database object from the list. For fields and indexes, you select the correct table and then the correct field or index object.
3. To work with fields and indexes, select the appropriate editor from the Schema menu and then choose the appropriate table from the list that appears.
4. To work with sequences, select the sequence editor from the Schema menu.
5. Choose the action you want to perform with the object from the commands listed at the bottom of the editor.

4.6 Introduction to the sports Database Schema

Now that you have accessed the Data Dictionary and connected your copy of the sports database, take a moment to review the descriptions of the sports database tables found in Table 4–2. Understanding the purpose of tables gives you a general idea about the function of the database.

Table 4–2: Tables in the sports Database

Table	Contents
Customer	Customer information, such as customer name, address, phone number, and credit limit.
Invoice	Accounting information, such as order number, invoice number, and billing total.
Item	Inventory information, such as item name, catalog description, and price.
Local-Default	Information for working with international customers, such as country name, currency, and postal format.
Order	Order form information, such as order date, shipping date, and postal carrier.
Order-Line	Information relevant to a request for one item in a particular order, such as item number, quantity, price, and associated order number.
Ref-Call	Customer call information, such as call number and reason for call.
Salesrep	Sales representative information, such as name, region, and monthly quota.
State	State information, such as state name, abbreviation, and region.

As you can conclude from these descriptions, this database holds the information necessary for All Around Sports to take orders, process orders, bill for orders, and keep inventory information up to date.

4.7 Defining Database Objects

Earlier, the tutorial made a distinction between schema properties that define a database object and properties that establish application defaults. This section points out the essential defining properties of each object type.

4.7.1 Defining Tables

A table is a collection of information about a subject. As an example of table properties, choose the Schema Modify Table command and choose the Customer table from the list. The dialog box for the Customer table of the sports database is shown below.

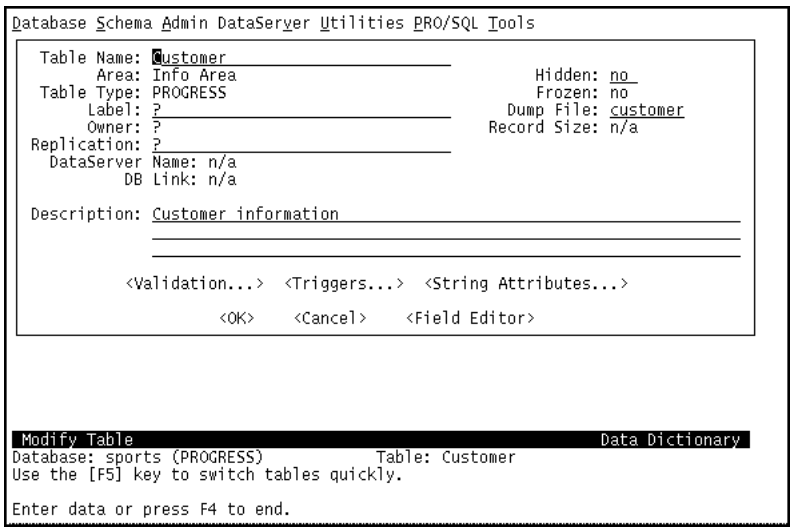


Figure 4–2: Modify Table Dialog Box

All you really need to define a table is a unique name. The rest of the table definition is actually the definition of the field and index objects that make up the table.

4.7.2 Defining Fields

A field is a particular item of information that helps describe the subject of a table. For example, the name of a customer, the address, and the credit limit are information items that help define a customer. As an example of field properties, access the Cust-Num field by choosing Modify from the Field Editor and then choosing the field.

This is shown in Figure 4–3.

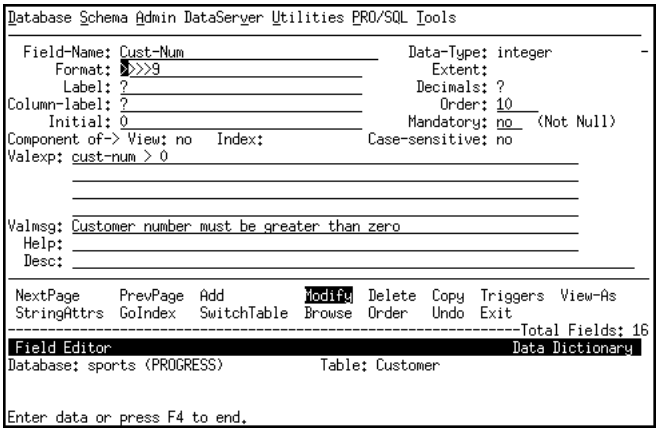


Figure 4–3: Field Editor

Two properties make up the essential definition of a field object: the field name and the data type.

- Field names may be up to 32 characters long and can consist of alphabetic characters (A–Z, a–z), digits (0–9), and the special characters $, &, #, %, -, and _. A field name must begin with an alphabetic character.
- The data type defines what type of data a field can store.

Table 4–3 describes the Progress data types.

Table 4–3: Data Types *(1 of 2)*

Data Type	Description
CHARACTER	Character fields contain any type of data (letters, numbers, and special characters). Some character fields in the Customer table are Name, Address, and Phone.
INTEGER	Integer fields contain whole numbers. They can be positive or negative, from roughly -2 billion to 2 billion. The Cust-Num field from the Customer table is an integer field.
DECIMAL	Decimal fields hold decimal numbers up to 50 digits in length, including up to 10 digits to the right of the decimal. The Balance field in the Customer table is a decimal field.

Table 4–3: Data Types *(2 of 2)*

Data Type	Description
DATE	Date fields contain dates. The Invoice-Date field in the Invoice table is a date field.
LOGICAL	Logical fields contain values that evaluate to YES/NO or TRUE/FALSE. The Backorder field of the Order-Line table is a LOGICAL field.

4.7.3 Defining Indexes

An index is a field or combination of fields from one table that Progress uses as the basis for searching, sorting, or otherwise processing the records of that table. As an example of index properties, access the Index Editor for the Customer table and select the Comments index, as sh9own in Figure 4–4.

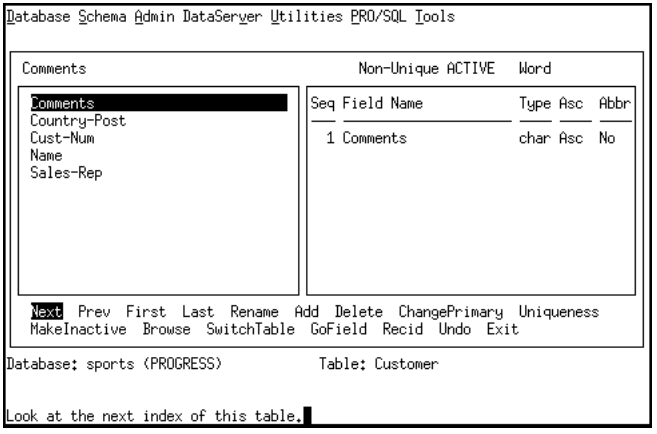


Figure 4–4: Index Editor

An index is defined by several important properties. First, the index must have a unique name. If it is a simple index, based on one field, then name the index the same as the field component. If it is a compound index, based on several fields, give the index a descriptive name that describes the components of the index.

Next, the properties of an index include each component field and whether the component is ascending or descending. The display lists the component fields and their ascending or descending status. When a field component is ascending, Progress sorts the index values from first to last (for example, A to Z or 1 to 10). When a field component is descending, Progress sorts the index values from last to first (for example, Z to A or 10 to 1). Note that you cannot

change these properties after you create the index. Changing components or sort direction is equivalent to building a new index.

Finally, an index has a series of logical properties that define how the index works. These properties are also shown in the display. Table 4–4 describes these properties.

Table 4–4: Index Properties

Property	Description
Primary	The *primary* index is the one that Progress uses by default. Each table has one and only one primary index. Progress uses the primary index when retrieving records or ordering records for a list (like a report) if you don't specify another index. You want your primary index to reflect the most common or natural sort order.
Unique	A *unique* index is one where each index value must be different. For example, each customer needs a unique number to identify them. Thus, the Cust-Num index is a unique index. When you define an index as unique, Progress will not allow you to enter an index value that's already in use.
Active	An *active* index is one that Progress updates when you create, delete, or modify a record. If the index were inactive, you could still find the definition of the index in the Data Dictionary, but Progress would not update it until you made it active again.
Word	A *word index* is one based on a character field where each distinct word in the data becomes a separate index entry. Word indexes allow you to search character fields quickly for key words.

4.7.4 Defining Sequences

A *sequence* is an algorithm for creating an incremental number series. Sequences can generate sequential values within any integer range with either positive or negative increments. The database holds the sequence definition and keeps track of the next available value. However, you use sequences programmatically. That is, when you create a new record, your procedures can use sequences to provide automatically generated sequential values for fields.

For example, All Around Sports customer numbers begin at 1 and increment by 1. Every time you add a new customer record, you can access the Next-Cust-Num sequence to generate a new, legal, unique customer number. You would then assign the value to the Cust-Num field of a new Customer record.

As an example of sequence properties, access the Sequence Editor and select the Next-Cust-Num sequence of the Customer table, as shown in Figure 4–5.

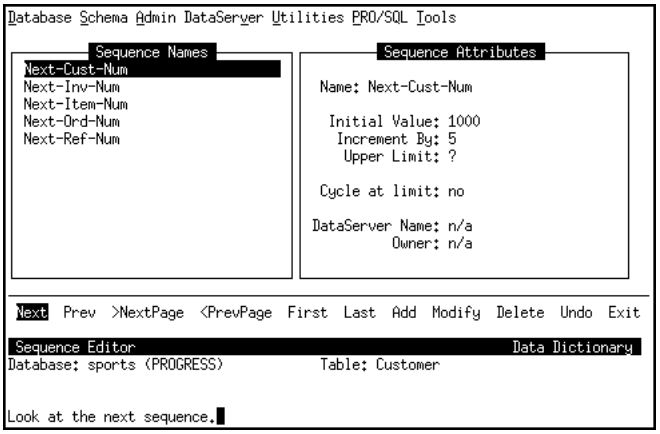


Figure 4–5: Sequence Editor

Sequence properties include a unique name and properties that provide a number of ways to alter the basic sequential algorithm. Table 4–5 describes the properties.

Table 4–5: Sequence Properties

Property	Description
Initial Value	The starting value of the sequence. You can enter any positive or negative integer.
Increment by	The value used to determine the next sequence number. You can use a positive or negative integer.
Upper Limit	The highest acceptable number for the sequence. You can use any positive or negative integer.
Cycle at Limit	Determine whether the initial value is re-used when the sequence reaches the upper limit.

By using the properties, you can create one of three types of sequences:

- **Nonterminating Sequence** — A sequence that begins at an initial value and increments in one direction with no limit. Accordingly, you must provide a value for the Initial Value property and the Increment by property. Do not specify a limit (? designates no limit) and Cycle at Limit must be NO. A non-terminating sequence provides continual unique values.

- **Terminating Sequence** — A sequence that begins at an initial value and increments in one direction until it reaches a designated limit. At this point, the sequence cannot create new values unless you explicitly reset the initial value. Accordingly, you must provide a value for the Initial Value property, the Increment by property, and the appropriate limit property. Cycle at Limit must be NO. A terminating sequence provides unique values until the limit is reached.

- **Cycling Sequence** — A sequence that begins at an initial value and increments in one direction until it reaches a designated limit. On reaching the limit, the sequence restarts with the initial value. You must provide a value for the Initial Value property, the Increment by property, and the appropriate limit property. Cycle at Limit must be YES. A cycling sequence does not provide unique values.

4.8 Establishing Application Defaults

In the last section, you learned about the schema properties that define database objects. Of the remaining properties, this section looks at those that establish important application defaults. These specific defaults contribute to a few general areas of functionality, including:

- Validation
- Custom behavior based on special interactions with database objects
- Display formatting

4.8.1 Validation Expressions and Messages

Validation is the process of checking criteria before allowing the user to take an action. There are two types of validation: table validation and field validation. Table validation allows you to establish criteria that must be met before a user can delete a record. For example, you wouldn't want to delete a customer record if that customer has unpaid bills.

To see an example of table validation, access the Modify Table dialog box for the Customer table and choose the Validation button. A dialog box appears, as shown in Figure 4–6.

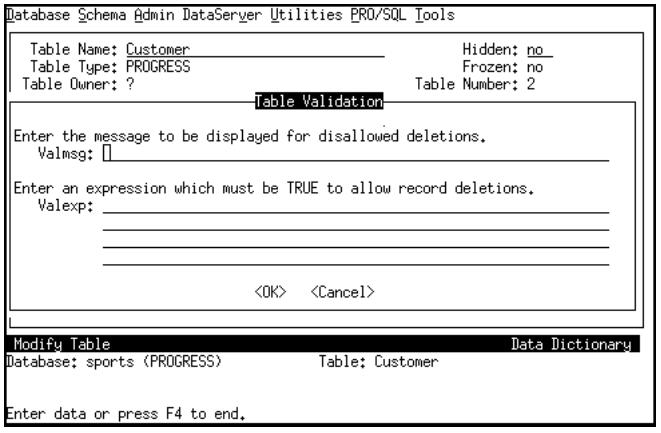


Figure 4–6: Table Validation Dialog Box

Field validation allows you to establish what is and is not an acceptable entry for a field. For example, you can check to make sure an entry falls with high and low numeric limits.

To see an example of field validation, access the Modify command of the Field Editor and choose Cust-Num field of the Customer table. The validation fields appear at the bottom of the display.

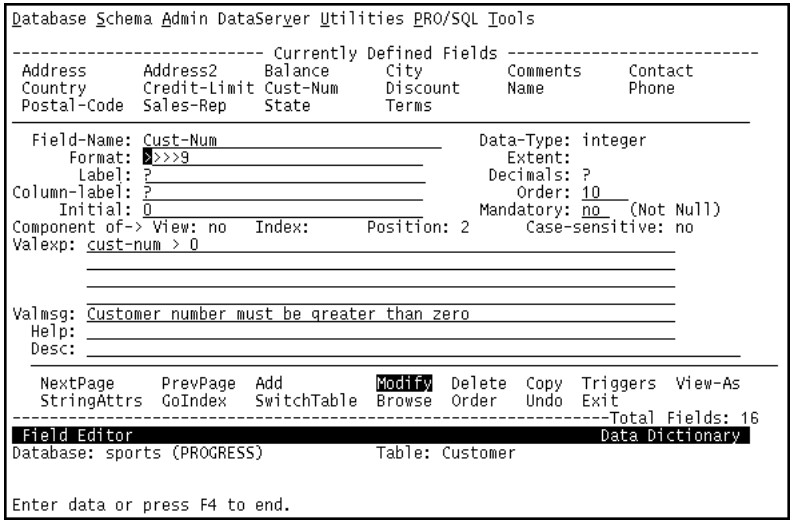


Figure 4–7: Field Validation Dialog Box

Both of these displays have space where you can define the validation criteria using Progress 4GL statements. This code is known as the *validation expression*.

The fill-in widget contains text that Progress displays to the user when a user action fails the validation expression. This message is known as the *validation message*.

4.8.2 Triggers and Database Events

In Chapter 3, you learned about events and triggers. These triggers and events are called *user-interface events* and *user-interface triggers*. Tables and fields in a Progress database can receive a different type of event known as *database events*. Tables and fields respond to these events with a different type of trigger known as a *database trigger*.

When discussing database triggers, it's also important to distinguish two types: schema triggers and session triggers. A *schema trigger* is a database trigger stored in the schema of a database. A *session trigger* is a database trigger used in code.

There are five database events, as described in Table 4–6.

Table 4–6: Database Events

Event	Object Type	Description
CREATE	Table	Executes on creation of a new record. For example, use a CREATE trigger to insert data, like the current date, into record fields.
DELETE	Table	Executes on deletion of a record. For example, use a DELETE trigger to prompt the user to confirm the deletion.
FIND	Table	Executes when Progress locates a record that satisfies a search request. For example, use a FIND trigger to increment a field that keeps track of how many times a record has been accessed.
WRITE	Table	Executes whenever a record is written to the database. For example, use a WRITE trigger to make a copy of the old record before writing the new data.
ASSIGN	Field	Executes when an individual field has new data written to it. For example, use an ASSIGN trigger to make a copy of old data before copying new data to a field.

To specify a trigger, access the appropriate table or field dialog box and choose the Triggers button. Figure 4–8 shows the Table Triggers dialog box for the Customer table.

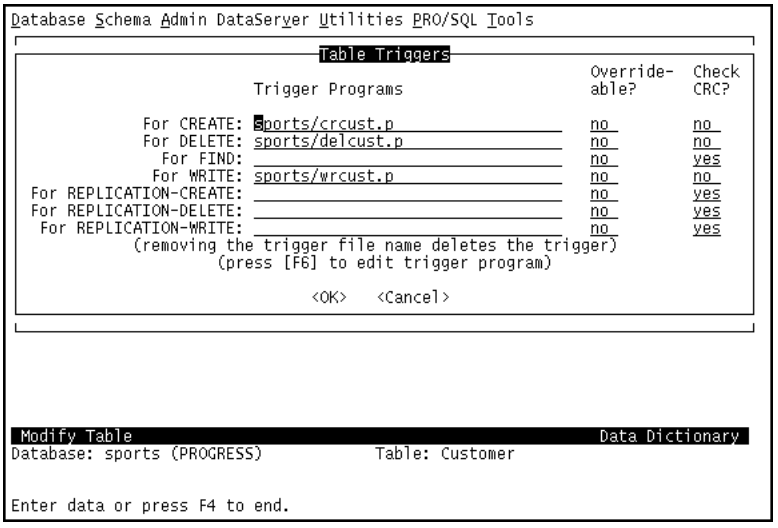


Figure 4–8: Table Triggers Dialog Box

The tutorial doesn't cover how to write and implement database triggers. Still it's important to know about them. Database triggers give 4GL programmers the ability to define custom default behaviors and business rules for your data. The database then becomes a central storage place for these rules.

For information on using database trigger dialog boxes, the *Progress Basic Development Tools* manual.

4.8.3 Field Formats

The format of a field describes the way that Progress displays data on your screen and in printed reports. The format is a string made up of literals and symbols. *Literals* are extra characters that you want to display with data and *symbols* are place holders for data. When you create a field in the Data Dictionary, Progress supplies a default format string for the particular data type. If that format string doesn't suit your needs, then you can specify another.

To create format strings, you first need to understand the format symbols. Table 4–7 describes these symbols.

Table 4–7: Field Format Symbols

Symbol	Description
X	Represents an alphanumeric character position.
A	Represents a letter position. A blank, digit, or punctuation mark is not allowed.
N	Represents a letter or number position. A blank is not allowed.
!	Represents a letter (A) that must be displayed in uppercase. A blank is not allowed.
9	Represents a numeric character position. A blank is not allowed.
–	Places a minus sign in front of numeric data if the value is negative or a blank if the value is positive.
>	Designates zero suppression. That is, Progress replaces > with a number, as long as it's not a leading zero. If it's a leading zero, Progress replaces > with a blank.
,	Displays a comma, unless a > precedes it and the character is a leading zero. If the character is a leading zero, Progress replaces the comma with a null.
.	Represents a decimal point.

To create a format string, just put together the symbols to describe how to display the data. For example, if a character field contains the word "Wonderful", one format string could be "AAAAAAAAA". If you want the data to display in uppercase, change the format string to "!!!!!!!!!". If the field can contain numbers too, change the format string to "NNNNNNNNN". If the field can contain any number, letter, or character, including blanks, then use the format string, "XXXXXXXXX".

Instead of typing long strings of symbols, you can use this shortcut:

1. Specify the symbol.

2. Indicate the number of total places enclosed in parentheses.

So, "XXXXXXXXX" becomes "X(9)".

Using “X(9)” displays “Wonderful”. If you specify “X(6)”, only “Wonder” displays. Specifying “X(3)” displays “Won”. Note that even though the last two format strings truncate the value on the screen, they do not affect the actual data stored in the field or limit the storage capacity of the field. Format strings affect displays only.

While the format strings help display existing data, they also limit user input. For example, if a field has a format string that converts all data to uppercase, then when the user enters data, it will be converted to uppercase. Similarly, a format string that specifies numeric data prevents a user from entering letters and special characters into the field.

4.8.4 VIEW-AS Phrases

As you learned in the last chapter, the default data widget for a database field is a fill-in field. If you don’t plan to display a particular field as a fill-in field, consider using the VIEW-AS property to establish a different default data widget. For example, if you want a particular database field to hold notes and miscellaneous information in sentence form, then a fill-in field would be a poor data representation for that field. It would be better to define it as an editor widget.

You access the VIEW-AS property by choosing the VIEW-AS command in the Field Editor and choosing the appropriate field. The VIEW-AS Phrase dialog box appears, as shown in Figure 4–9.

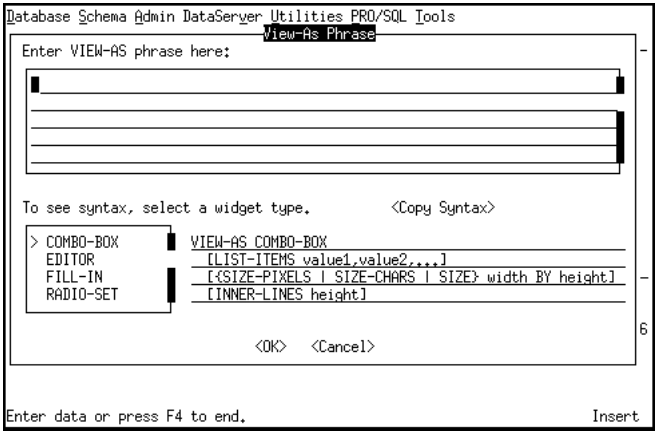


Figure 4–9: VIEW-AS Phrase Dialog Box

The selection list contains the data widgets that are legal for the field. When you select a widget type, the editor displays a syntax diagram for in the bottom editor, which you can copy into the editor at the top of the dialog box.

4.8.5 Labels

When you display a field, the field name is the default screen label. You can specify a different label using the Label property in the Field Editor. Typically, you want your default label to be more descriptive than the field name, which most programmers like to keep short. You can also specify a different label in the Column Label field for use when Progress displays data in columns. Both the label and column label can consist of any text, including blanks.

To keep the default label (field name), enter ? in the label fields.

The Field Editor also contains the Help Text property. Progress displays the text entered here in the default status area of the window when the field gets input focus. You can specify any text you like. Consistently providing help text for database fields significantly reduces the burden of programming help messages.

4.9 Generating Reports

As you're coding, it helps to have hard copy reports of database properties for reference. The Data Dictionary Reports menu has a variety of options, which you can use to output reports to the screen or to a printer. Table 4–8 lists and describes the Reports menu options.

Table 4–8: Reports Menu Options *(1 of 2)*

Menu Option	Description
Detailed Table	Create a complete report of the properties of a table and its constituent fields.
Quick Table	Create a short summary of the properties of the database tables.
Quick Index	Create a short summary of the indexes, index components, and index properties of a table.
Quick Field	Create a short summary of the fields, from one or all tables.
SQL View	List the defined SQL views in a database.
Sequence	Fully describe the sequences defined in a database
Trigger	Summarize the fields that have defined database triggers.

Table 4–8: Reports Menu Options *(2 of 2)*

Menu	Description
User	List the defined users for the database.
Table Relations	Examine the database and report the natural relationships found among database tables.

4.10 Summary

The Data Dictionary is a tool that you use to create, view, and modify the properties of a database and database objects: tables, fields, indexes, and sequences. Database properties include those that define database objects and those that establish application defaults. Collectively, database properties are referred to as schema.

Databases have two names:

- The physical database name is the operating system filename of the database.
- The logical database name is the internal Progress name of the database. Procedures reference the logical name. If you don't specify a logical name, Progress uses the physical name by default.

The Data Dictionary allows you to connect databases. When a database is connected, any Progress tool or application can access the data or schema of the database.

Table objects are defined with a unique name. The definitions of constituent fields and indexes make up the bulk of a table definition.

Field objects are defined with a unique name and by specifying a data type. Data types determine what kind of data a field can store:

- CHARACTER fields can contain any kind of text data.
- INTEGER fields can contain positive or negative whole numbers.
- DECIMAL fields can contain any number.
- DATE fields contain dates.
- LOGICAL fields contain values that evaluate as YES/NO or TRUE/FALSE.

Index objects are defined by specifying a name and the component fields. Other index properties define how the index works:

- The primary index is the index that Progress uses by default.
- A unique index cannot contain two identical index values.
- An active index is one that Progress keeps updated and ready for use.
- A word index is an index of a character field that allows you to search for words within the field.

Sequences are objects that generate automatic incremental sequences. Using sequence properties, you can define three types of sequences:

- Nonterminating sequences start at an initial value and increment continually in one direction with no limit.
- Terminating sequences start at an initial value and increment in one direction toward a limit. Upon reaching the limit, a terminating sequence ceases to generate new values.
- A cycling sequence starts at an initial value and increments in one direction toward a limit. Upon reaching the limit, the sequence starts over at the initial value.

Tables and fields have a validation property, which allow you to check criteria before allowing record deletions in a table or new values in a field. The validation property is made up of:

- A validation expression that establishes the logic that allows or disallows record deletions in a table or new values in a field.
- A validation message that informs the user that an action failed a validation check.

Databases have special events for which you can write triggers:

- A CREATE event occurs when a new record is created.
- A DELETE event occurs when a record is deleted.
- A FIND event occurs when any search successfully finds a record.

- A WRITE event occurs when Progress writes a record back to the database.
- An ASSIGN event occurs when Progress writes new data to an individual field.

The database trigger property allows you to add default behaviors to a database, eliminating the need to code the behaviors in your procedures.

Field formats are symbols that tell Progress how you want to display data by default. Formats do not affect the actual data, just the display of the data.

5

Working with Expressions

This chapter discusses different ways to represent and manipulate values. A value can be a variable or a database field, as you've already seen. A value can also be the result of an arrangement of language elements. An arrangement of language elements that represents a single value is known as an *expression*.

Specifically, this chapter contains information on:

- Understanding expressions
- Using operands
- Using operators
- Using functions
- Understanding precedence in expressions
- Using expressions for common tasks

5.1 Understanding Expressions

An expression is a combination of 4GL language elements that results in a single value. At a minimum, an expression consists of one operand, but can include multiple operands, operators, and functions.

An *operand* is a language element that represents a value. Operands include:

- Variables — A location in memory that holds a value that may change during the execution of a procedure.
- Database fields — An individual value from a database record.
- Widget attributes — A value that describes a characteristic of a widget like its width, height, and so on.
- Constants — A value that does not change during the execution of a procedure.

An *operator* is a language element that represents an action carried out on one or more operands. The successful combination of operands and operators, called an *operation*, always returns a value. For example, the equal sign (=) is a 4GL operator that assigns the value from the right side of the operator to the field or variable on the left side of the operator. The plus sign (+) is another operator that adds the operands on either side of the operator and returns the result.

A *function* is a prewritten program that performs a specific task and returns a value. A function is like an operand in that it represents a value, but it is also like an operator in that it represents an action. You can think of a function as a prepackaged solution to a task that can be used in an expression. Because the code already exists, either as a *pre-defined function* or one that you create as a *user-defined function*, it offers a shortcut to providing a programmed solution to a coding task or problem.

Progress supports pre-built Progress-supplied functions and user-defined functions. The 4GL contains many functions that perform a wide range of common tasks. For example, the ASC(*character*) function returns the ASCII integer code for a supplied character. Also, you can use the FUNCTION statement to create specific functions that suit your programming needs.

The sections that follow provide more information about expression components.

5.2 Using Operands

Operands are values. They are the raw material that operators and functions act on to determine the final result of an expression. This section provides the information you need to successfully use operands in expressions.

5.2.1 Using Variables

A *variable* is a temporary storage place for data. The value contained in a variable may change during the execution of the program that contains it. Typically, variables contain data from the user, copies of database fields or other variables, or the results of expressions.

Defining Variables

The partial syntax for the define statement below contains more options than you encountered in Chapter 3.

SYNTAX

```
DEFINE VARIABLE variable-name
  { AS datatype | LIKE field }
  [ EXTENT n ]
  [ FORMAT string  ]
  [ LABEL string [ , string ] ]
  [ INITIAL { constant | [ constant [ , constant ] ... ] } ]
```

Table 5–1 describes the new syntax components of the DEFINE VARIABLE statement:.

Table 5–1: Syntax Components of DEFINE VARIABLE *(1 of 2)*

Component	Description
AS *datatype*	Specifies the data type of the variable. Valid data types include: CHARACTER, INTEGER, DECIMAL, DATE, LOGICAL and WIDGET-HANDLE
LIKE	Copies the attributes of an existing field or variable. When you use LIKE, the new variable gets the data type, format, label, extent, and initial value specification of the target field or variable.
EXTENT	Makes a variable an array. Specify an integer after the keyword to indicate the number of elements in the array.
FORMAT	Specifies the data format of the variable. If you do not specify FORMAT, the variable uses the default format for its data type.

Table 5–1: Syntax Components of DEFINE VARIABLE *(2 of 2)*

Component	Description
LABEL	Specifies the display label for the variable. For arrays, if you specify a single label, then every element of the array uses that label. You can also specify a different label for each element of the array using a comma-separated list of labels after the keyword.
INITIAL	Gives the variable an initial value. For arrays, if you specify a single value, then every element of the array has that initial value. You can also specify a different initial value for each element using a comma-separated list of values enclosed in brackets.

Using LIKE

When defining a variable, you must specify either the data type or a database field or another variable whose format and attributes you want to copy to the new variable. When you specify the LIKE option, the new variable acquires the format, label, and initial value attributes of the referenced field or variable. You can override an inherited attribute by using the FORMAT, LABEL, and INITIAL options.

NOTE: To use a LIKE phrase that references a database field, you must be connected to the database that contains the field you want to copy.

Examine the following code:

```
/*1*/  DEFINE VARIABLE New-Name1 AS CHARACTER LABEL "NAME" FORMAT "X(20)"
                              INITIAL "All Around Sports".

/*2*/  DEFINE VARIABLE New-Name2 LIKE New-Name1.

/*3*/  DEFINE VARIABLE New-Name3 LIKE customer.name.
```

Here's what happens:

1. You create a variable to hold the name of a company.

2. Your second variable is identical to the first; that is, New-Name2 has the same value, format, and label that you specified in the New-Name1 definition.

3. In this statement, you create a third variable that inherits many of the attributes of a database field. Notice the general syntax for making a specific reference to a database field: *table-name.field-name*. The period separates the table and field names.

Using Arrays

Arrays are fields or variables that contain multiple data elements. The *extent* of an array is the number of elements it contains.

To define a field or variable as an array, specify the extent. If you want to define a database field as an array, use the extent option in the Data Dictionary. To define a variable as an array, use the EXTENT option in the DEFINE VARIABLE statement.

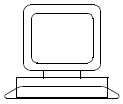
To initialize the array with values, you can use the INITIAL option with the values listed between square brackets as shown in the following code:

```
DEFINE VARIABLE Letters AS CHARACTER EXTENT 3 INITIAL ["a", "b", "c"].
DEFINE VARIABLE Numbers AS INTEGER EXTENT 5 INITIAL [1, 2, 3, 4, 5].
```

If you do not supply enough values to fill up the elements of the array, Progress copies the last specified value into the remaining elements of the array. If you supply too many values, Progress returns an error message.

Within a procedure, you can:

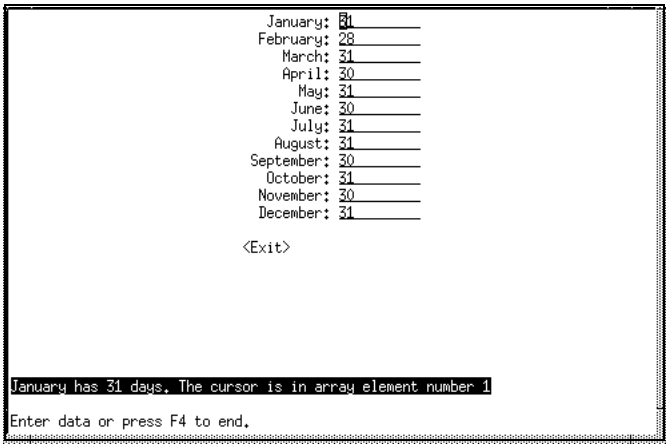
- Refer to all the elements in the array at once. Simply specify the array name.
- Refer to a specific array element. Specify the array name and the element number, called the *index*, enclosed in square brackets. For example: Letters[2]. Note that the index value of the first array element is 1.



Exercise

Follow these steps to see how to set up an array, present the whole array for user interaction, and reference the array in a trigger:

1 ♦ Open `lt-05-01.p` and run it. The interface shown below appears:



The array of 12 elements contains the number of days in each month. Each element has an individual label that corresponds to the name of the month.

2 ♦ Press **TAB** to move through the array elements. Notice the message area of the window. As you enter each element, the message references the label, data, and index of the element.

3 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

Here is the code that created the display:

lt-05-01.p

```
      /**********  DEFINE WIDGETS  **********/
/*1*/  DEFINE VARIABLE Months AS INTEGER EXTENT 12
/*2*/       LABEL "January", "February", "March", "April", "May",
                "June", "July", "August", "September", "October",
                "November", "December"
/*3*/       INITIAL [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31].
      DEFINE BUTTON btn-Exit LABEL "Exit".

      /**********  DEFINE FRAMES  **********/
      DEFINE FRAME Frame1
/*4*/       Months COLON 11 SKIP(1)
            btn-Exit
             WITH SIDE-LABELS NO-BOX CENTERED THREE-D.

      /**********  DEFINE TRIGGERS  **********/
/*5*/  ON ENTRY OF Months
       DO:
/*6*/       MESSAGE SELF:LABEL "has" SELF:SCREEN-VALUE "days."
                  "The cursor is in array element number" SELF:INDEX.
       END.

      /**********  MAIN LOGIC  **********/
      DISPLAY Months WITH FRAME Frame1.
      ENABLE ALL WITH FRAME Frame1.
      WAIT-FOR CHOOSE OF btn-Exit.
```

NOTE: The THREE-D option is relevant only on a Windows client; it is ignored by a character client.

These notes describe the array techniques used in this procedure:

1. The DEFINE VARIABLE statement with the EXTENT option defines the array.

2. To give each element an individual label, use a comma-separated list of strings after the LABEL keyword.

3. To initialize the elements, use a bracketed, comma-separated list of values after the INITIAL keyword.

4. An unsubscripted reference to an extent variable references all of the array elements. In a DEFINE FRAME statement, Progress includes room in the frame for each element of the array as a fill-in field.

   When you use the COLON option of the format phrase, Progress positions the field by the colon that terminates the widget label. When used with an array, the COLON option stacks the elements neatly.

5. Here, you use the array variable name (Months) as a reference to the whole array. If the user enters a fill-in field associated with any of the elements, the trigger executes.

6. The SELF keyword is called a system handle. The LABEL, SCREEN-VALUE, and INDEX keywords are widget attributes. A *system handle* is a global variable that holds a widget handle. The system maintains system handles for you. SELF is a system handle that holds the widget handle of the current widget (a widget that has input focus or a widget for which a trigger is firing). SELF is a convenient way of referring to an individual array element without having to keep track of the index. You'll learn more about SELF later in the tutorial.

   The LABEL attribute holds the widget label. The SCREEN-VALUE attribute holds the value shown on the screen, which may not be the same as the value in the underlying variable. (Perhaps the user changed the screen value or the procedure changed the field value.) The tutorial covers screen values and field values more thoroughly later. Finally, the INDEX attribute holds the index value of the array element where the cursor currently resides. If you need to know explicitly where the cursor is, as opposed to relying on SELF, the INDEX attribute is your tool.

5.2.2 Using Database Fields

Before you can reference a database field, you must connect the database that contains it. To reference a field in your code, specify its name. You may also need to specify the table name that contains the field if more than one table in the database contains a field of the same name. To avoid ambiguity, add the table name and a period before the field name.

SYNTAX

```
table-name.field-name
```

In many cases, programmers feel that supplying both the table and field name makes the code more readable, even if it is not required.

At other times, you may need to specify the database name that contains the table. Add the database name and a period before the table name.

SYNTAX

```
database-name.table-name.field-name
```

This example adds $500 to the credit limit of a customer:

```
sports.Customer.Credit-Limit = sports.Customer.Credit-Limit + 500.
```

5.2.3 Using Widget Attributes

Since a widget attribute is a value, it can be an operand in an expression. For some widget attributes, you can assign the result of an expression to a widget attribute, thereby changing that characteristic of the widget. For example, you could move a widget over five columns by adding five to the current value of the COL attribute.

This is the syntax for referencing a widget attribute value.

SYNTAX

```
widget-name:attribute-name [ IN FRAME frame-name ]
```

You only need the IN FRAME option when two or more frames have a widget of the same name. Here's an example of using a widget attribute in an expression:

```
Mywidget:COL IN FRAME Myframe = Mywidget:COL IN FRAME Myframe + 5.
```

5.2.4 Using Constants

A *constant* is a value that does not change. A constant can be of any data type: Numeric (DECIMAL or INTEGER), CHARACTER, DATE, or LOGICAL.

Table 5–2 shows how to represent constants.

Table 5–2: Constant Data Types

Data Type	Constant Example
CHARACTER	`My-variable = "Second Skin Scuba":`
Numeric	`My-variable = 5000:`
DATE	`Myvariable = 12/31/93:`

LOGICAL	`Myvariable = TRUE:`
	`Myvariable = T:`
	`Myvariable = YES`
	`Myvariable = Y`

As you can see for LOGICAL values, TRUE, T, YES, and Y all represent the TRUE state. Similarly, FALSE, F, NO, and N all represent the FALSE state.

5.3 Using Operators

An *operator* is a language element that represents an action carried out on one or more operands, resulting in the return of a value. Operator categories include:

- Numeric
- Comparison
- Date
- Character
- Logical

Numeric Operators

Use numeric operators with numeric data to perform calculations. Table 5–3 describes the operators.

Table 5–3: Numeric Operators

Name	Operator	Description
Unary Negative	-	Reverses the sign of a numeric expression. `Variable1 =` Credit-Limit.
Unary Positive	+	Preserves the positive or negative value of a numeric expression. `Credit-Limit =` + (Credit-Limit - Balance).
Division	/	Divides one numeric expression by another numeric expression. `Variable1 = Extended-Price /` Quantity.
Multiplication	*	Multiplies two numeric expressions. `Extended-Price = Price *` Quantity.

Subtraction	-	Subtracts one numeric expression from another.
		`New-Price =`
		`  New-Price  (Price * (Discount / 100)).`

Addition	+	Adds two numeric expressions. `Balance = Balance +` Variable1.

Comparison Operators

Use comparison operators to compare fields, variables, constants, or expressions. The result is always a logical value. Table 5–4 describes the comparison operators.

Table 5–4: Comparison Operators

Name	Operator	Description
Less Than	< or LT	Returns TRUE if the first expression is less than the second. `FOR EACH Customer WHERE Balance LT Credit-Limit:`
Less Than or Equal To	<= or LE	Returns TRUE if the first expression is less than or equal to the second. `FOR EACH Customer WHERE Balance LE Credit-Limit:`
Greater Than	> or GT	Returns TRUE if the first expression is greater than the second. `FOR EACH customer WHERE balance > credit-limit:`
Greater Than or Equal To	>= or GE	Returns TRUE if the first expression is greater than or equal to the second. `FOR EACH customer WHERE balance >= credit-limit:`
Equal To	= or EQ	Returns TRUE if two expressions are equal. `FOR EACH customer WHERE country EQ "Finland":`
Not Equal To	<> or NE	Returns TRUE if two expressions are not equal. `FOR EACH customer WHERE country <> "France":`

Date Operators

You use date operators to manipulate date values. Table 5–5 describes the date operators.

Table 5–5: Date Operators

Name	Operator	Description
Date Subtraction	-	Subtracts a number of days from a date to produce a date result, or subtracts one date from another to produce an integer result that represents the number of days between the two dates. `days-over = ship-date` promise-date.
Date Addition	+	Adds a number of days to a date to produce a date result. `due-date = invoice-date` + 30.

Character Operator

You use the character concatenation operator to combine two character expressions. Table 5–6 describes the operator.

Table 5–6: Character Operator

Name	Operator	Description
Concatenation	+	Produces a character value by joining, or concatenating, two character strings or expressions. `DISPLAY "ATTN: "` + contact.

Logical Operators

The logical operators allow you to make compound Boolean conditions. Table 5–7 describes the logical operators.

Table 5–7: Logical Operators

Operator	Description
NOT	Returns a TRUE value if an expression is false, and FALSE if an expression is true. `IF` NOT AVAILABLE customer THEN DO:
AND	Returns a TRUE value if all expressions are true. `FOR EACH customer WHERE balance LE credit-limit` AND country = "Japan":
OR	Returns a TRUE value if at least one expression is true. `FOR EACH customer WHERE balance` > credit-limit OR credit-limit > 50000:

5.4 Using Functions

A *function* is a prewritten program that performs a specific task and returns a value. Functions enable code to be defined once but re-used repeatedly. Some functions require input values.

Progress supports two kinds of functions:

- Pre-defined function
- User-defined function

The following sections provide more details about each of these kinds of functions.

5.4.1 Pre-defined Functions

The Progress 4GL provides numerous pre-defined functions that provide shortcuts for accomplishing many common programming tasks. For example, **Arithmetic** functions (such as RANDOM or EXP) perform mathematical operations on numeric values and **Character** functions (such as LENGTH and SUBSTRING) manipulate character strings or expressions.

The example below uses the pre-defined, Progress-supplied function for calculating square roots: SQRT( ):

```
A = 100 + SQRT(9).
```

Here, 9 is the input parameter and the SQRT function returns 3, the square root. You could write code with Progress to calculate square roots, but using the prebuilt function is far more convenient.

For more information about pre-defined functions, see the function entries of the *Progress Language Reference*.

5.4.2 User-defined Functions

As its name implies, a user-defined function is a function that you can create based on a programming task that you want to solve. Like the pre-defined function, the value of a user-defined function is that you can write the code once and reuse it. You define user-defined functions using the FUNCTION statement. You'll learn more about the FUNCTION statement, its syntax, and its role in relationship to other parts of the Progress program in Chapter 6.

5.5 Understanding Precedence in Expressions

When you have a complex expression of many operators and functions, it's important to know the order in which Progress evaluates parts of the expression. For example, the following nonsensical jumble of language elements is a legal expression:

```
var1 <= var2 + var3 / (ASC("z") - ASC("a")
```

Assume that the variables have the following values:

```
ASSIGN var1 = 20
       var2 = 10
       var3 = 250.
```

Progress follows these steps to evaluate the expression:

1. Progress evaluates functions first. So, Progress executes the ASC functions, which return the ASCII integer code for a supplied character as shown in the following code:

```
var1 <= var2 + var3 / (122 - 97).
```

2. Progress evaluates any part of the expression contained inside parentheses:

```
var1 <= var2 + var3 / 25.
```

   If Progress encounters nested parentheses, Progress evaluates the elements in the innermost set first and then works out from there.

3. Progress has predefined levels of operator importance based on the laws of mathematics and logic. These levels of *precedence* define which operators Progress executes first. Table 5–8, which follows this description, shows the levels of precedence. Operators with a higher number are executed first. In this example, the division operator has the highest level of precedence:

```
var1 <= var2 + 10
```

   Among operators of equal precedence, Progress evaluates them from left to right.

4. The addition operator has the next highest level of precedence:

```
var1 <= 20
```

5. Finally, the less than or equal to operator remains. Since this is a logical operator, the final result of the expression will be TRUE or FALSE. In this case the final value is TRUE.

Table 5–8 depicts the levels of operator precedence which Progress follows from highest (7) to lowest (1):

Table 5–8: Operator Precedence

Level	Operator	Symbol
7	Unary Negative Unary Positive	- +
6	Modulo Division Multiplication	MODULO / *
5	Date Subtraction Subtraction Date Addition Concatenation Addition	- - + + +
4	MATCHES Less Than Less Than or Equal To Greater Than Greater Than or Equal To Equal To Not Equal To BEGINS	MATCHES LT or < LE or <= GT or >> GE or >= = <> BEGINS
3	Not	NOT
2	And	AND
1	Or	OR

5.6 Using Expressions

To help you learn the rules of writing expressions, the tutorial looks at several of the most common uses of expressions, including:

- Calculating values
- Evaluating conditions
- Defining selection criteria
- Manipulating widget attributes

5.6.1 Calculating Values

When you calculate a value, you usually need to save the calculation result. In other words, you need a way to *assign* a new value to a variable or a field. In Progress, the *assignment operator* is the equal sign as shown in the following code:

```
A = 2 + 2.
```

When you use the assignment operator in this way, it is a complete Progress statement. Now suppose you need to make several calculations such as those shown in this code:

```
A = 2 + 2.
B = A + 10.
C = A + B.
```

You've created three Progress statements to perform three calculations. Progress can perform any number of assignments using the ASSIGN statement. This is a partial syntax for the ASSIGN statement.

SYNTAX

```
ASSIGN [ expression ] ...
```

This example recreates the last example using ASSIGN:

```
ASSIGN A = 2 + 2
       B = A + 10
       C = A + B.
```

Note that there is only one period—at the end of the last assignment. Using ASSIGN is more efficient than using multiple assignment statements.

5.6.2 Evaluating Conditions

You can think of a conditional expression as an expression that Progress ultimately evaluates to either TRUE or FALSE. Conditional expressions make up the heart of conditional branching statements, like the IF statement and the CASE statement. (You'll learn about conditional statements in the next chapter.) The conditional expression is the test that determines how a conditional statement branches. Conditional expressions do not make up separate statements, but are usually part of another statement. Here are some examples using the IF statement:

```
/*1*/  IF A = 2 + 2 THEN . . .
/*2*/  IF B > A THEN . . .
/*3*/  D = B > A.
```

These notes explain the code:

1. It appears at first that there is an assignment embedded in the IF statement. Actually, it is a conditional expression. Read the statement like this: Is the current value of A equal to 2 plus 2? The result is either YES or NO—TRUE or FALSE.

2. This second expression uses the greater than operator: Is B greater than A?

3. This example is a statement, not a conditional expression. Here, you are assigning the result of the B > A test to a LOGICAL variable named D.

5.6.3 Defining Selection Criteria

Conditional expressions used to select particular records from a database table or tables are called the *selection criteria*. Progress has many options for specifying selection criteria, but the most commonly used is the WHERE phrase. You'll learn about selecting records in later chapters. This chapter shows you how to create the expressions you'll use with WHERE. This is the general syntax for WHERE.

SYNTAX

```
WHERE [ expression ]
```

Earlier you saw that the FOR EACH statement goes through every record in a database table. When you attach WHERE to FOR EACH, FOR EACH goes through each record and evaluates the WHERE expression using each record. If the expression for a particular record evaluates to TRUE, then Progress executes the statements inside the FOR EACH block. If it evaluates to FALSE, Progress moves on to the next record in the table. Thus, FOR EACH selects records based on the WHERE expression, as shown in the following example:

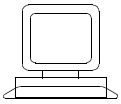
```
FOR EACH Customer WHERE Customer.Cust-Num > 30:
  DISPLAY Cust-Num Name Balance.
END.
```

So, as Progress cycles through the Customer records, it displays information from only those records where the customer number is greater than 30.

5.6.4 Manipulating Widget Attributes

Widgets have many attributes that you can access and change even when the widget is visible on screen. By changing attributes, you can change the appearance or function of the widget. For example, you might want to:

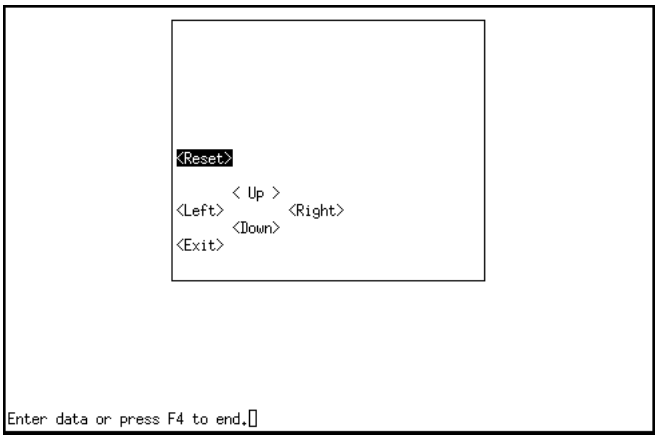
- Move a widget by manipulating the ROW and COL attributes. You can actually create simple animation by incrementing these values.
- Change the size of a widget by manipulating the WIDTH and HEIGHT attributes.
- Change the value of a widget by manipulating the SCREEN-VALUE attribute.



Exercise

Follow these steps for a demonstration of attributes in expressions:

1 ♦ Open `lt-05-02.p` and run it. The interface is shown below:



2 ♦ Experiment by choosing from the Up, Down, Left, Right button group. The Reset button moves. You can move the button around within the frame, but can't move the button beyond the borders of the frame. The ROW and COL attributes, which the buttons manipulate, position a widget within a frame.

3 ♦ Choose the Reset button to move it back to its starting location.

4 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

Here is the code that created the display:

lt-05-02.p

```
      /**********  DEFINE WIDGETS  **********/
      DEFINE BUTTON btn-Up    LABEL " Up ".
      DEFINE BUTTON btn-Down  LABEL "Down".
      DEFINE BUTTON btn-Right LABEL "Right".
      DEFINE BUTTON btn-Left  LABEL "Left".
      DEFINE BUTTON btn-Reset LABEL "Reset".
      DEFINE BUTTON btn-Exit  LABEL "Exit".
      /**********  DEFINE FRAMES  **********/
      DEFINE FRAME Frame1
        btn-Reset AT ROW  8 COLUMN 1
        btn-Up    AT ROW 10 COLUMN 8
        btn-Down  AT ROW 12 COLUMN 8
        btn-Left  AT ROW 11 COLUMN 1
        btn-Right AT ROW 11 COLUMN 15
        btn-Exit  AT ROW 13 COLUMN 1
            WITH SIZE-CHARS 40 BY 14 CENTERED THREE-D.
      /**********  DEFINE TRIGGERS  **********/
/*1*/  ON CHOOSE OF btn-Reset IN FRAME Frame1 DO:
      ASSIGN btn-Reset:ROW = 8
               btn-Reset:COL = 1.
       END.
/*2*/  ON CHOOSE OF btn-Up DO:
        IF btn-Reset:ROW <> 1 THEN
          ASSIGN btn-Reset:ROW = btn-Reset:ROW - 1.
       END.
/*3*/  ON CHOOSE OF btn-Down DO:
        IF btn-Reset:ROW <> 8 THEN
          ASSIGN btn-Reset:ROW = btn-Reset:ROW + 1.
       END.
/*4*/  ON CHOOSE OF btn-Right DO:
        IF btn-Reset:COL <> 30 THEN
          ASSIGN btn-Reset:COL = btn-Reset:COL + 1.
       END.
/*5*/  ON CHOOSE OF btn-Left DO:
        IF btn-Reset:COL <> 1 THEN
          ASSIGN btn-Reset:COL = btn-Reset:COL - 1.
       END.
      /**********  MAIN LOGIC  **********/
      ENABLE ALL WITH FRAME Frame1.
      WAIT-FOR CHOOSE OF btn-Exit.
```

The following notes help explain the code:

1. Choosing the Reset button moves the widget back to its starting location. The trigger assigns the constant values that correspond to the original coordinates to the ROW and COL attributes.

2. This trigger moves the Reset button up one row by decrementing ROW.

3. This trigger moves the Reset button down one row by incrementing ROW.

4. This trigger moves the Reset button right one column by incrementing COL.

5. This trigger moves the Reset button left one column by decrementing COL.

Practice Problems

If you feel like you need some practice writing expressions, complete the exercises below. The filename listed next to each problem contains a sample solution. You can load the file into the Procedure Editor.

Problem 5-1 **lt-05-s1.p**

Using a FOR EACH block, increase all customer credit limits by 10%. Display the customer name, old credit limit, and new credit limit.

Problem 5-2 **lt-05-s2.p**

Display all customer names and balances where the balance is $5000 or more.

Problem 5-3 **lt-05-s3.p**

Display all orders that were placed between 1/14/93 and today.

Problem 5-4 **lt-05-s4.p**

Display all orders that have not been shipped within five days of the promised date.

Problem 5-5 **lt-05-s5.p**

Based on today's date, display the number of the month, weekday, and year.

Problem 5-6 **lt-05-s6.p**

Define an array of dates called Holidays. Initialize the array with the dates you get off from work. Label each element with the holiday name. Display the result.

5.7 Summary

An expression is a combination of 4GL language elements that results in a value. At a minimum, an expression consists of one operand, but can include multiple operands, operators, and functions.

An operand is a language element that represents a value. Operands include:

- **Variables** — A location in memory that holds a value that may change during the execution of a procedure.
- **Database fields** — An individual value from a database record.
- **Widget attributes** — A value that describes a characteristic of a widget like its width, height, and so on.
- **Constants** — A value that does not change during the execution of a procedure.

An operator is a language element that represents an action carried out on one or more operands.

An operation, is the successful combination of operands and operators that results in a value.

A function is a prewritten program that performs a specific task and returns a value. There are two types of functions in Progress: pre-defined functions that Progress provides and user-defined functions that you can create.

Arrays are fields or variables that contain multiple data elements. The extent of an array is the number of elements it contains.

6

Working with Control Structures

In Chapter 3, “Programming the Progress Way,” you learned about the parts of a Progress procedure that define and enable an interface. This chapter discusses the various Progress components that make up the functional part of an application—what’s going on behind the interface. The discussion describes both the structures you can use for containing code and the structures you can use to manage flow of control within a procedure.

Specifically, the chapter discusses:

- Procedures and include files
- Sharing information
- Control statements (blocks)
- Conditional processing statements

6.1 Modularizing Code with Procedures and Include Files

When you tackle an application, your first inclination might be to write a number of individual, potentially lengthy procedures where each procedure performs a complete and perhaps complex task. You can save yourself time and make your application easier to maintain by using modular code. In Progress, you can use different types of procedures to modularize your code. The type of procedure you use depends on the needs of your application. This section discusses procedure files, external procedures, internal procedures, and include files.

6.1.1 Using Procedures

A *procedure* is a functional module of Progress source code. A *procedure file* is an operating system file in text format that contains one or more procedures. By convention, procedure files have a .p file extension.

Since a procedure file can contain many procedures, you could fit an entire application in one procedure file. However, this large procedure file would quickly become hard to read and maintain. Instead, you can create many procedure files, each with one or more procedures in them. One of these procedure files contains the initial, or *main procedure* , which calls other procedures as needed by the application. The procedures called by the main procedure are called *subprocedures*.

Subprocedures let you:

- Minimize your application development time. You don't have to retype a sequence of statements each time you need those statements—you call the subprocedure as many times as you like.
- Achieve consistent application behavior. Actions performed by a set of statements behave in a predictable way when called by, or included within, another procedure.
- Maintain your application easily. If you want to change code, you can change the one subprocedure that contains it, instead of correcting the code in many different areas of the main procedure.

To run a procedure from within another procedure, you use the RUN statement. Here's a partial syntax of the RUN statement.

SYNTAX

```
RUN procedure-name
```

To demonstrate this process, suppose your main procedure creates an interface that displays four buttons. When the user chooses a button, a distinct task executes. You store each of these tasks, or modules, in a separate procedure file. Each module then becomes a subprocedure of the main procedure. Figure 6–1 illustrates this concept.

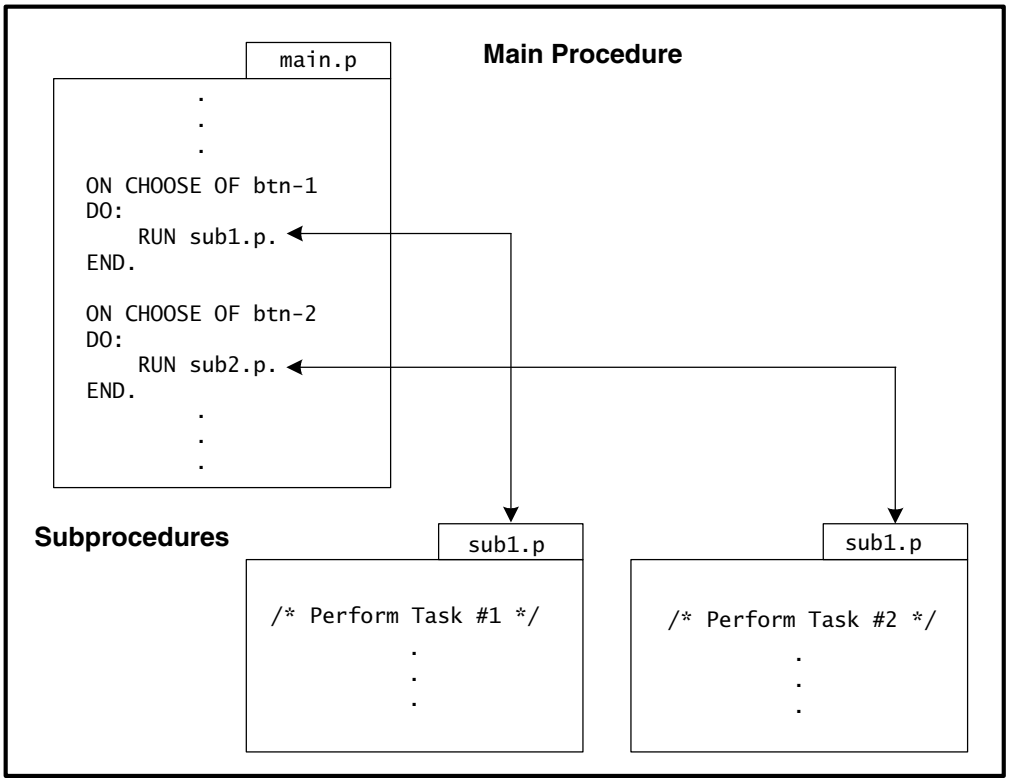


Figure 6–1: Main Procedure and Subprocedures

In the example shown in Figure 6–1, each procedure file contains one procedure. The RUN statement references the procedure filename to access the procedure contained within the file. When you store a subprocedure in a separate procedure file, it is called an *external procedure*.

Progress also allows you to store subprocedures with the main procedure in one procedure file. These subprocedures are called *internal procedures*. You can have many internal procedures in a procedure file.

This is a partial syntax for defining an internal procedure.

SYNTAX

```
PROCEDURE procedure-name :

  statements

END [ PROCEDURE ].
```

By convention, you store internal procedures at the bottom of a procedure file, after the main procedure. Internal procedures do not require a file extension (.p). You access internal procedures in exactly the same way as external procedures—with the RUN statement.

Figure 6–2 shows a simple example of a procedure file that contains two internal procedures.

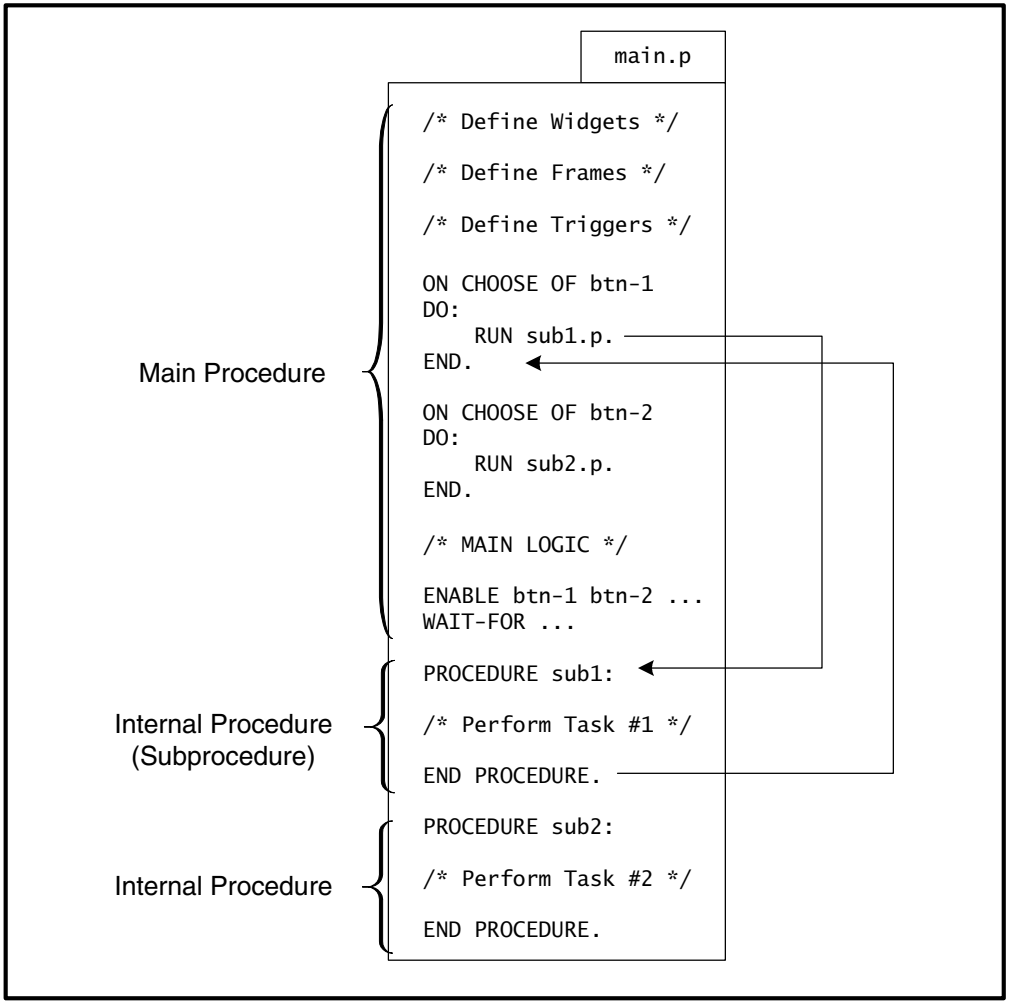


Figure 6–2: Main Procedure and Internal Procedures

All procedures define resources like variables, widgets, frames, and triggers. By default, the main procedure cannot use the resources defined in an internal or external subprocedure. Also, an external procedure cannot use the resources defined in the main procedure. An internal procedure can use the resources defined in the main procedure. This feature is the major advantage that internal procedures have over external procedures.

Now that you know the difference between internal and external procedures and how each of these procedures relates to the main procedure, let's resume the topic of user-defined functions that was introduced in Chapter 5, "Working with Expressions.".

6.1.2 Using User-defined Functions

When you define a function using the FUNCTION statement, the statement is treated like an internal procedure. The FUNCTION statement can use the resources defined in the main procedure, helping you to simplify the actual statement that you need to code. However, user-defined functions differ from internal procedures because the function can return a value, and the function can be called from within a Progress expression. In addition, user-defined functions have certain restrictions such as you cannot define work tables, temporary tables or shared variables in a user-defined function.

Here's a partial syntax of the FUNCTION statement.

SYNTAX

```
FUNCTION function-name
 [ RETURNS ] data-type
 [ ( param [ , param ] ... ) ]
```

Table 6–1 describes elements of the partial FUNCTION statement:.

Table 6–1: Elements of the FUNCTION Statement

Component	Description
function-name	The name you are using to identify the function.
data-type	Describes the data type of the value returned from the function.
param	Indicates a parameter of the function. If you are including parameters in a given FUNCTION statement, you can include as many parameters as you need.

The FUNCTION statement offers you the advantages of defining an internal procedure while also returning a value, a capability supported by a function. A function applies the logic that defines the function to the values that are passed to it, and provides a single return value to the calling procedure (that is, the procedure that contacted it to perform its function).

NOTE: You cannot define temp-tables, work-tables, shared variables, shared frames, shared browses or streams within a user-defined function.

To gain a better understanding of the user-defined function, let's review the code in Figure 6–3.

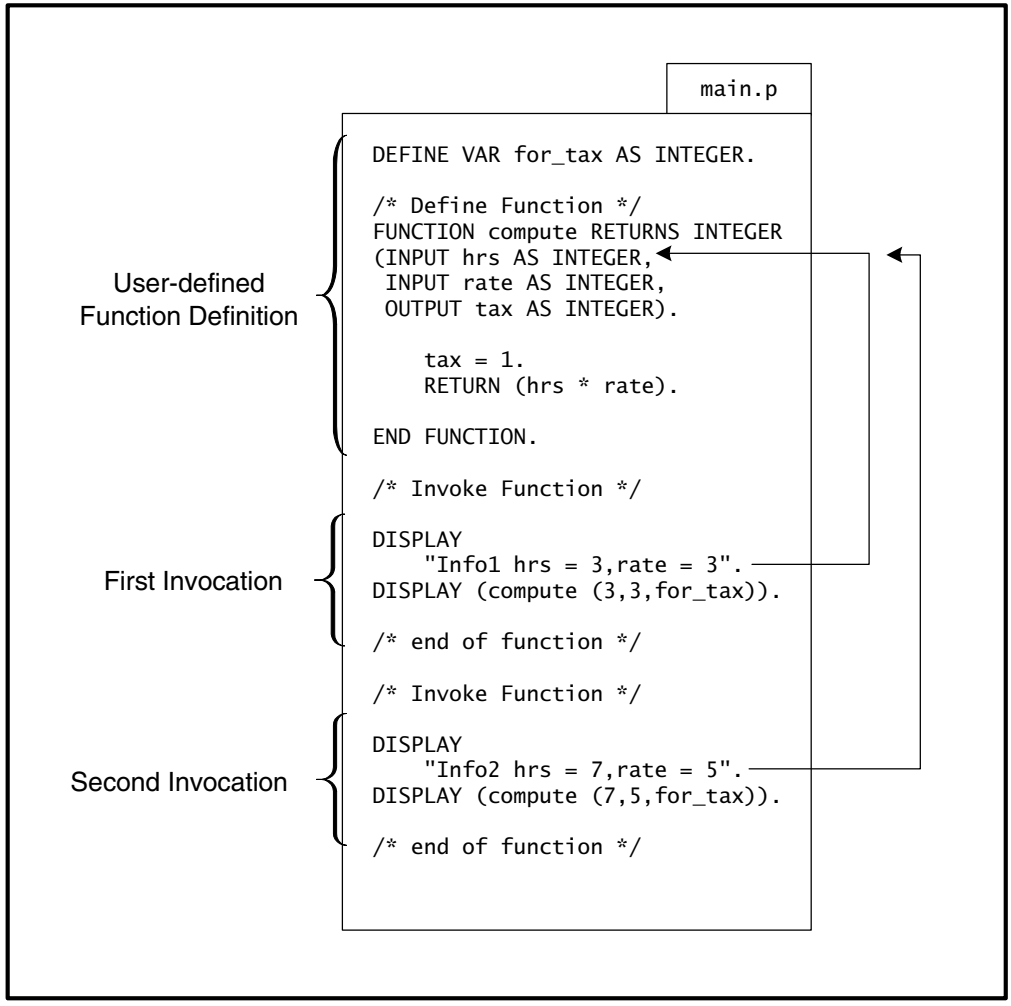


Figure 6–3: Defining and Using a User-defined Function

This example shows an extract of a main procedure in which the user-defined function compute is defined and then used, or invoked, in two separate instances. In this example, the function accepts input and output parameter details for each function call, and substitutes the actual parameters to perform its defined operation. As you can see from the two invocations of the function, the values passed to the function are different, but the function's defined operation is applied in the same way; within the function definition, the value for hours that is supplied from the calling procedure is multiplied by the value supplied for rate from the same calling procedure. The single return value derived is returned to the calling procedure. The parameters

associated with this user-defined function are further discussed in the "Using Parameters to Pass Values" section presented later in this chapter.

6.1.3 Using Include Files

A subprocedure is a complete functional module of code—you can plug it into a larger application wherever you want. What if you have common code that does not form a complete module? Even though this code doesn't form a functional module, you may want to modularize the common code for the same space-saving and maintenance reasons that make subprocedures valuable. For example, you may have standard interface layouts.

An *include file* is a text file that contains Progress source code. When you reference include files in your procedures, Progress replaces the reference with the contents of the include file at compile time. Thus, the include filename is only a placeholder for the actual code. Because Progress compiles the main procedure and the include file together, they can share the same variables, frames, and so on. Include files have an .i file extension by convention.

To reference an include file, add the include filename with its file extension to the procedure and enclose it between braces { }.

Figure 6–4 illustrates how include files work.

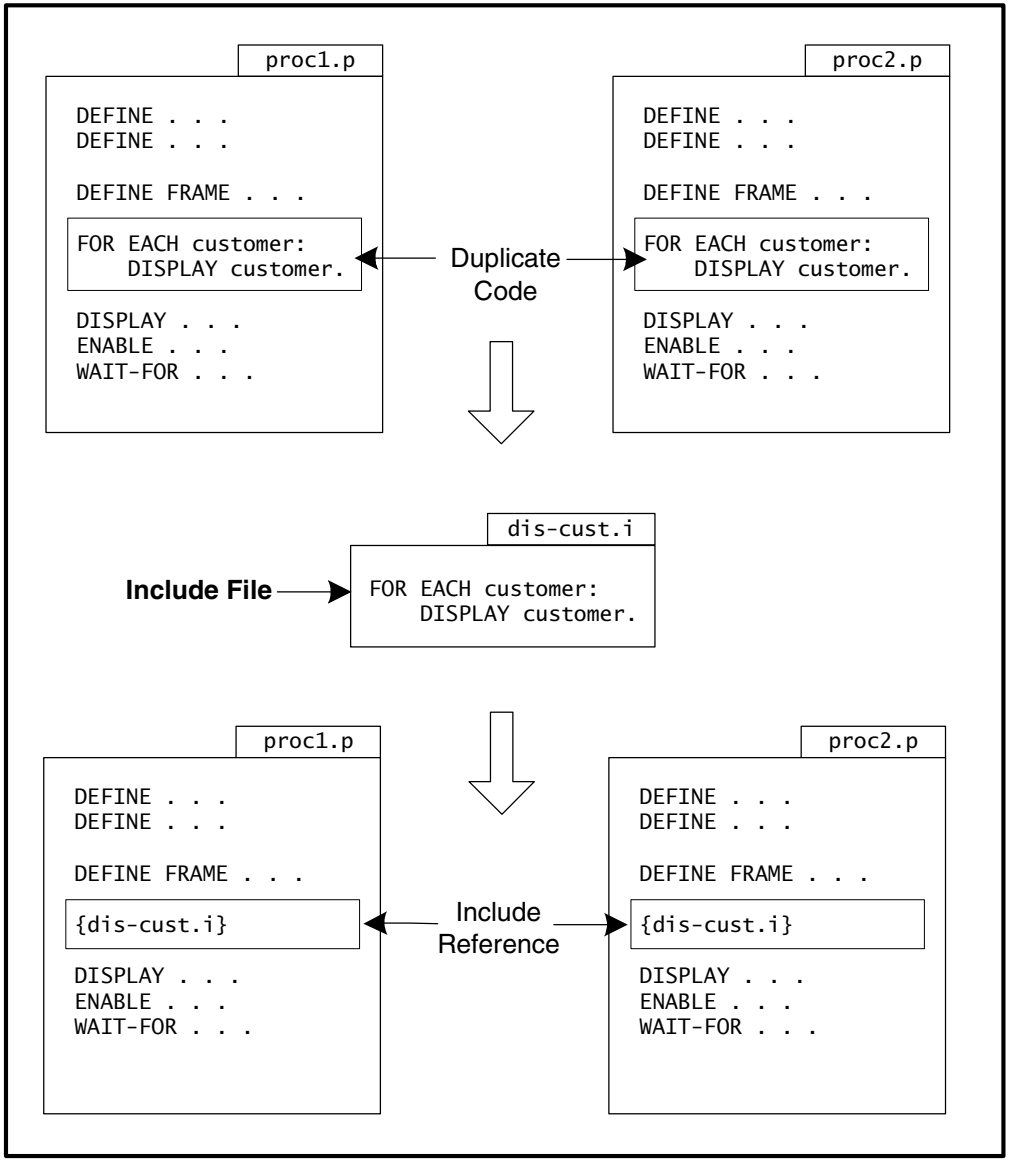


Figure 6–4: Using an Include File

6.2 Sharing Information Between Procedures

When you run external procedures, you often want to share information among them. There are three ways to share information among procedures:

- Define shared variables to make a variable and its value available to other procedures
- Pass data between two procedures using a parameter
- Pass programming literals (like field names or table names) as arguments

When you run internal procedures from a main procedure, the variables defined in the main procedure are always available to the internal procedure. Therefore, the information-sharing techniques listed above are not usually necessary for internal procedures.

6.2.1 Using Shared Variables

When you define a variable in one procedure, you can only use it in that procedure or the internal subprocedures of that procedure. This kind of variable is known as a *local* variable. If you want more than one procedure to use a variable, then you need to use a *shared* variable. To defined a shared variable, use the DEFINE VARIABLE statement.

SYNTAX

```
DEFINE [ [ NEW ] SHARED ] VARIABLE variable-name
 { AS data-type | LIKE field }
```

When you use a shared variable, only one procedure can own the shared variable. That procedure creates it, and when that procedure ends, so does the availability of the shared variable. The owner procedure must use the two keywords NEW SHARED in the DEFINE VARIABLE statement, as in this example:

```
DEFINE NEW SHARED VARIABLE Field1 AS CHARACTER.
```

Any other procedure that uses the shared variable must also define it, but without the keyword NEW:

```
DEFINE SHARED VARIABLE Field1 AS CHARACTER.
```

Note that any format phrase option, such as a label or format string, specified in the original definition (NEW SHARED), does not apply to the variable in subsequent definitions (SHARED). You can specify different options for the variable in different procedures.

Also note that you cannot define shared variables in a user-defined function.

Figure 6–5 shows the relationship among three procedures and the shared variable that two of these procedures access. The `proc1.p` procedure defines a NEW SHARED variable called Shareinfo and runs a procedure called `proc2.p`. The `proc2.p` procedure does not use Shareinfo but calls `proc3.p`, which does. Procedure `proc3.p` needs to define Shareinfo as a shared variable. Even though you don't run `proc3.p` from `proc1.p`, it still can still access the shared variable. Once `proc1.p` defines the variable, any procedure that also defines that variable can access the data stored there.

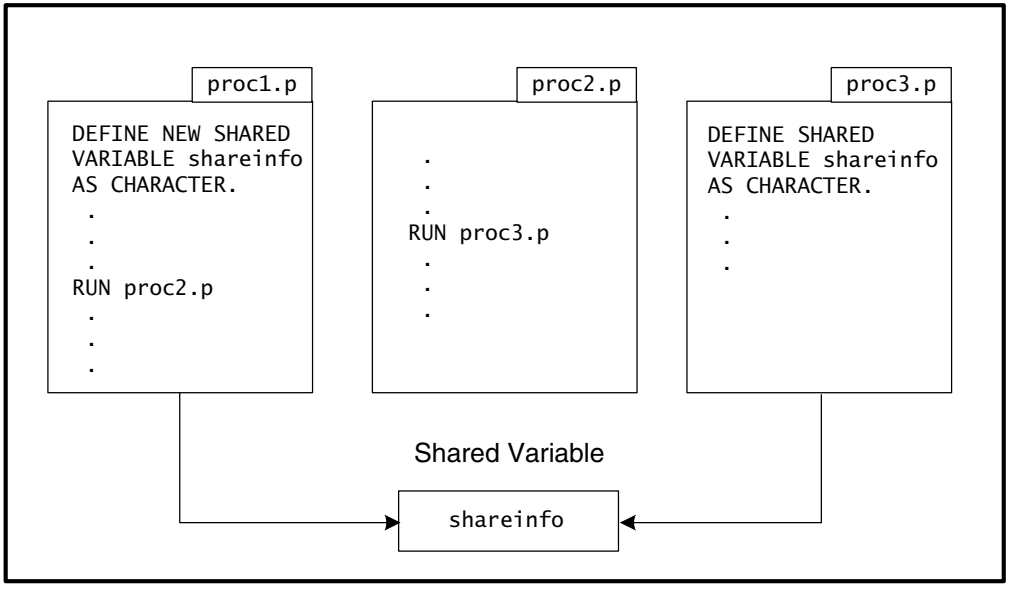


Figure 6–5: Shared Variables

6.2.2 Using Parameters to Pass Values

Shared variables are one way that procedures share information, although perhaps not the best way. It may not be clear to other programmers who read your code what relationship each procedure has to the shared variable. Another way of sharing values is to pass parameters. A *parameter* is a variable in a called procedure whose function is to accept an input value from the calling procedure, output a value to the calling procedure, or both. For example, you can use a parameter with an internal procedure to create something like a function that accepts input values and outputs return values. If you define such a procedure, you can use it many times in the main procedure.

You define a parameter with the DEFINE PARAMETER statement.

SYNTAX

```
DEFINE
  { INPUT | OUTPUT | INPUT-OUTPUT } PARAMETER parameter
  { AS data-type | LIKE field }
  { [ format-phrase ] }
```

The statement options are similar to the options you used with the DEFINE VARIABLE statement. The parameter types are:

- **INPUT** — A parameter defined in a called procedure that receives a value from the calling procedure.
- **OUTPUT** — A parameter defined in a called procedure that returns a value to the calling procedure.
- **INPUT-OUTPUT** — A parameter defined in the called procedure that can receive a value from the calling procedure and return a value back to it.

All procedure and function parameters (that is, pre-defined and user-defined functions) are passed by value. This means that for any INPUT-OUTPUT or OUTPUT parameter, the field or variable that receives the output value is not set by the called procedure until and unless the procedure returns without error.

To call a procedure that contains parameters, place a comma-separated list of parameters enclosed in parentheses at the end of the RUN statement. You can specify a type for each parameter in the list. Similarly, to call either type of function, you also define a comma-separated list of parameters enclosed in parentheses within a statement that invokes the function.

Parameters Used in the RUN Statement

The following syntax shows where to define parameters used within a RUN statement.

SYNTAX

```
RUN procedure-file
[ ( [ { INPUT | OUTPUT | INPUT-OUTPUT } expression ] , ... ) ]
```

Here's is a sample RUN statement:

```
RUN proc2.p (INPUT-OUTPUT Variable1, INPUT "Constant", OUTPUT Variable2).
```

If you do not specify the parameter type keywords in the RUN statement, Progress assumes that the parameters are INPUT parameters. The number and type of parameters in the called procedure must match the number and type of procedures in the RUN statement:

```
DEFINE INPUT-OUTPUT PARAMETER Param1 AS CHARACTER.
DEFINE INPUT PARAMETER Param2 AS CHARACTER.
DEFINE OUTPUT PARAMETER Param3 AS CHARACTER.
        .
        .
        .
DISPLAY Param1 Param2 Param3 WITH FRAME Frame1.
```

The called procedure matches the values passed by the RUN statement by associating the first value to the first defined parameter and so on.

Parameters Used in the FUNCTION Statement

The following is a partial syntax of the FUNCTION statement. It presents the syntax that you can use to define parameters for a user-defined function using the FUNCTION statement.

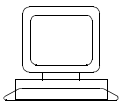
SYNTAX

```
FUNCTION function-name
 [ RETURNS ] data-type
 { { [ INPUT ] |  OUTPUT | INPUT-OUTPUT }
       param-name AS data-type
   }
```

The FUNCTION statement matches parameters by associating the first value to the first defined parameter, the second value to the second defined parameter, and so on. The number and type of parameters in the calling procedure must match the number and type of procedures in the FUNCTION statement.

For example, in the user-defined function code presented in Figure 6–3 earlier in this chapter, the function called compute is written to accept two input parameters defined as INTEGER, and one output parameter that is also defined as INTEGER. In the first invocation of the function, the actual input parameter values 3 and 3 replace the input parameter values hr and rate. Within the function definition, these two values are multiplied (hrs * rate) and the function returns the result, 9, to the calling procedure. The output parameter for_tax is also returned to the calling procedure.

The second invocation in Figure 6–3 shows how this code is re-usable. The second invocation provides two different input parameter values, 7 and 5, but the same operation defined for the function is performed.



Exercise

Follow these steps to demonstrate passing parameters and shared variables:

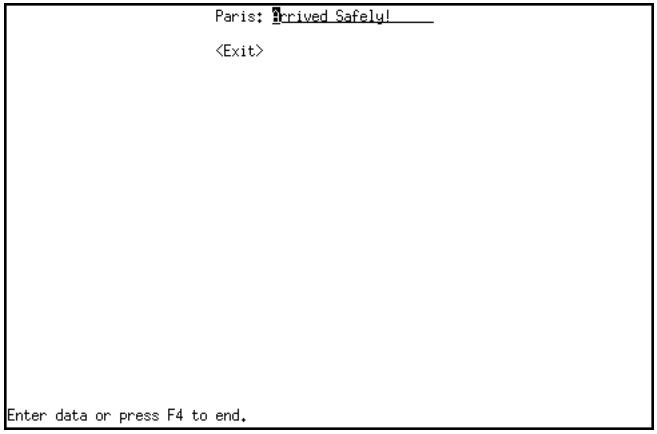
1 ♦ Open the following files:

- `lt-06-01.p`
- `lt-06-02.p`
- `lt-06-03.p`

2 ♦ Choose Buffer→ List.

3 ♦ Select `lt-06-01.p` from the Buffer List and choose OK.

4 ♦ Choose Compile→ Run. The following display appears:



The code in procedure `lt-06-03.p` creates this display, but the value "Paris" originates in the main procedure `lt-06-01.p`, passes through `lt-06-02.p`, and arrives safely for use in the display.

5 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

Here is the code for the three procedures that create the display:

lt-06-01.p

```
/*1*/  DEFINE NEW SHARED VARIABLE Boston AS CHARACTER
        INITIAL "Arrived Safely!".

/*2*/  RUN lt-06-02.p
```

lt-06-02.p

```
/*3*/   DEFINE SHARED VARIABLE Boston AS CHARACTER.
/*4*/   DEFINE VARIABLE Londton AS CHARACTER.

        ASSIGN London = Boston.

/*5*/   RUN lt-06-03.p (INPUT London).
```

lt-06-03.p

```
/*6*/  DEFINE INPUT PARAMETER Paris AS CHARACTER FORMAT "x(20)".
      DEFINE BUTTON btn-Exit LABEL "Exit".

      DEFINE FRAME Frame1
          SKIP(1)
          Paris SKIP(1)
          btn-Exit
              WITH SIDE-LABELS NO-BOX CENTERED THREE-D.

/*7*/  DISPLAY Paris WITH FRAME Frame1.
      ENABLE ALL WITH FRAME Frame1.
      WAIT-FOR CHOOSE OF btn-Exit.
```

NOTE: The THREE-D option is relevant only on a Windows client; it is ignored by a character client.

These notes explain the code and trace the value through the three procedures:

1. The main procedure creates a new shared variable and gives it an initial value.

2. The main procedure calls the second procedure. Note that the RUN statement uses no parameters.

3. For this procedure to access the shared variable, it too must define it.

4. This local variable passes on the value as a parameter. It is not necessary, because a shared variable can also be a parameter. The point here is to show how to pass the value of a local variable.

5. This RUN statement calls the next procedure and specifies the local variable London as a parameter. Note the parentheses.

6. The final procedure defines one parameter. During run time, Progress checks to make sure that both the calling and called procedures specify the same number of parameters.

7. Because of the defined parameter, you can use Paris as a valid reference anywhere you can use a variable name.

6.2.3 Using Arguments to Pass Literals

While parameters allow you to pass **values** to accomplish a **run-time** task, arguments allow you to pass **literals** to allow code to **compile**. For example, you can use arguments to change a procedure that works with only one database table to one that works with any database table. The following code creates a report of all customer names in the Customer table:

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

You can modularize this code to work for any table and field name pair by replacing the literals. *Literals* are programming symbols, such as table names, field names, or variable names. An *argument* passes a literal, not a value. To create a procedure that accepts arguments, replace the literals with argument references, as shown:

```
FOR EACH {1}:
  DISPLAY {2}.
END.
```

Progress cannot compile this code because {1} is not a table name and {2} is not a field name. However, if you place any valid table name at {1} and any valid field name at {2}, the procedure could compile. Once the procedure compiles, the table and field name become valid references to actual data, and the procedure runs.

What the external procedure needs is arguments (valid literal references) to complete the code. You can pass arguments to an external procedure by specifying them after the RUN statement, as shown in the following code:

```
RUN proc2.p "Customer" "Name".
```

“Customer” is argument 1 and “Name” is argument 2. To reference the arguments in the called procedure, enclose the argument number in braces { } as shown in the following code:

```
FOR EACH {1}:   /* {1} becomes Customer when compiled */
  DISPLAY {2}.  /* {2} becomes Name when compiled */
END.
```

Now the external procedure can compile by substituting the argument references for the arguments. Remember, you are literally passing the argument specified in the RUN statement, not the value of the argument. In this case, arguments turn out to be very useful because now you have a procedure that can create a report for any valid table and field pair.

You can also use variables in the RUN statement, if the values of the variables are valid literals for the called procedure, as shown in the following code:

```
DEFINE VARIABLE Table-Name AS CHARACTER.
DEFINE VARIABLE Field-Name AS CHARACTER.

ASSIGN Table-Name = "Customer"
       Field-Name = "Name".

RUN proc2.p Table-Name Field-Name.
```

The drawback to using arguments is that you cannot compile the external procedures that contain them until run time, which may slow your application down. In many cases, you can achieve the same kind of modularity by using parameters or shared variables. In other words, use arguments as a last resort.

Using Arguments with Include Files

Although arguments may not be the best way to modularize external procedures, they work well with include files. When Progress encounters an include file in a procedure, it retrieves the contents of the include file, substitutes the arguments, and compiles the statements as part of the main procedure. Therefore, Progress resolves all include file references to arguments at compile time.

The example below shows how to use arguments with an include file:

```
{include1.i "Variable1" "Customer.Name"}
```

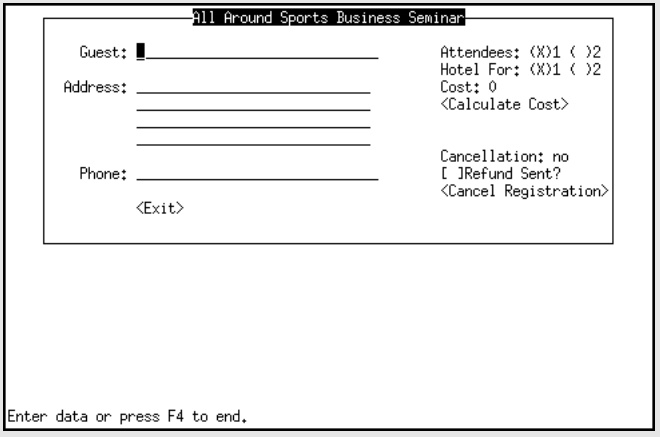
You reference the arguments in the include file the same way you reference arguments in called procedures, as shown in this example:

```
DISPLAY {1} {2} WITH FRAME Frame1.
```

Practice Problems

Problem 6-1: `lt-06-s1.p` `lt-06-i1.i` `lt-06-e1.p`

First, duplicate the interface shown below with a DEFINE FRAME statement. Cost and Cancellation should be text widgets. All the widgets should be enabled on startup except for Refund Sent.



6.3 About Blocks

The Progress 4GL is a block-structured language. A *block* is a series of 4GL statements grouped together and treated as a single unit. Blocks are important because Progress bases many default behaviors on blocks. For example, earlier you learned about the default frame. If you don't define a frame, Progress creates one for you (the default frame) when you display widgets. Actually, each block in your procedure can have a default frame; default frame behavior is based on blocks.

There are several types of blocks that you can use within your application. They include:

- Procedures
- Internal procedures
- Triggers
- Control blocks:
  - DO
  - REPEAT
  - FOR EACH

Here are some general statements about blocks, although you'll learn about some exceptions in the sections that follow:

- A block begins with a *block header* statement.
- A block ends with the END statement.
- Most block types come with implicitly defined services called *block properties*. Default frame allocation, for example, is a service, or block property, of many blocks. In the tutorial, you'll work with these block properties: looping, record reading, and frame allocation.
- Many blocks have syntax options that allow you to define services explicitly, either to override implicit services or to add new services.

- You can name blocks by adding a label and a colon before the block header as shown in the following code:



6.3.1 Procedure Block

Every procedure includes a procedure block, whether the procedure is a main procedure or an external or internal subprocedure. The procedure block encompasses the entire procedure file (a .p file). You can include any Progress 4GL statement inside the procedure block, and you can also nest other blocks inside the main procedure block. However, the main procedure block cannot be enclosed in another block. The procedure block does not have a block header statement—Progress automatically starts a new procedure block when you run a new procedure file. The procedure block does not require or allow an END statement to terminate it—the end of the procedure file marks the end of the procedure block.

Quite literally, the procedure block encompasses all the code in a procedure file. However, only the statements that are not part of another block belong to the procedure block. The procedure block is also characterized by these features:

- Although you can place internal procedures anywhere in a procedure file, by convention, you usually place them at the very end of the procedure file.
- Application blocking (WAIT-FOR) is normally part of the the procedure block.

A typical event-driven procedure contains definitions, triggers, and main logic. The task of blocking the interface with the WAIT-FOR statement usually falls to the procedure block, because it is the outermost block. Blocks within the procedure block normally work with the interface created and blocked for input in the procedure block.

The procedure block has implicit frame allocation, that is, any screen I/O statements in the procedure block that do not explicitly use a defined frame perform their functions in a default frame owned by the procedure block.

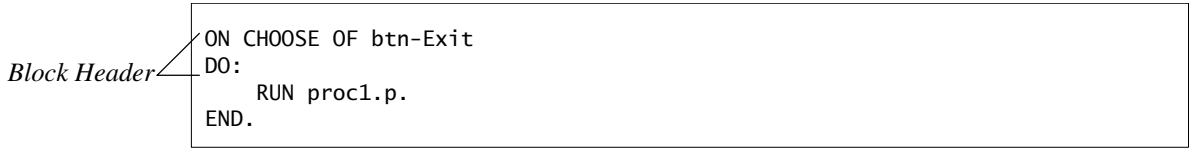
6.3.2 Trigger Blocks

This is the basic and partial syntax for triggers.

SYNTAX

```
ON event-list OF widget-list
  trigger-block
```

The trigger header statement is the ON statement. You enclose the trigger block statements between DO and END as shown in the following code:



The trigger block properties are identical to the procedure block's.

As you've seen, the trigger block allows the user to control an application by way of events.

You may, however, want to programmatically execute a trigger. In this case, you could use an APPLY statement. This is a partial syntax for the APPLY statement.

SYNTAX

```
APPLY event [ TO widget-phrase ]
```

Here's an example of using the APPLY statement:

```
APPLY "CHOOSE" TO btn-Exit.
```

Note that the event name in an APPLY statement is enclosed in quotation marks, which is not the case for the ON statement.

In general, you should avoid using APPLY because too many APPLY statements make your code hard to follow and also because APPLY is not a very efficient way to execute triggers. If you have code in a trigger and want to execute the same code from another location, move the trigger code into an internal or external procedure. Then, you can access the code with a RUN statement from inside a trigger or anywhere else.

6.4 Working with Control Blocks

Program control blocks let you group statements together as a smaller unit within a larger block such as procedures and trigger blocks.

The control blocks include:

- DO
- REPEAT
- FOR EACH

6.4.1 Using the DO Block

You use the DO block to group statements together and treat them as a single unit. DO has no implicit block properties. However, you can add explicit looping and frame allocation properties to the DO block by using the syntax shown below.

SYNTAX

```
[ label: ]
DO [ variable = expression1 TO expression2 ]
     [ WHILE expression ]
  { [ frame-phrase ] } :
  END
```

Here are some common uses of the DO block:

- To group together statements within a conditional statement like IF:

```
IF Balance > 1000 THEN
    DO:
        MESSAGE "Large Outstanding Balance".
        Freeze-Account = YES.
    END.
```

- To provide explicit services to another block. For example, triggers have implicit frame allocation, but do not have the frame phrase option on the header statement. So, if the statements in your trigger operate with a frame other than one Progress expects, you can nest a DO block within the trigger block. As shown in the following code, this technique eliminates having to specify the frame on every screen-output statement in the trigger:

```
ON CHOOSE OF btn-New
    DO:
        DO WITH FRAME Frame2:
            ASSIGN Cost = Price + Tax.
            DISPLAY Price Tax Cost.
        END.
    END.
```

- To provide looping with an incrementing variable using the TO syntax as shown in this code:

```
DO i = 1 TO 31:
    ASSIGN Total = Total + January[i].
END.
```

- To provide conditional looping with the WHILE syntax. WHILE allows you to specify an expression. On each iteration, the block evaluates the WHILE expression and then executes the block, if the expression evaluates to TRUE. Review the WHILE expression in this code:

```
ASSIGN i = 1.
DO WHILE(i < 32):
    ASSIGN Total = Total + January[i]
           i = i +1.
END.
```

It's important to note that the DO and END syntax of a trigger block does not make it a DO block. As described in this section, DO is a control structure that has no block properties unless you specify them with optional syntax. The trigger block DO has no explicit syntax options.

6.4.2 Using the REPEAT Block

The REPEAT block provides implicit looping and frame allocation, as well as syntax options for explicit looping and frame allocation. This is the syntax for the REPEAT block.

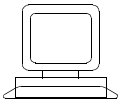
SYNTAX

```
[ label: ]
REPEAT [ variable = expression1 TO expression2 ]
         [ WHILE expression ]
   { [ frame-phrase ] } :
 END
```

Like the DO block, the REPEAT block can define the iterations of the block by incrementing a variable to a desired value or by specifying a WHILE expression. If the block contains neither of these options, then the REPEAT block loops continuously. In other words, there is no implicit terminating condition.

Progress provides the user with an implicit method of ending the iterations of a REPEAT block: the **END-ERROR** key. This feature makes REPEAT useful for coding repetitive tasks, especially where you cannot predict how many iterations of the task the user will want to complete. The terminating condition is in the hands of the user. By default, Progress displays a message instructing the user to press **END-ERROR** to end the block.

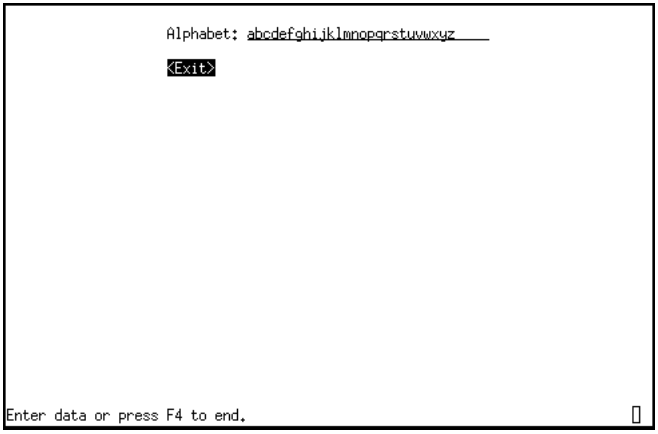
You can also programmatically control the end of a REPEAT block using the LEAVE statement. The LEAVE statement is useful when you want to evaluate a condition within a REPEAT block and exit the block based on the outcome. After Progress encounters the LEAVE statement, it continues execution with the first statement after the end of the block.



Exercise

This exercise uses a REPEAT block to type characters in a fill-in field. Follow these steps for a demonstration of the REPEAT block:

1 ♦ Open `lt-06-04.p` and run it. The following display appears:



As soon as the interface displays, the letters of the alphabet quickly fill the Alphabet field. Of course, if you have a very fast computer, you may only see the end result of the loop—the entire alphabet.

2 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

Here is the code that created the display:

lt-06-04.p

```
      /**********  DEFINE WIDGETS  **********/
      DEFINE VARIABLE i AS INTEGER.
      DEFINE VARIABLE Alphabet AS CHARACTER FORMAT "x(30)".
      DEFINE BUTTON btn-Exit LABEL "Exit".

      /**********  DEFINE FRAME  **********/
      DEFINE FRAME Frame1
          SKIP(1)
          Alphabet SKIP(1)
          btn-Exit
              WITH SIDE-LABELS NO-BOX CENTERED THREE-D.

      /**********  MAIN LOGIC  **********/
/*1*/  ASSIGN i = ASC("a").
      ENABLE ALL WITH FRAME Frame1.
       REPEAT:
/*2*/      APPLY CHR(i) TO Alphabet.
/*3*/      IF CHR(i) = "z" THEN
            LEAVE.
        i = i + 1.
       END.
      WAIT-FOR CHOOSE OF btn-Exit.
```

These notes help explain the code:

1. The ASC function returns the integer code for a single character. (The integer value is interpreted from the current character set)

2. The CHR function returns the character equivalent of an integer code. Applying a character to a fill-in field is equivalent to a user typing that character.

3. When the loop reaches “z,” the LEAVE statement terminates the loop.

In addition, the REPEAT block:

- Can use explicit looping with either the TO or WHILE syntax. Using either of these options creates a loop with an explicit terminating condition.

- Can use a frame other than the default by specifying one with the frame phrase.

6.4.3 Using the FOR EACH Block

The FOR EACH block has implicit looping, frame allocation, and record reading block properties. FOR EACH is an essential structure for moving through the records of a database table. Each iteration of the block reads a new record from the table. When no records are left, the block ends. While the FOR EACH header reads a record, the statements inside the FOR EACH block typically work with the data of that record. Therefore, the main use of this structure is to move through and process the data of a database table. Chapter 8, "Using Database Records," and Chapter 9, "Selecting, Sorting, and Relating Records," extensively describe the use of FOR EACH as a record-reading tool. This section introduces the properties of the FOR EACH block.

This is the basic syntax of FOR EACH.

SYNTAX

```
[ label: ] FOR EACH [ record-phrase ]
  [ variable = expression1 TO expression2 ]
  [ WHILE expression ]
  { [ frame-phrase ] } :
END
```

This simple example of FOR EACH shows the implicit block properties:

```
/*1*/  FOR EACH Customer:  /*2*/
/*3*/      DISPLAY Name.
       END.
```

The following notes describe the implicit block properties:

1. The record phrase defines the implicit looping of a FOR EACH block. The actual number of iterations of the FOR EACH statement is the sum of all records of all tables referenced in the record phrase. However, the FOR EACH statement does not necessarily execute the statements inside the block for each record of the database table. FOR EACH executes these statements when the current record matches the criteria specified in the record phrase. In this example, the statements execute for every record, since no selection criteria exist.

2. The record phrase is a Progress language structure that lets you specify options for selecting, sorting, and relating records. In the last chapter, you learned how to create record-selection expressions using the WHERE syntax. You've also seen that specifying a table name, as in this example, selects all the records of the table. Both of these techniques are uses of the record phrase.

3. This statement performs screen output, so Progress creates a default frame for the FOR EACH block to use.

This example of FOR EACH shows the explicit block properties:

```
                  /*1*/                   /*2*/
FOR EACH Customer WHERE Balance < 1000 WITH FRAME Frame3 NO-BOX:
    DISPLAY Name.
END.
```

These notes describe the explicit block properties:

1. The WHERE expression in the record phrase restricts the number of records that execute the statement of the block body. Although Progress reads every record and compares it to the WHERE expression, Progress executes the block statements for only those records where the Balance field is less than 1000. On the other hand, the default implicit record reading property executes the block for every record.

2. The frame phrase here names a frame for use with the screen-output statements of the block. Frame3 could be either a previously defined frame or a new frame created for the block.

   Although the TO and WHILE syntaxes for explicit looping are valid with FOR EACH, they are not commonly used.

6.5 Block Property Summary

Table 6–2 shows the properties that Progress associates with each type of block. Implicit properties are those that come by default. Explicit properties are those that you can associate with the block by adding the Progress syntax listed in the table.

Table 6–2: Block Properties

Property	REPEAT	FOR EACH	DO	Procedure	Internal Procedure	Trigger
Looping	Implicit Explicit: WHILE TO	Implicit Explicit: WHILE TO	 Explicit: WHILE TO	None	None	None
Record reading	None	Implicit Explicit: Record Phrase	None	None	None	None
Frame allocation	Implicit Explicit: WITH FRAME	Implicit Explicit: WITH FRAME	 Explicit: WITH FRAME	Implicit	Implicit	Implicit

Practice Problems

These problems give you some practice with blocks.

Problem 6-2: `lt-06-2a.p and lt-06-2b.p`

Using a FOR EACH block and a DO block, loop through all the customer records and count the number of customers whose credit limit is greater than $15,000. Display the customer number, name, and credit limit.

Problem 6-3: `lt-06-s3.p`

Using the REPEAT block, increment a counter by 1 up to 10 times and display the counter. Once the counter has a value of 10 leave the REPEAT block.

6.6 Conditional Processing

Conditional processing structures are those that allow the flow control within a procedure to branch based on a condition. There are two statements that allow you to test conditions and define what code to execute under what circumstances: the IF statement and the CASE statement.

6.6.1 Using the IF Statement

The IF statement allows you to test for a condition and then execute code if the expression is TRUE. The ELSE option allows you to specify a second block of code to execute if the expression is FALSE. This is the syntax.

SYNTAX

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

The code for either the THEN or ELSE branch can be either a statement or a block, as this example shows:

```
      FOR EACH Customer:
          IF Balance < 10000 THEN
/*1*/             DO:
                   Small-Accounts = Small-Accounts + 1.
                   Small-Total = Small-Total + Balance.
                  END.
           ELSE
/*2*/           DO:
                  Big-Accounts = Big-Accounts + 1.
                  Big-Total = Big-Total + Balance.
                 END.
       END.

      DISPLAY "There are" Small-Accounts "accounts under $10,000."
              "The total outstanding balance is" Small-Total SKIP(1).

      DISPLAY "There are" Big-Accounts "accounts over $10,000."
              "The total outstanding balance is" Big-Total.
```

The block at point 1 executes when the expression is TRUE, while the block at point 2 executes when the expression is FALSE.

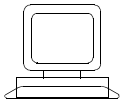
6.6.2 Using the CASE Statement

Another means of conditional control is the CASE statement. Use the CASE statement when you want to test the result of an expression against several values. You could use nested IF statements, but the CASE statement is much clearer.

This is the syntax for CASE.

SYNTAX

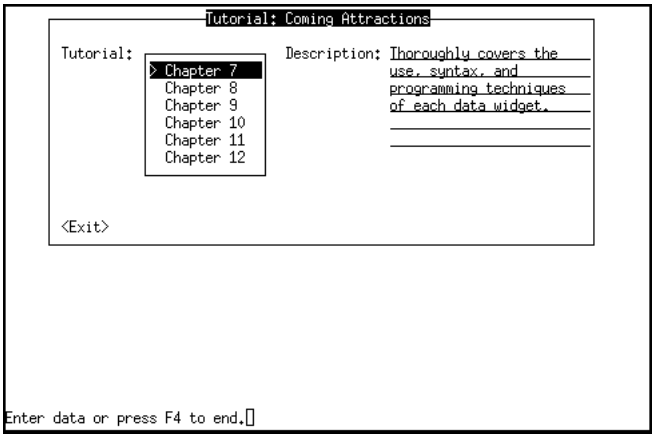
```
CASE expression :
  { WHEN value [ OR WHEN value ] ...
    THEN { block | statement }
  } ...
  [ OTHERWISE { block | statement } ]
END [ CASE ]
```



Exercise

Follow these steps to see how a CASE statement and a selection list can work together:

1 ♦ Open `lt-06-05.p` and run it. The display shown below appears:



The selection list allows you to choose a chapter. A short description of the chapter then appears in the editor widget.

2 ♦ Use ↑ and ↓ to move the highlight bar in the selection list.

3 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

This is code that created the display:

lt-06-05.p

```
      /**********  DEFINE WIDGETS  **********/
/*1*/  DEFINE VARIABLE Chapter AS CHARACTER LABEL "Description"
        VIEW-AS EDITOR INNER-CHARS 25 INNER-LINES 6.
/*2*/  DEFINE VARIABLE Tut-List AS CHARACTER LABEL "Tutorial"
        INITIAL "Chapter 7" VIEW-AS SELECTION-LIST INNER-CHARS 12
        INNER-LINES 6 LIST-ITEMS "Chapter 7", "Chapter 8", "Chapter 9",
        "Chapter 10", "Chapter 11", "Chapter 12".
      DEFINE BUTTON btn-Exit LABEL "Exit".

       /**********  DEFINE FRAMES  **********/
      DEFINE FRAME Frame1
        Tut-List AT ROW 2 COLUMN 2 Chapter AT ROW 2 COLUMN 30
         btn-Exit AT ROW 12 COLUMN 2
           WITH SIDE-LABELS TITLE "Tutorial: Coming Attractions" CENTERED
              THREE-D.

      /**********  DEFINE TRIGGERS  **********/
      ON VALUE-CHANGED OF Tut-List DO:
/*3*/     CASE Tut-List:SCREEN-VALUE:
/*4*/      WHEN "Chapter 7" THEN
             ASSIGN Chapter = "Thoroughly covers the use, syntax, and " +
                         "programming techniques of each data widget.".
           WHEN "Chapter 8" THEN
             ASSIGN Chapter = "Covers the Progress data handling " +
                              "statements.".
           WHEN "Chapter 9" THEN
             ASSIGN Chapter = "Covers record selection, sorting, and " +
                              "relating.".
           WHEN "Chapter 10" THEN
             ASSIGN Chapter = "Summarizes what you already know about " +
                              "frames and introduces some new options.".
           WHEN "Chapter 11" THEN
             ASSIGN Chapter = "Shows you how to define and use a " +
                              "menu bar, menus, and menu items.".
           WHEN "Chapter 12" THEN
             ASSIGN Chapter = "Covers the 4GL features used to create " +
                              "reports.".
           END CASE.
          DISPLAY Chapter WITH FRAME Frame1.
       END.

      /**********  MAIN LOGIC  *********/
      Tut-List:SCREEN-VALUE = "Chapter 7".
      ENABLE ALL WITH FRAME Frame1.
      APPLY "VALUE-CHANGED" TO Tut-List IN FRAME Frame1.
      WAIT-FOR CHOOSE OF btn-Exit.
```

These notes help explain the code:

1. This DEFINE VARIABLE sets up the editor widget.
2. This DEFINE VARIABLE statement sets up the selection list using the VIEW-AS phrase.
3. This syntax makes the screen value of the selection list the conditional expression of the CASE statement.
4. For each value of the selection list, the branches of the CASE statement assign an appropriate value to the editor widget.

Practice Problems

These problems give you some practice with conditional processing.

Problem 6-4: `lt-06-s4.p`

Using the IF statement, identify the number of customers that have BBB for their sales rep, the number of customers that have the sales rep SLS, and the number of customers that have any other sales rep. Use the Customer.Sales-Rep field.

Problem 6-5: `lt-06-s5.p`

Do the same thing as in Problem 6-4, except use the CASE statement.

6.7 Summary

This chapter discussed those features of the Progress 4GL that allow you to control the flow of execution within your application. Here's a summary of what you've learned.

Procedures and Include Files

In the Progress programming model the main procedure can access modules stored in separate subprocedures. A subprocedure can be either an internal or external procedure:

- The main procedure accesses an internal procedure with the RUN statement.
- An external procedure is a procedure file that another procedure calls.
- An internal procedure is a procedure file that is stored with the main procedure in one procedure file.
- A user-defined function is like an internal procedure that has some additional capabilities. It can return a value and the function can be called from within a Progress expression. You use the FUNCTION statement to create a user-defined function.
- An include file is a text file that contains Progress source code. When you reference include files in your procedures, Progress replaces the reference with the contents of the include file at compile time.

Sharing Information between Procedures

There are three basic strategies for passing data between procedures:

- **Using shared variables** — Shared variables are specially defined variables that more than one procedure can use.
- **Using parameters** — Parameters are variables in a called procedure or function whose responsibility is to accept an input value from the calling procedure, output a value to the calling procedure, or both. Parameters allow you to pass values to accomplish a run-time task for a procedure or a function.
- **Using arguments to pass literals** — Literals are programming symbols, such as table names, field names, or variable names. You can replace the literals with argument references in your procedure to make your code more generic. Arguments let you pass literals that allows code to compile.

Blocks

Blocks are structures that let you group segments of code. Typically a block consists of a block header statement, Progress 4GL statements, and an END statement. Progress blocks include:

- **Procedure blocks** — Encompass the entire procedure file (a .p file), but only those statements that don't belong to another block actually belong to the procedure block.
- **Internal procedure blocks** — Begin with the PROCEDURE header statement and are accessed with the RUN statement.
- **Trigger blocks** — Allow the user to control an application by directing processing with events.
- **Control blocks** — Have specific implicit properties that affect: loping, frame allocation, and record reading.

The control blocks include:

- **DO** — Groups statements together for execution as a single unit. It has no implicit block properties.
- **REPEAT** — Provides implicit looping and frame allocation. It has no explicit terminating condition, but Progress allows a user to end a REPEAT block with the **END-ERROR** key, or you can end it programmatically with the LEAVE statement.
- **FOR EACH** — Provides implicit looping, frame allocation, and record reading. Each iteration of the block reads a new record from a database table.

Conditional Processing Statements

There are two statements that allow you to test conditions and define which code branch to execute under which circumstances:

- The IF statement lets you test for a condition and then execute code if the expression is TRUE. The ELSE option lets you specify a second block of code to execute if the expression is FALSE.
- The CASE statement lets you test the result of an expression against several values.

7

Representing Data with Widgets

Using the right data widget with a particular variable or database field makes all the difference in the world to your end users. The nature of many widgets suggests the kind of data they contain and the ways that users can interact with them. For example, when users see a slider widget, they know that they are working with numeric data. On the other hand, some widgets, like the fill-in field, can work with several data types, so a user relies on other cues in your interface to determine the expected input.

First, this chapter introduces you to some common widget issues. Then the chapter discusses the syntax, events, attributes, and programming techniques used with:

- Fill-in fields
- Text widgets
- Toggle boxes
- Radio sets
- Sliders
- Selection lists
- Combo boxes
- Editors

7.1 Programming Widgets

Before you work with each individual widget, it's important to understand the programming concepts and structures that support your ability to define and modify the characteristics of a widget. Table 7–1 organizes these concepts and structures around the life span of a widget.

Table 7–1: Ways to Define or Modify a Data Widget

Stage	Description
Widget definition	For variables, the DEFINE VARIABLE statement supports the VIEW-AS phrase. This phrase enables you to specify a widget type and define its major characteristics. For fields, the Data Dictionary supports properties that match all the options of the DEFINE VARIABLE statement.
Container definition	When you define your frames and dialog boxes, you specify the widgets found in the containers. After each widget reference, there are several 4GL options you can specify to further define the characteristics of the widget. Collectively, these options are referred to as the *format phrase*. The DEFINE FRAME statement supports the frame phrase, which you use to define the characteristics of the frame widget. Some frame phrase options modify the characteristics of the widgets contained in the frame.
Before display	Progress screen input and output statements, like DISPLAY, support both the format phrase and the frame phrase. You can override earlier format and frame phrase options by respecifying them on a screen I/O statement. Note that some control blocks also support the frame phrase.
After display	After a widget appears on screen, you can no longer modify many characteristics with 4GL statements. Instead, you must access the corresponding widget attributes directly. The *Progress Language Reference* contains an appendix that details which attributes you can set in this way.

7.1.1 Using the VIEW-AS Phrase

The VIEW-AS is a programming structure for associating a data widget with a variable or database field. The syntax below shows the VIEW-AS phrase at the simplest level.

SYNTAX

```
VIEW-AS widget-phrase
```

The widget phrase consist of a keyword that identifies the widget type and the options that go with that particular type. This chapter covers each widget phrase in detail.

You can specify a VIEW-AS phrase at several points as noted in the following list:

1. Data Dictionary
2. DEFINE VARIABLE statement
3. Format phrase in a DEFINE FRAME statement
4. Format phrase on a screen I/O statement (DISPLAY, ENABLE, UPDATE, SET, and PROMPT-FOR)

A programmatic VIEW-AS phrase overrides a Data Dictionary VIEW-AS phrase. A VIEW-AS phrase that occurs later in the execution of a procedure overrides one that occurs earlier.

7.1.2 Using the Format Phrase

Throughout the tutorial, you've seen the appearance of widgets affected by programming options included in a DEFINE FRAME statement or on a screen I/O statement. These options are part of a Progress language structure called the *format phrase*. Not to be confused with the FORMAT option, the format phrase is a collection of options that you can specify after a field, variable, or expression reference in a DEFINE FRAME statement or screen I/O statement.

This simplified syntax for the DEFINE FRAME statement shows how the format phrase fits into the definition of a frame.

SYNTAX

```
DEFINE FRAME frame-name
  [ frame-item [ format-phrase ] ] ...
  format-phrase
```

A frame item can be a constant, a database field, or a variable. After each, you can specify one or more options from the format phrase to affect, for example, the label, format, or position of the frame item. The partial syntax below shows the most frequently used options of the format phrase.

SYNTAX

```
[ at-phrase | COLON n | TO n ]
[ LABEL string | NO-LABELS]
[ COLUMN-LABEL string ]
[ FORMAT string ]
[ HELP string ]
[ view-as-phrase ]
[ VALIDATE ( condition , msg-expression ) ]
```

These options all help you design attractive and intuitive displays. Table 7–2 describes the options.

Table 7–2: Format Phrase Options

Option	Use	Example
at-phrase	Specifies the row and column position for the top left corner of the widget.	`field1 AT ROW 2 COLUMN 40`
COLON *n*	Specifies the column position of the colon in the widget's label.	`field1 COLON 40`
TO *n*	Specifies the column position of the right edge of the widget.	`field1 TO 40`
LABEL *string*	Specifies a display label for the widget.	`field1 LABEL "Display Label"`
NO-LABELS	Suppresses all display labels for the widget.	`field1 NO-LABELS`
COLUMN-LABEL *string*	Specifies the display label to use when the frame is in column label mode.	`field1 COLUMN-LABEL "Top Label"`
FORMAT *string*	Specifies a format for the data of the widget.	`field1 FORMAT "x(12)"`
HELP *string*	Specifies a message to display in the message area of the window when the widget has input focus.	`field1 HELP "Enter an even number."`
view-as-phrase	Specifies the data widget for a field or variable.	`field1 VIEW-AS FILL-IN`
VALIDATE (*condition*, *msg-expression*)	Verifies that input matches the supplied condition. If so, Progress allows the input. Otherwise, Progress displays the message expression in an alert box.	`field1 VALIDATE(field1 > 0, "Must be greater than zero")`

You can specify one or many format phrase options after each field, variable, or constant at these points:

1. DEFINE FRAME statement

2. Screen I/O statement

Also, you can specify format phrase options for an expression in a screen I/O statement, but not in a DEFINE FRAME statement. Note the following code example:

```
DISPLAY (A + B) LABEL "Sum of A and B" WITH FRAME Frame1.
```

Note that the LABEL, COLUMN-LABEL, FORMAT, and VIEW-AS phrase options are also available on the DEFINE VARIABLE statement.

7.1.3 Positioning Widgets

From Table 7–2, you saw that you have three options for positioning a widget in a frame: AT, COLON, and TO. You can think of these options as left, center, and right alignment options for widgets. Figure 7–1 contrasts the three positioning options.

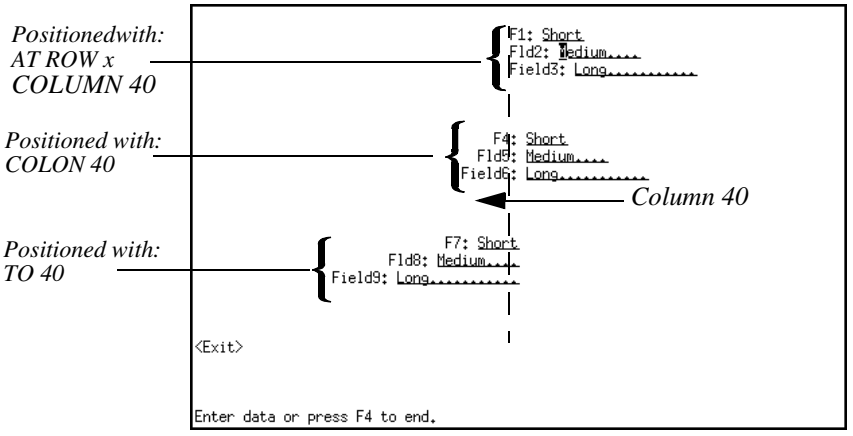


Figure 7–1: Format Phrase Positioning Options

As you might surmise from Figure 7–1, the COLON option is very useful for aligning columns of fields.

7.1.4 Using the Frame Phrase

The frame phrase is the major structure for defining the characteristics of a frame. However, some of the frame phrase options affect the contents of the frame. In this respect, the frame phrase is also a structure for defining the characteristics of widgets. In some cases, it may be more convenient to use a frame phrase option rather than specifying a similar option for each individual widget. The partial syntax below shows the most frequently used options of the frame phrase that affect widget characteristics.

SYNTAX

```
WITH
  [ SIDE-LABELS ]
  [ NO-LABELS ]
  [ n COLUMNS ]
  [ USE-TEXT ]
```

Table 7–3 describes these options.

Table 7–3: Frame Phrase Options

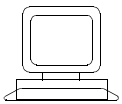
Option	Use	Example
SIDE-LABELS	Puts the frame in side label mode. The widget labels then appear on the left sides of the widgets. The default mode, column label mode, positions all labels above the widgets.	`WITH FRAME Frame1 SIDE-LABELS`
n COLUMNS	Puts the frame in side label mode and uses the COLON option to create (*n*) number of neat columns.	`WITH FRAME Frame1 2 COLUMNS`
NO-LABELS	Suppresses all widget labels in the frame.	`WITH FRAME Frame1 NO-LABELS`
USE-TEXT	Makes all fill-in fields in the frame text widgets. Use this option to display read-only data.	`WITH FRAME Frame1 USE-TEXT`

You can specify a frame phrase at these points:

1. DEFINE FRAME statement
2. Screen I/O statement
3. Control blocks (FOR, DO, REPEAT)

A frame phrase that occurs later in the execution of a procedure overrides one that occurs earlier.

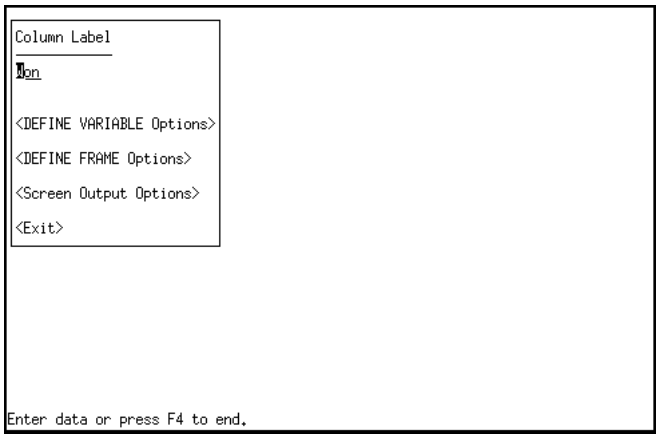
7.1.5 Changing Widget Characteristics Programming Example



Exercise

This example changes the characteristics of a widget at different points in a procedure:

1 ♦ Open `lt-07-01.p` and run it. The following display appears:



The characteristics of the fill-in field widget in this example are either default characteristics or characteristics defined with the DEFINE VARIABLE statement.

2 ♦ Choose the second button. The new display shows the same variable with a new set of characteristics established by the format phrase and frame phrase in a DEFINE FRAME statement.

3 ♦ Choose the third button. This display shows the variable with another set of characteristics. A format phrase and frame phrase on a screen I/O statement (ENABLE) defined these characteristics.

4 ♦ Choose Exit and press **SPACEBAR** to return to the Procedure Editor.

Here is the code for this example:

lt-07-01.p

```
      /**********  DEFINE WIDGETS  **********/
/*1*/  DEFINE VARIABLE Var1 AS CHARACTER LABEL "Column Label"
          INITIAL "Wonderful" FORMAT "x(3)" VIEW-AS FILL-IN.
      DEFINE BUTTON btn-one LABEL "DEFINE VARIABLE Options".
      DEFINE BUTTON btn-two LABEL "DEFINE FRAME Options".
      DEFINE BUTTON btn-three LABEL "Screen Output Options".
      DEFINE BUTTON btn-exit LABEL "Exit" TOOLTIP "This is a ToolTip".
      /**********  DEFINE FRAMES  **********/
/*2*/  DEFINE FRAME Frame1 Var1 THREE-D.
/*3*/  DEFINE FRAME Frame2
          Var1 AT ROW 3 COLUMN 10 NO-LABELS FORMAT "x(6)"
              VIEW-AS TEXT SKIP(2)
          btn-one SKIP btn-two SKIP btn-three SKIP
          btn-exit WITH SIDE-LABELS THREE-D.
      /********** MAIN LOGIC  **********/
      FRAME Frame1:HIDDEN = TRUE.
      FRAME Frame2:HIDDEN = TRUE.
      ENABLE Var1 SKIP(2) btn-one SKIP btn-two SKIP btn-three SKIP
          btn-exit WITH FRAME Frame1.
      DISPLAY Var1 WITH FRAME Frame1.
      ENABLE ALL WITH FRAME Frame2.
      DISPLAY Var1 WITH FRAME Frame2.
/*4*/  ENABLE Var1 AT ROW 2 COLUMN 2 LABEL "Side Label" FORMAT "x(9)"
          VIEW-AS EDITOR SIZE 12 BY 3 SKIP(2) btn-one SKIP btn-two SKIP
          btn-three SKIP btn-exit WITH FRAME Frame3 SIDE-LABELS THREE-D.
      FRAME Frame3:HIDDEN = TRUE.
      DISPLAY Var1 WITH FRAME Frame3.
      VIEW FRAME Frame1.
/*5*/  /********** DEFINE TRIGGERS  **********/
      ON CHOOSE OF btn-one IN FRAME Frame1, btn-one IN FRAME Frame2,
          btn-one IN FRAME Frame3 DO:
              HIDE FRAME Frame2 FRAME Frame3.
              VIEW FRAME Frame1. END.
      ON CHOOSE OF btn-two IN FRAME Frame1, btn-two IN FRAME Frame2,
          btn-two IN FRAME Frame3 DO:
              HIDE FRAME Frame1 FRAME Frame3.
              VIEW FRAME Frame2. END.
      ON CHOOSE OF btn-three IN FRAME Frame1, btn-three IN FRAME Frame2,
          btn-three IN FRAME Frame3 DO:
              HIDE FRAME Frame1 FRAME Frame2.
              VIEW FRAME Frame3. END.
      /**********  WAIT-FOR  **********/
      WAIT-FOR CHOOSE OF btn-Exit IN FRAME Frame1,
          btn-Exit IN FRAME Frame2, btn-Exit IN FRAME Frame3.
```

NOTE: The THREE-D option is relevant only on a Windows client; it is ignored by a character client.

These notes explain the code highlights:

1. The DEFINE VARIABLE statement establishes the default characteristics and the optional characteristics of the widget associated with the variable.

2. This DEFINE FRAME statement changes no characteristics of the variable.

3. This DEFINE FRAME statement alters the characteristics of the widget with both the format and frame phrases.

4. This run-time screen I/O statement defines a new frame and alters the characteristics of the widget with both the format and frame phrases.

5. Because Progress creates one of the frames used in this procedure at run time, the trigger definitions that reference that frame must follow the run-time frame definition.

NOTE: The TOOLTIP attribute noted in the code example is ignored when this code is run on a character client.

7.1.6 Referencing Widgets

A field or variable name is both a valid reference to the value associated with the field or variable and to the widget associated with the field or variable. For example, in the first statement below, referencing Field1 and Variable1 in an expression manipulates their values. In the second and third statements, referencing the names in a screen I/O statement manipulates the widgets associated with them. Review these statements in the following code:

```
New-Total = Field1 + Variable1.
ENABLE Field1 Variable1 WITH FRAME Frame1.
HIDE Field1 IN FRAME Frame1 Variable1 IN FRAME Frame1
```

This example also demonstrates the difference between using the frame phrase or IN FRAME to make an unambiguous reference to a widget. A single screen I/O statement can only work with the widgets in one frame. Thus the reference to the frame in the frame phrase creates an unambiguous reference to all widgets in the frame. On the other hand, you can append the IN FRAME syntax to any widget reference in any other statement to make a single widget reference unambiguous.

Referencing Widget Attributes

You also use the IN FRAME syntax to make unambiguous widget attribute references. Recall that the basic syntax for accessing a widget attribute is as follows.

SYNTAX

```
widget-name:attribute-name
```

You append the IN FRAME syntax to the widget attribute reference to make the reference unambiguous, as follows:

SYNTAX

```
widget-name:attribute-name IN FRAME frame-name
```

For example, the assignment statement below assigns an attribute value to a variable:

```
Myvar = Mywidget:ROW IN FRAME Myframe
```

Referencing Widget Methods

A method is a function associated with a particular type of widget that you access to manipulate an individual widget. You access methods in the same way that you access attributes.

SYNTAX

```
widget-name:method-name( ) IN FRAME frame-name
```

Like a function, a method always returns a value. For methods, that value is usually LOGICAL and indicates whether or not Progress successfully applied the method to the widget. Methods also may use input and output parameters like a function.

Later in the chapter you'll see programming examples that use methods.

Using Widget Handles

Using the field or variable name as a reference to both value and widget is convenient and makes readable code. At times, you may need to specifically reference the widget and not its associated field or variable. In these cases, you need to provide the widget handle. A *widget handle* is a unique internal identifier for a widget. Every widget has an attribute named HANDLE that contains the widget handle for that widget.

To access the handle, use the syntax presented in this example:

```
Widgetname:HANDLE
```

You may also want to save the handle of a widget in a variable. Since Progress supports widget handles as a separate data type, you need to specify the WIDGET-HANDLE data type in your DEFINE VARIABLE statements as this shows:

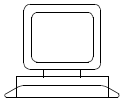
```
DEFINE VARIABLE Myhandle AS WIDGET-HANDLE.
```

Progress maintains several global variables that contain widget handles. These variables are part of a group of language elements known as *system handles*. System handles allow you to access information about the system. Table 7–4 describes the system handles that you can use to access information about widgets.

Table 7–4: Widget System Handles

System Handle	Description
DEFAULT-WINDOW	Contains the widget handle of the window that all Progress applications create by default. In single-window applications, the default window is the only window. In multiple-window applications, you can choose to make another window the default window.
CURRENT-WINDOW	Contains the widget handle of the currently active window.
FOCUS	Contains the handle of the widget that currently has input focus.
SELF	Contains the handle of the widget for which a trigger is currently executing.

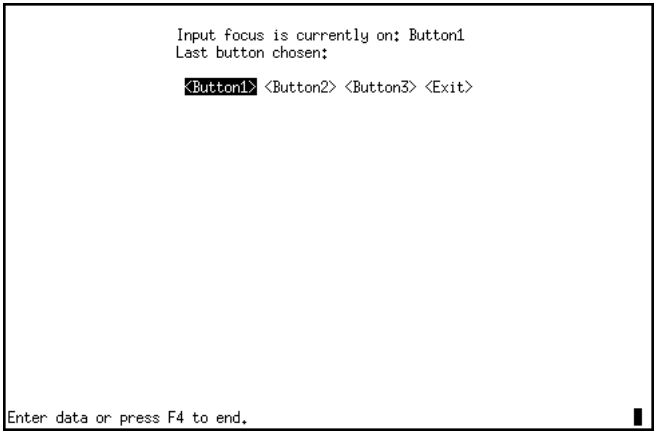
Widget Referencing Programming Example



Exercise

The following exercise demonstrates widget referencing and system handles:

1 ♦ Open `lt-07-02.p` and run it. The following display appears:



2 ♦ Press **TAB** or use the mouse to move input focus or choose a button. Notice that the variables at the top of the screen keep track of your actions.

3 ♦ Choose Exit and press **SPACEBAR** to return to the Procedure Editor.

Here is the code for this procedure:

lt-07-02.p

```
      /**********  DEFINE WIDGETS  **********/
      DEFINE VARIABLE Where-focus AS CHARACTER
          LABEL "Input focus is currently on" VIEW-AS TEXT.
      DEFINE VARIABLE Which-chosen AS CHARACTER
          LABEL "Last button chosen" VIEW-AS TEXT.
      DEFINE BUTTON Button1.
      DEFINE BUTTON Button2.
      DEFINE BUTTON Button3.
      DEFINE BUTTON Exit.

      /**********  DEFINE FRAMES  **********/
/*1*/  DEFINE FRAME Frame1
          SKIP(1)
          Where-focus SKIP
          Which-chosen SKIP
              WITH SIDE-LABELS NO-BOX CENTERED THREE-D.
/*2*/  DEFINE FRAME Frame2
          SKIP(1)
         Button1 Button2 Button3 Exit SKIP
              WITH NO-BOX CENTERED THREE-D.

      /**********  DEFINE TRIGGERS  **********/
/*3*/  ON ENTRY OF Button1 IN FRAME Frame2, Button2 IN FRAME Frame2,
                   Button3 IN FRAME Frame2, Exit IN FRAME Frame2
       DO:
/*4*/      APPLY "ENTRY" TO SELF.
/*5*/      ASSIGN Where-focus = FOCUS:LABEL.
/*6*/      DISPLAY Where-focus WITH FRAME Frame1.
       END.
/*7*/  ON CHOOSE OF Button1 IN FRAME Frame2, Button2 IN FRAME Frame2,
                  Button3 IN FRAME Frame2
       DO:
/*8*/      ASSIGN Which-chosen = SELF:LABEL.
           DISPLAY Which-chosen WITH FRAME Frame1.
       END.

      /**********  MAIN LOGIC  **********/
      VIEW FRAME Frame1.
      ENABLE ALL WITH FRAME Frame2.
      WAIT-FOR CHOOSE OF Exit IN FRAME Frame2.
```

The following notes describe the code highlights:

1. The first frame contains the text widgets that track the user's actions.

2. The second frame contains the buttons.

3. This trigger executes when any of the four buttons receive the ENTRY event. Notice the IN FRAME syntax for specifying the location of the widgets.

4. The ENTRY event occurs before Progress actually moves input focus. This APPLY statement forces Progress to fully move focus to the new widget. Notice the SELF system handle. No matter which button executes the trigger, Progress evaluates this statement using the correct button.

5. You can access an attribute of FOCUS or SELF exactly as you would with an explicit widget reference.

6. On this screen I/O statement, the frame phrase option specifies the correct frame.

7. Choosing any button except Exit executes this trigger.

8. Again, you access the system handle attribute exactly as you would a named widget.

7.1.7 Working with Widget Values

There is one very important case when you want to make a distinction between the field a widget represents and the widget itself. When you display a value and enable a widget for input, the value in the field and in the widget are the same. From that point forward, however, the user can change the value in the widget, which is also known as the *screen value*.

When users manipulate screen values, they are not manipulating the underlying field or variable values. Your code has to explicitly assign changes in the widget to the field or variable. There are two ways to do this.

1. Use the ASSIGN statement as shown in this code example:

```
ASSIGN Variable1 Field1.
```

   This form of the ASSIGN statement can be interpreted as “write the screen values of these widgets to their associated fields or variables.”

2. Access the SCREEN-VALUE attribute as shown in this code example:

```
Field1 = INTEGER(Variable1:SCREEN-VALUE) + 200.
```

Notice the use of the INTEGER data conversion function. No matter what data type the underlying field or variable may be, the screen value is always of type CHARACTER. Assuming that Variable1 is an integer variable, you need to use the appropriate data conversion function to make the expression compatible.

Checking for Changed Values

When you save screen values, it may be useful to check and see if the values have changed since you enabled the widgets. This check saves your application from doing unnecessary work. All data widgets have the MODIFIED attribute. MODIFIED is a LOGICAL attribute that contains a TRUE value if the value of the widget has changed since it was last enabled. Therefore, you can check to see if a value has changed before writing, as shown below:

```
IF Field1:MODIFIED THEN
    ASSIGN Field1.
```

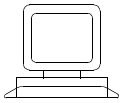
Validating Input

You saw earlier that the format phrase has a VALIDATE option. You can use this option to establish validation criteria for variables or to change default validation for database fields. Although you can use this option instead of the corresponding Data Dictionary properties, it is far more valuable to use the Data Dictionary as a central source for validation information.

To use the VALIDATE option, specify a condition and a message expression, as shown in this example:

```
DEFINE FRAME Frame1
    Var1 VALIDATE(Var1 > 0, "Entry must be greater than zero.")
        WITH SIDE-LABELS
```

Programming Example



Exercise

This programming example demonstrates the techniques discussed in this section:

1 ♦ Open `lt-07-03.p` and run it. The following display appears:



2 ♦ Type a new value for CharField that does not begin with the letter "A."

3 ♦ Try to move input focus with **TAB** or the mouse. An alert box appears telling you that the entry must begin with the letter "A."

4 ♦ Choose OK. You return to the main display with input focus on CharField. You cannot move from this field until you make a valid entry.

5 ♦ Type a new value for CharField that does begin with "A" and move input focus. This time you can.

6 ♦ Try the same experiment with IntField using a number less than 100.

7 ♦ Now that you have changed the screen values, choose Reset. The original values appear.

8 ♦ Type new valid entries in the fields and choose Save and then Reset. As you can see, your new values are now also the values stored in the variables.

9 ♦ Choose Exit and press **SPACEBAR** to return to the Procedure Editor.

Here is the code for this example:

lt-07-03.p

```
      /**********  DEFINE WIDGETS  **********/
      DEFINE VARIABLE CharField AS CHARACTER INITIAL "A string".
      DEFINE VARIABLE IntField AS INTEGER INITIAL 200.
      DEFINE BUTTON btn-Save LABEL "Save".
      DEFINE BUTTON btn-Reset LABEL "Reset".
      DEFINE BUTTON btn-Exit LABEL "Exit".

      /********** DEFINE FRAMES  **********/
      DEFINE FRAME Frame1
/*1*/      SKIP(2) CharField COLON 10 VALIDATE( CharField BEGINS "a",
             "Entry must begin with the letter A.") SKIP
/*2*/      IntField COLON 10 VALIDATE(IntField > 100,
              "Entry must be greater than 100.") SKIP(1)
        btn-Save btn-Reset btn-Exit
            WITH NO-BOX CENTERED SIDE-LABELS THREE-D.

      /**********  DEFINE TRIGGERS  **********/
/*3*/    ON CHOOSE OF btn-Save
       DO:
/*4*/      IF CharField:MODIFIED THEN
              ASSIGN CharField.
        IF IntField:MODIFIED THEN
              ASSIGN  IntField.
       END.
/*5*/  ON CHOOSE OF btn-Reset
       DO:
          DISPLAY CharField IntField WITH FRAME Frame1.
       END.

      /**********  MAIN LOGIC  **********/
      DISPLAY CharField IntField WITH FRAME Frame1.
      ENABLE ALL WITH FRAME Frame1.
      WAIT-FOR CHOOSE OF btn-Exit.
```

The following notes explain the code highlights:

1. The VALIDATE option of the format phrase establishes a condition that will allow only input that begins with the letter “A.”

2. Here the VALIDATE option establishes a condition that disallows input less than 100.

3. This trigger moves the screen values into the associated variables.

4. The IF statement checks the MODIFIED attribute of the widget to see if any changes were made, and thus whether it is necessary to write a new value.

5. This trigger copies the values of the variables to the screen with the DISPLAY statement.

7.2 Working with Fill-in Fields

The fill-in field is a labeled display and input area for a database field or variable. The fill-in field displays and accepts input as text strings. Figure 7–2 shows the parts of a fill-in field.

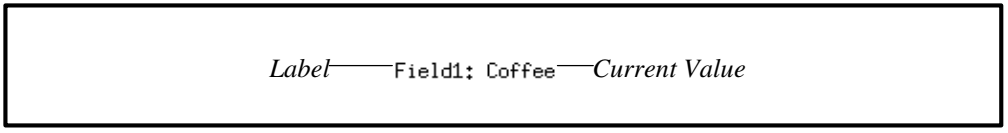


Figure 7–2: Parts of a Fill-in Field

The fill-in field is the all-purpose data representation. Think of the fill-in field as the data widget you use unless another data widget is better. Since the fill-in field is the default data widget, you don't need to write a VIEW-AS phrase when you want a fill-in field, unless you want to be explicit. This is the syntax for specifying a fill-in field.

SYNTAX

```
VIEW-AS FILL-IN [ size-phrase ]
```

The size phrase is a standard Progress way of describing how much room a widget should occupy. Normally, the size phrase uses this syntax.

SYNTAX

```
SIZE-CHARS width BY height
```

Where *width* and *height* are integer constants. For fill-in fields, the height is always 1.

There is an important relationship between the size-phrase and the format string associated with a fill-in field. As you learned in Chapter 4, "Understanding the Database Environment," a format string establishes the defaults for the number of characters a widget can display or accept as input. Progress uses this information to determine the default size of a fill-in field widget. You can override the default Progress format string with the FORMAT option on the DEFINE

VARIABLE statement or the format phrase. You can override the default widget size by using the size phrase of the VIEW-AS phrase.

Examine the code fragment below:

```
/*1*/  DEFINE VARIABLE Region-Code AS CHARACTER.
/*2*/  DEFINE VARIABLE First-Name AS CHARACTER FORMAT "x(20)"
/*3*/  DEFINE VARIABLE Last-Name AS CHARACTER FORMAT "x(40)"
          VIEW-AS FILL-IN SIZE 20 BY 1.
```

These notes explain the example highlights:

1. When Progress defines this variable, it uses the default format string of x(8) for CHARACTER variables, the default data widget (fill-in field), and creates a default size for the widget based on the format string. (The default size is the width of eight average-width characters.)

2. In this example, x(8) is not adequate, so the FORMAT option overrides the default and makes it x(20). Progress still determines the default size (20 average-width characters) based on the new FORMAT option.

3. This statement uses a format string of x(40). However, 40 characters take up too much room on the display, so the size phrase shortens the display area to 20 characters. Now the widget displays 20 characters, but can accept up to 40. When a user types beyond the 20th character, the data in the field scrolls to allow the extra input.

7.2.1 Fill-in Field Events

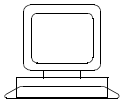
When an enabled fill-in field appears on screen, it can accept a whole range of events. The fill-in's default response to most events will be exactly what you intend, requiring no programming on your part. The fill-in field:

- Accepts input focus by way of the navigation key functions (like **TAB**) or the mouse
- Responds to the universal key functions (like **GO** or **HELP**)
- Accepts input from character keys, as governed by the format string

- Allows the user to move the text cursor within the field with the cursor keys or the mouse
- Allows the user to edit the field with the standard field editing keys of the native environment
- Responds to ENTRY and LEAVE triggers

ENTRY and LEAVE Event Functions

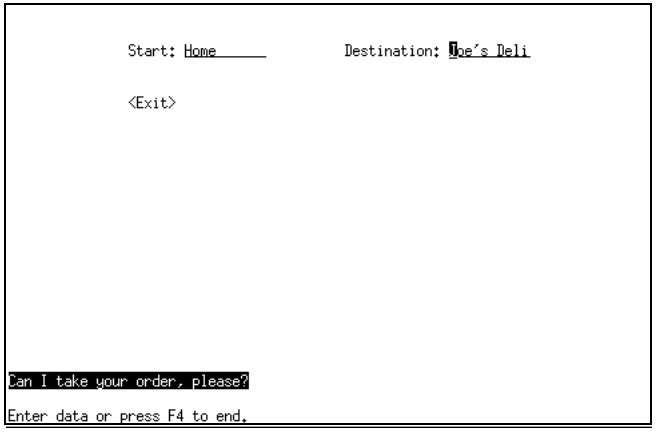
The ENTRY and LEAVE event functions are useful for attaching functionality to the widget. ENTRY executes when a widget receives input focus, while LEAVE executes when input focus moves from a widget. Use ENTRY when you need to do something before the user manipulates the widget. For example, you might want to issue a cautionary message to the user. Use LEAVE when you want to manipulate the widget after the user finishes with it. For example, you might want to update other widgets in the display based upon the new input.



Exercise

Follow these steps to use ENTRY and LEAVE:

1 ♦ Open `lt-07-04.p` and run it. The following display appears:



2 ♦ Press **TAB**. Notice the message that appears in the message area. A trigger attached to ENTRY displays this message.

3 ♦ Press **TAB** again. Another message appears: "Hey, where's the tip!" A trigger attached to LEAVE displays this message.

4 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

Here is the code that created this interface:

lt-07-04.p

```
      /**********  DEFINE WIDGETS  **********/
      DEFINE VARIABLE Field1 AS CHARACTER INITIAL "Home"
          FORMAT "x(10)" LABEL "Start".
      DEFINE VARIABLE Field2 AS CHARACTER INITIAL "Joe's Deli"
         FORMAT "x(10)" LABEL "Destination".
      DEFINE BUTTON btn-Exit LABEL "Exit".

      /**********  DEFINE FRAMES  **********/
      DEFINE FRAME Frame1
          SKIP(2) Field1 SPACE (10) Field2 SKIP(2)
          btn-Exit
             WITH NO-BOX CENTERED SIDE-LABELS THREE-D.

      /**********  DEFINE TRIGGERS  **********/
/*1*/  ON ENTRY OF Field2
       DO:
          MESSAGE "Can I take your order, please?".
       END.
/*2*/  ON LEAVE OF Field2
       DO:
          MESSAGE "Hey, where's the tip!".
       END.

      /***********  MAIN LOGIC  **********/
      DISPLAY Field1 Field2 WITH FRAME Frame1.
      ENABLE ALL WITH FRAME Frame1.
      WAIT-FOR CHOOSE OF btn-Exit.
```

As you can see at points 1 and 2, simple triggers on ENTRY and LEAVE are all that you need to extend the functionality of the widget.

NOTE: LEAVE does not execute on high-level event functions, such as GO or menu mnemonics.

7.3 Working with Text Widgets

The text widget is very similar to the fill-in field. You use a text widget when you want to display values as text strings, but do not want the user to change the data. In other words, text widgets are display-only fields.Figure 7–3 shows the parts of a text widget.

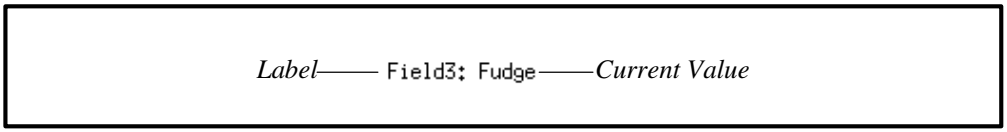


Figure 7–3: Parts of a Text Widget

The text widget has these advantages:

- Text widgets take up less screen space on many platforms and print more compactly, making them valuable for printed reports.
- You can easily switch between fill-in fields and text widgets. The new widget inherits many of the attributes of the old widget, such as format and label options.
- You can use text widgets to display static text in your interface display without creating a variable to hold the data.

This is the syntax for specifying a text widget.

SYNTAX

```
VIEW-AS TEXT
```

Converting Fill-in Fields with the USE-TEXT Option

When you design a frame for data entry, but then decide to use the same frame for display only, you have to place a VIEW-AS TEXT phrase on every fill-in field in the frame to make the fields static (display only). Progress includes the USE-TEXT frame phrase option to make this conversion easy. Specifying USE-TEXT in a frame phrase converts the fill-in fields in the frame to text widgets.

The following code example shows this technique:

```
DEFINE FRAME Frame1
    Field1 SKIP
    Field2 SKIP
    Button1 Button2
        WITH SIDE-LABELS THREE-D.
        .
        .
        .
DISPLAY Field1 Field2 WITH FRAME Frame1 USE-TEXT.
```

The USE-TEXT option on the screen I/O statement converts the fill-in fields in the frame to text widgets. The USE-TEXT option does not affect any other type of widget.

Static Text without Variables

To include text as part of a frame without first storing the text in a variable, simply place the text between quotes in a DEFINE FRAME or screen I/O statement. Progress creates a text widget to contain the string. You can then apply the options of the format phrase to the constant as this code example shows:

```
DEFINE FRAME Frame1
    Field1 SKIP
    "Constant Text String" TO 50
    Button1 Button2
        WITH SIDE-LABELS THREE-D.
        .
        .
        .
DISPLAY Field1 "More Text" AT ROW 9 COLUMN 30 WITH FRAME Frame1.
```

Practice Problems

You've covered quite a bit of new material so far in this chapter. It's time to cement your new knowledge in place by putting it to use.

Problem 7-1: `lt-07-s1.p`

Create a procedure that accepts input for the List Price, Discount, and Tax Rate with fill-in fields. Using a LEAVE trigger on the Tax Rate field, calculate the adjusted Price, Tax, and Total with text widgets.

7.4 Working with Toggle Boxes

A toggle box is a widget that graphically represents the state of a LOGICAL value. Progress displays a box in front of the widget label. When the value of the widget is TRUE, the box is filled. When the value of the toggle box is FALSE, the box is empty. Figure 7–4 shows the parts of a toggle box.

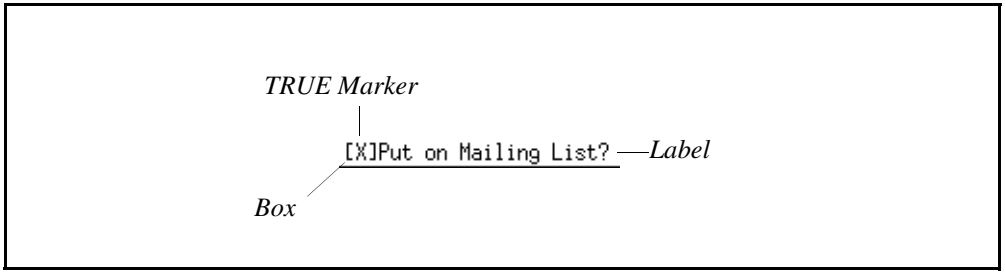


Figure 7–4: Parts of a Toggle Box

Instead of relying on textual value pairs like TRUE/FALSE or YES/NO, the toggle box graphically represents the value of a single logical field. A filled or checked box indicates a positive choice and is said to be activated. An empty box indicates a negative choice and is said to be deactivated.

This is the syntax for defining a toggle box.

SYNTAX

```
VIEW-AS TOGGLE-BOX
```

7.4.1 Toggle-box Events

An enabled toggle box accepts several events. The toggle box:

- Accepts input focus by way of the navigation key functions (such as **TAB**) or the mouse
- Responds to the universal key functions (such as **GO** or **HELP**)
- Accepts input from the **SPACEBAR** key or the mouse
- Responds to ENTRY, LEAVE, and VALUE-CHANGED triggers

VALUE-CHANGED Event Function

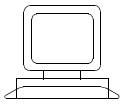
The toggle box responds to another important event function: VALUE-CHANGED. Each time the user changes the state of a toggle box, the VALUE-CHANGED event function executes.

7.4.2 Checked Attribute

The toggle box has an additional attribute that's useful: CHECKED. The CHECKED attribute holds a logical value. The value is TRUE when the on-screen state of the widget is solid or checked. It's FALSE when the on-screen state is empty. Although you can use either, using CHECKED instead of SCREEN-VALUE with toggle boxes creates more readable code. The following examples are equivalent, assuming that the solid state of the widget corresponds to YES:

```
IF Mywidget:SCREEN-VALUE = "YES" THEN . . .
IF Mywidget:CHECKED = YES THEN . . .
```

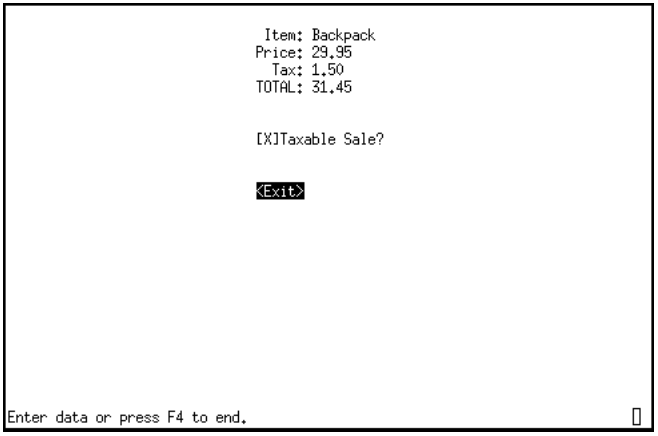
7.4.3 Toggle Box Programming Example



Exercise

Follow these steps to see how to use VALUE-CHANGED:

1 ♦ Open `lt-07-05.p` and run it. The following display appears:



2 ♦ Only the toggle box and Exit button are sensitive. Activate the toggle box. A VALUE-CHANGED trigger calculates the tax and new total.

Note that if you pause your mouse pointer over the toggle box a ToolTip is displayed.

3 ♦ Deactivate the toggle box. The VALUE-CHANGED trigger executes again and calculates the total without the tax.

4 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

Here is the code that created the display:

lt-07-05.p

```
      /**********  DEFINE WIDGETS  **********/
      DEFINE VARIABLE Item AS CHARACTER INITIAL "Backpack".
      DEFINE VARIABLE Price AS DECIMAL INITIAL 29.95.
      DEFINE VARIABLE Tax AS DECIMAL INITIAL 0.00.
      DEFINE VARIABLE TOTAL AS DECIMAL.
/*1*/  DEFINE VARIABLE Taxable AS LOGICAL LABEL "Taxable Sale?"
          VIEW-AS TOGGLE-BOX TOOLTIP "Checkmark indicates tax is
           included".
      DEFINE BUTTON btn-Exit Label "Exit".

      /**********  DEFINE FRAMES  **********/    DEFINE FRAME Frame1
          SKIP(1) Item COLON 7 SKIP
          Price COLON 7 SKIP
          Tax COLON 7 SKIP
          TOTAL COLON 7 SKIP(2)
          Taxable SKIP(2)
          btn-Exit
/*2*/           WITH NO-BOX CENTERED SIDE-LABELS USE-TEXT THREE-D.

      /**********  DEFINE TRIGGERS  **********/
/*3*/  ON VALUE-CHANGED OF Taxable
       DO:
          ASSIGN Taxable.
          IF Taxable = YES THEN
              ASSIGN Tax = Price * 0.05
                   TOTAL = Price + Tax.
          ELSE
              ASSIGN Tax = 0.00
                    TOTAL = Price + Tax.
          DISPLAY Tax Total WITH FRAME Frame1.
       END.

      /**********  MAIN LOGIC  **********/
      DISPLAY Item Price Tax TOTAL Taxable WITH FRAME Frame1.
      ENABLE ALL WITH FRAME Frame1.
      WAIT-FOR CHOOSE OF btn-Exit.
```

These notes explain the code highlights:

1. This statement creates the toggle box.
2. USE-TEXT converts the fill-in fields to text widgets without affecting other widgets.
3. Whenever you change the screen value of the widget, VALUE-CHANGED executes, forcing the appropriate calculation and update.

NOTE: The TOOLTIP attribute noted in the code example is ignored when this code is run on a character client.

7.5 Working with Radio Sets

A radio set is a set of radio buttons. Each button represents one possible value of a limited set of values for a single variable or field. You can select only one radio button in a radio set at a time. Choosing a button deselects the previously selected button. Figure 7–5 shows the parts of a radio set.

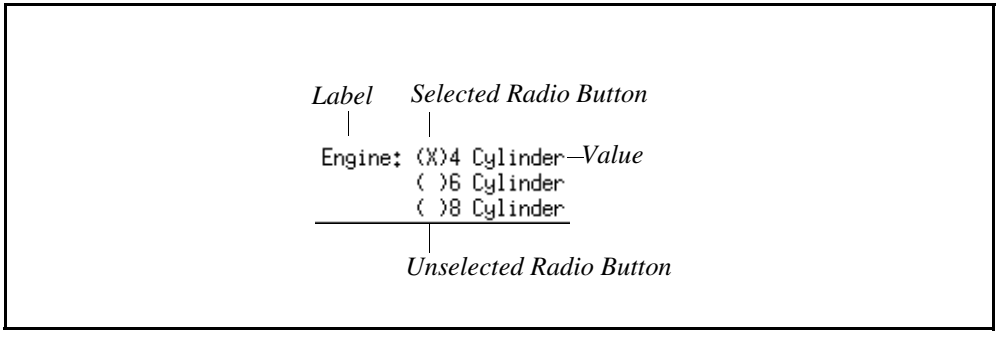


Figure 7–5: Parts of a Radio Set

You can use the radio-set widget with a CHARACTER, INTEGER, DECIMAL, LOGICAL, or DATE field or variable. This is the syntax for defining a radio set with the VIEW-AS statement.

SYNTAX

```
VIEW-AS RADIO-SET
  [ HORIZONTAL | VERTICAL ]
  [ size-phrase ]
  RADIO-BUTTONS label , value [ label, value ] ...
```

Table 7–5 explains the parts of the radio set syntax:

Table 7–5: Radio Set Syntax

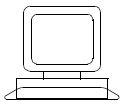
Component	Description
HORIZONTAL VERTICAL	Progress orients the radio buttons vertically by default. Use the HORIZONTAL keyword to specify horizontal orientation.
size-phrase	You probably won't use the size phrase for radio sets-Progress radio sets use the native look and spacing of your platform.
label, *value*	For each radio button, specify the screen label and the corresponding value. Enclose strings in quotes.

7.5.1 Radio Set Events

An enabled radio set accepts many events. The radio set:

- Accepts input focus by way of the navigation key functions (such as **TAB**) or the mouse
- Responds to the universal key functions (such as **GO** or **HELP**)
- Moves input focus among the radio buttons with the cursor keys
- Accepts input from **RETURN** and **SPACEBAR** keys or the mouse
- Responds to ENTRY, LEAVE, and VALUE-CHANGED triggers

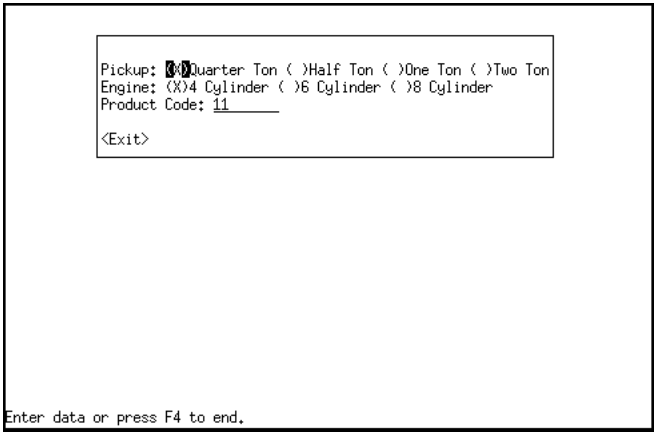
7.5.2 Radio Set Programming Example



Exercise

Follow these steps to demonstrate radio sets:

1 ♦ Open `lt-07-06.p` and run it. The display shown below appears:



2 ♦ Press **TAB** to move through the widgets.

3 ♦ Use the arrow keys to move among radio buttons.

4 ♦ Press **RETURN** or **SPACEBAR** to select a radio button.

5 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

Here is the code that created the display. This code also shows how to set up the radio sets:

lt-07-06.p

```
      /**********  DEFINE WIDGETS  **********/
/*1*/  DEFINE VARIABLE Pickup AS INTEGER INITIAL 1 VIEW-AS RADIO-SET
          HORIZONTAL RADIO-BUTTONS "Quarter Ton", 1, "Half Ton", 2,
          "One Ton", 3, "Two Ton", 4 TOOLTIP "Select one Pickup.". .
      DEFINE VARIABLE Engine AS INTEGER INITIAL 1 VIEW-AS RADIO-SET
          HORIZONTAL RADIO-BUTTONS "4 Cylinder", 1, "6 Cylinder", 2,
          "8 Cylinder", 3 TOOLTIP "Select one Engine.".
/*2*/  DEFINE VARIABLE P-code AS CHARACTER
          LABEL "Product Code" INITIAL "11".
      DEFINE BUTTON btn-Exit LABEL "Exit".

      /**********  DEFINE FRAMES  **********/
      DEFINE FRAME Frame1
          SKIP(1)
          Pickup SKIP
          Engine SKIP
          P-code SKIP(1)
          btn-Exit
              WITH SIDE-LABELS CENTERED ROW 2 THREE-D.

      /**********  DEFINE TRIGGERS  **********/
/*3*/  ON VALUE-CHANGED OF Pickup, Engine
       DO:
          ASSIGN Pickup
                 Engine
                 P-code = STRING(Pickup) + STRING(Engine).
         DISPLAY Product-code WITH FRAME Frame1.
       END.

      /**********  MAIN LOGIC  **********/
      DISPLAY Pickup Engine P-code WITH FRAME Frame1.
      ENABLE ALL WITH FRAME Frame1.
      WAIT-FOR CHOOSE OF btn-Exit.
```

These notes help explain the code:

1. For each label-value pair, Progress creates a radio button. In this case, the screen labels describe a product and the values represent an internal code. A radio set has vertical orientation by default. The keyword HORIZONTAL in the VIEW-AS phrase changes the orientation.

2. The INITIAL option sets the value of the P-code variable to "11".

3. Whenever you select a different radio button, the VALUE-CHANGED event executes and Progress concatenates and reassigns the values of the radio-set variables to P-code.

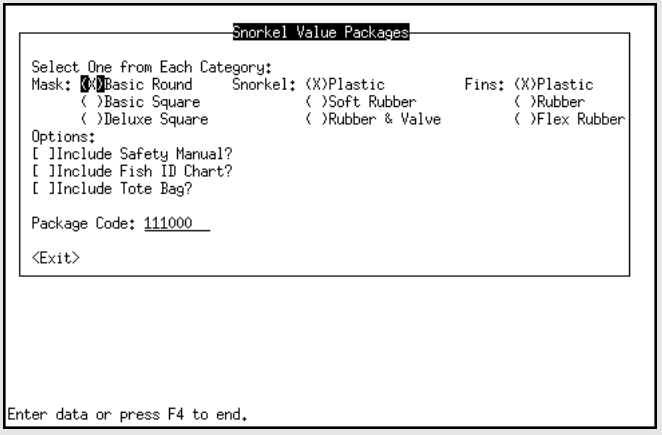
NOTE: The ToolTip attribute noted in the code is ignored when this code is run on a character client.

Practice Problems

You've now added two more widgets to your repertoire. Use the problem below to practice what you've learned about the widgets.

Problem 7-2: `lt-07-s2.p`

Duplicate this display using variables as the basis for the widgets. Use a VALUE-CHANGED trigger to update the Package Code field. The first three digits come from the index of the radio sets. The last three digits are "1" if the corresponding toggle box is TRUE and "0" if it is FALSE. (HINT: Use the online help system to look up the IF... THEN... ELSE function.)



7.6 Working with Sliders

A slider is a graphical representation of a numeric range composed of a rectangular area and a marker that shows the current value within the range.

Figure 7–6 shows the parts of a slider.

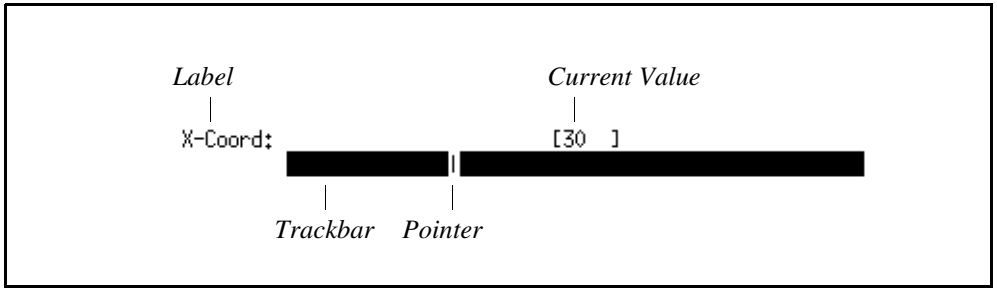


Figure 7–6: Parts of a Slider

The slider shows the current value and the range of acceptable values for an INTEGER field or variable. This is the VIEW-AS syntax for defining a slider widget.

SYNTAX

```
VIEW-AS SLIDER
  MAX-VALUE max-value MIN-VALUE min-value
  [ HORIZONTAL | VERTICAL ]
  [ NO-CURRENT-VALUE ]
  [ LARGE-TO-SMALL ]
  [ size-phrase ]
```

NOTE: A slider works best with a mouse. If your environment does not support the mouse, your users may prefer another data widget.

Table 7–6 explains the elements of the slider syntax.

Table 7–6: Slider Syntax

Element	Description
MAX-VALUE	Specify the MAX-VALUE keyword and an integer constant to set the upper limit of the slider's range.
MIN-VALUE	Specify the MIN-VALUE keyword and an integer constant to set the lower limit of the slider's range.
HORIZONTAL VERTICAL	Sliders have horizontal orientation by default. The keyword VERTICAL defines a slider with vertical orientation.
NO-CURRENT-VALUE	The default is to display the current value for a given position on the slider 's trackbar. The NO-CURRENT-VALUE option allows you to override this default behavior to indicate that the slider will not automatically display its current value.
LARGE-TO-SMALL	The default numeric range that a slider displays is small (minimum) to large (maximum). The LARGE-TO-SMALL option allows you to override this default behavior as follows: When the slider is positioned horizontally, the left most position on the trackbar displays the maximum value and the right most position displays the minimum value. When the slider is positioned vertically, the bottom most position on the trackbar displays the maximum value and the top most position displays the minimum value.
size-phrase	The size phrase has a few different syntaxes. The tutorial uses the most portable syntax: SIZE-CHARS *width* BY *height* where *width* and *height* are integer constants. If the slider has horizontal orientation, the height has to accommodate the trackbar, pointer, and labels. For character interfaces, the width should be a multiple or factor of the range to ensure that the pointer moves in even increments. If the slider has vertical orientation, the width has to accommodate the label. The height should be a multiple or factor of the range to ensure that the pointer moves in even increments.

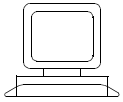
7.6.1 Slider Events

An enabled slider accepts many events. The slider:

- Accepts input focus by way of the navigation key functions (such as **TAB**) or the mouse
- Responds to the universal key functions (such as **GO** or **HELP**)
- Moves the slider pointer with either the cursor keys or when a user drags it with a mouse
- Responds to ENTRY, LEAVE, and VALUE-CHANGED triggers

Note that VALUE-CHANGED works differently in response to the cursor keys and mouse drags. Every keystroke is a separate event, so VALUE-CHANGED executes after each keystroke. This behavior effectively means that VALUE-CHANGED executes on every increment or decrement of the slider value. A mouse drag is a single event, so the user could move from one end of the slider to the other and VALUE-CHANGED would execute once, at the conclusion of the drag.

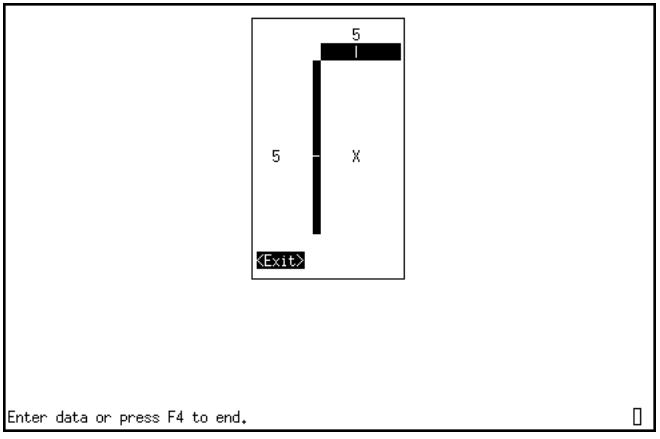
7.6.2 Slider Programming Example



Exercise

Follow these steps to contrast horizontal and vertical sliders.

1 ♦ Open `lt-07-07.p` and run it. The display shown below appears:



2 ♦ Press **TAB** to move to the second slider.

3 ♦ Use the cursor keys to move the pointer. At each increment, the value changes, a trigger executes, and the X character moves.

4 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

Here is the code that created the sliders:

lt-07-07.p

```
/**********  DEFINE WIDGETS  **********/
DEFINE VARIABLE X-Coord AS INTEGER INITIAL 0 VIEW-AS SLIDER
    MIN-VALUE 1 MAX-VALUE 10 HORIZONTAL SIZE-CHARS 10 BY 2.
DEFINE VARIABLE Y-Coord AS INTEGER INITIAL 0 VIEW-AS SLIDER
    MIN-VALUE 1 MAX-VALUE 10 VERTICAL SIZE-CHARS 8 BY 10.
DEFINE VARIABLE Point AS CHARACTER INITIAL "X"  FORMAT "x"
    VIEW-AS TEXT.
DEFINE BUTTON btn-Exit LABEL "Exit".
/**********  DEFINE FRAMES  **********/
DEFINE FRAME Frame1
    X-Coord NO-LABEL AT ROW 1 COLUMN 9 SKIP
    Y-Coord NO-LABEL
    btn-Exit AT ROW 14 COLUMN 1
    Point NO-LABEL AT ROW 12 COLUMN 9
        WITH CENTERED.
/**********  DEFINE TRIGGERS  **********/
ON VALUE-CHANGED OF X-Coord, Y-Coord
DO:
    ASSIGN X-Coord
           Y-Coord
           Point:COL = ((X-Coord:COL) + (X-Coord - 1) - 2)
                       /* (Starting Position) + (Offset) */
           Point:ROW = (Y-Coord:ROW + Y-Coord:HEIGHT - 1)
                         - (Y-Coord - 1).
                       /* (Starting Position) - (Offset) */
END.
/**********  MAIN LOGIC  **********/
DISPLAY X-Coord Y-Coord Point WITH FRAME Frame1.
ENABLE ALL WITH FRAME Frame1.
WAIT-FOR CHOOSE OF btn-Exit.
```

These notes help explain the code highlights:

1. The first slider is horizontal and displays the X coordinate of the character that moves within the area of the sliders.

2. The second slider is vertical and displays the Y coordinate of the character.

3. On VALUE-CHANGED of either slider, this trigger moves the character to the appropriate position.

4. The ROW and COL attributes specify the location of the widget within a frame. The HEIGHT attribute specifies the height of the widget. All widgets have these attributes.

7.7 Working with Selection Lists

A selection-list widget presents the user with a list of choices in a rectangle. The user can make single or multiple selections by manipulating a highlight bar. The selection list can contain more values than the rectangle can hold. In this case, Progress provides scrolling behaviors. Figure 7–7 shows the parts of a selection list.

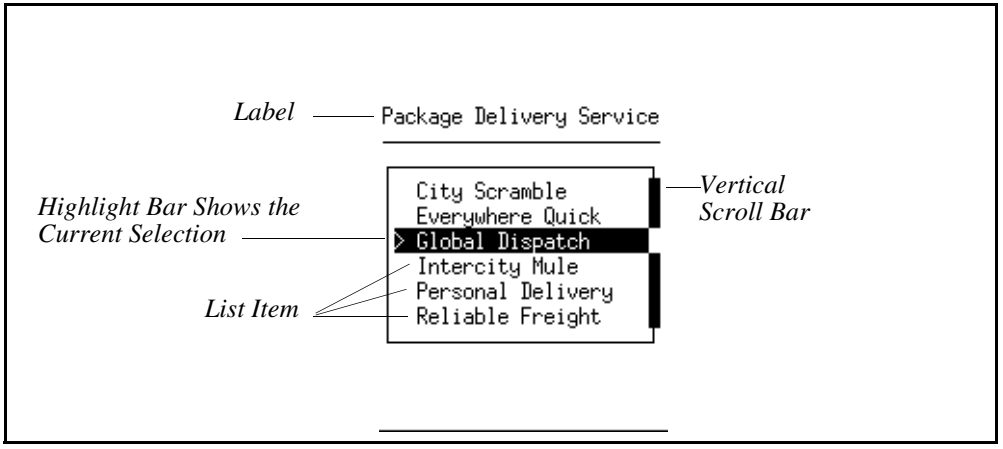


Figure 7–7: Parts of a Selection List

You can use the selection list widget with a CHARACTER field or variable. The choices in a list are always CHARACTER strings and the values returned to the procedure from the widget are CHARACTER strings. (You can effectively create a selection list of a different data type by using the data conversion functions on the returned CHARACTER string.)

Selection lists are valuable for restricting values to a predefined set. Selection lists are also valuable for creating indexes to data sets. For example, suppose you needed to search through several records. As you search, each record displays-that's a lot of work on the computer's part. Instead, you could create a selection list that contains a key field in a record, such as Name in the Customer table. You can then search more quickly through the selection list and use the result to display the whole record. This technique of creating a run-time selection list from a data set is called *populating* a selection list.

The selection list is an extremely flexible widget, as you can see by the options of the VIEW-AS syntax.

SYNTAX

```
VIEW-AS SELECTION-LIST
  [ SINGLE | MULTIPLE ]
  LIST-ITEMS item-list
  [ DELIMITER character ]
  [ SCROLLBAR-HORIZONTAL ]
  [ SCROLLBAR-VERTICAL ]
  { size-phrase | INNER-CHARS cols INNER-LINES rows }
  [ SORT ]
```

Table 7–7 explains some of the elements of the selection list syntax:

Table 7–7: Selection List Syntax *(1 of 2)*

Element	Description
SINGLE MULTIPLE	The SINGLE and MULTIPLE options allow you to specify whether the user can select a single value from the list or multiple values. Normally, selection lists have single selection and that is the default. For more information about MULTIPLE lists, see the *Progress Programming Handbook*.
item-list	After the keyword LIST-ITEMS, include a comma-separated list of strings. The strings are both the values that appear in the widget and the values the widget returns to your procedure.
DELIMITER	After the keyword DELIMITER, you can specify a character other than the comma to use as the separator in the item list. This option is necessary when your data could contain commas.
SCROLLBAR-HORIZONTAL SCROLLBAR-VERTICAL	If the selection list has more items than the confining rectangle can show, then the selection list automatically scrolls the values as the user accesses the top and bottom values in the selection rectangle. Despite this behavior, specifying a vertical scrollbar with the SCROLLBAR-VERTICAL option makes it explicit to your users that the selection list contains more items than they can see. Specify a horizontal scrollbar with the SCROLLBAR-HORIZONTAL option when the selection list items are longer than the confining rectangle. Scrollbars only appear if Progress cannot fit all the items in the display rectangle.

Table 7–7: Selection List Syntax *(2 of 2)*

Element	Description
size-phrase	Although available, you'll find the INNER-CHARS and INNER-LINES syntax more useful than the size phrase for setting up selection lists.
INNER-CHARS	After the keyword INNER-CHARS, specify the number of characters of the list items you want to appear in the selection list rectangle. Typically, this value should equal the length of the longest list item plus one. Progress can then determine the width of the widget.
INNER-LINES	After the keyword INNER-LINES, specify the number of list items you want to appear in the rectangle of the selection list. Progress will then determine the height of the widget.
SORT	This option sorts and displays the list items in alphabetical order. This option is especially valuable when populating selection lists at run time.

7.7.1 Selection-list Events

An enabled selection list accepts many events. The selection list:

- Accepts input focus by way of the navigation key functions (such as **TAB**) or the mouse
- Responds to the universal key functions (such as **GO** or **HELP**)
- Moves input focus among the list items with the cursor keys or the mouse
- Accepts input from **SPACEBAR** or **RETURN**.
- Allows contiguous mouse selections in MULTIPLE mode when the DRAG-ENABLED attribute is TRUE (the default).
- Allows discontiguous selections in MULTIPLE mode with the NO-DRAG option by selecting with **SPACEBAR**. The highlight does not indicate a selection with this setup. Instead, it indicates input focus.
- Responds to ENTRY, LEAVE, VALUE-CHANGED, and DEFAULT-ACTION triggers

DEFAULT-ACTION Event

A *default action* is an interaction with a widget that signals the widget to perform a task associated with the widget. For example, issuing a default action with a selection list might make the display update with new information about the selection. The default action for a selection list is to press **ENTER** or **RETURN** after selecting a list item or to double-click the desired list item. When a user issues the default action, the DEFAULT-ACTION event occurs, making this event the one to use for associating tasks with widgets that support a default action.

In MULTIPLE mode, the user selects all items and positions to the last item, but does not select it. When the user presses **ENTER** or **RETURN** on the last choice, VALUE-CHANGED occurs first, selecting the last item, and then DEFAULT ACTION occurs.

7.7.2 LIST-ITEMS, NUM-ITEMS, and DELIMITER Attributes

Three attributes are important for use with selection lists, as shown in Table 7–8.

Table 7–8: Selection List Attributes

Attribute	Description
LIST-ITEMS	This attribute contains all the list items in a selection list. The attribute is formatted as a comma-separated list of CHARACTER strings. The list does not require quotes around the list items.
NUM-ITEMS	This attribute is an INTEGER value that indicates how many list items are in the selection list.
DELIMITER	This attribute defines the character that separates the items in the LIST-ITEMS attribute. By default, it is a comma. However, since data frequently contains commas and would therefore corrupt the format of the LIST-ITEMS attribute, you can use the DELIMITER attribute to select a different delimiter.

7.7.3 ADD-LAST( ) and LOOKUP( ) Methods

The selection list supports a variety of methods to give you greater flexibility during runtime. Table 7–9 defines two important methods.

Table 7–9: Selection List Methods

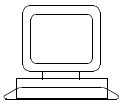
Attribute	Description
ADD-LAST(*list-item*)	Adds a new list item to the bottom of the selection list. This method returns TRUE if the method successfully added the item to the selection list and FALSE if it failed.
LOOKUP(*list-item*)	Returns the INTEGER index of the supplied list item in the selection list.

7.7.4 Selection-list Programming Example

Although the selection list is valuable for presenting a list of known choices to a user, it becomes essential as a way to display acceptable choices that aren't known until run time. The process of filling the list of choices at run time is known as populating. Populating involves two major steps:

1. Use the VIEW-AS phrase without the LIST-ITEMS option. This allows the Progress compiler to define the widget for use at runtime.

2. Define the list items of the widget before you display it.

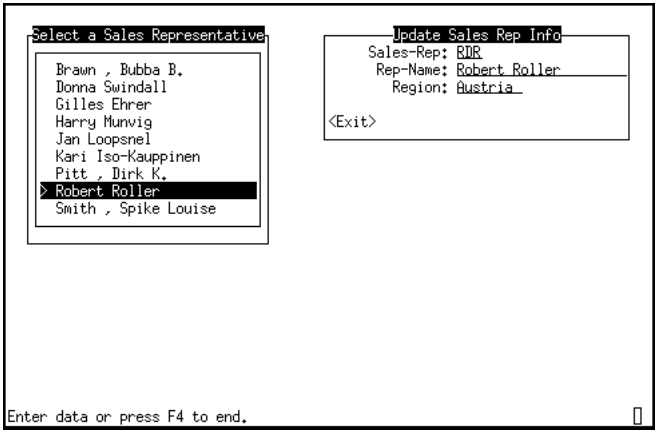


Exercise

All Around Sports is a growing business, and the responsibilities of its sales people are changing to meet the new challenges. The sales director asks you to design a simple interface so that she can quickly view and change sales representative assignments.

Follow these steps to view a sales representative selection list:

1 ♦ Open `lt-07-08.p` and run it. The display shown below appears:



2 ♦ Use the cursor keys to move the highlight bar. Since this is a single-selection list, highlighting is synonymous with selection. As you move through the sales representative names, Progress accesses the appropriate record from the Sports database and displays some of the fields in the second frame.

3 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

This is the code that created the interface:

lt-07-08.p

```
      /**********  DEFINE WIDGETS  **********/
/*1*/  DEFINE VARIABLE Reps AS CHARACTER VIEW-AS SELECTION-LIST
          INNER-CHARS 25 INNER-LINES 9 SORT.
      DEFINE VARIABLE Stat AS LOGICAL.
      DEFINE BUTTON btn-Exit LABEL "Exit".
      /**********  DEFINE FRAMES  **********/
      DEFINE FRAME Frame1
          Reps NO-LABEL      AT ROW 2 COLUMN 3
         Salesrep.Sales-Rep AT ROW 2 COLUMN 35
         Salesrep.Rep-Name  FORMAT "x(20)" AT ROW 3 COLUMN 36
          Salesrep.Region    AT ROW 4 COLUMN 38
          btn-Exit           AT ROW 9 COLUMN 3 SKIP(1)
             WITH SIDE-LABELS CENTERED ROW 2 TITLE "Update Sales Rep Info"
                 THREE-D.
      /**********  DEFINE TRIGGERS  **********/
/*2*/  ON DEFAULT-ACTION OF Reps
       DO:
           ASSIGN Reps.
          FIND FIRST Salesrep WHERE Salesrep.Rep-Name = Reps.
         DISPLAY Salesrep.Sales-Rep Salesrep.Rep-Name Salesrep.Region
            WITH FRAME Frame1.
       END.
      /**********  MAIN LOGIC  **********/
/*3*/  Reps:DELIMITER = "*".
/*4*/  FOR EACH Salesrep BY Salesrep.Rep-Name:
/*5*/      Stat = Reps:ADD-LAST(Salesrep.Rep-Name).
       END.
/*6*/  FIND FIRST Salesrep.
     DISPLAY Reps Salesrep.Sales-Rep Salesrep.Rep-Name Salesrep.Region
          WITH FRAME Frame1.
      ENABLE ALL WITH FRAME Frame1.
      WAIT-FOR CHOOSE of btn-Exit.
```

These notes help to explain the code:

1. Note that the initial definition of the selection list does not include the LIST-ITEMS.

2. By using DEFAULT-ACTION, the fields in the form do not update until the user double-clicks a list item or presses **RETURN** or **SPACEBAR**. The FIND statement, which you'll learn about in the next chapter, accesses the appropriate sales representative record.

3. Since the names in the Salesrep table may contain commas, you have to set the LIST-ITEM delimiter to some character that won't be in the data. You choose the asterisk.

4. The FOR EACH block cycles through each record in the Salesrep table.

5. The ADD-LAST( ) method adds the current Salesrep name to the bottom of the list. Since the SORT option is also employed, Progress will sort the list items before you display the widget.

6. This FIND statement accesses a record so that some data appears at startup.

7.8 Working with Combo Boxes

A combo box consists of a fill-in field or variable, a button, and a selection list. A combo box can display any single value from the selection list in the fill-in field. This single value can also be assigned to the underlying field or variable.

Progress provides the following types of combo-box widgets:

- A SIMPLE combo-box widget with a read/write edit control and a selection list that is always visible. This widget is supported in graphical interfaces only, and only in Windows. If you specify a SIMPLE combo-box widget in a character interface, Progress treats it as a DROP-DOWN-LIST combo-box widget.

- A DROP-DOWN combo-box widget with a read/write edit control and a selection list that appears when you click the drop-down button. This option is supported in graphical interfaces only, and only in Windows. If you specify a DROP-DOWN combo-box widget in a character interface, Progress treats it as a DROP-DOWN-LIST combo-box widget.

- A DROP-DOWN-LIST combo-box widget with a read-only edit control and a selection list that appears when you click the drop-down button. This is the default.

You can use the combo-box widget with a CHARACTER, INTEGER, DECIMAL, LOGICAL, or DATE field or variable. The value representations in the drop-down list conform to the data type of the underlying field or variable. Like radio sets and selection lists, combo boxes are useful for representing fields or variables that have a limited number of possible values. One advantage of combo boxes is that they take up less screen space than radio sets and selection lists.

To select a value in the character interface, the end user can position to the value in the drop-down list using the arrow keys and press SPACEBAR or RETURN to confirm the selection.

When the user selects an item, it triggers the VALUE-CHANGED event.

Thus, a combo box combines the functionality of a fill-in field, selection list, and radio set. It uses a fill-in field to display the selected item, and like a selection list displays a list of available values. Like a radio set, it supports sets of values for any Progress data type and allows selection of any one value at a time.

Figure 7–8 shows the parts of a combo box.

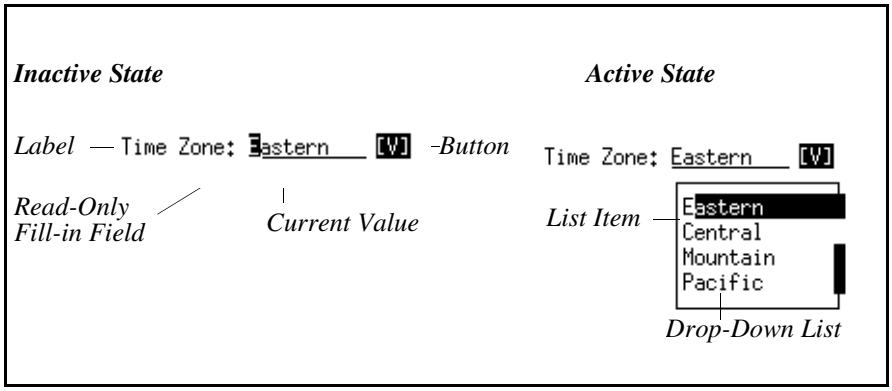


Figure 7–8: Parts of a Combo Box

This is the syntax for defining a combo box.

SYNTAX

```
VIEW-AS COMBO-BOX
  [ LIST-ITEMS item-list | LIST-ITEM-PAIRS item-pair-list ]
  [ INNER-LINES lines ] [ size-phrase ] [ SORT ]
```

Table 7–10 describes the combo box syntax elements.

Table 7–10: Combo Box Syntax

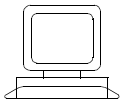
Element	Description
LIST-ITEMS	Specify a comma separated lists of values to populate the drop down list. Note that you can populate a combo box at run time in the same way you populate a selection list.
LIST-ITEM-PAIRS	Specify a list of label-value pairs. Each pair represents the label and value of a field or variable. When the user selects a label, Progress assigns the corresponding value to the field or variable.
INNER LINES	Specify the number of entries that the drop-down list can display. If you set INNER-LINES to less than the actual number of entries, the user can scroll through the drop-down list.
size-phrase	Specify the size of the widget in the interface. By default, Progress sizes the widget to correspond to the natural look of the native platform. Note that the height is always 1 on all platforms.
SORT	Sorts the LIST-ITEMS into alphabetical order.

7.8.1 Combo Box Events

An enabled combo box accepts many events. The combo box:

- Accepts input focus by way of the navigation key functions (such as **TAB**) or the mouse
- Responds to the universal key functions (such as **GO** or **HELP**)
- Moves input focus among the drop-down list items with the cursor keys
- Accepts input from **RETURN** and **SPACEBAR** keys or the mouse
- Responds to ENTRY, LEAVE, and VALUE-CHANGED triggers

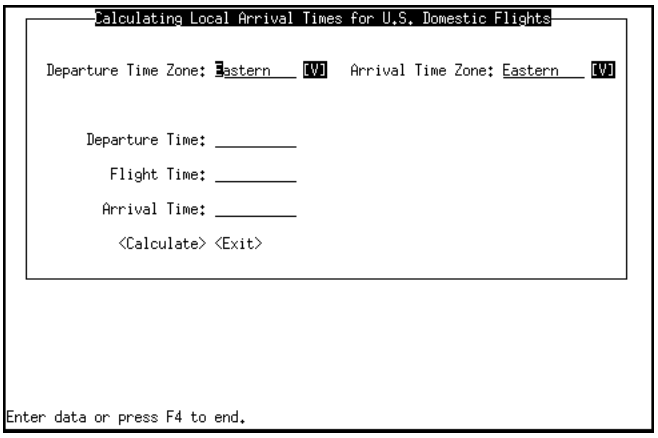
7.8.2 Combo Box Programming Example



Exercise

This example creates a simple combo box to represent a variable that lists U.S. time zones. Follow these steps to view the combo box:

1 ♦ Open `lt-07-09.p` and run it. The display shown below appears:



This application calculates the correct local arrival time of a U.S. domestic flight.

2 ♦ Choose the button for the Departure Time Zone combo box. The drop-down list appears. You select a value as you do with a selection list.

3 ♦ Select a value for the Arrival Time Zone combo box.

4 ♦ Enter a value for Departure Time and Flight Time.

5 ♦ Choose the Calculate button. The application displays the local arrival time.

6 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

This is the code that created the display:

lt-07-09.p

```
      /**********  DEFINE WIDGETS  **********/
/*1*/  DEFINE VARIABLE Zone1 AS CHARACTER LABEL "Departure Time Zone"
        FORMAT "x(10)" INITIAL "Eastern" VIEW-AS COMBO-BOX
      LIST-ITEMS "Eastern","Central","Mountain","Pacific".
/*2*/  DEFINE VARIABLE Zone2 LIKE Zone1 LABEL "Arrival Time Zone".
      DEFINE VARIABLE Dtime AS DECIMAL LABEL "Departure Time".
      DEFINE VARIABLE Ftime AS DECIMAL LABEL "Flight Time".
      DEFINE VARIABLE Atime AS DECIMAL LABEL "Arrival Time".
      DEFINE BUTTON btn-Calc LABEL "Calculate".
      DEFINE BUTTON btn-Exit LABEL "Exit".
      /**********  DEFINE FRAMES  **********/
      DEFINE FRAME Frame1
      SKIP(2) Zone1 COLON 22 SPACE(2) Zone2 SKIP(3)
      Dtime COLON 22 SKIP(1) Ftime COLON 22 SKIP(1) Atime COLON 22 SKIP(1)
      btn-Calc TO 22 btn-Exit SKIP(1)
        WITH SIDE-LABELS CENTERED THREE-D
          TITLE "Calculating Local Arrival Times for
             U.S. Domestic Flights".
      /**********  DEFINE TRIGGERS  **********/
      ON CHOOSE OF btn-Calc
       DO:
/*3*/      Atime = (DECIMAL(Dtime:SCREEN-VALUE) +
                  DECIMAL(Ftime:SCREEN-VALUE) +
                 (Zone1:LOOKUP(Zone1:SCREEN-VALUE) -
                 Zone2:LOOKUP(Zone2:SCREEN-VALUE))) MOD 24.
          DISPLAY Atime WITH FRAME Frame1.
       END.
      /**********  MAIN LOGIC  **********/
      DISPLAY Zone1 Zone2 WITH FRAME Frame1.
      ENABLE ALL WITH FRAME Frame1.
      WAIT-FOR CHOOSE OF btn-Exit.
```

These notes help to explain the code:

1. This DEFINE VARIABLE statement sets up the first combo box with the VIEW-AS phrase.

2. This DEFINE VARIABLE statement uses the LIKE option to inherit the characteristics of the first combo box. The other options override specific inherited characteristics.

3. This expression uses departure time and flight time to calculate arrival time. It also uses the LOOKUP method of the combo box to determine the index values of the selected time zones. The expression uses these values to adjust the arrival time to the arrival time zone. Finally, the MOD operator keeps the result in 24 hour format.

7.9 Working with Editors

An editor widget presents the user with a large rectangular area for manipulating text. Actually, the Procedure Editor is a Progress application based on the editor widget. Figure 7–9 shows the parts of an editor.

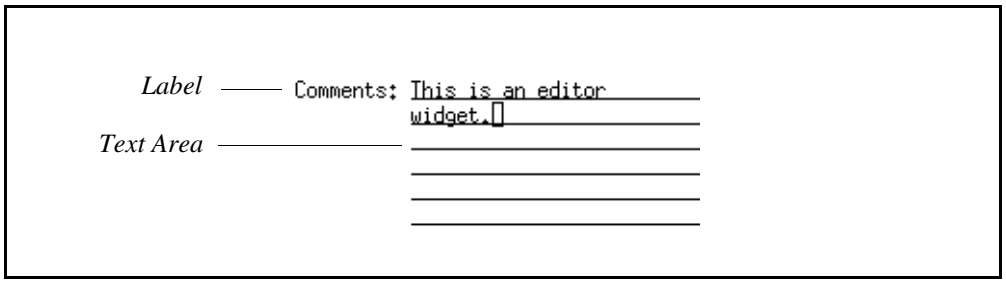


Figure 7–9: Parts of an Editor

The editor serves as a means to display and update lengthy CHARACTER data. However, you could also use the editor widget as the basis for a word processor. This section sticks to the basic functionality, which is essential for many applications. See the *Progress Programming Handbook* for information on the sophisticated applications of the editor widget.

This is the VIEW-AS syntax for creating a simple editor.

SYNTAX

```
VIEW-AS EDITOR
  { size-phrase | INNER-CHARS chars INNER-LINES lines }
  [ BUFFER-CHARS chars ]
  [ BUFFER-LINES lines ]
  [ MAX-CHARS chars ]
  [ SCROLLBAR-VERTICAL ]
  [ NO-WORD-WRAP [ SCROLLBAR-HORIZONTAL ] ]
```

Table 7–11 describes the editor syntax components.

Table 7–11: Editor Syntax *(1 of 2)*

Component	Description
size-phrase	You might want to use the size phrase with an editor to fit an editor into a particular area of a display. In this case, use this syntax: SIZE-CHARS WIDTH *cols* HEIGHT *rows* Where *cols* and *rows* are integer constants.
INNER-CHARS	Instead of the size phrase, you can use the INNER-CHARS option with an integer constant to specify the number of characters that the editor can display on one line.
INNER-LINES	Use INNER-LINES with an integer constant to specify the number of lines for the editor to display.
BUFFER-CHARS	Use the two buffer options when the text area of an editor widget is larger than the rectangular area set up by the previous options. In this case, the editor rectangle shows only part of the text area at one time. Use BUFFER-CHARS with an integer constant to specify the number of characters that one line of the editor can hold.
BUFFER-LINES	Use BUFFER-LINES with an integer constant to specify the number of lines that the editor can hold.
LARGE	Specifies that Progress uses a large editor widget rather than a normal editor widget in Windows. A normal Windows editor can contain up to 20K of data. The LARGE option enables the editor to contain data up to the limit of your system resources. However, it also consumes more internal resources and lacks some functionality. Use the LARGE option only if you have to edit very large sections of text. The LARGE option applies only to Windows; other interfaces allow for larger editors by default. This option is ignored in those other interfaces.
MAX-CHARS	Use this option to establish a maximum number of characters that the editor can hold.

Table 7–11: Editor Syntax *(2 of 2)*

Component	Description
SCROLLBAR-VERTICAL	When an editor can hold more lines than it can display, use this option to enable a vertical scrollbar.
NO-WORD-WRAP	The editor has default behavior to prevent words from splitting across line breaks. If a word cannot fit on the previous line, the editor moves it to the beginning of the new line as the user types. Specifying this option prevents word wrap. The editor instead scrolls horizontally until the user enters a line break.
SCROLLBAR-HORIZONTAL	When you use the NO-WORD-WRAP option, you can specify a horizontal scrollbar with SCROLLBAR-HORIZONTAL.

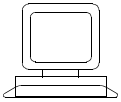
7.9.1 Editor Events

An enabled editor accepts many events. The editor:

- Accepts input focus by way of the navigation key functions (such as **TAB**) or the mouse
- Responds to the universal key functions (such as **GO** or **HELP**)
- Moves the text cursor with the arrow keys or the mouse
- Accepts input from any character key
- Creates a line break when the user presses **RETURN**
- Responds to ENTRY, LEAVE, and VALUE-CHANGED triggers

For information about the attributes and methods associated with more sophisticated editors, see the *Progress Programming Handbook*.

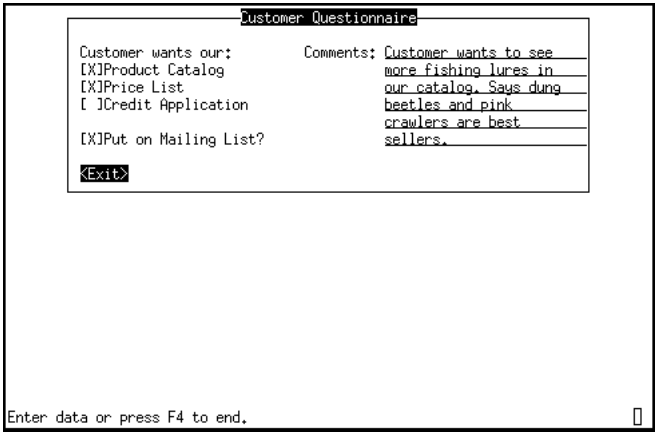
7.9.2 Editor Programming Example



Exercise

Follow these steps for a demonstration of an editor widget:

1 ♦ Open `lt-07-10.p` and run it. The display shown below appears:



2 ♦ Use **TAB** to move to the editor widget. The text cursor appears in the widget.

3 ♦ Type some comments. Notice the word-wrap behavior.

4 ♦ Press **RETURN** to enter a manual line break.

5 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

This is the code that created the display:

lt-07-10.p

```
      /**********  DEFINE WIDGETS  **********/
      DEFINE VARIABLE Catalog AS Logical LABEL "Product Catalog"
          VIEW-AS TOGGLE-BOX.
      DEFINE VARIABLE Price AS Logical LABEL "Price List"
          VIEW-AS TOGGLE-BOX.
      DEFINE VARIABLE Credit AS Logical LABEL "Credit Application"
          VIEW-AS TOGGLE-BOX.
      DEFINE VARIABLE Mail AS Logical LABEL "Put on Mailing List?"
          VIEW-AS TOGGLE-BOX.
/*1*/  DEFINE VARIABLE Comments AS CHARACTER LABEL "Comments"
          VIEW-AS EDITOR INNER-CHARS 25 INNER-LINES 6.
      DEFINE BUTTON btn-Exit LABEL "Exit".

      /**********  DEFINE FRAMES  **********/
      DEFINE FRAME Questions
          "Customer wants our:" AT ROW 2 COLUMN 2
           Catalog   AT ROW 3 COLUMN 2
           Price     AT ROW 4 COLUMN 2
           Credit    AT ROW 5 COLUMN 2
           Mail      AT ROW 7 COLUMN 2
           btn-Exit  AT ROW 9 COLUMN 2
           Comments  AT ROW 2 COLUMN 30
              WITH SIDE-LABELS CENTERED TITLE "Customer Questionnaire"
                   THREE-D.

      /**********  MAIN LOGIC  **********/
      DISPLAY Catalog Price Credit Mail Comments WITH FRAME Questions.
      ENABLE ALL WITH FRAME Questions.
      WAIT-FOR CHOOSE OF btn-Exit.
```

Simply by defining the size with the INNER-CHARS and INNER-LINES options at point 1, Progress creates an editor with the text-handling behavior the user expects.

Practice Problems

The problem below gives you some practice with sliders, selection lists, combo boxes, and editors.

Problem 7-3: `lt-07-s3.p`

It seems that All Around Sports needs to add another display to its on-line questionnaire application. It needs widgets to record customer responses to these questions.

A. What percentage of your business is in sports equipment?

B. What percentage of your business is in sports supplies?

C. What percentage of your business is in sports apparel?

D. Of the following categories, which best describes your business?

> Department store, general sports retailer, specialty sports retailer, secondary distributor, mail order distributor, sports club

E. What is your best sales season of the year?

F. Do you have any suggestions for All Around Sports?

Create the display All Around Sports needs, using these directions.

1. Use sliders to implement questions A, B, and C.
2. Use a selection list to implement question D.
3. Use a combo box to implement question E.

Use an editor capable of holding 25 lines of 60 characters to implement question F. However, make the editor small enough to fit in the same display as the other widgets.

7.10 Summary

This chapter covered the language elements used to define general widget characteristics as well as the specific elements used to define data widgets.

Defining Widget Characteristics

Progress provides different methods for determining the characteristics of widgets. When you can use the different methods depends of the stage of the widget:

- **Widget definition** — For variables, the DEFINE VARIABLE statement supports the VIEW-AS phrase and other options to define the major characteristics of a widget. For database fields, the Data Dictionary supports properties which correspond to all the options of the DEFINE VARIABLE statement.
- **Container definition** — When you define your frames and dialog boxes, you specify the widgets found in those containers and define the characteristics of the widget with the format phrase. The DEFINE FRAME statement also supports the frame phrase, which has options that modify the characteristics of the widgets contained in the frame.
- **Before display** — Progress screen I/O statements support both the format phrase and the frame phrase.
- **After display** — After a widget appears on screen, you access widget attributes directly to redefine widget characteristics.

Programming Widgets

Data widgets are representation of database fields or variables. The data widgets include:

- **Fill-in field** — The default widget representation for all fields and variables. It presents data as modifiable text.
- **Text widget** — Basically a fill-in field for read-only data. Text widgets are useful because they take up less room than fill-ins on some operating platforms and in reports.
- **Toggle box** — A graphical representation of a logical field or variable. You program toggle boxes so that a filled or checked toggle box represents a TRUE value and an empty toggle box represents a FALSE value.
- **Radio set** — A graphical representation of a field or variable that has a limited number of possible values. Only one value can be true at a time.

- **Slider** — A graphical representation of an INTEGER field or variable value and the acceptable range of values. By moving a slide along the range, the user manipulates the current value.
- **Selection list** — Presents the user with a scrolling list of possible values for a CHARACTER field or variable.
- **Combo box** — A combination of a read-only fill-in field, a button, and a drop-down selection list. Choosing the button displays the drop-down list of possible values, which the user manipulates like a selection list.
- **Editor** — A large rectangular area for displaying and editing long text.

8

Using Database Records

Up to this point, you've learned about interfaces and procedure without learning much about database access. The tutorial uses this approach because database access techniques make more sense in the context of fully functional procedures. In this chapter, you'll learn complete database access techniques that draw on what you already know about interfaces and procedures.

Specifically, this chapter describes:

- How Progress stores and moves data
- Designing displays for interactive database access
- Retrieving data
- Using queries
- Modifying data
- Releasing records
- Creating records
- Deleting records
- Related database access statements
- Using the browse widget

8.1 How Progress Stores and Moves Data

Before discussing the Progress language statements that manipulate data, it's helpful to know the main locations where Progress stores data. Once you understand where your data is, it's easier to visualize how Progress statements change or move that data.

To start working with data, you first have to copy it from the database. The database is the first data location.

When your procedure copies data from a database, Progress stores that data in memory. An area of memory used to store data is called a *buffer*. When you request data from a database record, Progress places a copy of the entire database record or a group of fields from the record (called a *field list*) in a buffer. These buffers are called *record buffers*.

For each table that you work with, Progress creates and maintains a separate record buffer. In general, you won't have to work directly with record buffers. Progress understands what to do with the buffers based on your code.

Before your end users can view or work with data, that data has to be visible on the screen. Data visible on screen is also stored in memory. Memory reserved for on-screen data is called a *screen buffer*. As with record buffers, Progress maintains screen buffers for you.

Figure 8–1 shows the icons that the tutorial uses to represent these data locations.

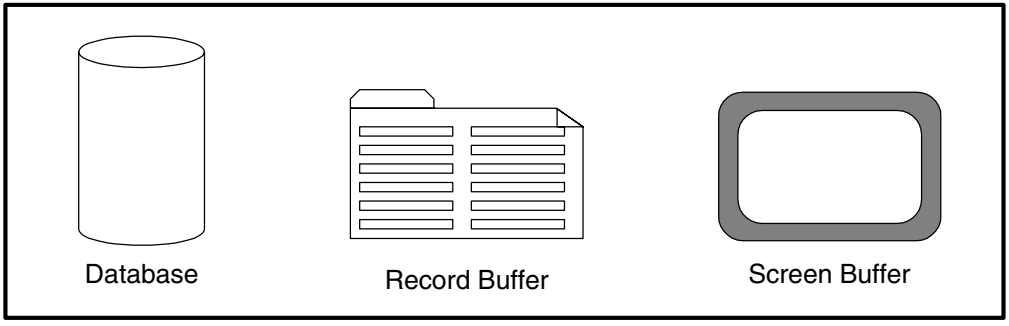


Figure 8–1: Data Locations

These data locations are important to understand because they relate to three fundamental rules of data movement in Progress. The following list describes these rules of data movement:

1. Your procedure cannot interact with data until the procedure copies the data from the **database** to a **record buffer**. Once in the record buffer, the procedure can manipulate that data.

2. End users cannot interact with data until your procedure copies the data from the **record buffer** to the **screen buffer**. Once in the screen buffer, the data is visible on screen.

3. Progress does not automatically copy changes made in a screen buffer or a record buffer to the related buffer. For example, if a copy of a particular field exists in both a record buffer and a screen buffer, changing the data in the screen buffer does not change the data in the record buffer. **Your procedure programmatically controls the movement of data between buffers**.

Think of the data locations as stops along a two-way street. To display a database record, Progress must first move it to a record buffer and then into a screen buffer. Similarly, Progress cannot write new screen buffer data to a database without first writing it to a record buffer. Figure 8–2 shows how data moves in Progress.

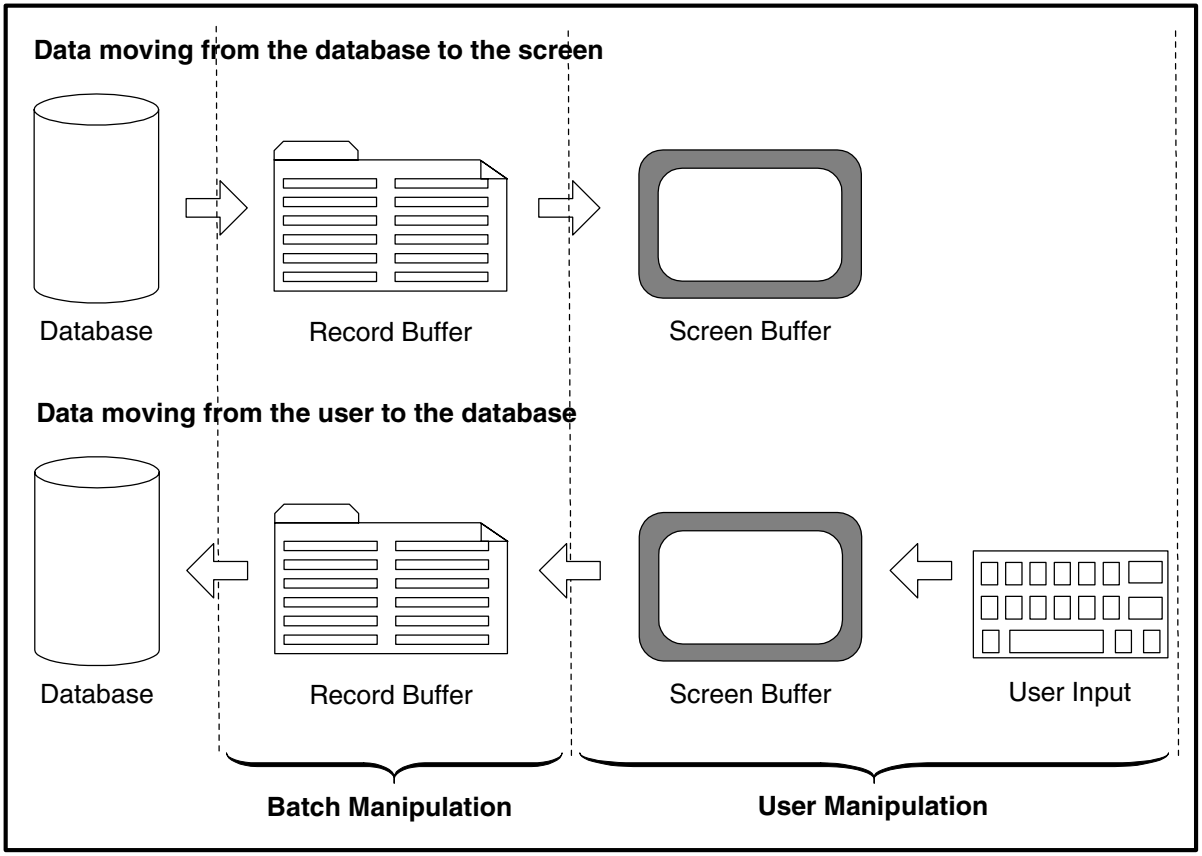


Figure 8–2: Data Movement in Progress

Up to this point, the tutorial used the terms “underlying value” to refer to a record buffer value and “screen value” to refer to a screen buffer value. (The screen value of a widget is a screen buffer value.)

Notice the distinction in Figure 8–2 between batch and user manipulation. If you write a procedure that processes records behind the scenes, then your procedure works exclusively with record buffers. If you write a procedure that allows users to interact with the data, the procedure also works with screen buffers. At some point, your interactive procedure copies data between the screen buffers and record buffers. This chapter focuses on the techniques of interactive database access.

8.2 Designing Displays for Interactive Database Access

Event-driven interfaces maximize the user’s choices so that control of an application resides with the user. In the context of database access, you also need to safeguard data from unintended changes. This chapter shows you how to implement an interface, also called a *form*, that balances user freedom with data protection. Figure 8–3 shows the database access form that this chapter implements.

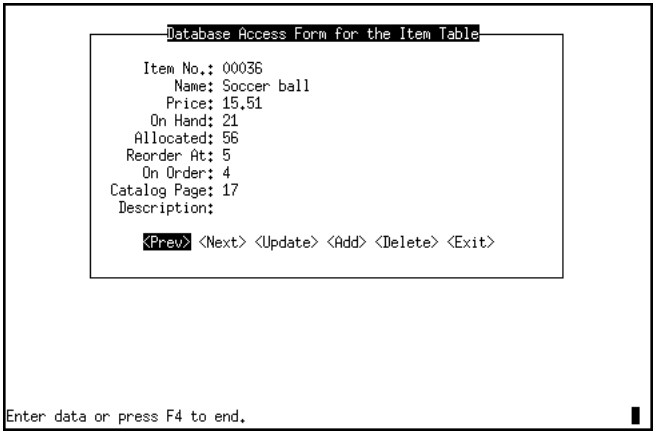


Figure 8–3: Basic Database Access Form

A basic database access form has several key features:

- Allows the user to navigate through a set of records
- Displays key data so users know that they are interacting with the correct record
- Separates update mode from display mode and lets users decide whether to save or cancel their changes on a record-by-record basis
- Lets users create new records and lets the user decide whether to keep or discard the new record
- Allows users to delete records, but asks for confirmation before deleting the record
- Handles all foreseeable user errors

In Figure 8–3, the form's buttons execute triggers that perform these user tasks: navigate and display, update, add, and delete. Figure 8–4 shows how the form and triggers work together.

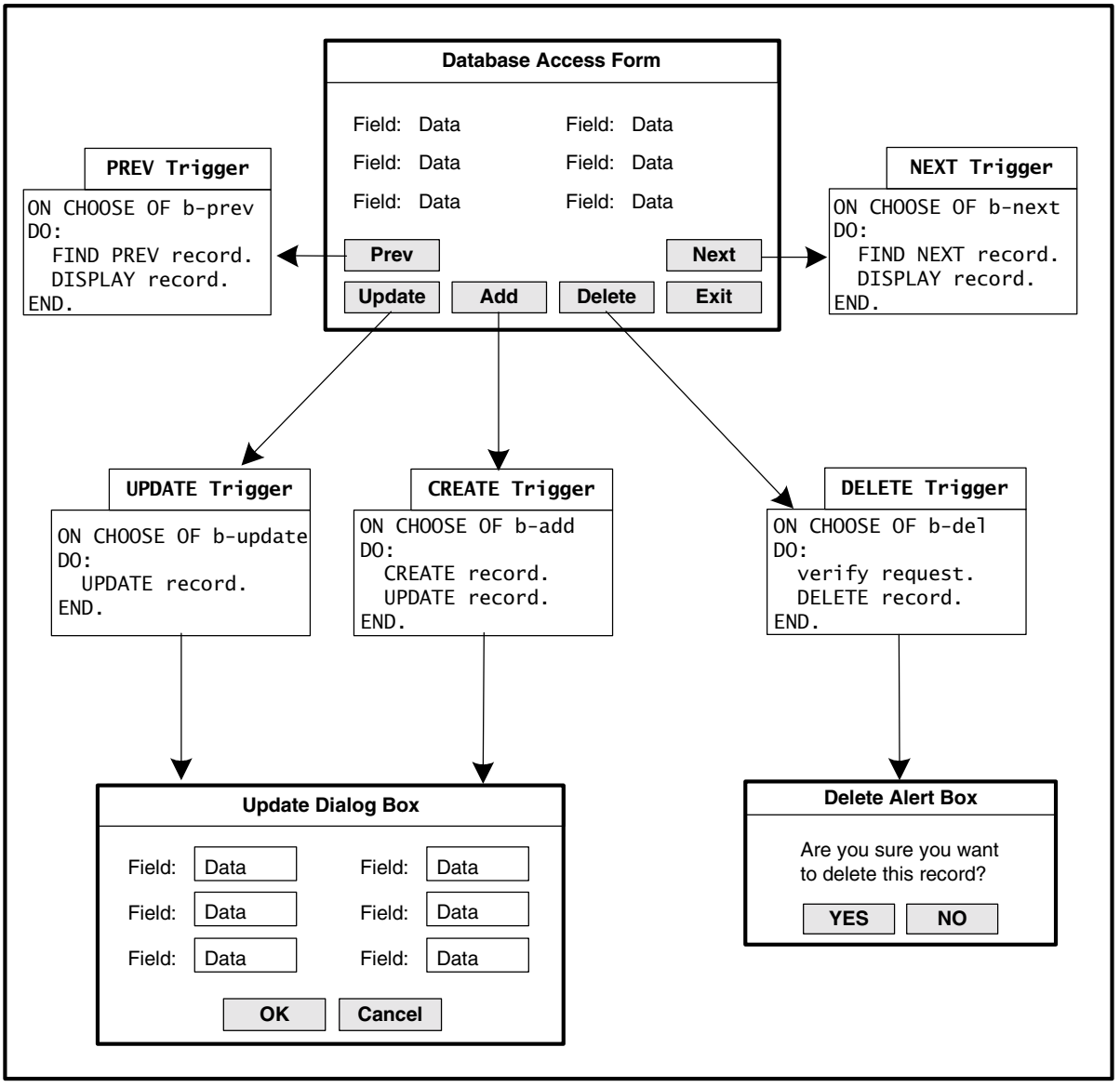


Figure 8–4: How a Database Access Form Works

This chapter shows you how to build this fully functional database access form. This simple model is one that you can use in most situations. The form is also flexible enough to accommodate many of the advanced techniques that you can learn about in the *Progress Programming Handbook*.

8.3 Retrieving Data from a Database

An enterprise's data is a vital asset. Progress has a system for protecting that data called the *relational database management system* (RDBMS). When you start a Progress application, the RDBMS uses the commands in the startup script to connect the necessary database or databases. All the objects (fields, tables, indexes, and so on) contained in the connected databases make up the set of objects that your application can reference. Your procedures interact with the RDBMS behind the scenes to access these database objects.

Earlier in the chapter, you learned that your procedures cannot interact with database data until you copy data into a record buffer. This section teaches you how to copy a single record or a group of fields to a record buffer with the database access form shown earlier. This is the basic syntax of a database request. Note that the combination of the *table-name* and *record-phrase-options* in the syntax expression comprise the record phrase.

SYNTAX

```
statement-name
  table-name [ record-phrase-options ] [ statement-options ]
```

Retrieving Field Lists

When you request data from a database record, you can request the entire record or a specific subset of fields from the record using a field list. A *field list* is a subset of the fields that define a database record. Field lists can be beneficial when you are running Progress on a network; if you only retrieve the fields you need, you can reduce network traffic. However, field lists are only useful if you are reading records without making any changes.

You can use field lists in your code with the FIELDS option in the following Progress 4GL statements:

- FOR statement
- DEFINE QUERY statement
- DO PRESELECT statement
- REPEAT PRESELECT statement

It is a good rule of thumb to use field lists only when you are referencing less than 90% of the record. When you are referencing more than 90%, it makes more sense to fetch the entire record. Also keep in mind these rules when you use field lists:

- Specify all the fields you plan to reference in your field lists.
- Specify the entire database record if you are updating or deleting the record. Otherwise, if you delete or update a record from a Progress database, Progress rereads the complete record before completing the transaction.

The `lt-08-08.p` procedure shows a revision of an example from Chapter 7, "Representing Data with Widgets," that uses the FIELDS option of the FOR statement to read only the fields referenced in this application. This is the revised code:

lt-08-08.p

```
      /**********  DEFINE WIDGETS  **********/
      DEFINE VARIABLE Reps AS CHARACTER VIEW-AS SELECTION-LIST
          INNER-CHARS 25 INNER-LINES 9 SORT.
      DEFINE VARIABLE Stat AS LOGICAL.
      DEFINE BUTTON btn-Exit LABEL "Exit".

      /**********  DEFINE FRAMES  **********/
      DEFINE FRAME Frame1
          Reps NO-LABEL      AT ROW 2 COLUMN 3
          Salesrep.Sales-Rep AT ROW 2 COLUMN 35
         Salesrep.Rep-Name  FORMAT "x(20)" AT ROW 3 COLUMN 35
         Salesrep.Region    AT ROW 4 COLUMN 38
          btn-Exit           AT ROW 9 COLUMN 3 SKIP(1)
        WITH SIDE-LABELS CENTERED ROW 2 TITLE "Update Sales Rep Info"
         THREE-D.

      /**********  DEFINE TRIGGERS  **********/
      ON DEFAULT-ACTION OF Reps
       DO:
          ASSIGN Reps.
          FIND FIRST Salesrep WHERE Salesrep.Rep-Name = Reps.
          DISPLAY Salesrep.Sales-Rep Salesrep.Rep-Name Salesrep.Region
              WITH FRAME Frame1.
       END.

      /**********  MAIN LOGIC  **********/
      Reps:DELIMITER = "*".
/*1*/  FOR EACH Salesrep FIELDS (Rep-Name) BY Salesrep.Rep-Name:
          Stat = Reps:ADD-LAST(Salesrep.Rep-Name).
       END.
      FIND FIRST Salesrep.
/*2*/  DISPLAY Reps Salesrep.Sales-Rep Salesrep.Rep-Name Salesrep.Region
          WITH FRAME Frame1.
      ENABLE ALL WITH FRAME Frame1.
      WAIT-FOR CHOOSE of btn-Exit.
```

NOTE: The THREE-D option is relevant only on a Windows client; it is ignored by a character client.

These notes help explain the code:

1. This FOR statement uses the FIELDS option to specify a field list instead of fetching the entire record.

2. The Sales-Rep and Region fields of the Salesrep table are also referenced in the application, but they do not appear in the field list because they are referenced outside of the FOR statement.

Referencing Database Records

The record phrase is the language structure for referencing the table you want to work with and for helping to specify an individual record or a set of records.

The reference to the table name is both a reference to the database table and the record buffer associated with the table. When two connected databases contain identically named tables, you also need to specify the database name in the record phrase to make the reference *unambiguous*. The diagram below shows the correct syntax for unambiguous record references. Note that the period separates the names.

SYNTAX

```
database-name.table-name
```

Progress flags all ambiguous references at compilation time. Many programmers always specify the full reference to avoid ambiguity and make more readable code.

Referencing Database Fields

After you retrieve a record, you'll want to work selectively with some of the fields in the record buffer. To avoid ambiguity at this level, use the syntax shown below.

SYNTAX

```
database-name.table-name.field-name
```

It is also important to remember that the expanded reference to a database field is also a reference to the widget that represents it. In most cases, Progress knows from the context of your code whether you are working with the record buffer (database value) or screen buffer (widget screen value).

The following code, which comes from the include file, sets up the database access form:

lt-08-f1.i

```
/* Form definition for Chapter 8 programming examples */
        .
        .
        .
DEFINE FRAME Frame1
    SKIP(1)
    sports.Item.Item-Num LABEL "Item No." COLON 15
    sports.Item.Item-Name LABEL "Name" COLON 15
    sports.Item.Price COLON 15
    sports.Item.On-Hand LABEL "On Hand" COLON 15
    sports.Item.Allocated  COLON 15
    sports.Item.Re-Order LABEL "Reorder At" COLON 15
    sports.Item.On-Order LABEL "On Order" COLON 15
    sports.Item.Cat-Page LABEL "Catalog Page" COLON 15
    sports.Item.Cat-Description LABEL "Description" FORMAT "x(40)"
        COLON 15  SPACE(2) SKIP(1)
    btn-Prev TO 12
    btn-Next
    btn-Update
    btn-Add
    btn-Delete
    btn-Exit SKIP(1)
        WITH SIDE-LABELS CENTERED ROW 2 THREE-D
            TITLE "Database Access Form for the Item Table".
```

The frame in this code fragment is the main display for the database access form. Notice how the database references here refer to the widgets associated with the database fields. You can think of this frame as a formatted representation of the screen buffer.

About Queries

A *query* is a request for database data that results in one or more records. A *results* or *results list* is the record or set of records defined by a query. Some Progress statements query the database directly. Other statements work with the results list of a predefined query to query the database. A results list points to all the records that satisfy a predefined query. For example, the FIND statement queries the database directly, while the GET statement queries through the results list of a predefined query.

In general, the Progress documentation uses “query” to refer to a predefined query created with the DEFINE QUERY and OPEN QUERY statements.

8.3.1 Using the FIND and DISPLAY Statements

The first step in implementing the database access form, shown in Figure 8–5, is to let users move through, or *navigate*, the records of a database table. In this example, when the procedure starts, it finds the first record in the Item table and displays data from that record. Choosing the Next button finds and displays the next Item record. The Prev button finds and displays the previous Item record. The Next and Prev buttons implement the navigation and display functions of the form.

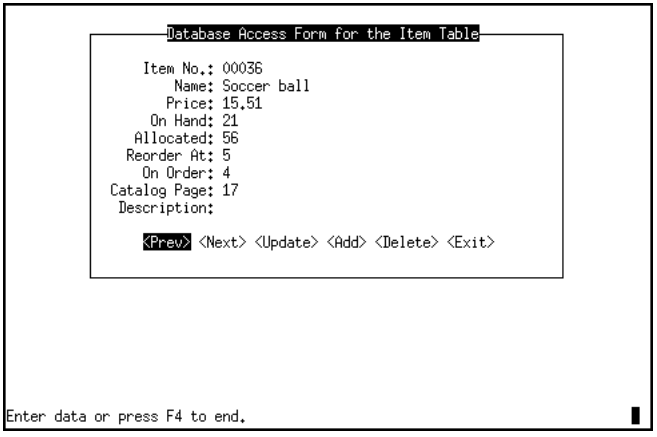


Figure 8–5: Data Form with Navigation Buttons Enabled

The FIND statement retrieves an individual record and is the basis for the triggers associated with the navigation buttons. FIND copies a record from a database to a record buffer, as shown in Figure 8–6.

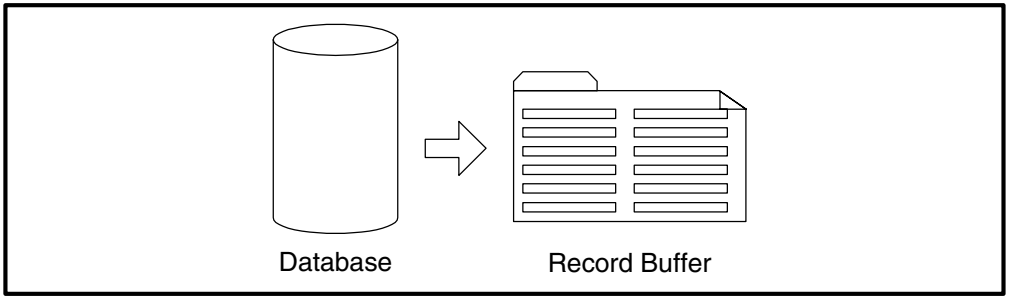


Figure 8–6: Data Movement with the FIND Statement

This is a partial syntax for the FIND statement.

SYNTAX

```
FIND
  { FIRST | NEXT | PREV | LAST | CURRENT } record-phrase
  NO-ERROR
```

When you think about it, there are four records that you search for most often in database applications. They are the first record, the record next after the one currently in the record buffer, the record previous to the one currently in the record buffer, and the last record. Progress supports four keywords for the FIND statement to specify these records: FIRST, NEXT, PREV, and LAST.

With just these four keywords, you have enough context to create an application that interactively searches one by one through the records of a database table. The record phrase only has to specify a table name to complete a valid statement, as shown below:

```
FIND FIRST Item.
```

The FIND statement uses the primary index of the database table to determine which record is first and last. In the next chapter, you'll learn how to navigate through the records in different orders.

The NO-ERROR option suppresses the normal error messages and default behavior that the RDBMS executes if the FIND attempt fails. Use NO-ERROR when you want to handle error conditions in your procedure. The programming example at the end of this section demonstrates error handling.

Once you have a record in the buffer, you can display that record. You've already seen the DISPLAY statement several times. The function of the DISPLAY statement is to move data from a record buffer into a screen buffer, and thus make the data visible to the user, as shown in Figure 8–7.

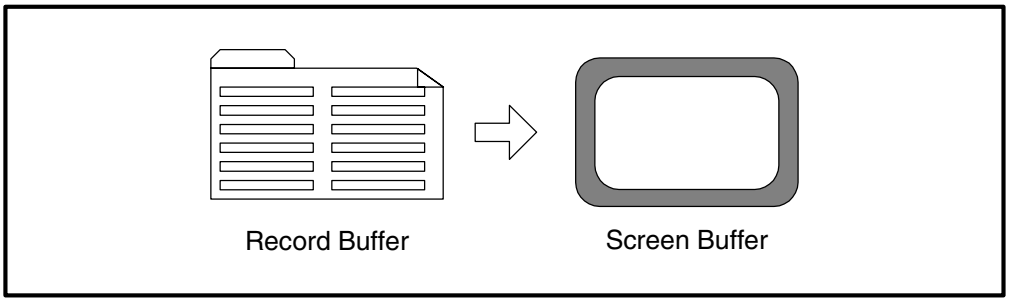


Figure 8–7: Data Movement with the DISPLAY Statement

As you learned in Chapter 3, "Programming the Progress Way," a widget and the data it represents are two different things. DISPLAY manipulates the data in a widget. But, if the widget that contains the data is not already visible when Progress executes DISPLAY, Progress makes the widget visible. After a widget is visible, you can use the DISPLAY statement repeatedly to update the data in the screen buffer, which refreshes the widget with the most current data. Remember, when data changes in the record buffer, Progress does not update the associated data in the screen buffer. If you want your user to see the current data, use another DISPLAY statement.

This is a partial syntax for the DISPLAY statement.

SYNTAX

```
DISPLAY
  [   expression [ format-phrase ]
    | SPACE [ ( n ) ]
    | SKIP [ ( n ) ]
  ] ...
  { [ frame-phrase ] }
```

This is an alternate syntax (partial) of the DISPLAY statement for displaying whole records.

SYNTAX

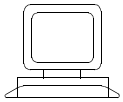
```
DISPLAY record
  [ EXCEPT field ... ]
  { [ frame-phrase ] }
```

Table 8–1 describes the DISPLAY statement syntax components:

Table 8–1: DISPLAY Statement Syntax

Component	Description
expression	*Expression* can be a database field, variable, constant, or any legal combination of these. For each expression in the DISPLAY statement, you can use the options provided by the format phrase.
SPACE (*n*)	Specify a number of spaces to place after an expression.
SKIP (*n*)	Specify a number of lines to skip after an expression.
frame-phrase	The frame phrase lets you specify a particular frame in which to display the data, as well as frame-formatting options.
EXCEPT	Specify fields from the record to not display with the record.

FIND and DISPLAY Programming Example



Exercise

Follow these steps for a demonstration of the FIND statement and the NO-ERROR option using the database access form for the Item table:

1 ♦ Open `lt-08-01.p` and run it. The Database Access Form appears.

2 ♦ Choose the Next button several times and notice the changes to the Item No. field. The procedure navigates through the records starting with the lowest item number and moving to the highest. FIND uses the Item-Num field for the navigation order because it is the primary index of the Item table.

3 ♦ Choose Prev until the form displays item number 1. Now choose Prev again. Normally, trying to find the previous record of the first record would be an error. This procedure, however, defines the LAST record as the PREV record of the FIRST record. The procedure also defines the NEXT record of the LAST record as the FIRST record.

4 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

Here is the code for the form:

lt-08-01.p

```
       /**********  DEFINE FORM  **********/
/*1*/  {lt-08-f1.i}

       /**********  DEFINE TRIGGERS  **********/
/*3*/   ON CHOOSE OF btn-Prev
        DO:
/*4*/      FIND PREV Item NO-ERROR.
/*5*/      IF NOT AVAILABLE(Item) THEN FIND LAST Item.
/*6*/      DISPLAY Item.Item-Num Item-Name Price On-Hand
               Allocated Re-Order
             On-Order Cat-Page Cat-Description WITH FRAME Frame1.
        END.
/*7*/   ON CHOOSE OF btn-Next
        DO:
/*8*/      FIND NEXT Item NO-ERROR.
/*9*/      IF NOT AVAILABLE(Item) THEN FIND FIRST Item.
           DISPLAY Item.Item-Num Item-Name Price On-Hand
                Allocated Re-Order
              On-Order Cat-Page Cat-Description WITH FRAME Frame1.
        END.
       /**********  MAIN LOGIC  **********/
/*2*/   FIND FIRST Item.
       DISPLAY Item.Item-Num Item-Name Price On-Hand Allocated Re-Order
          On-Order Cat-Page Cat-Description WITH FRAME Frame1 USE-TEXT.
       ENABLE btn-Next btn-Prev btn-Exit WITH FRAME Frame1.
        WAIT-FOR CHOOSE OF btn-Exit.
```

These notes help to explain the code:

1. This statement includes the file that contains the frame and button definitions that make up the interface.

2. The first FIND statement (in the Main Logic section) creates the Item buffer and copies the first record from the database to the record buffer. The NO-ERROR option is not necessary here. As long as the table contains one record, Progress will find a record that satisfies the request. If the RDBMS encounters any other errors, then you want the default messages and behaviors to execute.

3. This trigger executes when the user chooses the Prev button. It finds the previous record, which is the record that comes before the current record according to the primary index. If the current record is the first record (no previous record exists) it finds the last record.

4. This FIND statement clears the Item buffer, locates the previous record, and copies it to the Item buffer. The NO-ERROR option suppresses the normal error response. Normally, if Progress cannot find the previous record, the RDBMS sends a message to the user and the procedure stops execution of the current code block, which is the trigger. With the NO-ERROR option, the user gets no error message and the procedure continues executing the code block. However, because Progress clears the Item buffer during the FIND request, there is no current record. Any attempt to access the buffer later results in an error.

5. This IF statement checks the Item buffer with the AVAILABLE function. This test allows you to handle error conditions. If there is no record in the Item buffer, then the FIND statement failed. If it failed, then it probably failed because the user tried to find the previous record of the first record. In this situation, finding the last record puts data into the empty buffer.

6. The DISPLAY statement refreshes the form with the new data in the record buffer.

7. This trigger executes when the user chooses the Next button. It finds the next record, which is the record that comes after the current record according to the primary index. If the current record is the last record (no next record exists) it finds the first record.

8. This FIND statement copies the next record to the Item buffer.

9. This IF statement defines the normal behavior for trying to find the next record of the last record—find the first record.

8.4 Using Defined Queries

Progress has another statement that works similarly to FIND: the GET statement. To put it simply, FIND returns a record from the database and GET returns a record from the database by way of a defined query. So before you can use GET, you need to learn about defined queries.

A defined query lets you build a results list of records that satisfy some criteria. When you use FIND, Progress only knows about a single record that satisfies your criteria. When you use a defined query, Progress knows about all the records that satisfy the criteria. This results list is the basis for many Progress features.

In the context of an interactive database form, these steps show how to program a defined query:

1. Established the query with the DEFINE QUERY statement at the top of the procedure in the definitions section.

2. Initialize the results list with the OPEN QUERY statement at the beginning of your main code block.

3. Navigate through the results list with the GET statement, or by using a browse widget, but not with both.

4. Release the query resources with the CLOSE QUERY statement at the end of your main code block.

To start with, the DEFINE QUERY statement lets you name your new query and requires you to list all the tables that the query uses. This is a partial syntax for the DEFINE QUERY statement.

SYNTAX

```
DEFINE QUERY query-name FOR table-name
  [ , table-name ] ...
  [ SCROLLING ]
```

Use the SCROLLING option if you plan to use a browse widget or the REPOSITION statement with the query. SCROLLING is almost always necessary for an interactive database procedure.

Next, you need to initialize the results list with the OPEN QUERY statement. This is a partial syntax for the OPEN QUERY statement.

SYNTAX

```
OPEN QUERY query-name FOR EACH record-phrase
```

Each instance of the FIND statement uses a record phrase. With a defined query, the record phrase does its work up front in the OPEN QUERY statement. Therefore, the GET statement has no need to support the record phrase.

Now, you're ready to navigate through the results list. The next few sections of this chapter show you how to do this.

Finally, when you finish working with a query, you should close it. This is the syntax for the CLOSE QUERY statement.

SYNTAX

```
CLOSE QUERY query-name
```

8.5 Retrieving Data Through a Query with the GET Statement

The GET statement works similarly to the FIND statement, but the GET statement works with a defined query. With FIND, you provide enough information for your procedure to access the database and find the record that satisfies your criteria. With a defined query, the DEFINE QUERY and OPEN QUERY statements provide enough information for Progress to build a results list of all the records that match your criteria. The GET statement works with the results list to determine the correct record and then copies that record from the database to a record buffer, as shown in Figure 8–8.

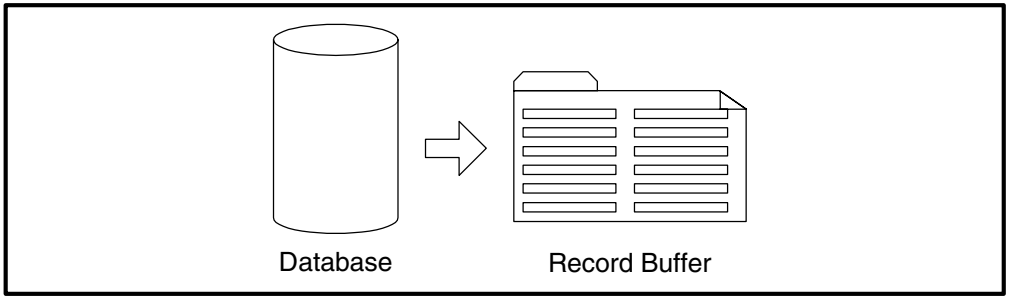


Figure 8–8: Data Movement with the GET Statement

As you can see, GET accomplishes the same thing as the FIND statement. While FIND goes directly to the database, GET works through the results list. GET manipulates the *cursor* of the results list. The cursor is an agreement between the query results list and the database about which record is the current one. When GET moves the cursor of the results list, Progress updates the associated record buffer. So, the position of the cursor always indicates which record is in the record buffer. This is the syntax for the GET statement.

SYNTAX

```
GET { FIRST | NEXT | PREV | LAST | CURRENT } query-name
```

As with FIND, the four keywords specify which record you are trying to get. Following the keyword, you provide the name of the query, not the actual table name:

```
GET FIRST My-Query.
```

Notice that the GET statement does not support the NO-ERROR option. In the last programming example, you saw that NO-ERROR is very useful for suppressing undesirable error behavior with the FIND statement. GET does not support the NO-ERROR option because a failed GET does not invoke an error response. For example, suppose you try to get the next record of the last record. The GET statement moves the results list cursor to a null position, and clears the associated record buffer, which remains empty. This condition is known as being *off the end* of the results list.

You can use the AVAILABLE function to check for this error condition (an empty record buffer), or you can use a special query function: QUERY-OFF-END. This is the syntax for QUERY-OFF-END.

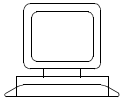
SYNTAX

```
QUERY-OFF-END ( query-name )
```

The function takes the query name as a string and returns TRUE if the cursor is off either the top or bottom end of the results list. Here is a code example using this query function:

```
IF QUERY-OFF-END ( "My-Query" ) THEN GET FIRST My-Query.
```

GET and DISPLAY Programming Example



Exercise

This exercise is based on the code used to demonstrate FIND, but instead it uses a query, GET statements, and the QUERY-OFF-END function. Follow these steps to see the results:

1 ♦ Open `lt-08-02.p` and run it. The database access form appears.

2 ♦ Choose the Prev and Next buttons and notice the changes to the Item No. and Name fields. The code works precisely the same as the FIND statement version.

3 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

This the code that created the display:

lt-08-02.p

```
      /**********  DEFINE QUERY  **********/
/*1*/  DEFINE QUERY Item-Query FOR Item.
      /**********  DEFINE FORM  **********/
      {lt-08-f1.i}

      /**********  DEFINE TRIGGERS  **********/
/*4*/  ON CHOOSE OF btn-Prev
       DO:
          GET PREV Item-Query.
         IF QUERY-OFF-END(Item-Query) THEN GET LAST Item-Query.
         DISPLAY Item.Item-Num Item-Name Price On-Hand Allocated Re-Order
            On-Order Cat-Page Cat-Description WITH FRAME Frame1.
       END.
/*5*/  ON CHOOSE OF btn-Next
       DO:
          GET NEXT Item-Query.
         IF QUERY-OFF-END(Item-Query) THEN GET FIRST Item-Query.
         DISPLAY Item.Item-Num Item-Name Price On-Hand Allocated Re-Order
            On-Order Cat-Page Cat-Description WITH FRAME Frame1.
       END.
      /**********  MAIN LOGIC  **********/
/*2*/  OPEN QUERY Item-Query FOR EACH Item.
/*3*/  GET FIRST Item-Query.
      DISPLAY Item.Item-Num Item-Name Price On-Hand Allocated Re-Order
          On-Order Cat-Page Cat-Description WITH FRAME Frame1 USE-TEXT.
      ENABLE btn-Prev btn-Next btn-Exit WITH FRAME Frame1.
      WAIT-FOR CHOOSE OF btn-Exit.
/*6*/  CLOSE QUERY Item-Query.
```

These notes help explain the code:

1. The DEFINE QUERY occurs at the beginning of the procedure.

2. OPEN QUERY at the beginning of the main code block initializes the results list that defines the matching subset of records. You cannot use GET until after you open the query.

3. The GET statement creates a record buffer and copies a record to the buffer from the database using the query results list.

4. This trigger performs the “retrieve next or retrieve first record” logic, exactly as in the FIND example. Here, the GET statement replaces the FIND statement and the QUERY-OFF-END function replaces the AVAILABLE function.

5. This trigger performs the “retrieve previous or retrieve last record” logic, exactly as in the FIND example.

6. The CLOSE QUERY statement after the WAIT-FOR statement releases the resources used by the query.

Practice Problems

The basic form used in the programming examples can quickly be refitted for use with any other table. Use the Customer table with the problems below to practice what you have learned so far.

Problem 8-1: `lt-08-s1.p`

Using the FIND and DISPLAY statements, create a procedure that lets you review records in the Customer table.

Problem 8-2: `lt-08-s2.p`

Make a copy olist:f the procedure you created in Problem 8-1. Replace the FIND statements with DEFINE QUERY and GET statements.

8.6 Modifying Data

Earlier, this chapter defined a good database access form as one that separates navigation and display of data from update of data. Separating the two functions into different modes makes clear to the user what’s going on and helps prevent unintended changes. This statement isn’t necessarily true in all cases. In some cases, you may want to use the display form as your update form.

This section describes two techniques for changing data: one that uses the main form as the display form and one that uses a separate dialog box as the update form.

Both techniques use the database access form and the Update button as shown in Figure 8–9.

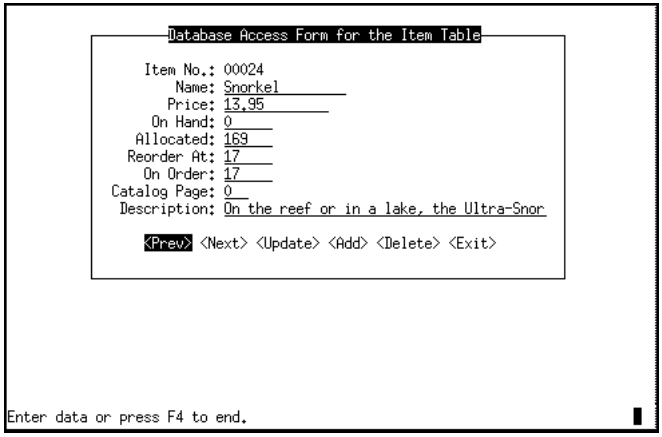


Figure 8–9: Database Access Form with Update Enabled

8.6.1 The DISPLAY, ENABLE, and ASSIGN Technique

When you worked with variables in previous chapters, you used this technique, as summarized below:

1. Use a DISPLAY statement to copy data from the variable to the widget that represents that variable.

2. Use an ENABLE statement to let the user interact with the widget and the screen value.

3. On some event, use an ASSIGN statement to copy the screen value back to the variable.

Using the form in Figure 8–9, you could follow the same sequence of steps to change database data. Choosing the Update button will be the event that copies the current screen buffer to the record buffer and eventually to the database.

To use this basic technique with database records, you need to learn more about locks, ROWIDs, ENABLE, and ASSIGN.

Locks

The technique listed above may not work well with database records because of the possibility that more than one user might try to use a record at the same time. Since most Progress applications are designed to work with multiple users, this technique needs modification to support *multi-user mode*. The Progress RDBMS has two basic rules for a database in multi-user mode:

1. Anyone can access a record at anytime to view its data.

2. Only one user at a time can access a record to change the data.

When a user retrieves a record, Progress puts the user's copy of the record in one of three *lock* states. A lock is a description on the user's freedom to use the record. Progress has three keywords that describe the lock states, as shown in Table 8–2.

Table 8–2: Locks

Lock	Description
NO-LOCK	The user can only view the data. Progress puts the user's record in this state if the NO-LOCK keyword is included in the record phrase.
SHARE-LOCK	In most cases, when a user requests a record, Progress puts it in this state. SHARE-LOCK means that the user can view the data or attempt to modify the data. If the user attempts to modify data and no other SHARE-LOCK exists, Progress upgrades the user's lock to an EXCLUSIVE-LOCK.
EXCLUSIVE-LOCK	An EXCLUSIVE-LOCK means that the user can view and change the data. While the user holds the EXCLUSIVE-LOCK, other users can only access the record in a NO-LOCK state.

To let the greatest number of users successfully access a particular record, it makes sense to keep the record in NO-LOCK state. The application should only upgrade the lock to EXCLUSIVE-LOCK during the act of changing data. This means that the record is tied up for the shortest possible period of time.

Row IDs (ROWID)

Every record (or row) in a database has a unique internal identifier which is specified by the Progress keyword ROWID. ROWIDs let you identify a particular record quickly and easily.

To work with ROWIDs, you have to get them and store them. Progress supports ROWIDs as a separate data type, which you can specify in a variable definition with the keyword ROWID. To retrieve a ROWID, you can use the ROWID( ) function.

SYNTAX

```
ROWID ( buffer-name )
```

This code fragment shows how to retrieve and store a ROWID for later use:

```
DEFINE VARIABLE Current-Record AS ROWID
             .
             .
             .
FIND FIRST Item.
Current-Record = ROWID(Item).
             .
             .
             .
```

The programming example at the end of this section demonstrates how to use ROWIDs and how to take specific control over locks. First, you need a more in-depth look at the ENABLE and ASSIGN statements.

ENABLE Statement

You saw in the last section that the DISPLAY statement moves data from the record buffer to the screen buffer. ENABLE allows users to enter data directly into the screen buffer, as shown in Figure 8–10.

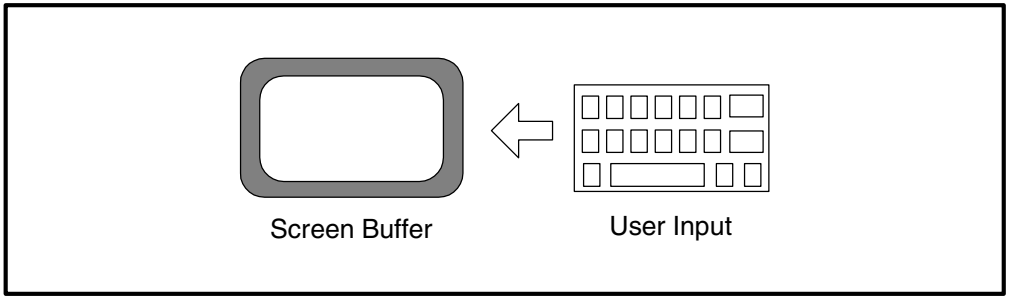


Figure 8–10: Data Movement with the ENABLE Statement

This is a partial syntax for the ENABLE statement.

SYNTAX

```
ENABLE { ALL [ EXCEPT widget ... ] | field ... }
  { [ frame-phrase ] }
```

Use the EXCEPT *widget* syntax to enable all the widgets in a frame, except for those you specify.

ASSIGN Statement

You also have encountered the ASSIGN statement. ASSIGN moves data from the screen buffer to the record buffer, as shown in Figure 8–11.

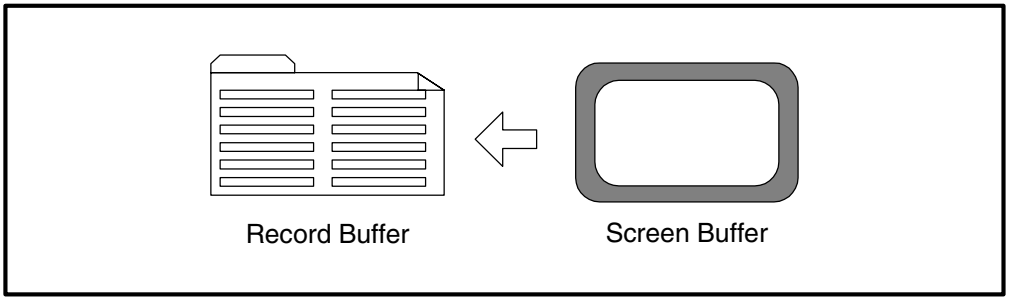


Figure 8–11: Data Movement with the ASSIGN Statement

Once you move changes from the screen buffer to the record buffer, or make changes directly to the record buffer, Progress understands that you want those changes written to the database. Progress does not immediately write the changes to the database. When the write occurs is based on other factors. The next section describes how Progress handles database writes.

This is the syntax for the ASSIGN statement.

SYNTAX

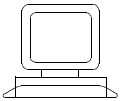
```
ASSIGN
  { { field | field = expression } ...
    record [ EXCEPT field ] ...
  }
```

You can assign fields or the entire record. The EXCEPT syntax is also valid with ASSIGN.

By using DISPLAY, ENABLE, and ASSIGN together, you create a chain of functionality that presents current data to your users, allows them to change it, and allows them to save the changes.

NOTE: Using the DISPLAY, ENABLE, and ASSIGN technique does not automatically invoke the validation behaviors stored in the Data Dictionary. To force the validation routines to run, you can query the VALIDATION attribute of a widget or frame. For more information, see the *Progress Programming Handbook*. The other techniques described in this section do invoke Data Dictionary validation.

DISPLAY, ENABLE, and ASSIGN Programming Example



Exercise

Follow these steps for a demonstration of the DISPLAY, ENABLE, and ASSIGN technique with explicit control of locks:

1 ♦ Open `lt-08-03.p` and run it. Note that the form now uses fill-ins instead of text widgets.

2 ♦ Select a record and change some of the fields.

3 ♦ Choose the Next followed by the Prev button. The record does not contain the new data. This procedure does not save your changes unless you choose the Update button.

4 ♦ Select a record, change some of the fields, and choose Update.

5 ♦ Choose the Next followed by the Prev button. The new data is now permanent.

6 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

Here is the code that created the display:

lt-08-03.p

```
      /**********  DEFINE QUERY  **********/
       DEFINE QUERY Item-Query FOR Item.
      /**********  DEFINE VARIABLES  **********/
/*3*/  DEFINE VARIABLE Current-Record AS ROWID.
      /**********  DEFINE FORM  **********/
      {lt-08-f1.i}

      /**********  DEFINE TRIGGERS  **********/
/*4*/  {lt-08-t1.i}

       ON CHOOSE OF btn-Update
       DO:
/*5*/     Current-Record = ROWID(Item).
/*6*/     FIND FIRST Item WHERE ROWID(Item) = Current-Record EXCLUSIVE-LOCK.
/*7*/     IF AVAILABLE(Item) THEN
           ASSIGN Item.Item-Name Price On-Hand Allocated Re-Order
                  On-Order Cat-Page Cat-Description.
/*8*/      ELSE DO:
           FIND FIRST Item WHERE ROWID(Item) = Current-Record NO-LOCK.
           MESSAGE "Record in use. Unable to update."
               VIEW-AS ALERT-BOX WARNING BUTTONS OK Title "Update Error".
           END.
       END.
      /**********  MAIN LOGIC  **********/
/*1*/  OPEN QUERY Item-Query FOR EACH Item NO-LOCK.
       GET FIRST Item-Query.
       DISPLAY Item.Item-Num Item-Name Price On-Hand Allocated Re-Order
          On-Order Cat-Page Cat-Description WITH FRAME Frame1.
/*2*/  ENABLE Item.Item-Name Price On-Hand Allocated Re-Order On-Order
        Cat-Page Cat-Description btn-Prev btn-Next btn-Update btn-Exit
            WITH FRAME Frame1.
      WAIT-FOR CHOOSE OF btn-Exit.
      CLOSE QUERY Item-Query.
```

The following notes help to explain the code:

1. In the main code block, the OPEN QUERY statement contains the NO-LOCK option. This means that navigating through the records or leaving the application running will not impede other users.

2. The fields in this example are specifically enabled.

3. This variable will hold the current ROWID when trying to upgrade a lock.

4. This include file contains the navigation triggers which you have already studied.

5. Once the user chooses Update, this trigger executes. Here, the trigger saves the ROWID of the current Item record.

6. Recall that FIND retrieves an individual record. This trigger uses FIND to get a copy of the record with the EXCLUSIVE-LOCK option. The FIND statement does not affect the defined query.

7. If any other user has EXCLUSIVE-LOCK or SHARE-LOCK, the FIND will fail, which is why this IF statement checks the record buffer. If the record is there, the user's changes are saved with the ASSIGN statement.

8. If the FIND failed, then refind the record with a NO-LOCK and inform the user.

One way to keep data accessible to all users is to keep the data in NO-LOCK and let the user change it. When it is time to assign the changes, use the FIND CURRENT or GET CURRENT statement to get a fresh copy of the record that the user has been modifying with an EXCLUSIVE-LOCK. Then, use the CURRENT-CHANGED function to test for changes to the record while the user was working with it. If the record was changed by another user, you can send a message to the user and display the updated record. If the record is unchanged, you can assign the changes.

You can use this trigger code in place of the trigger code for btn-Update in `lt-08-03.p` as the following code fragment illustrates:

```
          .
          .
          .
ON CHOOSE OF btn-Update
DO:
    GET CURRENT Item EXCLUSIVE-LOCK.
    IF AVAILABLE(Item) THEN
        IF NOT CURRENT-CHANGED Item THEN
           ASSIGN Item.Item-Name Price On-Hand Allocated Re-Order
                On-Order Cat-Page Cat-Description.
        ELSE DO:
           MESSAGE "Record has been changed by another user.
                Redisplaying new data.".
           DISPLAY Item.Item-Name Price On-Hand Allocated Re-Order
                On-Order Cat-Page Cat-Description.
        END.
    ELSE DO:
        FIND FIRST Item WHERE ROWID(Item) = Current-Record NO-LOCK.
        MESSAGE "Record in use. Unable to update."
            VIEW-AS ALERT-BOX WARNING BUTTONS OK Title "Update Error".
        DISPLAY Item.Item-Name Price On-Hand Allocated Re-Order
                On-Order Cat-Page Cat-Description.
    END.
END.
          .
          .
          .
```

8.6.2 The UPDATE Technique

A separate mode for updating data protects data from unintended changes. The technique shown in this section relies on a statement normally reserved for application-driven programs and a special type of frame. To understand the technique, you need to learn about the UPDATE statement and the dialog box.

UPDATE Statement

To put it succinctly, the UPDATE statement is a powerful statement that does the work of the DISPLAY, ENABLE, ASSIGN, and WAIT-FOR statements. The procedure below is a complete application that does much of the work of the last programming example:

```
FIND FIRST Item.

REPEAT:

  UPDATE Item WITH 2 COLUMNS.
  FIND NEXT Item.

END.
```

Here's what this application does:

1. Finds the first Item record.
2. The UPDATE statement:
   - Creates a default display.
   - Displays data from the record.
   - Enables the widgets.
   - Blocks execution to allow the user to interact with the widgets.
   - Waits for the GO event function.
   - Assigns changes after receiving the GO event function. (Pressing F2 executes GO. After finishing with each record, the user presses F2 to move on to the next record.)
3. Finds the next record.
4. Iterates until it encounters an error condition: either no more records exist or the user issues the ENDKEY (F5) event function. If the user issues ENDKEY, any changes made to the current record are discarded.

This application is not very flexible, but illustrates both the power of UPDATE and why UPDATE should normally not be used with an event-driven procedure. As you saw in the description above, UPDATE blocks application while waiting for GO. If you mixed an update statement with a WAIT-FOR statement, you would have explicit application blocking and implicit application blocking conflicting with each other. The result would be confusing to the end user. UPDATE should be used by itself. As you you'll see later in the section, you can combine UPDATE with a dialog box to provide a separate update mode for database records.

Figure 8–12 shows the data movement with UPDATE.

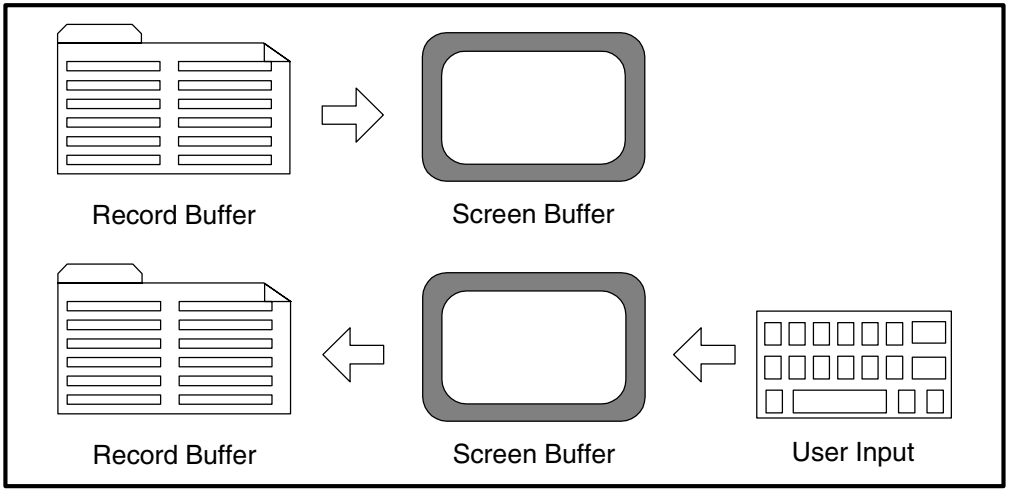


Figure 8–12: Data Movement with the UPDATE Statement

This is a partial syntax for the UPDATE statement.

SYNTAX

```
UPDATE
  {   field
    | SPACE [ ( n ) ]
    | SKIP [ ( n ) ]
  } ... { [ frame-phrase ] }
```

Dialog Boxes

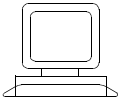
A dialog box is a container widget that disables all widgets located in other container widgets while the dialog box is enabled. Since all other widgets are disabled, you can execute an UPDATE statement with a dialog box from within a block of code that includes a WAIT-FOR statement. Since the implied application blocking of the UPDATE statement cannot interfere with the explicit application blocking of the WAIT-FOR, this combination elegantly creates an update mode.

You create a dialog box by specifying the VIEW-AS DIALOG BOX option in the frame phrase.

The dialog box appears on top of all other frames and stays there until the user or the application dismisses it. In this case, completing the UPDATE statement dismisses the dialog box. So, if the user presses **F2** (**GO**), UPDATE assigns the changes and the dialog box disappears. If the user presses **F4** (**ENDKEY**), the changes are discarded and the dialog box disappears.

The use of the **GO** and **ENDKEY** keystrokes are not very elegant in an event-driven application. Fortunately, the DEFINE BUTTON statement supports options that let buttons automatically execute the GO and ENDKEY event functions: AUTO-GO and AUTO-ENDKEY. If you define a button labeled OK with the AUTO-GO option and another button labeled Cancel with the AUTO-ENDKEY option, you have an interface that fits in nicely with the conventions of an event-driven application.

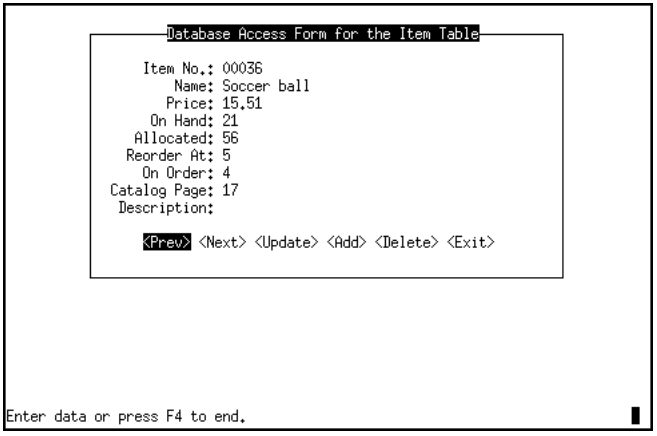
UPDATE and Dialog Box Programming Example



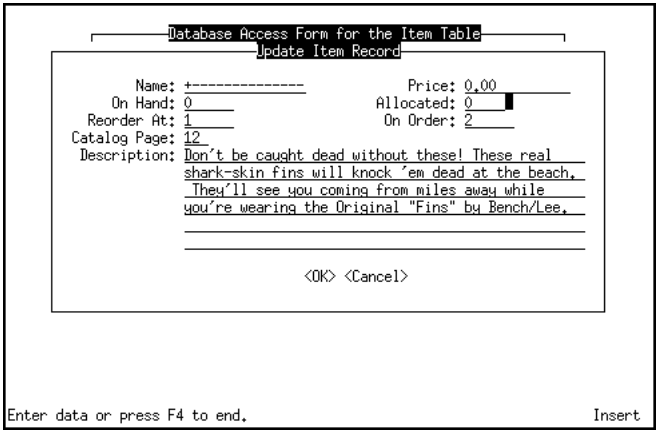
Exercise

Follow these steps for a demonstration:

1 ♦ Open `lt-08-04.p` and run it. The display appears as shown below, with the fields on the form represented as text widgets.



2 ♦ Choose Update. The update dialog box appears, as shown below:



Note that you cannot move focus to any widget outside the dialog box.

3 ♦ Change the data and choose Cancel. The dialog box disappears and the main form remains unchanged.

4 ♦ Choose Update again, change some data, and Choose OK. The dialog box disappears and the main form shows your changes.

5 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

This is the code for this application:

lt-08-04.p

```
       /**********  DEFINE QUERY  **********/
       DEFINE QUERY Item-Query FOR Item.

       /**********  DEFINE VARIABLES  **********/
       DEFINE VARIABLE Current-Record AS ROWID.

       /**********  DEFINE FORM  **********/
       {lt-08-f1.i}

      /**********  DEFINE TRIGGERS  **********/
       {lt-08-t1.i}

       ON CHOOSE OF btn-Update
        DO:
/*2*/      Current-Record = ROWID(Item).
          FIND FIRST Item WHERE ROWID(Item) = Current-Record
             EXCLUSIVE-LOCK.
          IF AVAILABLE(Item) THEN DO:
/*3*/          UPDATE Item.Item-Name Price On-Hand Allocated Re-Order
              On-Order Cat-Page Cat-Description btn-OK btn-Cancel
                  WITH FRAME Dialog1.
              DISPLAY Item-Num Item-Name Price On-Hand Allocated Re-Order
                On-Order Cat-Page Cat-Description WITH FRAME Frame1.
            END.
/*4*/       ELSE DO:
            FIND FIRST Item WHERE ROWID(Item) = Current-Record NO-LOCK.
            MESSAGE "Record in use. Unable to update."
                VIEW-AS ALERT-BOX WARNING BUTTONS OK Title "Update Error".
            END.
        END.

      /**********  MAIN LOGIC  **********/
/*1*/  OPEN QUERY Item-Query FOR EACH Item NO-LOCK.
      GET FIRST Item-Query.
     DISPLAY Item.Item-Num Item-Name Price On-Hand Allocated Re-Order
        On-Order Cat-Page Cat-Description WITH FRAME Frame1 USE-TEXT.
     ENABLE btn-Prev btn-Next btn-Update btn-Exit WITH FRAME Frame1.
      WAIT-FOR CHOOSE OF btn-Exit.
      CLOSE QUERY Item-Query.
```

These notes help to explain the code:

1. In the main code block, the NO-LOCK option lets all users have complete access.
2. The procedure saves the ROWID, as before.
3. If the upgrade to EXCLUSIVE-LOCK is successful, then the UPDATE statement and dialog box combination take over.
4. Again, if the lock upgrade fails, refind the record with NO-LOCK and notify the user.

8.7 Releasing Records

When you make changes to a record buffer, Progress knows that those changes are intended for the database. Progress does not, however, write changes to the database as it receives them in the record buffer.

Progress makes the changes to the database only when your procedure is done with the record. Progress understands when you are finished with a record because the rules that determine how long a record stays in a record buffer are enforced by the 4GL. The life span of a record in a record buffer is called the *scope* of the record. When the scope of a record ends, Progress *releases* the record and surrenders it to the RDBMS. The RDBMS has its own rules for determining when it is safe and efficient to write the record to the database.

The basic rule for determining the scope of a record is that the scope lasts from the moment the record is retrieved until:

- Another record is requested.
- The code block that owns the record buffer ends.

As a general rule, you could say that a record's scope ends on a new record request or an END statement. That statement is subject to many exceptions and some special situations where Progress allows you to bend the rules. As you work through this tutorial, the general rule of thumb holds. When you finish working with the tutorial, you can read more about record scoping in the *Progress Programming Handbook*.

Earlier in the tutorial, you learned that Progress can create and maintain a separate record buffer for each database table. Actually, each block that can own a record buffer can have a separate record buffer for the same database table. Blocks that can have buffers include:

- Procedures
- FOR EACH

- DO with the FOR option
- REPEAT with the FOR option

If a block is not capable of owning its own buffer then it uses the buffer of the block in which it resides. So, unless you have heavily nested blocks, the block uses its own buffer or the main procedure block's buffer.

The idea of multiple buffers is important because it means your procedures can work on more than one record from a table at a time.

Explicit Release

You can end the scope of a record prematurely by forcing Progress to release it with the RELEASE statement.

This is the syntax for the RELEASE statement.

SYNTAX

```
RELEASE record
```

Figure 8–13 shows the data movement of RELEASE.

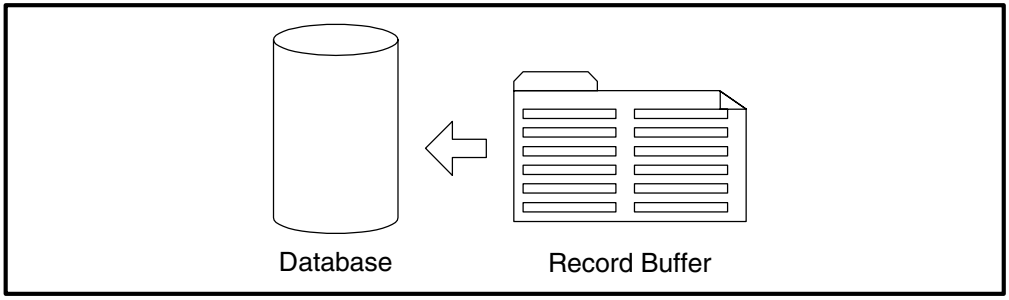


Figure 8–13: Data Movement with the RELEASE Statement

Practice Problems

Problem 8-3: `lt-08-s3.p`

Make a copy of your solution to Problem 8-2. Now, add the ability to update the fields of your customer review program using the UPDATE statement and a dialog box.

8.8 Creating Records

When you create a record, you are directly manipulating the database. Progress creates a new empty record and copies that record to the record buffer in EXCLUSIVE-LOCK state. The new record may not be entirely empty, however. Depending on your database schema, some of the fields may have default initial values or values generated by sequences and database triggers. Figure 8–14 shows the data movement of CREATE.

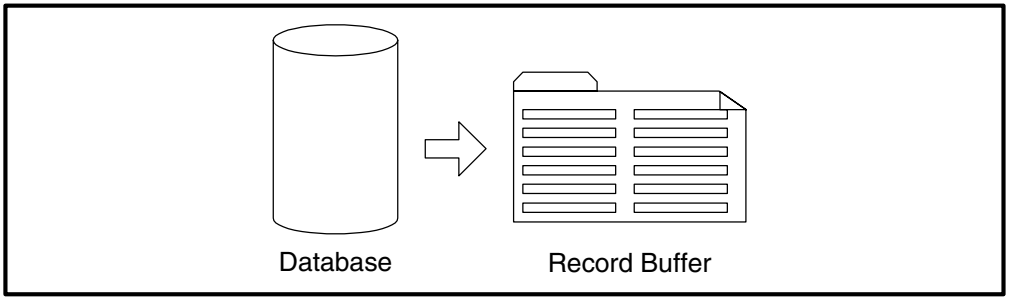


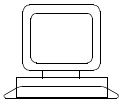
Figure 8–14: Data Movement with the CREATE Statement

This is the syntax for the CREATE statement.

SYNTAX

```
CREATE table-name
```

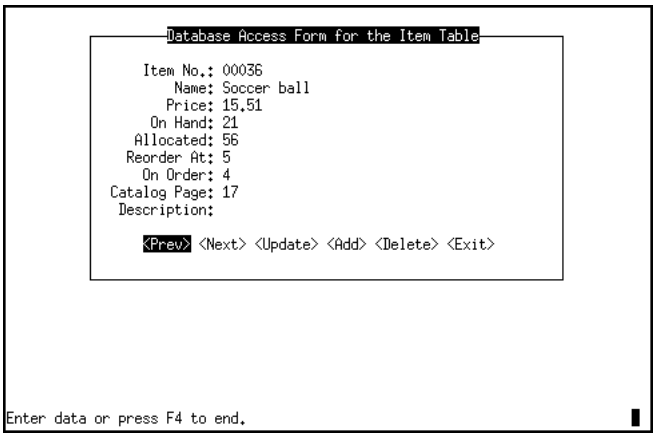
As soon as you create a record, you'll want to fill it with data. One way to acquire new data for the record is to use a dialog box with an UPDATE statement.



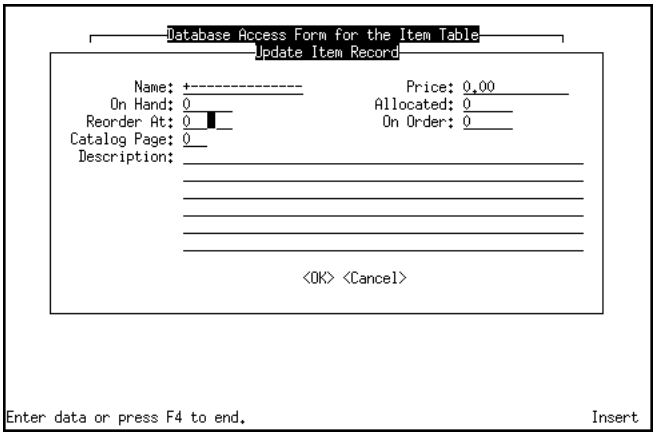
Exercise

Follow these steps for a demonstration of creating records:

1 ♦ Open the file named `lt-08-05.p` and run it. The following display appears:



2 ♦ Choose Add, enter data in the update dialog box, shown below, and choose OK:



The main form displays data from the new record.

3 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

Here is the code that created the display:

lt-08-05.p

```
      /**********  DEFINE QUERY  **********/
      DEFINE QUERY Item-Query FOR Item.


      /**********  DEFINE VARIABLES  **********/
      DEFINE VARIABLE Current-Record AS ROWID.

      /**********  DEFINE FORM  **********/
      {lt-08-f1.i}

      /**********  DEFINE TRIGGERS  **********/
      {lt-08-t1.i} {lt-08-t2.i}

      ON CHOOSE OF btn-Add
      DO:
/*1*/      CREATE Item.
/*2*/     UPDATE Item.Item-Name Price On-Hand Allocated Re-Order
            On-Order Cat-Page Cat-Description btn-OK btn-Cancel
                WITH FRAME Dialog1.
/*3*/     DISPLAY Item-Num Item-Name Price On-Hand Allocated Re-Order
            On-Order Cat-Page Cat-Description WITH FRAME Frame1.
      END.

     /**********  MAIN LOGIC  **********/
     OPEN QUERY Item-Query FOR EACH Item NO-LOCK.
     GET FIRST Item-Query.
    DISPLAY Item.Item-Num Item-Name Price On-Hand Allocated Re-Order
        On-Order Cat-Page Cat-Description WITH FRAME Frame1 USE-TEXT.
     ENABLE btn-Prev btn-Next btn-Update btn-Add btn-Exit
      WITH FRAME Frame1.
     WAIT-FOR CHOOSE OF btn-Exit.
     CLOSE QUERY Item-Query.
```

These notes help to explain the code:

1. The CREATE statement establishes the record and interacts with the schema to get default values.

2. Since the record has EXCLUSIVE-LOCK, you can move right to an UPDATE statement.

3. The DISPLAY statement refreshes the main form with the new data.

8.9 Deleting Records

The process for deleting records is simple. The following list identifies the steps in the delete process:

1. Find a record to delete.
2. Upgrade to an EXCLUSIVE-LOCK.
3. Use the DELETE statement.

Figure 8–15 shows the data movement for the DELETE statement.

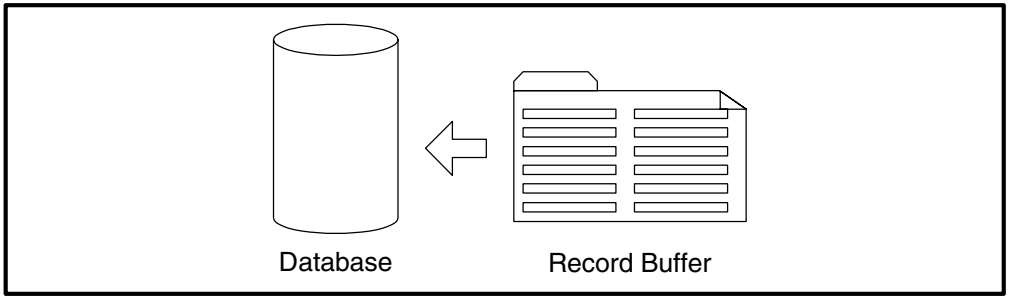


Figure 8–15: Data Movement with the DELETE Statement

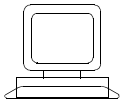
The figure may seem nonsensical. The DELETE statement doesn't copy data to the database. What it does do is mark the record for deletion and release the record to the RDBMS for eventual removal from the database. You could say that DELETE sends instructions, not data, back to the database.

Again, the syntax is simple, although delete has a new useful option. This is the syntax for the DELETE statement.

SYNTAX

```
DELETE record [ VALIDATE ( condition , msg-expression ) ]
```

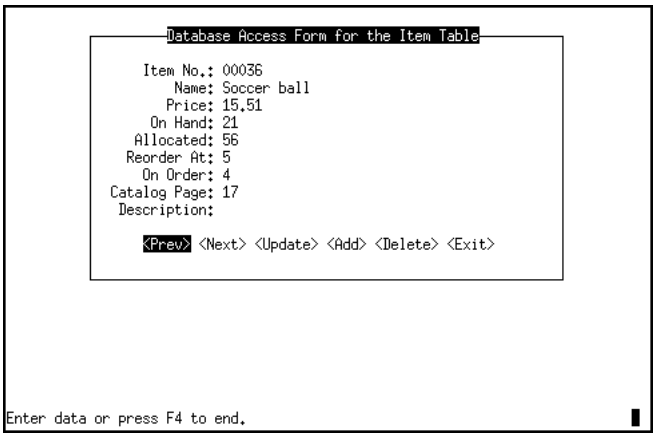
The VALIDATE function accepts an expression and a message to check and make sure that the deletion is allowable according to your business rules.



Exercise

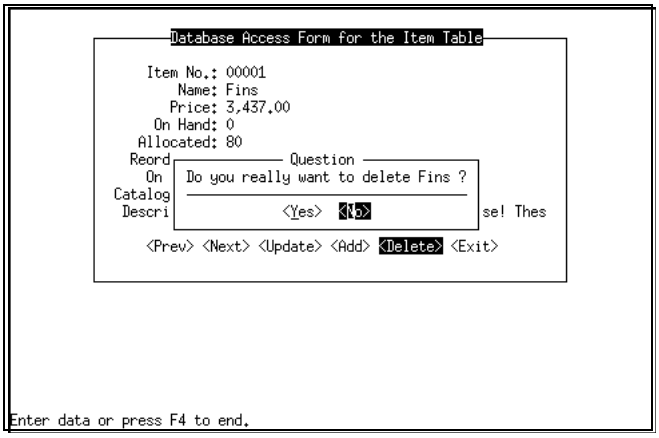
Follow these steps for a demonstration of the DELETE statement:

1 ♦ Open `lt-08-05.p` and run it. The following display appears:



2 ♦ Choose the Prev button to get one of the records you created in the last exercise. Since the new records were created with high customer numbers, they appear last in the record list. (Cust-Num is the primary index.)

3 ♦ Choose Delete Customer. An alert box appears, as show below.



An *alert box* is a sophisticated alternative for sending messages to the status area. The status area is an extension of the window widget and so is the alert box.

In event-driven programming, alert boxes ask questions of the user or present the user with important information. Since alert boxes are bearers of important news, they block the rest of the interface—like a dialog box—until the user responds to them.

The reason that a major new component is introduced right in the middle of a discussion on the DELETE statement has to do with good programming style. Good event-driven programming style dictates that you should always have the user confirm deletion of important data with an alert box. Chapter 12, "Managing Your Application," discusses how alert boxes work.

4 ♦ Choose Yes to delete the record, or choose No to cancel the deletion.

If you try to delete one of the records that come with the sports database, you may get alert boxes explaining why you can't delete the record. These behaviors are established by database triggers in the schema. They are a way of enforcing business rules.

5 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

Here is the code that created the display:

lt-08-06.p

```
    /**********  DEFINE QUERY  **********/
    DEFINE QUERY Item-Query FOR Item.
    /**********  DEFINE VARIABLES  **********/
    DEFINE VARIABLE Current-Record AS ROWID.
    /**********  DEFINE FORM  **********/
    {lt-08-f1.i}
    /**********  DEFINE TRIGGERS  **********/
   {lt-08-t1.i} {lt-08-t2.i} {lt-08-t3.i}
    ON CHOOSE OF btn-Delete
    DO:
/*1*/      MESSAGE "Do you really want to delete" Item.Item-Name "?"
             VIEW-AS ALERT-BOX QUESTION BUTTONS YES-NO UPDATE Answer.
/*2*/      IF Answer THEN DO:
             Current-Record = RECID(Item).
/*3*/         FIND FIRST Item WHERE RECID(Item) = Current-Record
             EXCLUSIVE-LOCK NO-ERROR.
/*4*/          IF AVAILABLE(Item) THEN DO:
                DELETE Item.
/*5*/               OPEN QUERY Item-Query FOR EACH Item NO-LOCK.
/*6*/                GET FIRST Item-Query.
                  DISPLAY Item.Item-Num Item-Name Price On-Hand Allocated
                      Re-Order On-Order Cat-Page Cat-Description
                         WITH FRAME Frame1.
                END.
/*7*/           ELSE DO:
                 FIND FIRST Item WHERE RECID(Item) = Current-Record
                     NO-LOCK.
                  MESSAGE "Record in use. Unable to update."
                    VIEW-AS ALERT-BOX WARNING BUTTONS OK
                       TITLE "Delete Error".
                END.
           END.
    END.
   /**********  MAIN LOGIC  **********/
   OPEN QUERY Item-Query FOR EACH Item NO-LOCK.
   GET FIRST Item-Query.
   DISPLAY Item.Item-Num Item-Name Price On-Hand Allocated Re-Order
        On-Order Cat-Page Cat-Description WITH FRAME Frame1 USE-TEXT.
   ENABLE btn-Prev btn-Next btn-Update btn-Add btn-Delete btn-Exit
        WITH FRAME Frame1.
   WAIT-FOR CHOOSE OF btn-Exit.
   CLOSE QUERY Item-Query.
```

These notes help explain the code:

1. A special VIEW-AS syntax on a MESSAGE statement reroutes a message from the status area to an alert box. The UPDATE option and variable captures the user's response to the question.
2. If the user answers YES, then the DELETE logic executes.
3. This FIND statement upgrades the lock.
4. If the record is available with an EXCLUSIVE-LOCK, then DELETE executes.
5. Reopening the query rebuilds the results list without the newly deleted record.
6. This statement resets the cursor to the first record.
7. This logic executes if the record is unavailable for deletion.

Practice Problems

Problem 8-4: `lt-08-s4.p`

Make a copy of your solution to problem 8-3. Add the ability to create new customer records and to delete customer records using the techniques covered in this chapter.

8.10 Other Database Access Statements

This section provides a quick summary of three other statements that work with database records: PROMPT-FOR, SET, and INSERT. These statements aren't used very often in event-driven applications.

8.10.1 The PROMPT–FOR Statement

The PROMPT-FOR statement accepts data from the keyboard to the screen buffer, as shown in Figure 8–16.

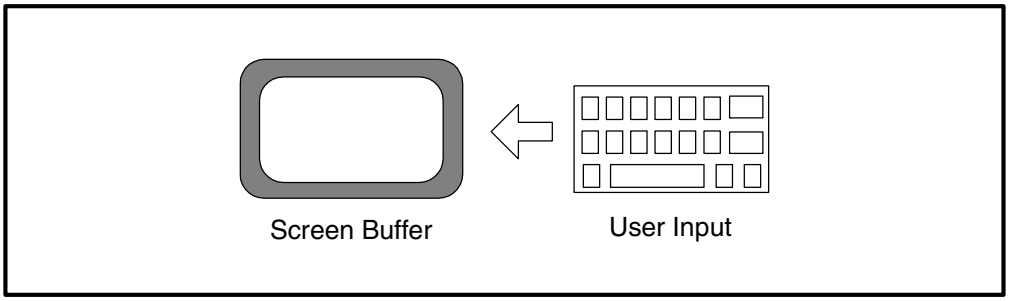


Figure 8–16: Data Movement with the PROMPT–FOR Statement

This is a partial syntax for the PROMPT-FOR statement.

SYNTAX

```
PROMPT-FOR
  {   field
    | SPACE [ ( n ) ]
    | SKIP [ ( n ) ]
  } ... { [ frame-phrase ] }
```

Like UPDATE, PROMPT–FOR has implied application blocking.

8.10.2 The SET Statement

SET works like a PROMPT-FOR followed by an ASSIGN statement. Figure 8–17 shows the data movement with SET.

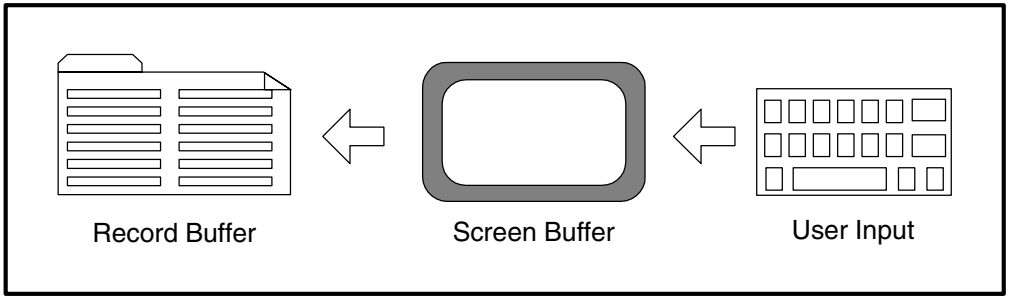


Figure 8–17: Data Movement with the SET Statement

You use SET when you don't need to execute any tasks between the time that the data is in the screen buffer and moves to the record buffer. This is a partial syntax for the SET statement.

SYNTAX

```
SET
  { field
    | SPACE [ ( n ) ]
    | SKIP [ ( n ) ]
  } ...
  { [ frame-phrase ] }
```

SET also has implied application blocking.

8.10.3 The INSERT Statement

The INSERT works like a combined CREATE and UPDATE statement. Typically, you use INSERT only when you are creating a lot of records, like when you create a new database from scratch. Figure 8–18 shows the data movement of the INSERT statement.

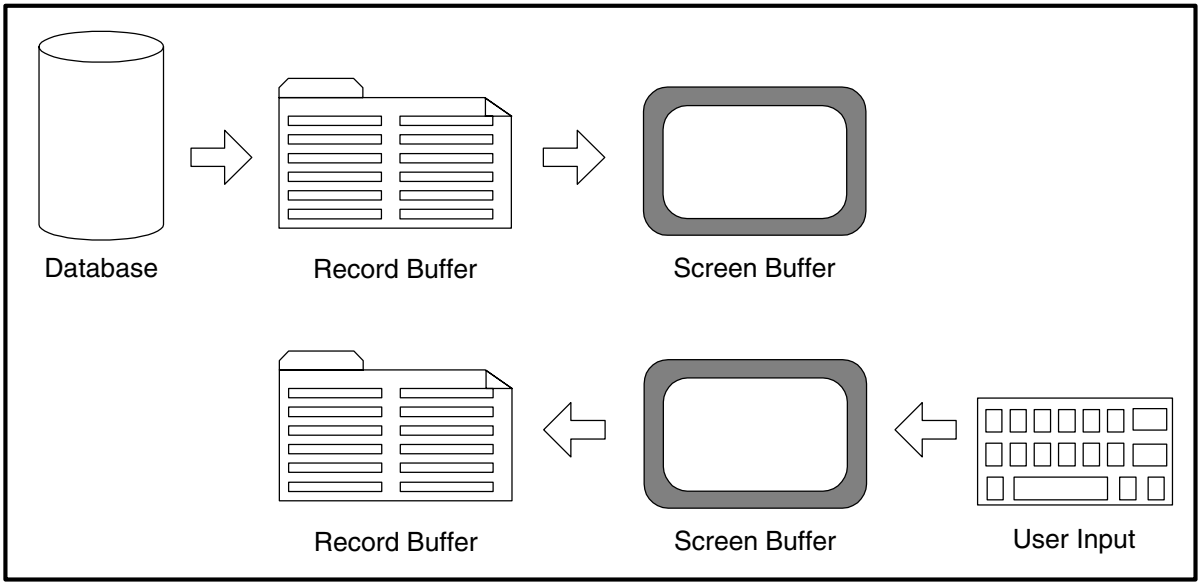


Figure 8–18: Data Movement with the INSERT Statement

This is a partial syntax for the INSERT statement.

SYNTAX

```
INSERT record
  [ EXCEPT field ... ] { [ frame-phrase ] }
```

INSERT also has implied application blocking.

8.10.4 Using the Browse Widget

The data widgets you've used so far represent a single variable or database field. The browse widget goes beyond that and represents the results list of a defined query. Basically, you can think of a browse widget as a selection list of database records. Instead of listing a single value on each line of the list, browse widgets let you list values from several fields.

You can use a browse widget to display or update record values. When you use a browse for display only, it is called a read-only browse. Figure 8–19 describes the parts of a read-only browse widget.

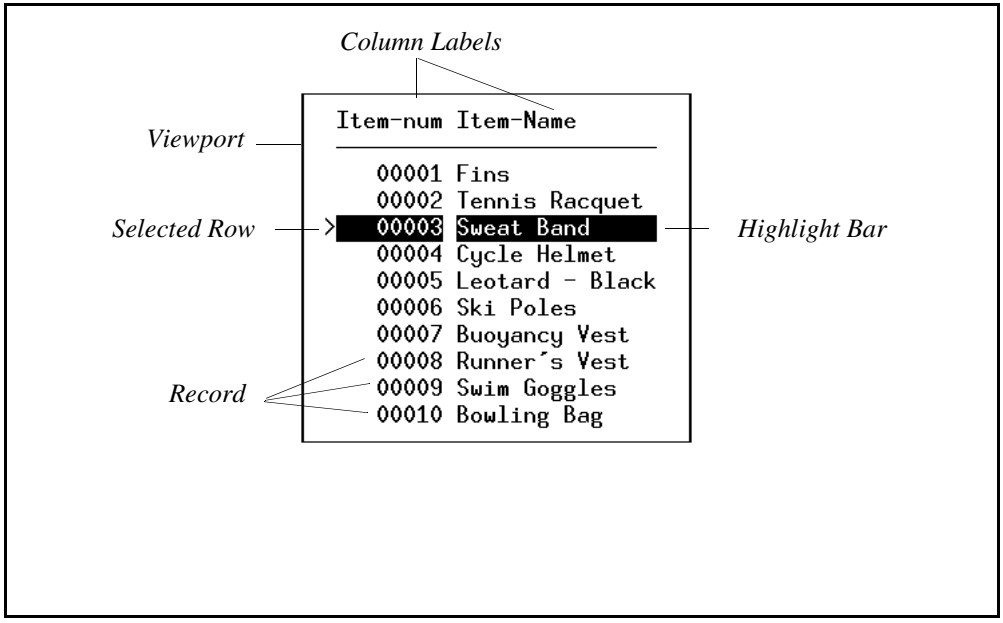


Figure 8–19: Parts of a Read-only Browse Widget

When you use a browse to let the user view and update record values, it is called an updatable browse. Figure 8–20 describes the parts of an updatable browse widget.

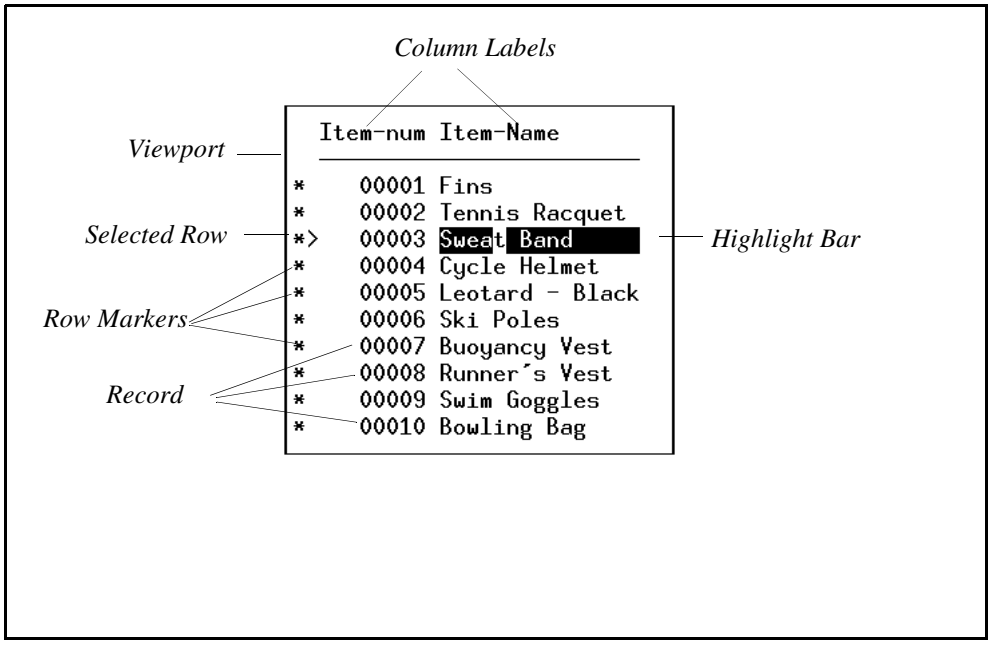


Figure 8–20: Parts of an Updatable Browse Widget

Like the other widgets you've used so far, the browse widget is a screen object. You can manipulate the browse widget in many of the same ways you manipulate the other widgets. The browse widget is a ready-made navigable interface to a query.

You can update the form you've built in this chapter by replacing the navigation buttons with an updatable browse widget, as shown in Figure 8–21.

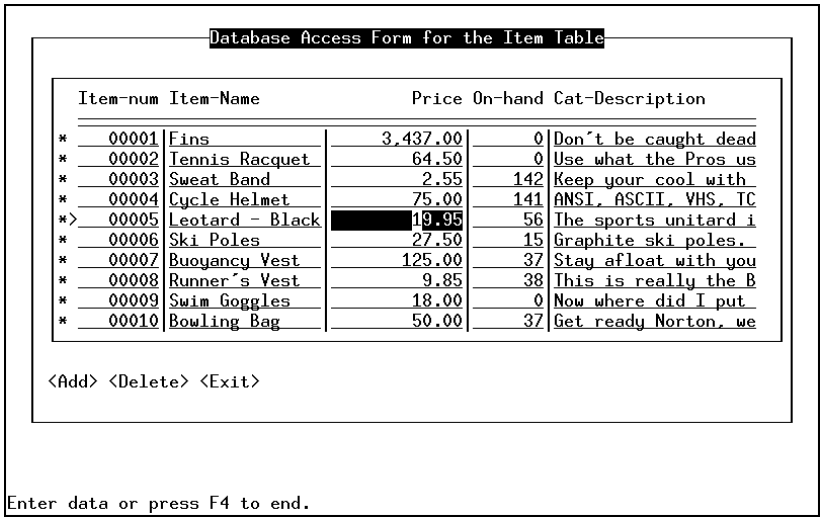


Figure 8–21: Database Access Form with a Browse Widget

To completely understand browse widgets, you need to understand how to work with the screen object and how to work with the associated query.

NOTE: Keep these points in mind when using the browse widget:

- The type of row marker indicator that displays can vary depending on the platform on which you are running the Progress character client.
- The mouse cannot be used to select and access records from either a read-only or an updatable browse widget when the browse widget is displayed in character client mode on Windows platforms. You must use the keyboard features.

The syntax for a browse widget looks like a DISPLAY statement with a frame phrase attached, but it's not. The browse widget mimics the syntax of the DISPLAY statement because it is intuitive and employs many of the same concepts. This is a partial syntax for defining a browse widget.

SYNTAX

```
DEFINE BROWSE browse-name QUERY query-name
  DISPLAY
    { expression [ format-phrase ] [ LABEL label ] } ...
  [ browse-enable-phrase ]
  [ browse-options-phrase ]
```

Table 8–3 defines some of the elements of the DEFINE BROWSE statement syntax.

Table 8–3: DEFINE BROWSE Statement Syntax

Element	Description
browse-name	Like a variable, each browse widget needs a unique name.
query-name	To specify the name of the query with which the browse is associated.
expression	To specify the field or expression for each column. You can include a format phrase and LABEL option for each field reference.
format-phrase	To override the database format string, include one using the FORMAT "*string*" syntax. You can use the WIDTH option to specify a column width.
LABEL	To override the database label options, include a LABEL option.
enable-phrase	To enable specific columns for input in an updatable browse.
options-phrase	To specify the number of records to show in the browse with the DOWN *rows* option after the WITH keyword. Otherwise, Progress sets the size of the browse. Alternately, you can use the *size-phrase* to determine a precise size, and Progress will determine the number of rows it can display in the browse viewport. For a complete list of options, see the *Progress Language Reference*. This syntax mimics the true frame phrase.

Read-only Browse

A read-only browse is a good mechanism for viewing records without allowing any changes. When you press the arrow keys (**CURSOR-DOWN** or **CURSOR-UP**), Progress selects the highlighted row.

NOTE: Mouse functionality is not supported in the read-only browse widget when the widget is displayed in character client mode on a Windows platform. You must use the keyboard features to select and display records.

Updatable Browse

An updatable browse is a good mechanism for applying changes to your data. You can add, delete, and update records with an updatable browse. The capabilities of the updatable browse are very similar to those of a spreadsheet application. You can view and update records, working in two browse modes—edit mode and row mode. Edit mode allows you to update one or more rows. Row mode allows you only to highlight and select rows.

When you first enter (tab into) an updatable browse, you are in edit mode with focus on the first enabled cell (first enabled column in the first row). When the browse enters edit mode, Progress selects the focused row and executes a ROW-ENTRY event. When you move focus away from that row, Progress executes a ROW-LEAVE event followed by a VALUE-CHANGED event, and writes any changes to the database. You can move focus and selection from row to row using the up and down arrow keys. Progress also executes ENTRY and LEAVE events for the columns you move in and out of within the focused row.

You can move from cell to cell throughout the browse using **EDITOR-TAB** (**CTRL-G**, by default) and **BACK-TAB** (**CTRL-U**, by default). These key functions execute the appropriate row and column events as they move focus through the browse.

NOTE: The **TAB** key only moves focus between the browse and neighboring widgets.

To change to row mode, press the **REPLACE** function key (**ESC-R**, by default). When you press the **REPLACE** key, Progress highlights the entire row that has focus. This highlight allows you to select the focused row using the cursor keys, but not change any row data.

To return to edit mode, press **REPLACE**. The highlight places focus in the first enabled cell of the focused row.

NOTE: Mouse functionality is not supported in the updatable browse widget when the widget is displayed in character client mode on a Windows platform. You must use the keyboard features to select and access records.

To define an updatable browse, use the *browse-enable-phrase* of the DEFINE BROWSE statement. You can enable one to all the columns in a browse for input with the ENABLE option. Remember not to allow changes to a unique index, such as Cust-Num or Item-Num.

Browse Control Keys

Table 8–4 summarizes the control keys that are specially defined for the browse. It also highlights the similarities and differences between common keyboard mappings that can be used when you are running a Progress character client on UNIX and when you are running a Progress character client on Windows.

Table 8–4: Browse Control Keys *(1 of 4)*

Key Function	Common Keyboard Mappings on UNIX	Common Keyboard Mappings on Windows	Description
(Space Bar)[1]	(Space Bar)	(Space Bar)	Edit mode: Enters a space or zeroes numeric data in the focused cell. Row mode: Fires the VALUE-CHANGED event.
CURSOR-LEFT	¨	¨	Edit Mode: Moves the cursor one character to the left in the cell. (Does not leave the cell.) Row mode: Scrolls the browser widget horizontally one column to the left.
CURSOR-RIGHT	Æ	Æ	Edit mode: Moves the cursor one character to the right in the cell. (Does not leave the cell.) Row mode: Scrolls the browser widget horizontally one column to the right.

Table 8–4: Browse Control Keys *(2 of 4)*

Key Function	Common Keyboard Mappings on UNIX	Common Keyboard Mappings on Windows	Description
CURSOR-DOWN	Ø	Ø	Edit mode: Moves focus down to the next cell in the column. Row mode: Moves focus down one row.
CURSOR-UP	¦	¦	Edit mode: Moves focus up to the previous cell in the column. Row mode: Moves focus up one row.
BACK-TAB	CTRL-U	CTRL-U	Edit mode: Moves focus to the previous enabled cell in the browse (right to left; bottom to top). Row mode: No function.
EDITOR-TAB	CTRL-G	CTRL-G	Edit mode: Moves focus to the next enabled cell in the browse (left to right; top to bottom). Row mode: No function.
END	ESC-.	END	Edit mode: Moves focus to the last cell in the current column. Row mode: Moves focus to the last row in the browse widget.

Table 8–4: Browse Control Keys *(3 of 4)*

Key Function	Common Keyboard Mappings on UNIX	Common Keyboard Mappings on Windows	Description
HOME	ESC-,	HOME	Edit mode: Moves focus to the first cell in the current column. Row mode: Moves focus to the first row in the browse widget.
PAGE-DOWN	ESC-CURSOR -DOWN	PAGE-DOWN	Edit mode: Pages down one full page of data. Row mode: Pages down one full page of data.
PAGE-UP	ESC-CURSOR -UP	PAGE-Up	Edit mode: Pages up one full page of data. Row mode: Pages up one full page of data.
REPLACE	ESC-R	CTRL-ALT-R	Edit mode: Changes the browser to row mode, with selection set to the currently focused row. Row mode: Changes the browser to edit mode and places focus in the first enabled cell of the currently focused row. If you define the browse with the NO-ROW-MARKERS option, REPLACE has no effect.

Table 8–4: Browse Control Keys *(4 of 4)*

Key Function	Common Keyboard Mappings on UNIX	Common Keyboard Mappings on Windows	Description
RETURN	RETURN	ENTER	Edit mode: Moves focus to the next cell in the current column. Row mode-Fires the DEFAULT-ACTION event.
TAB	TAB	TAB	Always leaves the browse widget 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 (" ").

Working with the Query

A query associated with a browse widget must be defined with the SCROLLING option.

When you open a query, Progress automatically fills the browse. When you reopen the query, Progress refreshes the browse with the latest data.

If a query is associated with a browse, do not use the GET statement with that query. If you do, the browse might get out of sync with the query.

Browse Events

An enabled browse widget accepts many events. The browse:

- Accepts input focus by way of the navigation key functions (such as **TAB**).
- Responds to the universal key functions (such as **GO** or **HELP**).
- Moves input focus among the records with the arrow keys.
- Responds to ENTRY, LEAVE, and VALUE-CHANGED triggers.
- Responds to SCROLL-NOTIFY triggers.
- Responds to HOME and END triggers.

The browse widget supports the following column-level events:

- **ENTRY** — Occurs when the user enters the named column in the focused row.
- **LEAVE** — Occurs when the user leaves the named column in the focused row.

The browse widget also supports the following row-level events:

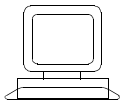
- **ROW-ENTRY** — Occurs when the user or system moves focus to a row.
- **ROW-LEAVE** — Occurs when the user or system moves focus away from a row.
- **ROW-DISPLAY** — Occurs when a row is displayed in the browse viewport.

Deleting a Row in an Updatable Browse

When you delete a row from an updatable browse widget, you must perform two distinct steps to complete the record deletion. First, reread the record with an EXCLUSIVE-LOCK and use the DELETE statement to remove the record from the database. Next, use the DELETE-SELECTED-ROWS( ) method to delete the record from the browse widget and the query results list.

It is important to remember that when you delete a row from a browse using the DELETE-SELECTED-ROWS( ) method, you are not deleting the record from the database. The DELETE-SELECTED-ROWS( ) method deletes the record from the results list, but the record is still in the database. Make sure that you remove a record from all sources when you delete a browse row.

Browse Programming Example



Exercise

In this example, the complete database access form developed so far is modified to use a browse widget.

1 ♦ Open `lt-08-7c.p` and run it. The Database Access Form appears.

2 ♦ Tab through the buttons (or back tab) into the browse and select an item with the arrow keys.

3 ♦ Select the Item-Name, Price, On-hand, or Cat-Description field using **EDITOR-TAB** and make a change to the field. (Note that you cannot select the Item-num field.)

4 ♦ Move focus to another row. Notice that Progress saved your changes to the updated field.

5 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

Here is the first section of code for the new form:

lt-08-07a.p

```
      /**********  DEFINE QUERY  **********/
/*1*/  DEFINE QUERY Item-Query FOR Item SCROLLING.
      /**********  DEFINE VARIABLES  **********/
      DEFINE VARIABLE Method-Status AS LOGICAL.
      DEFINE VARIABLE Current-Record AS ROWID.

      /**********  DEFINE BROWSE  **********/
/*2*/  DEFINE BROWSE Item-Browse QUERY Item-Query DISPLAY Item-Num
        Item-Name Price On-hand Cat-Description WIDTH 18
        ENABLE Item-Name Price On-hand Cat-Description WITH 10 DOWN
            SEPARATORS.

      /**********  DEFINE WIDGETS  **********/
/*3*/  DEFINE BUTTON btn-Add LABEL "Add".
      DEFINE BUTTON btn-Delete LABEL "Delete".
      DEFINE BUTTON btn-Exit LABEL "Exit".

      /**********  DEFINE FRAMES  **********/
      DEFINE FRAME Frame1
        btn-Add AT COLUMN 2 ROW 12 btn-Delete btn-Exit SKIP(1)
        Item-Browse AT ROW 2 COLUMN 2
         WITH SIDE-LABELS USE-TEXT CENTERED THREE-D
          ROW 2 TITLE "Database Access Form for the Item Table".

      /**********  DEFINE TRIGGERS  **********/
/*4*/  ON CHOOSE OF btn-Add DO:
        Method-Status = Item-Browse:INSERT-ROW("AFTER").
      END.
```

These notes explain the highlights of the first part of the programming example:

1. To use a browse with a query, include the SCROLLING option on the DEFINE QUERY statement.

2. This statement defines the browse widget, the fields it displays, and the number of rows it shows at one time. The ENABLE option defines this as an updatable browse where all fields except Item-Num can be changed.

3. These statements define the widgets and frames for the procedure.

4. This statement defines the trigger for button btn-Add.

Here is the second section of code for the new form:

lt-08-07b.p

```
          .
          .
          .
/*5*/  ON ROW-LEAVE OF BROWSE Item-Browse DO:
        IF Item-Browse:NEW-ROW IN FRAME Frame1 THEN DO:
         CREATE Item.
          ASSIGN Item-Num = NEXT-VALUE(next-item-num)
/*6*/          INPUT BROWSE Item-Browse Item-Name Price On-hand
               Cat-Description.
/*7*/     Method-Status = Item-Browse:CREATE-RESULT-LIST-ENTRY().
/*8*/      DISPLAY Item-Num WITH BROWSE Item-Browse.
       END.
    END.

    ON CHOOSE OF btn-Delete DO:
      GET CURRENT Item-Query EXCLUSIVE-LOCK NO-WAIT.
/*9*/    DELETE Item.
      Method-Status = Item-Browse:DELETE-SELECTED-ROWS().
    END.

   /**********  MAIN LOGIC  **********/
   OPEN QUERY Item-Query FOR EACH Item NO-LOCK.
   ENABLE ALL WITH FRAME Frame1.
   WAIT-FOR CHOOSE OF btn-Exit.
   CLOSE QUERY Item-Query.
```

These notes explain the highlights of the second half of the programming example.

1. This DO statement executes when changing focus from a row in browse Item-Browse. For new records, the IF...THEN statement updates the browse.

2. The INPUT BROWSE option of the ASSIGN statement writes the new information to the database.

3. The CREATE-RESULT-LIST-ENTRY( ) method updates the query results list to match the browse and database.

4. This statement displays the Item-Num assigned to a new record by the database.

5. This DELETE trigger on button btn-Delete deletes the selected rows.

Other Browse Techniques

The following list contains some of the techniques you can implement to make your browse more useable. See the chapter on using the browse widget in the *Progress Programming Handbook* for more information about how to use these features.

- Moving columns
- Locking columns
- Multiple row selection
- Calculated fields
- Deleting multiple rows
- Adding rows to an empty browse
- Repositioning the browse
- Overriding automatic database writes

8.11 Summary

Primarily, you work with three data locations:

- Databases
- Memory used to store records that the procedure is working with, called record buffers
- Memory used to store data that is visible on the screen, called screen buffers

Progress creates and maintains buffers for you, but you programmatically control the movement of data between buffers. Some rules about data and buffers:

- Procedures cannot interact with data until the data is copied to the record buffer from the database.
- End users cannot work with data until it is copied from the record buffer to the screen buffer.
- Changes made to one buffer do not automatically affect the other buffer.

To make an unambiguous reference to a database field, specify the database name, table name, and field name using this syntax: `database.table.field`

The following list summarizes the data handling statements:

- **FIND** — Copies data from the database to the record buffer.
- **GET** — Copies data from the database to the record buffer. You use GET with a defined query. Progress has three statements that control defined queries:
  - **DEFINE QUERY** — Specifies the tables included in the query.
  - **OPEN QUERY** — Uses the record phrase to define the records that are part of the query.
  - **CLOSE QUERY** — Closes a query.
- **DISPLAY** — Moves data from the record buffer to the screen buffer.
- **ENABLE** — Allows user input into the screen buffer.
- **ASSIGN** — Moves data from the screen buffer to the record buffer.
- **UPDATE** — Combines the functions of SET and DISPLAY.
- **RELEASE** — Prematurely releases a record from the record buffer to the RDBMS.

- **CREATE** — Adds a new record to the database and copies the empty record to the record buffer.
- **DELETE** — Marks a record for deletion and releases the record to the RDBMS for eventual removal from the database.
- **PROMPT-FOR** — Allows user input from the user to the screen buffer.
- **SET** — Combines the functions of PROMPT-FOR and ASSIGN.
- **INSERT** — Combines the functions of CREATE and UPDATE.

When working with records, your procedures should handle the basic error processing associated with not finding a record. Using the NO-ERROR option suppresses the Progress default error response. Use the AVAILABLE function to test the record buffer before attempting to use the buffer.

The PROMPT-FOR, SET, UPDATE, and INSERT statements all have implied blocking properties. These statements block execution until the user issues either a GO event or an ENDKEY event. Because of this behavior, these statements must never be mixed with WAIT-FOR statements.

To facilitate the GO and ENDKEY events more easily, you can associate buttons with the events using the AUTO-GO and AUTO-ENDKEY options in a DEFINE BUTTON statement.

The length of time a record stays in the record buffer is called the scope of the record. As a very general rule of thumb, a record's scope ends when a new record is requested or the block in which the buffer was created ends.

Alert boxes are extensions to the window widget that provide important information to users and disable the rest of the interface until the user responds to the alert box.

A browse widget is a specialized selection list for displaying data from database tables:

- Browse widgets work with defined database queries.
- The most common event function for the browse widget is VALUE-CHANGED. VALUE-CHANGED executes every time you change the "row focus" of the browse widget from one row to another.

9

Selecting, Sorting, and Relating Records

This chapter describes the important options of the record phrase and how to use them in the context of an interactive database form. The options contribute to your ability to select a record or set of records, to sort records, and to relate records from different tables.

Specifically, the chapter covers:

- Searching for a record using a screen value (USING)
- Sorting records (BY)
- Sorting records with a specific index (USE-INDEX)
- Relating records (EACH, OF)
- Searching for records and relating records with expressions (WHERE)

For more information on the record phrase and database access techniques, see the *Progress Programming Handbook*.

9.1 Uses of the Record Phrase

This chapter focuses on the record phrase. The record phrase is the language structure that supports most of your options for selecting, sorting and relating records. Before focusing on the record phrase itself, take a moment to review the different contexts which employ the record phrase:

- The FIND statement locates an individual record. The record phrase options help specify the record.
- The FOR EACH block is most valuable in batch applications where records are processed linearly. The options of the record phrase help select which records are executed with the statements inside the block. You can also sort and relate records with the FOR EACH block.
- The OPEN QUERY statement establishes a results list of records. Record phrase options help select, sort, and relate the records to form the results list.

To continue the focus on interactive database applications, this chapter teaches record phrase options in the context of a defined query. Notes along the way point out any significant differences in the use of an option with FIND or FOR EACH.

9.2 Selecting Records

Record selection is the process of distinguishing which records are part of the set you want to work with and which are not. In a defined query, the results list represents the set of records you want to work with. With the results list established, you can ignore all the other records in the table. For example, you might want to select just those customers who have a balance due, or those items that are on order. This section describes the two main options for selecting records.

As a refresher, this is the basic syntax for the OPEN QUERY statement.

SYNTAX

```
OPEN QUERY query-name FOR EACH record-phrase
```

This syntax diagram shows a partial syntax for the record phrase, listing the selection options.

SYNTAX

```
record-name
  [ WHERE expression ]
  [ USING [ FRAME frame ] field
    [ AND [ FRAME frame ] field ] ...
  ]
```

Table 9–1 introduces the record phrase selection options.

Table 9–1: Record Selection Options

Component	Description
record-name	The table name.
WHERE	Specifies an expression to use as record-selection criteria. For example, WHERE Cust-Num < 11 finds all the customers with a customer number of less than 11. You saw many examples of WHERE expressions in Chapters 5 and 6.
USING	Specifies a screen value to use as the selection criteria. For example, you can have the user enter a region name, and then use that data to create a list of all the customers from that region.

9.2.1 Selecting with a WHERE Expression

Any valid Progress expression can be used with WHERE to select records. The code fragment below shows an example:

```
OPEN QUERY My-Query FOR EACH Customer WHERE Customer.Country = "USA".
```

In addition to the expressions you learned about in Chapter 5, "Working with Expressions," WHERE supports operators that make working with CHARACTER data easier:

- BEGINS
- =
- MATCHES
- CONTAINS

This example of BEGINS finds any record where the field reference begins with the same letters as specified in the option:

```
OPEN QUERY My-Query FOR EACH Customer WHERE Customer.Name BEGINS "Smith".
```

The output from this selection criteria includes Smith, Smithers, Smithy, and so on.

This example of MATCHES finds any record where there is an exact match between the supplied string and the database field reference:

```
OPEN QUERY My-Query FOR EACH Customer WHERE Customer.Name MATCHES "Tim".
```

This example finds Tim. However, MATCHES is very slow compared to using BEGINS. In most cases, substituting the = operator is far more efficient than using MATCHES.

You can use two wildcards with the MATCHES options, and this is the only use where MATCHES has an advantage over the other comparison operators. Use asterisk (*) to take the place of many characters and period (.) to take the place of a single character as shown in the following example:

```
OPEN QUERY My-Query FOR EACH Customer WHERE Customer.Name MATCHES "T.m".
```

This example finds Tam, Tim, and Tom. Remember, do not use MATCHES unless there is no other way to write the expression with another operator. The most common use of MATCHES is to search for an expression that begins with a wildcard. For example "*berry". This example would find Strawberry, Blueberry, Raspberry, and so on in a list of fruits.

NOTE: You can use BEGINS and MATCHES in any CHARACTER expression.

When a field is defined as a word index in the database schema, you can use the CONTAINS option. CONTAINS checks whether the supplied string is contained anywhere in the field reference. Review the following code example that presents the CONTAINS option:

```
OPEN QUERY My-Query FOR EACH Customer WHERE Customer.Comments CONTAINS
"volleyball".
```

If any of the comment fields contains the specified string, it is selected.

9.2.2 Selecting with the USING Option

The USING option allows you to match a screen value for a database field with records that have that value. In this example, the new query will build a results list of Customer records that have the same postal code as the current screen value of Postal-Code:

```
OPEN QUERY My-Query FOR EACH Customer USING Postal-Code.
```

In this example, the new query will build a results list of Customer records that have the same postal code and the same country name as the current screen value of Postal-Code and Country:

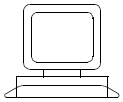
```
OPEN QUERY My-Query FOR EACH Customer USING Country AND Postal-Code.
```

The examples assume that a user has entered a value for Postal-Code and Country. Progress attempts to match these values to database records.

You can code similar functionality by prompting the user to enter a value into a variable, then using that variable in a WHERE clause, as shown below:

```
OPEN QUERY My-Query FOR EACH Customer WHERE Cust-Num = My-Variable.
```

9.2.3 Query and Selection Programming Example

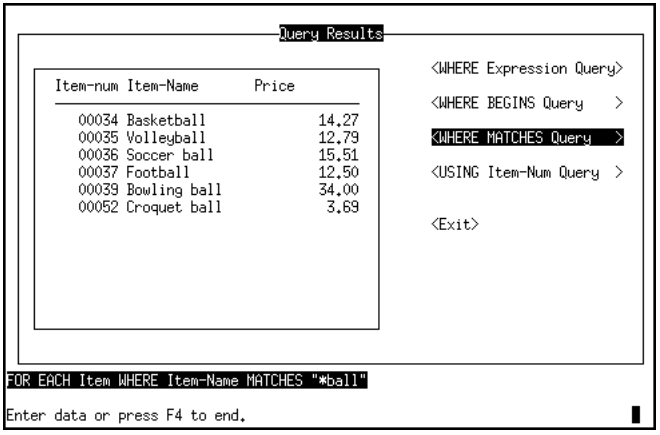


Exercise

This example demonstrates selecting records with a defined query:

1 ♦ Open `lt-09-01.p` and run it.

2 ♦ Choose the first Query button. The following browse widget appears with the results of the query. In the status area, notice the message that shows the selection syntax used in the particular OPEN QUERY statement:



3 ♦ Choose the second and third Query buttons.

4 ♦ Choose the fourth Query button. This example uses the USING option, so you have to provide a value to search for. A dialog box appears to prompt you for the Item-Num.

5 ♦ Type an integer from 1 to 55, then choose OK. The browse widget displays the results.

6 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

Here is the code that created the display:

lt-09-01.p

```
      /**********  DEFINE QUERIES  **********/
/*1*/  DEFINE QUERY New-Query FOR Item
          FIELDS (Item-Num Item-Name price).
      /**********  DEFINE WIDGETS  **********/
/*2*/  DEFINE BROWSE New-Browse QUERY New-Query
      DISPLAY Item-Num Item-Name Price WITH 22 DOWN.
      DEFINE BUTTON btn-Q1 LABEL "WHERE Expression Query".
      DEFINE BUTTON btn-Q2 LABEL "WHERE BEGINS Query".
      DEFINE BUTTON btn-Q3 LABEL "WHERE MATCHES Query".
      DEFINE BUTTON btn-Q4 LABEL "USING Item-Num Query".
      DEFINE BUTTON btn-OK LABEL "OK" AUTO-GO.
      DEFINE BUTTON btn-Cancel LABEL "Cancel" AUTO-ENDKEY.
      DEFINE BUTTON btn-Exit LABEL "Exit".
      /**********  DEFINE FRAMES  **********/
      DEFINE FRAME Frame1
        SKIP(1) btn-Q1 COLON 48 SPACE(2) SKIP(1) btn-Q2 COLON 48 SKIP(1)
        btn-Q3 COLON 48 SKIP(1) btn-Q4 COLON 48 SKIP(2) btn-Exit COLON 48
        New-Browse AT ROW 2 COLUMN 2 SKIP(1)
          WITH TITLE "Query Results" AT ROW 2 COLUMN 2.
    /**********  DEFINE TRIGGERS  **********/
      ON CHOOSE OF btn-Q1
       DO:
/*3*/       OPEN QUERY New-Query FOR EACH Item WHERE Item-Num > 45.
            DISPLAY New-Browse WITH FRAME Frame1.
            MESSAGE "FOR EACH Item WHERE Item-Num > 45".END.
      ON CHOOSE OF btn-Q2
       DO:
/*4*/       OPEN QUERY New-Query FOR EACH Item
              WHERE Item-Name BEGINS "ski".
            DISPLAY New-Browse WITH FRAME Frame1.
            MESSAGE "FOR EACH Item WHERE Item-Name BEGINS ""ski""".END.
      ON CHOOSE OF btn-Q3
       DO:
/*5*/  OPEN QUERY New-Query FOR EACH Item WHERE Item-Name MATCHES "*ball".
         DISPLAY New-Browse WITH FRAME Frame1.
        MESSAGE "FOR EACH Item WHERE Item-Name MATCHES ""*ball""".END.
      ON CHOOSE OF btn-Q4
       DO:
         PROMPT-FOR Item.Item-Num SKIP btn-OK btn-Cancel
              WITH FRAME Frame3 VIEW-AS DIALOG-BOX TITLE "New Query".
/*6*/       OPEN QUERY New-Query FOR EACH Item USING Item-Num.
          DISPLAY New-Browse WITH FRAME Frame1.
          MESSAGE "FOR EACH Item USING Item-Num". END.
      /**********  MAIN LOGIC  **********/
      ENABLE ALL WITH FRAME Frame1.
      WAIT-FOR CHOOSE OF btn-Exit.
```

NOTE: The THREE-D option is relevant only on a Windows client; it is ignored by a character client.

These notes help explain the code, describing how to query and use the browse widgets:

1. Even though the procedure makes several queries, each query uses the same table (Item) and only one query is active at a time. Therefore, you need only one defined query.

2. Similarly, although the procedure uses the browse widget to display several queries, only one is displayed at a time.

3. The first query uses a WHERE expression to display all the inventory items with item numbers greater than 45.

4. The second query uses a WHERE BEGINS query to list all the items with names that begin with "ski".

5. The third query uses a WHERE MATCHES query to see if any of the item names end with "ball".

6. The fourth query uses the USING option to allow the user to define the query. The USING item accepts a screen value for a table field, then finds the records that have that value.

9.3 Sorting Records

This is the syntax for using the OPEN QUERY statement's sorting options.

SYNTAX

```
OPEN QUERY query FOR EACH record-phrase
  [ BY expression [ DESCENDING ] ] ...
  record-name [ USE-INDEX index ]
```

Table 9–2 describes the sorting options.

Table 9–2: Record Sorting Options

Component	Description
BY *expression*	Specifies an expression to use as the basis for sorting the selected records. The FIND statement does not support BY.
ASCENDING DESCENDING	Ascending sorts begin with the lowest value and go to the highest value (A, B, C, D). Descending sorts start with the highest value and go toward the lowest value (Z, X, Y, W). By default, sorting is always ascending, so you don't need to specify the ASCENDING keyword. To use descending order, you must specify the DESCENDING keyword in the BY phrase.
USE-INDEX	Forces Progress to sort the selected records using a defined index.

9.3.1 Sorting Records with the BY Phrase

You use the BY option to sort by a field that is not indexed. The other sort options work efficiently with indexed fields, but work less efficiently with non-indexed fields. BY can work with either, which makes it the most useful sorting option.

There are three basic techniques to sorting with BY:

- BY *field*
- BY *field* BY *field* BY *field* . . .
- BY *expression*

The simplest technique is to supply a database field name as this code example shows:

```
OPEN QUERY My-Query FOR EACH Customer WHERE Country = "USA" BY Postal-Code.
```

The first example selects all United States-based companies and sorts the records by postal code, which is the ZIP code. This code could be used with an envelope-printing procedure to meet the sorting requirements of the U. S. Postal Service.

You can use multiple BY phrases in the same query as this example shows:

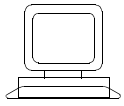
```
FOR EACH Customer BY Country BY Postal-Code
```

The first BY option is the primary sort order, and the second is the secondary sort order, and so on. In other words, this code first sorts the customers by their base country. Records within each country group are then sorted by postal code. With just a little more code, you've internationalized your envelope application.

Using Expressions with BY

The BY option can accept more than just database fields. You can also create expressions based on fields and sort by the results. See Chapter 5, "Working with Expressions," for more information on valid expressions.

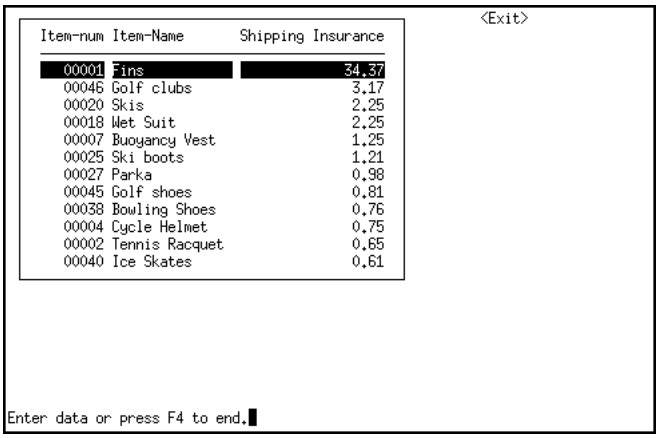
For example, suppose you need to code a browse widget to show the insurance charge for shipping a particular item. Insurance costs either $1 or 1% of the list price, whichever is higher. You can calculate the insurance based on the Price database field. If you sort the records in the browse descending by the insurance charge, then the sales representatives can quickly see which items have an insurance charge of more than $1.



Exercise

Follow these steps to view the sorting strategy previously discussed:

1 ♦ Open `lt-09-02.p` and run it. The following display appears:



As you can see, for each record, the procedure calculated the cost of shipping insurance. Since this is a descending list, you can quickly pick out which items cost more than $1 to insure.

2 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

Here is the code that created the display:

lt-09-02.p

```
      /**********  DEFINE QUERIES  **********/
      DEFINE QUERY New-Query FOR Item FIELDS (Item-Num Item-Name Price).

      /**********  DEFINE WIDGETS  **********/
      DEFINE BROWSE New-Browse QUERY New-Query
/*1*/      DISPLAY Item-Num Item-Name (Price / 100)
                LABEL "Shipping Insurance" WITH 12 DOWN.
      DEFINE BUTTON btn-Exit LABEL "Exit".

      /**********  DEFINE FRAMES  **********/
      DEFINE FRAME Frame1
          New-Browse SKIP(1) btn-Exit WITH NO-BOX THREE-D.

      /**********  MAIN LOGIC  **********/
/*2*/  OPEN QUERY New-Query FOR EACH Item BY (PRICE / 100) DESCENDING.
      DISPLAY New-Browse WITH FRAME Frame1.
      ENABLE ALL WITH FRAME Frame1.
      WAIT-FOR CHOOSE OF btn-Exit.
```

NOTE: The THREE-D option is relevant only on a Windows client; it is ignored by a character client.

These notes help explain the code:

1. In the widget definition statement, you include space for a *calculated field* (Price / 100), which is a temporary new field based on the value of other fields.

2. In the OPEN QUERY statement, you use the expression that corresponds to the calculated field in order to sort the calculated field. The descending option places those products with higher insurance charges at the top of the list.

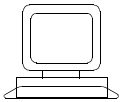
9.3.2 Sorting Records with the USE-INDEX Option

The USE-INDEX option of the record phrase allows you to choose the defined index by which you want to access the selected records. Usually, it's more efficient to use BY and let Progress pick the indexes for you. Since FIND does not support BY, USE-INDEX comes in handy with this statement. Review the following code example that presents the USE-INDEX option:

```
OPEN QUERY My-Query FOR EACH Item WHERE Price > 50 USE-INDEX Item-Name
```

In this example, the records for all items over $50 are sorted by product name.

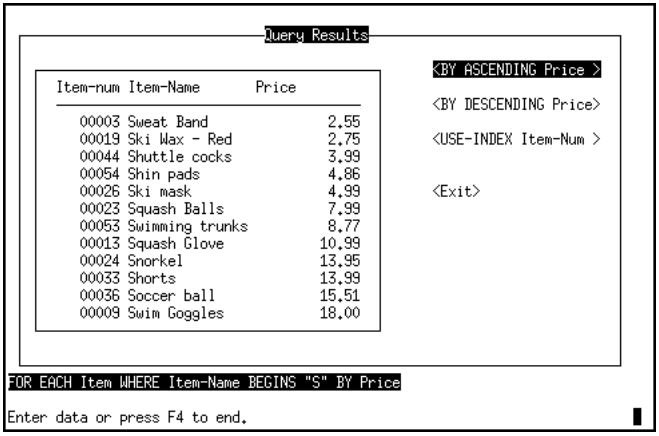
9.3.3 Querying and Sorting Example



Exercise

Follow these steps for a demonstration of combining sorting options with selection options:

1 ♦ Open `lt-09-03.p` and run it. The following interface appears:



2 ♦ Choose the BY ASCENDING Price button. A browse widget appears with the results of the query. In the status area, notice the message that shows the selection and sorting syntax used in the OPEN QUERY statement.

3 ♦ Choose the next two buttons.

4 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

Here is the code that created the display:

lt-09-03.p

```
    /**********  DEFINE QUERIES  **********/
    DEFINE QUERY New-Query FOR Item FIELDS (Item-Num Item-Name Price).

    /**********  DEFINE WIDGETS  **********/
    DEFINE BROWSE New-Browse QUERY New-Query
        DISPLAY Item-Num Item-Name Price WITH 22 DOWN.
    DEFINE BUTTON btn-Q1 LABEL "BY ASCENDING Price ".
    DEFINE BUTTON btn-Q2 LABEL "BY DESCENDING Price".
    DEFINE BUTTON btn-Q3 LABEL "USE-INDEX Item-Num ".
    DEFINE BUTTON btn-Exit LABEL "Exit".

    /**********  DEFINE FRAMES  **********/
    DEFINE FRAME Frame1
      SKIP(1) btn-Q1 COLON 50 SPACE(2) SKIP(1) btn-Q2 SKIP(1)
      btn-Q3 SKIP(2) btn-Exit
      New-Browse AT ROW 2 COLUMN 2
          WITH TITLE "Query Results" AT ROW 2 COLUMN 2 THREE-D.

    /**********  DEFINE TRIGGERS  **********/
    ON CHOOSE OF btn-Q1
    DO:
/*1*/      OPEN QUERY New-Query FOR EACH Item WHERE Item-Name BEGINS "S"
             BY Price.
          DISPLAY New-Browse WITH FRAME Frame1.
          MESSAGE "FOR EACH Item WHERE Item-Name BEGINS ""S"" BY Price".
    END.
    ON CHOOSE OF btn-Q2
    DO:
/*2*/      OPEN QUERY New-Query FOR EACH Item WHERE Item-Name BEGINS "S"
             BY Price DESCENDING.
          DISPLAY New-Browse WITH FRAME Frame1.
           MESSAGE
       "FOR EACH Item WHERE Item-Name BEGINS ""S"" BY Price DESCENDING".
    END.
    ON CHOOSE OF btn-Q3
    DO:
/*3*/     OPEN QUERY New-Query FOR EACH Item WHERE Item-Name BEGINS "S"
            USE-INDEX Item-Num.
          DISPLAY New-Browse WITH FRAME Frame1.
           MESSAGE
       "FOR EACH Item WHERE Item-Name BEGINS ""S"" USE-INDEX Item-Num".
    END.
    /**********  MAIN LOGIC  **********/
    ENABLE ALL WITH FRAME Frame1.
    WAIT-FOR CHOOSE OF btn-Exit.
```

These notes describe querying and sorting to help explain the code:

1. The BY phrase sorts the selected records by a non-indexed field, Price. The sort is ascending by default.

2. This BY phrase also sorts the selected records by Price, but the DESCENDING option forces a descending sort.

3. The USE-INDEX option makes clear to the query which of the defined indexes to use for the sort order. Again, this sort is ascending by default.

Practice Problems

Problem 9-1:

The code for the first example, `lt-09-01.p`, provides a nice shell for creating your own queries. Make a copy of the procedure and use it to experiment with querying. Use the different selection and sorting techniques. Also try using a different database table.

9.4 Relating Records

Imagine that a sales manager at All Around Sports discovers that a shipment of a best-selling product is a month late from the manufacturer and may be delayed for another three weeks. The sales manager wants to write to the affected customers about the delay. All Around Sports asks you to come up with a quick computer solution. You decide to write a procedure to generate a letter for each customer. However, the information you need for this project is in different tables. You need to:

- Find each Order-Line record that references the delayed product.
- Find the Order record for each matching Order-Line.
- Find the Customer record of each matching Order.

In a paper system, finding **related** data in different tables would be a tedious manual task. In some databases, you'd also have a hard time accomplishing this task. You need to use a relational database to put together data from related files, and you've got one with Progress. This process is also called *joining* tables or creating a *join*. For more information on joins, see the section on inner and outer joins later in this chapter.

9.4.1 Understanding Table Relationships

The most powerful advantage an electronic filing system has over a paper filing system is the ability to process data based on the relationships of the database tables to each other. When a database uses table relationships in this way, it is called a *relational database*.

Database table relationships are called relations. Two records from different database tables are said to have a *relation* if each contains a field with identical data, as shown in Figure 9–1.

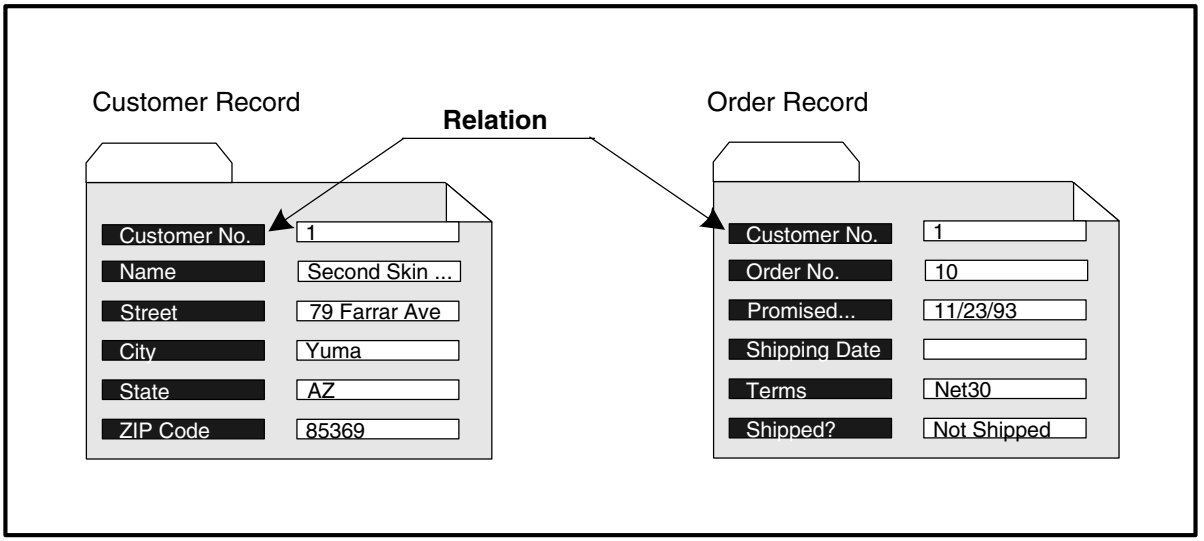


Figure 9–1: Table Relations

Two fields in two different tables contain the same data. Although the field names in Figure 9–1 are identical, they don't have to be to form a relation. Relations are the result of good database design. When you read the *Progress Database Design Guide*, you'll find a lot of information about normalization through the development of relations. *Normalization* is the database design principle that seeks to eliminate redundant data. In Figure 9–1, notice that the Order record does not contain the name or address of the customer. It doesn't need to because that information is already stored in the Customer record. However, you do need one field to associate this particular order with a particular customer. The Customer Number field serves this purpose. Now you've minimized the redundant data and established a common link between the two tables.

It's important to understand that normalizing your database in this manner does not create relationships. Relationships are created programmatically. However, good database design sets the preconditions for effective relationships.

Table records can have several types of relations, but for the most part, programmers work with two: one-to-one relationships and one-to-many relationships. Relationship types describe the way a particular record in one table can relate to one or more records in another table.

The first relationship type is called a *one-to-one relationship* and occurs when one record can relate to only one instance of a record in another table. For example, each customer order contains one or more order lines. An order line is a request for a quantity of a particular item. Therefore for each order line, there is only one inventory item, as shown in Figure 9–2.



Figure 9–2: One-to-one Relationship

When one record can relate to many records in another table, it is called a *one-to-many relationship*. For example, each customer can have several orders. In other words, each customer number can occur only once in the Customer table, but can occur many times in the Order table. This relationship is shown in Figure 9–3.

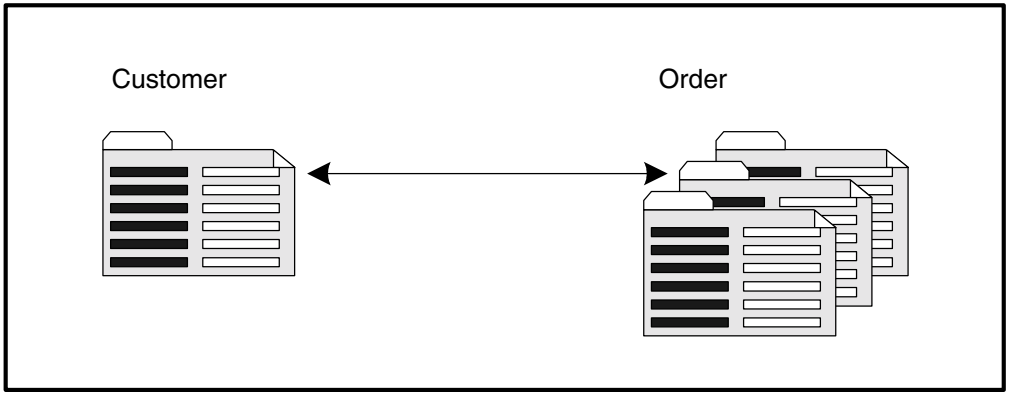


Figure 9–3: One-to-many Relationship

You may also suspect that there is a *many-to-one relationship*. However, a many-to-one relationship is just an inverted one-to-many relationship. In other words, instead of saying one customer can have many orders, you can say many orders relate to one customer. Both statements define a one-to-many relationship.

9.4.2 Relating Record Options

Your tools for creating relationships among tables are the same statements that you've been working with throughout this chapter. Instead of working with a single table, you can use the same statements to work with more than one. This section introduces the new options that help with this task.

This is the relevant FOR EACH syntax.

SYNTAX

```
FOR EACH record-phrase [ , EACH record-phrase ] ...
```

This is the relevant OPEN QUERY syntax.

SYNTAX

```
OPEN QUERY query FOR EACH record-phrase
  [ , EACH record-phrase ] ...
```

This is the relevant record phrase syntax.

SYNTAX

```
record [ OF table ] [ WHERE expression ]
```

Table 9–3 describes the syntax options.

Table 9–3: Record Relation Options

Component	Description

OF *table*	Allows you to relate two tables. You can use OF only when the two tables share an identical field and that field is indexed in at least one of the tables.
	`FOR EACH Customer:`
	`  FOR EACH Order OF Customer`
	OF is considered a shortcut syntax for an explicit WHERE expression when OF is used to express this type of relationship:
	`WHERE order.cust.num = customer.cust.num`

WHERE *expression*	Relates two tables in the same way OF does. With WHERE, however, the fields do not need to be indexed and they don't have to have the same name.
	`FOR EACH Customer:`
	`  FOR EACH Order WHERE Order.Cust-Num = Customer.Cust-Num`

EACH *record-phrase*	Allows you to build compound FOR EACH and OPEN QUERY statements. For example, you can simplify this code:
	`FOR EACH Customer:`
	`  FOR EACH Order OF Customer.`
	`by using the EACH phrase`
	`FOR EACH Customer, EACH Order OF Customer:`

9.4.3 Relating with OF

Relating records in a query means that you are building a meaningful data set with data from more than one table. The examples for the rest of this chapter display the data sets you build in browse widgets, but the task of combining data for reports is also a very important one. Chapter 10, "Creating Reports," discusses this important topic.

One of the sales reps asks you to write a browse procedure that shows all the outstanding orders for each customer. That way, when a customer calls about an order, the customer doesn't necessarily have to have the order number.

The name of each customer is in the Customer table in the Name field. The order number is in the Order table in the Order-Num field. To put the data set together, you need to search for and compare records from two tables in one process. You can't make the search and comparison unless there is a key between the two tables. A *key* is a field that holds common data.

Both the tables have a Cust-Num field, so a key does exist. If the key has the same name in both tables and is indexed in at least one, you can use the easiest method for relating the records, the OF option.

NOTE: For the OF keyword to properly detect a relationship between two tables, only one such relationship is allowed. If there are two or more fields in two tables with the same name and at least one of each is indexed, a relationship will not be detected.

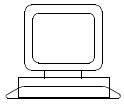
The OPEN QUERY statements you've seen so far search through one table. You need to search through two. So, it makes sense that you need two search phrases as the following code example shows:

```
OPEN QUERY My-Query FOR EACH Customer, EACH Order
```

This is where you use the EACH phrase. Each EACH searches a different table. But, this syntax still doesn't **relate** the tables. You have to use OF as the following code example shows:

```
OPEN QUERY My-Query FOR EACH Customer, EACH Order OF Customer
```

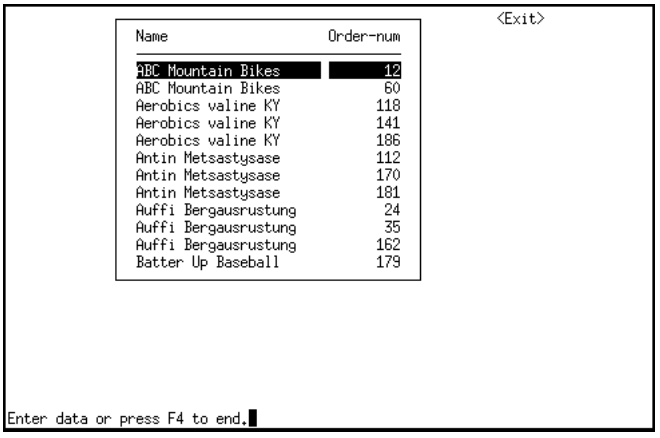
OF relates the second EACH back to the first EACH. The result is that the first Customer record is found. Next, Progress finds every Order record that has the same Cust-Num as the first Customer record.



Exercise

Follow these steps for a demonstration that implements the above scenario:

1 ♦ Open `lt-09-04.p` and run it. The following display appears:



2 ♦ Scroll through the records. The browse lists only Customers with current orders, and lists every order number for customers with multiple orders.

3 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

Here is the code that created the display:

lt-09-04.p

```
      /**********  DEFINE QUERIES  **********/
/*1*/  DEFINE QUERY New-Query FOR Customer FIELDS (Name Cust-Num),
                    Order FIELDS (Order-Num Cust-Num).

      /**********  DEFINE WIDGETS  **********/
      DEFINE BROWSE New-Browse QUERY New-Query
          DISPLAY Name SPACE(3) Order-Num WITH 15 DOWN.
      DEFINE BUTTON btn-Exit LABEL "Exit".

      /**********  DEFINE FRAMES  **********/
      DEFINE FRAME Frame1
          New-Browse AT ROW 1 COLUMN 2
          btn-Exit AT ROW 5 COLUMN 50
              WITH NO-BOX CENTERED THREE-D.

      /**********  MAIN LOGIC  **********/
/*2*/  OPEN QUERY New-Query FOR EACH Customer,
                           EACH Order OF Customer BY Name.
      DISPLAY New-Browse WITH FRAME Frame1.
      ENABLE ALL WITH FRAME Frame1.
      WAIT-FOR CHOOSE OF btn-Exit.
```

These notes help to explain the code:

1. Since you need to work with more than one table, you have to list the tables in the DEFINE QUERY statement.

2. This code searches for and does an implicit comparison on the Cust-Num key field.

9.4.4 Relating with WHERE

You can also relate tables with WHERE. There are two scenarios where you must use WHERE:

- Your key field is not indexed in either table.
- Your key field has a different name in each table.

In both of these cases, you may want to reconsider your design. Consider adding an index or changing the name of the field.

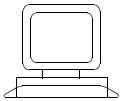
You can also use WHERE to do any relation that OF can do. The following example uses OF:

```
OPEN QUERY My-Query FOR EACH Customer, EACH Order OF Customer
```

This equivalent example uses WHERE:

```
OPEN QUERY My-Query FOR EACH Customer,
            EACH Order WHERE Order.Cust-Num = Customer.Cust-Num.
```

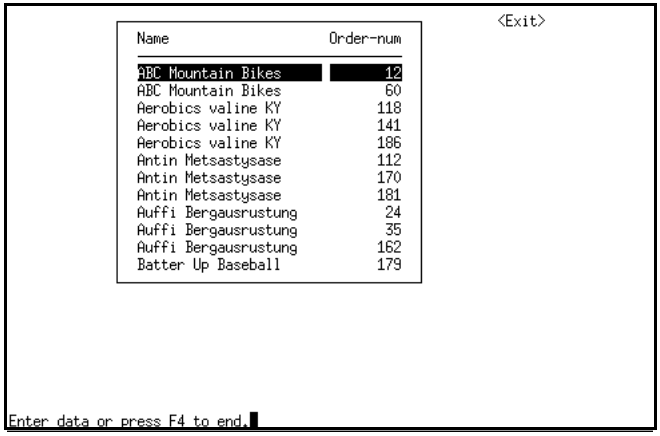
OF implicitly compares the keys, while WHERE explicitly compares the keys. At compile time, Progress turns an OF into a WHERE expression. For that reason, you should consider OF a shortcut and use WHERE most of the time.



Exercise

Follow these steps for a demonstration of the last exercise reworked to use WHERE:

1 ♦ Open `lt-09-05.p` and run it. The same display as in the last exercise appears:



2 ♦ Scroll through the records. As you can see, the same data is present.

3 ♦ Choose Exit, then press **SPACEBAR** to return to the Procedure Editor.

Here is the code that created the display:

lt-09-05.p

```
      /**********  DEFINE QUERIES  **********/
      DEFINE QUERY New-Query FOR Customer FIELDS (Name Cust-Num),
                    Order FIELDS (Order-Num Cust-Num).

      /**********  DEFINE WIDGETS  **********/
      DEFINE BROWSE New-Browse QUERY New-Query
          DISPLAY Name SPACE(3) Order-Num WITH 12 DOWN.
      DEFINE BUTTON btn-Exit LABEL "Exit".

      /**********  DEFINE FRAMES  **********/
      DEFINE FRAME Frame1
          New-Browse AT ROW 1 COLUMN 2
          btn-Exit   AT ROW 1 COLUMN 50
              WITH NO-BOX CENTERED THREE-D.

      /**********  MAIN LOGIC  **********/
/*1*/  OPEN QUERY New-Query FOR EACH Customer, EACH Order
                         WHERE Order.Cust-Num = Customer.Cust-Num BY Name.
      DISPLAY New-Browse WITH FRAME Frame1.
      ENABLE ALL WITH FRAME Frame1.
      WAIT-FOR CHOOSE OF btn-Exit.
```

As shown at point 1, you still need the additional EACH to search through the second table. The WHERE expression causes an explicit mathematical comparison between key values.

9.4.5 Inner and Outer Table Joins

You have learned that you can refer to multiple tables in a single statement using a join. A join is a binary operation that selects and combines the records from multiple tables so that each result in the results list contains a single record from each table. A join combines the records from one table with those from another table or a previous join.

When you specify a join, you also specify conditions that determine how one table relates to another. These are called *join conditions*. Join conditions control which records selected from one table join with the records in the other table.

Progress supports two types of joins:

- **Inner join** — Supported in all statements capable of reading multiple tables, including the FOR, DO, REPEAT, and OPEN QUERY statements. An *inner join* returns the records selected for the table (or join) on the left side of the join with the related records selected from the table on the right side of the join. If a value in the left table does not have a corresponding value in the right table, Progress does not return that value or a result for the results list. This is an example of an inner join with the FOR statemen:.

```
FOR EACH Table1, EACH Table2 WHERE Field1 = Field3
```

  In this join, Table1 is on the left side of the join and Table2 is on the right, and Field1 = Field3 is a join condition. For any records not selected from Table2, the join returns no records from either the Table1 or Table2. Only records that are selected from both tables are returned for an inner join.

- **Left outer join** — Supported only in the OPEN QUERY statement. An *outer join* returns the same set of records selected for an inner join. However, outer joins also return all the records from the left table. When a value from the left table does not have a corresponding value in the right table, Progress returns the left-table value with an unknown value (?) from the right table. This is an example of a left outer join with the OPEN QUERY statement:

```
OPEN QUERY Query1 FOR EACH Table1, EACH Table2 OUTER-JOIN
  WHERE Field1 = Field3
```

  In this join, Table1 is on the left side and Table2 is on the right, and Field1 = Field3 is a join condition.

Figure 9–4 illustrates the results of an inner join. Note that the only values included from Table1 are the values that match the join conditions. Also, the value where Field1 = 2 appears twice in the join because there are two instances where Field3 = 2 in Table2. For any values in Field1 that do not match a value in Field3, the entire Table1 record is excluded from the join. Likewise, for any values in Field3 that do not match any value in Field1, the entire Table2 record is excluded from the join.

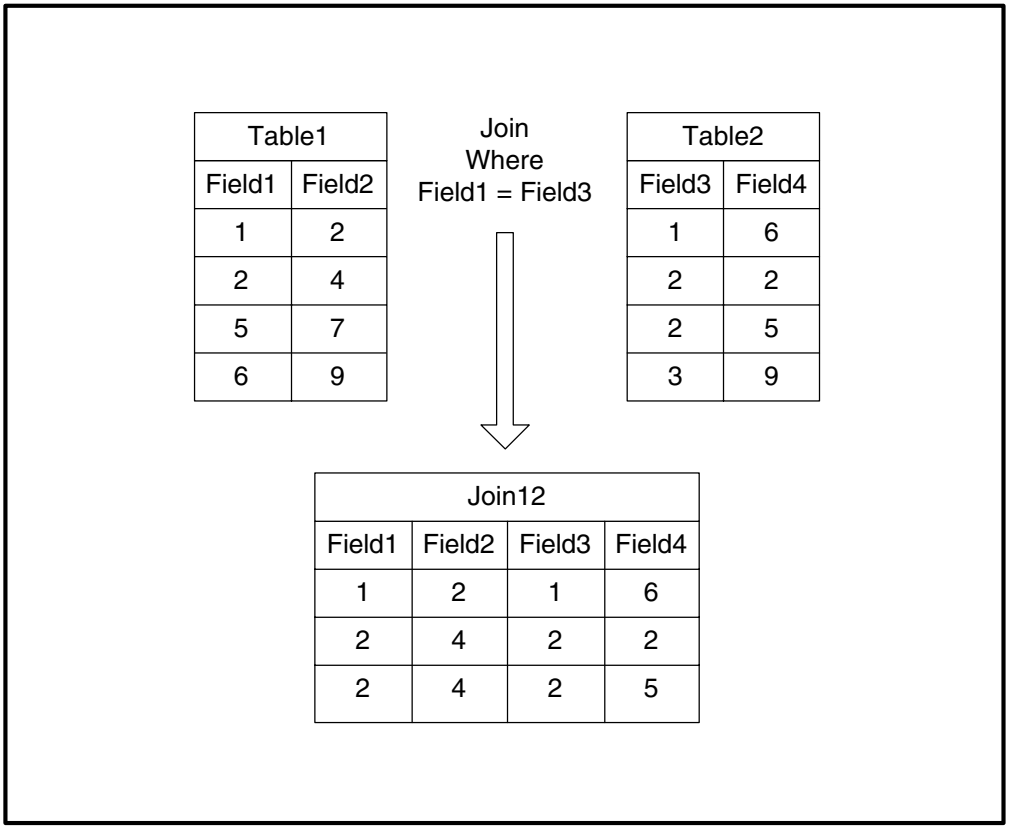


Figure 9–4: Inner Join

Figure 9–5 illustrates the results of a left outer join. Note that all values in Table1 are included in the join, even where they do not have a corresponding value in Table2. The unknown value is used in Field3 and Field4 where there is no value in Field3 corresponding to the value of Field1. Thus, the values of Field3 that do not match any values in Field1 are still excluded from the join.

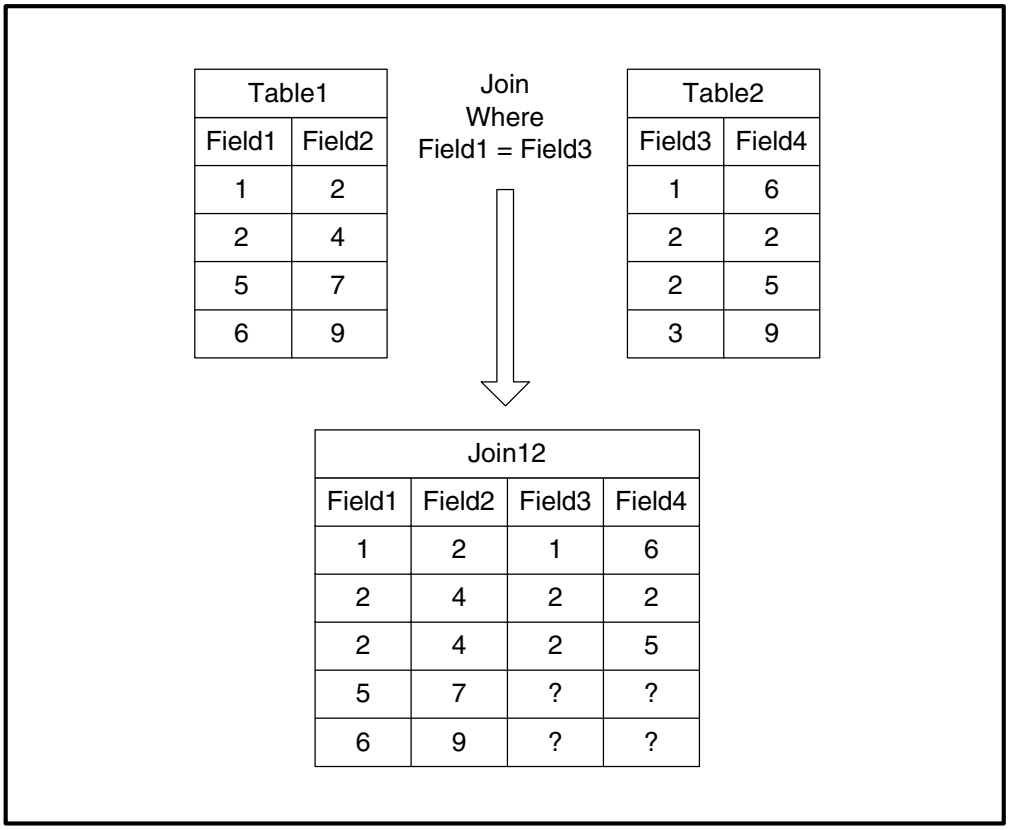


Figure 9–5: Left Outer Join

Practice Problems

Using a copy of procedure `lt-09-04.p` as a starting point, figure out the correct syntax for these queries:

Problem 9-2: `lt-09-s2.p`

Display the customer name and the sales representative for each customer. What kind of relationship is this?

Problem 9-3: `lt-09-s3.p`

For each customer, list all the product names currently on order. What type of relationship is this?

9.5 Summary

This chapter explained how to select, sort, and relate records from different tables.

Selection Options

These options allow you to specify which records to retrieve:

- The WHERE option uses any valid Progress expression to select records. WHERE supports options that allow you to specify how to select CHARACTER data:
  - BEGINS finds records where the field begins with a particular string.
  - MATCHES finds records where the field matches a string, but should only be used with wildcards, since the = operator is much more efficient for full string matches.
  - CONTAINS finds records where the field contains a string.
- The USING option selects records by one or more indexed fields.

Sorting Options

These options allow you to specify how to sort the records:

- The BY phrase allows you to sort by one or more fields (these don't have to be indexed) or by expressions based on fields.
- The USE-INDEX option sorts by an indexed field.

Relating Records

Database tables have a relationship if they have a field with identical data in common. There are two kinds of relationships between tables:

- A one-to-one relationship occurs when one record in a table relates to only one record in another table.
- A one-to-many relationship occurs when one record can relate to many records in another table.

The field with identical data that is held in common between two tables is called the key.

Three options allow you to establish relationships between database tables:

- The EACH phrase allows you to relate multiple tables.
- The OF option allows you to relate tables in a search by specifying the key field. The key must be an indexed field in at least one of the tables.
- The WHERE option relates tables in which the key is not an indexed field, or the key field doesn't have the same name in both tables.

10

Creating Reports

A database management system provides information that helps an organization run efficiently. A report—whether printed on paper, viewed on a workstation, or stored on media—is a common tool for distributing database information.

This chapter introduces you to the 4GL statements that you can use to generate reports, both simple and complex. Specifically, it discusses these topics:

- Report basics
- Designing an event-driven interface for viewing reports
- Converting widget values to textual data
- Generating sorted reports with aggregates and control breaks
- Generating reports from multiple tables
- Directing report output
- Designing frames for reports

10.1 Report Basics

The event-driven model provides a highly flexible environment for interactive applications. Reports collect and organize static data without interaction from the user, so traditional top-down programming techniques work better for this task. Typically, you can integrate a report-generating procedure into an event-driven application using a trigger with a RUN statement attached to a button or menu command (covered later).

The components of a basic report procedure include:

- A special type of frame, called a down frame, that can hold more than one iteration of data
- Frame phrase options to format the frame and included widgets
- Text widgets
- Format phrase options to format individual widgets
- An iterating control block
- An output statement

10.1.1 Down Frames

Like an interactive display, a frame is the data container for reports. However, unlike a display frame, a report usually contains more than one iteration of data. A frame that can contain more than one iteration of data is known as a *down frame*. (Sometimes a frame with a single iteration of data is called a *one-down frame*.)

Early on, you learned that a frame is a container for field-level widgets (data widgets, action widgets, graphic widgets). Actually, it's a bit more complicated than that. All the widgets included in a one-down frame are contained in a special widget called a *field-group widget*. The frame contains the field-group widget. When you display several iterations of data, the widgets of each iteration belong to a single field-group widget, and the frame contains the set of field group widgets.

The behind-the-scenes work of the field-group widget is rarely something that you need to track. The explanation above simply serves to explain the mechanics of a down frame in terms of the programming model.

The default frame of an iterating control block, like FOR EACH, is a down frame. You can also specify a down frame with the DOWN keyword of the frame phrase. DOWN used alone indicates that you want Progress to fit as many iterations in the frame as the screen can hold. Here is an example:

```
DEFINE FRAME Frame1
  sports.Customer.Name
    WITH DOWN CENTERED THREE-D.
```

Specifying an integer before DOWN indicates the maximum number of iterations the frame can hold. Here is an example:

```
DEFINE FRAME Frame1
  sports.Customer.Name
    WITH 12 DOWN CENTERED THREE-D.
```

10.1.2 Text Widgets

The text widget displays textual data without the decorations native to your windowing system. This means that data displays more compactly, so more iterations fit in a single display or printed page.

If you are working with fields or variables that use the fill-in field as the default data widget, then the USE-TEXT option of the frame phrase quickly converts the fill-in fields to text widgets. Here is an example:

```
DEFINE FRAME Frame1
  sports.Customer.Name
    WITH DOWN USE-TEXT CENTERED THREE-D.
```

If the default data widget is anything other than a fill-in field or text widget, you need to use the VIEW-AS TEXT option of the format phrase on each widget to convert the widget to a text widget. Here is an example:

```
DEFINE VARIABLE Bal-due AS LOGICAL LABEL "Balance Due?"
    VIEW-AS TOGGLE-BOX.

DEFINE FRAME Frame1
  sports.Customer.Name  /* default is a fill-in field */
  Bal-due VIEW-AS TEXT /* default is a toggle box */
    WITH DOWN USE-TEXT CENTERED THREE-D.
```

Remember that the default data representation for a field comes from the schema of the database, while the default for a variable comes from the DEFINE VARIABLE statement.

10.1.3 Control Blocks and Output Statements

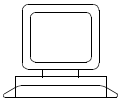
Compiling a report usually involves a straightforward process of moving through a set of table records, calculating data, and outputting results. A control block best handles the report-generating process. Most reports use the FOR EACH block because of its record-reading properties. Here is an example:

```
FOR EACH Customer FIELDS (Name) WITH FRAME Frame1:
  IF BALANCE > 0 THEN Bal-due = YES.
    ELSE Bal-due = NO.
  DISPLAY Name Bal-due.
END.
```

The default frame of an iterating control block is a down frame.

The DISPLAY statement is Progress's main programming statement for output. DISPLAY can output to printers and media as well as the screen. Later, you'll learn about another output statement and how to direct output.

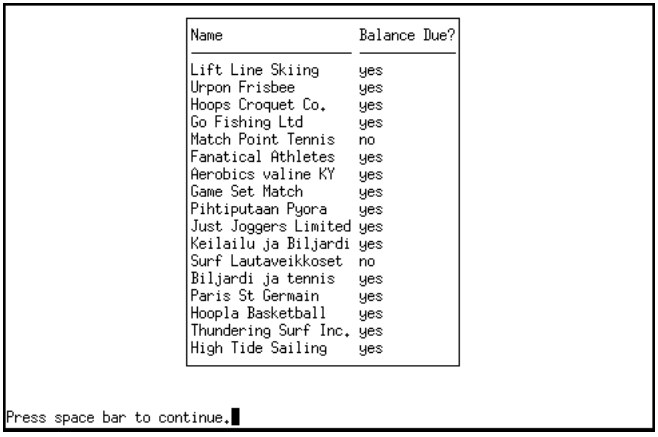
10.1.4 Basic Report Demonstration



Exercise

Follow the steps below for a demonstration of Progress behavior when displaying a report to your screen:

1 ♦ Open `lt-10-01.p` and run it. Progress creates a down frame that occupies all the vertical space it can and fills the frame with iterations of data, as shown below:



Name	Balance Due?
Lift Line Skiing	yes
Urpon Frisbee	yes
Hoops Croquet Co.	yes
Go Fishing Ltd	yes
Match Point Tennis	no
Fanatical Athletes	yes
Aerobics valine KY	yes
Game Set Match	yes
Pihtiputaan Pyora	yes
Just Joggers Limited	yes
Keilailu ja Biljardi	yes
Surf Lautaveikkoset	no
Biljardi ja tennis	yes
Paris St Germain	yes
Hoopla Basketball	yes
Thundering Surf Inc.	yes
High Tide Sailing	yes

Notice the message at the bottom of the screen. Progress pauses to allow you to view the first frame of data.

2 ♦ Press **SPACEBAR**. Progress clears the down frame and fills it with a new set of iterations.

3 ♦ Press **END-ERROR** and then **SPACEBAR** to return to the Procedure Editor.

Here is the code for this example:

lt-10-01.p

```
      DEFINE VARIABLE Bal-due AS LOGICAL LABEL "Balance Due?"
          VIEW-AS TOGGLE-BOX.

      DEFINE FRAME Frame1
        sports.Customer.Name
/*1*/      Bal-due VIEW-AS TEXT
/*2*/        WITH DOWN USE-TEXT CENTERED THREE-D.

/*3*/  FOR EACH Customer FIELDS (Name Balance) WITH FRAME Frame1:
        IF BALANCE > 0 THEN Bal-due = YES.
          ELSE Bal-due = NO.
/*4*/   DISPLAY Name Bal-due.
      END.
```

NOTE: The THREE-D option is relevant only on a Windows client; it is ignored by a character client.

The following notes help to explain the code:

1. The default data widget for this LOGICAL variable is a toggle box, making the VIEW-AS TEXT phrase necessary to convert it to a text widget.

2. The frame phrase of the DEFINE FRAME statement contains two key options: DOWN and USE-TEXT. The DOWN keyword specifies a down frame that automatically expands to fit as many iterations as the screen can hold. Specifying an integer before DOWN sets the maximum number of iterations the frame can hold. USE-TEXT converts all fill-in fields into text widgets for a compact display.

3. Since the great majority of reports compile and manipulate table data, the FOR EACH statement is the most common control block used for report procedures. The frame phrase here replaces the default down frame with the named down frame defined earlier.

4. The DISPLAY statement is the most commonly used output statement. As you'll see later, DISPLAY can output to terminals, printers, or files.

The type of report display created by procedure `lt-10-01.p` does not fit well with the event-driven model because the user:

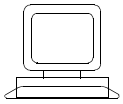
- Does not control the display of data
- Can view data in only one direction
- Does not have an intuitive way to dismiss the report

The next section describes a technique for viewing reports that better suits the event-driven model.

10.2 Designing an Interface for Viewing Report Data

In contrast to the Progress-controlled interface you saw in the last example, here's what you need to create a user-controlled report viewing interface:

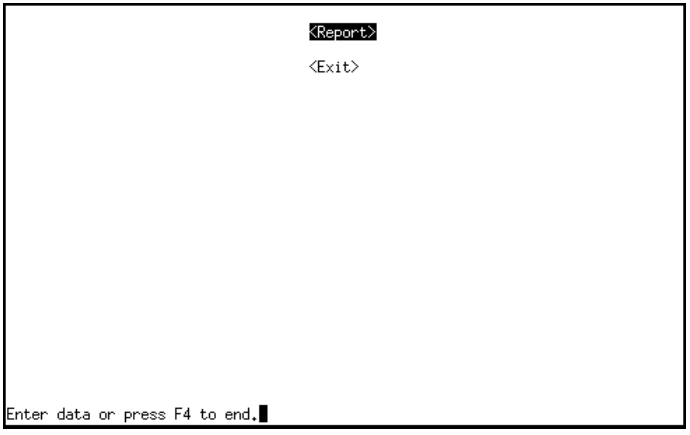
1. A dialog box to contain the output. Therefore, you won't need to leave room on your main display for the report data.
2. An OK button in the dialog box to let the user dismiss the report.
3. A temporary file to contain the report data. Having the control block output to a temporary file eliminates the pausing behavior.
4. An editor widget with scrollbars to read the file into. The scrollbars let the user navigate the report in two directions.



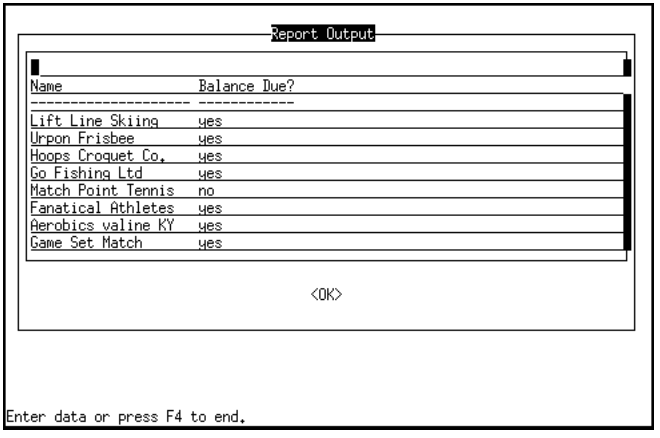
Exercise

The next example creates a dialog box interface for the same report you saw in the last example. The notes that follow the exercise explain the new code techniques you need to make this interface work.

1 ♦ Open `lt-10-02.p` and run it. The following screen displays:



2 ♦ Choose Report. The report dialog box appears as shown on the next page:



The report data appears in an editor that you can scroll through.

3 ♦ Choose OK to dismiss the dialog box.

4 ♦ Choose Exit and then press **SPACEBAR** to return to the Procedure Editor.

Here is the code for this example:

lt-10-02.p

```
      /**********  DEFINE WIDGETS  **********/
      DEFINE VARIABLE Bal-due AS LOGICAL LABEL "Balance Due?"
          VIEW-AS TOGGLE-BOX.
/*1*/  DEFINE VARIABLE Rep-Editor AS CHARACTER VIEW-AS EDITOR
          SCROLLBAR-VERTICAL SIZE 79 BY 16.
      DEFINE VARIABLE Stat AS LOGICAL.
      DEFINE BUTTON b-rep LABEL "Report".
      DEFINE BUTTON b-exit LABEL "Exit".
      DEFINE BUTTON b-ok LABEL "OK" AUTO-GO.
      /**********  DEFINE FRAMES  **********/
      DEFINE FRAME Frame1
          SKIP(1)
          b-rep SKIP(1)
          b-exit
              WITH NO-BOX CENTERED THREE-D.
      DEFINE FRAME Dialog1
          Rep-Editor SKIP(1)
          b-ok TO 40 SKIP(1)
              WITH NO-LABELS VIEW-AS DIALOG-BOX SCROLLABLE.
      /**********  DEFINE TRIGGERS  **********/
      ON CHOOSE of b-rep
      DO:
/*2*/      OUTPUT TO "tut-temp.txt".
/*3*/      FOR EACH Customer WITH STREAM-IO:
             IF Balance > 0 THEN Bal-due = YES.
             ELSE Bal-due = NO.
             DISPLAY Name Bal-due.
             END.
/*4*/      OUTPUT CLOSE.
/*5*/      ASSIGN Rep-Editor:READ-ONLY IN FRAME Dialog1 = YES
               Rep-Editor:SENSITIVE IN FRAME Dialog1 = YES
               FRAME Dialog1:TITLE = "Report Output"
               Stat = Rep-Editor:READ-FILE("tut-temp.txt") IN FRAME Dialog1.
/*6*/      IF Stat THEN DO:
              ENABLE Rep-Editor b-ok WITH FRAME Dialog1.
               WAIT-FOR GO OF FRAME Dialog1.
/*7*/          HIDE FRAME Dialog1.
           END.
      END.
      /**********  MAIN LOGIC  **********/
/*8*/  ASSIGN Rep-Editor:FONT = 3.
      ENABLE ALL WITH FRAME Frame1.
      WAIT-FOR CHOOSE OF b-exit.
```

These notes explain the new code techniques used in this example:

1. This editor is for the report data.

2. The OUTPUT TO statement is your tool for directing output. This example directs output to the named file. This chapter covers directing output more fully in a later section.

3. This FOR EACH uses the default down frame, and the STREAM-IO option reduces all widgets to textual data without decorations. Using the STREAM-IO option is a requirement when outputting to any destination other than the screen.

4. Once you direct output to a destination, you need to close the output stream to return output to the default destination, which is the screen.

5. This ASSIGN statement manipulates three editor attributes and one method to set up the dialog box. The READ-ONLY attribute prevents changes to the report content. The SENSITIVE attribute enables the editor, making the scrolling functions available. The TITLE attribute contains the dialog box title text. The READ-FILE method takes a valid filename and returns YES or NO to indicate if Progress successfully read the file contents into the editor.

6. You only want to bring up the dialog box if the logical variable stat equals YES, which means that Progress successfully read the file into the editor.

7. If you use a WAIT-FOR statement in a dialog box, you must use the HIDE statement to dismiss the dialog box.

8. On all platforms that Progress supports, the output font designated by 3 is always a monospaced font, which are frequently used in printed reports. Assigning this font makes the editor contents mimic a printed report.

10.3 Converting Widget Values to Report Data

When reporting, you want to output data, but not the widget that represents that data. Of the data widgets, you can directly output the data of these widgets to all devices: fill-in fields, text widgets, and editors. You can output these data widgets to the screen only: toggle-box, radio set, slider, combo box, and selection list.

In the last section, you learned about the USE-TEXT and VIEW-AS text options to convert other widgets to text widgets. In this section, you'll learn about two more important techniques for handling widgets:

- Using the STREAM-IO option of the frame phrase to convert all data widgets in a frame to text widgets
- Using the editor for formatting long text strings as blocks of text

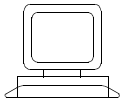
10.3.1 Printing Reports and the STREAM-IO Option

For toggle boxes, radio sets, sliders, combo boxes, and selection lists, trying to output the widget to any other device besides the screen yields nothing. Neither the widget nor the value it contains appears in files or printed reports.

Rather than tack a VIEW-AS TEXT phrase onto every widget, you can use the STREAM-IO option of the frame phrase to reduce every widget to textual data. Here is an example:

```
DEFINE VARIABLE Bal-due AS LOGICAL LABEL "Balance Due?"
  VIEW-AS TOGGLE-BOX.

DEFINE FRAME Frame1
  sports.Customer.Name
  Bal-due
    WITH DOWN STREAM-IO CENTERED THREE-D.
```

You must use the STREAM-IO option on every output frame intended for a destination other than the screen.



Exercise

Although primarily a tool for sending data to files and reports, the STREAM-IO option also reduces all widgets to textual data for screen output. This exercise demonstrates the effects of the STREAM-IO option on screen data. Perform the following steps:

1 ♦ Open `lt-10-03.p` and run it. This report is the same as the one from the last exercise, but without the special interface.

2 ♦ Press **END-ERROR** and then **SPACEBAR** to return to the Procedure Editor.

3 ♦ Remove STREAM-IO from the frame phrase and run the procedure again.

Note that the Bal-Due variable displays as a disabled toggle box in the iterations.

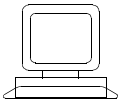
4 ♦ Press **END-ERROR** and then **SPACEBAR** to return to the Procedure Editor.

If you scroll through the text of the procedure, you can see that the STREAM-IO option on the frame phrase of the iterating block is the only change.

10.3.2 Formatting Long Text Strings

The remaining problem for simple output is how to handle long text strings. A text widget that contains a long string may cause formatting problems such as skipping to a new line or truncating. For screen design, you saw earlier that the editor widget is a good choice for holding long text strings. The editor is also the solution for reports. With the editor, you can output text in blocks.

For example, the Comments field in the Customer table has a format of x(60). Reserving 60 spaces takes up most of a line in a standard printed report. You can override this default behavior by specifying a VIEW-AS EDITOR phrase for the Comments field. When you output a field or variable value as an editor widget, Progress uses the INNER-CHARS attribute of the editor to format and wrap the text into a block. The INNER-LINES syntax sets the minimum number of lines the block occupies. If there is enough data to fill more lines than specified by INNER-LINES, then Progress provides the extra room.

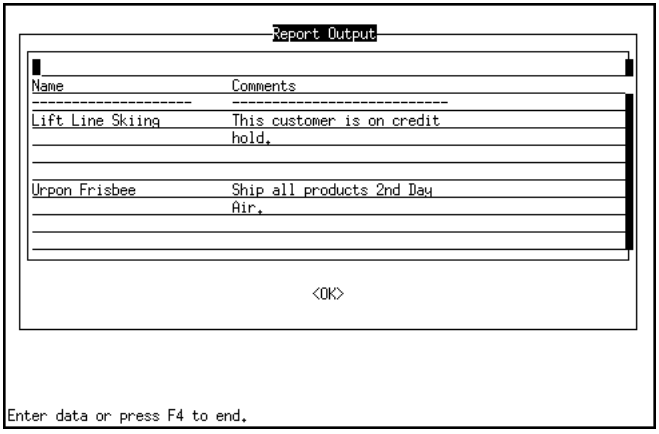


Exercise

The following exercise demonstrates editor output in reports:

1 ♦ Open `lt-10-04.p` and run it.

2 ♦ Choose Report. The report dialog box appears as shown in the following screen:



As you scroll through the editor, notice the blocks of Comments text.

3 ♦ Choose OK, then Exit, and then press **SPACEBAR** to return to the Procedure Editor.

Here is the code:

lt-10-04.p

```
    {lt-10-in.i} /* Common Interface Setup Code */

    /**********  DEFINE TRIGGERS  **********/
    ON CHOOSE of b-rep
    DO:
        OUTPUT TO "tut-temp.txt".
        FOR EACH Customer WITH STREAM-IO:
            DISPLAY Name SPACE(5) Comments VIEW-AS EDITOR
/*1*/            INNER-LINES 3 INNER-CHARS 20.
        END.
        OUTPUT CLOSE.

        ASSIGN Rep-Editor:READ-ONLY IN FRAME Dialog1 = YES
             Rep-Editor:SENSITIVE IN FRAME Dialog1 = YES
             FRAME Dialog1:TITLE = "Report Output"
             Stat = Rep-Editor:READ-FILE("tut-temp.txt") IN FRAME Dialog1.

        IF Stat THEN DO:
            ENABLE Rep-Editor b-ok WITH FRAME Dialog1.
            WAIT-FOR GO OF FRAME Dialog1.
            HIDE FRAME Dialog1.
        END.
    END.

    /**********  MAIN LOGIC  **********/
    ENABLE ALL WITH FRAME Frame1.
    WAIT-FOR CHOOSE OF b-exit.
```

INNER-LINES reserves at least three lines per iteration of the report. INNER-CHARS sets the length of the text block at 25 characters.

INNER-LINES reserves at least three lines per iteration of the report. INNER-CHARS sets the length of the text block at 20 characters.

This section concludes the basic discussion on creating a reporting procedure. The next few sections concentrate on the techniques for populating reports with more complex data.

10.4 Generating Reports with Control Breaks and Aggregates

In Chapter 9, "Selecting, Sorting, and Relating Records," you learned to sort data using the BY option of the record phrase. For example, this block header in a report procedure sorts the Customer records by the Sales-rep code:

```
FOR EACH Customer BY Sales-Rep:
```

A *control break* separates sorted data into meaningful groups. Breaking sorted data by Sales-rep separates each sales rep's customers into a separate *break group*. Break groups allow you to calculate summary data on subsets of data. For example, you could total the number of customers that each sales rep serves.

The BREAK option of the record phrase sets up your data for control breaks, as shown in this block header statement:

```
FOR EACH Customer BREAK BY Sales-Rep:
```

Once you've set up the control break, you can use the *aggregate phrase* on an output statement item to calculate summary information. The diagram below shows the syntax of the aggregate phrase.

SYNTAX

```
[ aggregate-option ] ... [ BY break-group ] ...
```

Table 10–1 describes the aggregate options.

Table 10–1: Aggregate Phrase Options *(1 of 2)*

Aggregate Option	Description
AVERAGE	Calculates the average of all the expression's values, and calculates the overall average for the break groups.
SUB-AVERAGE	Calculates the average of all the expression's values, but does not calculate the overall average.
COUNT	Counts the number of items in the break group, and calculates the overall count of the break groups.

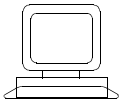
Table 10–1: Aggregate Phrase Options *(2 of 2)*

Aggregate Option	Description
SUB-COUNT	Counts the number of items in the break group, but does not calculate the overall count.
MAXIMUM	Finds the maximum value of the expression in the break group, and finds the overall maximum of the break groups.
SUB-MAXIMUM	Finds the maximum value of the expression in the break group, but does not find the overall maximum.
MINIMUM	Finds the minimum value of the expression in the break group, and finds the overall minimum of the break groups.
SUB-MINIMUM	Finds the minimum value of the expression in the break group, but does not find the overall minimum.
TOTAL	Calculates the total of all the expression's values, and calculates the overall total for the break groups.
SUB-TOTAL	Calculates the total of all the expression's values, but does not calculate the overall total for the break groups.

For example, in the code below, Sales-rep is the control break, Balance is the expression, and the aggregate phrase with the TOTAL option appears in parenthesis immediately after the expression in the DISPLAY statement:

```
FOR EACH Customer FIELDS (Balance Sales-Rep) BREAK BY Sales-rep:
  DISPLAY Customer Balance (TOTAL BY Sales-Rep).
END.
```

The TOTAL aggregate option totals the balances for each break group. Each time the value of Sales-rep changes in the sorted data, Progress creates another break group, and the TOTAL BY aggregate function calculates and outputs the total of customer balances for that sales rep's customers. Finally, at the end of the report, the TOTAL option calculates and outputs the total balances of all the break groups.

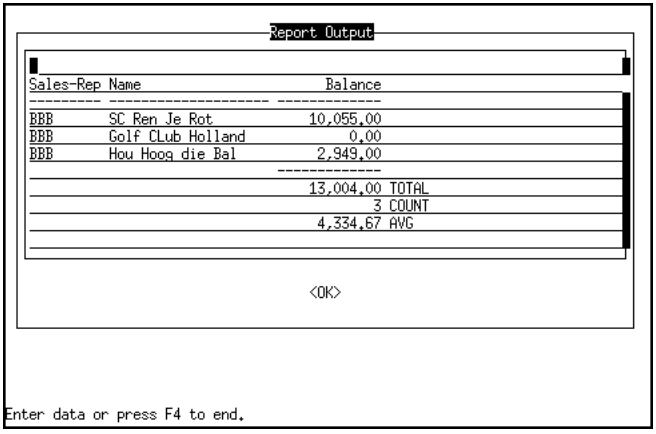


Exercise

Follow these steps to demonstrate control breaks and aggregate values:

1 ♦ Open `lt-10-05.p` and run it.

2 ♦ Choose Report. The report dialog box appears as shown in the following screen:



The members of each break group all have the same Sales-rep value. Notice that the report separates each break group with white space and outputs the result of the aggregate phrase options immediately following the break groups.

If you scroll to the bottom of the report, you can see the aggregate values for all the break groups.

3 ♦ Choose OK, then Exit, and then press **SPACEBAR** to return to the Procedure Editor.

Here is the code for this procedure:

lt-10-05.p

```
    {lt-10-in.i} /* Common Interface Setup Code */

    /**********  DEFINE TRIGGERS  **********/
    ON CHOOSE of b-rep
    DO:
        OUTPUT TO "tut-temp.txt".
/*1*/    FOR EACH Customer BREAK BY Sales-rep WITH STREAM-IO:
           DISPLAY Sales-rep Name
/*2*/           Balance (COUNT TOTAL AVERAGE BY Sales-rep).
        END.
        OUTPUT CLOSE.

        ASSIGN Rep-Editor:READ-ONLY IN FRAME dialog1 = YES
             Rep-Editor:SENSITIVE IN FRAME Dialog1 = YES
             FRAME dialog1:TITLE = "Report Output"
             Stat = Rep-Editor:READ-FILE("tut-temp.txt") IN FRAME Dialog1.

        IF Stat THEN DO:
            ENABLE Rep-Editor b-ok WITH FRAME Dialog1.
            WAIT-FOR GO OF FRAME Dialog1.
            HIDE FRAME Dialog1.
        END.
    END.

    /**********  MAIN LOGIC  **********/
    ENABLE ALL WITH FRAME Frame1.
    WAIT-FOR CHOOSE OF b-exit.
```

These notes point out the new features:

1. The BREAK option of the record phrase sets up the control break.

2. The aggregate phrase after the Balance expression contains three aggregate options.

10.5 Generating Reports from Multiple Tables

You've already learned all the language syntax you need to:

- Sort information
- Relate information from different tables
- Derive new data with calculations and aggregate options

The reports you have seen so far display data from a single table (the Customer table) and demonstrate one or two reporting features. Real world reports, however, frequently do a little bit of everything as they strive to organize lots of information and calculate useful summary information.

For example, suppose the All Around Sports accounting department needs to analyze delinquent accounts. To do their jobs, the accountants need information on customers, orders, and inventory. To get all the data they need, the report requires information from four tables: the Customer, Order, Order-line, and Item tables. The rest of this section builds a report that gathers and organizes information from these four tables and performs some summary calculations on the data.

10.5.1 Reporting Information from One Table

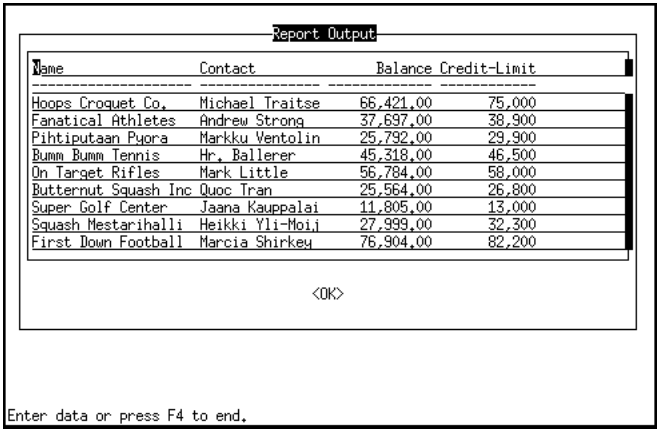
First, the accounting department needs to sort out which customers have exceeded their credit limits or those who are close to exceeding the limits. All Around Sports wants accounting to look at all customers that have balances equal to or greater than 85 percent of their credit limits. All of this information is in the Customer table.

A single FOR EACH with a WHERE clause is what you need to gather this information. The code fragment below from `lt-10-06.p` shows this structure:

lt-10-06.p

```
      .
      .
      .
FOR EACH Customer FIELDS (Balance Credit-Limit Name Contact)
    WHERE Balance >= (Credit-Limit * .85)
    WITH STREAM-IO:
    DISPLAY Name FORMAT "x(20)" Contact FORMAT "x(15)"
        Balance Credit-Limit WITH NO-BOX.
END.
      .
      .
      .
```

The WHERE clause selects just the customers over the 85 percent threshold, and the DISPLAY statement outputs the key information about these customers. Running this procedure yields the output shown below:



10.5.2 Reporting Information from Two Tables

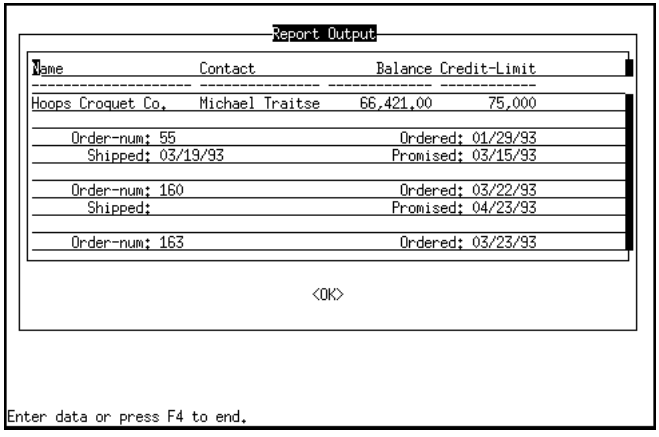
The next step is to look at the orders for each selected customer. This information is in the Order table. Another FOR EACH inside the first accomplishes this task. For each iteration of the outer block, the inner block finds all the related information. The WHERE clause establishes the relationship between the two tables.

The code below from `lt-10-07.p` shows the nested FOR EACH blocks:

lt-10-07.p

```
.
.
.
  FOR EACH Customer FIELDS (Balance Credit-Limit Name Contact Cust-Num)
      WHERE Balance >= (Credit-Limit * .85)
      WITH STREAM-IO:
      DISPLAY Name FORMAT "x(20)" Contact FORMAT "x(15)"
          Balance Credit-Limit WITH NO-BOX.

      FOR EACH Order FIELDS (Cust-Num Order-Num Order-Date Ship-Date
          Promise Date) WHERE Order.Cust-Num = Customer.Cust-Num
          WITH STREAM-IO:
          DISPLAY Order-Num Order-Date Ship-Date
              Promise-Date SKIP(1) WITH 2 COLUMNS.
      END.
  END.
.
.
.
```

The output from this procedure (following) shows how the related information ends up grouped together:



Notice that for a single iteration of the Customer data, there may be several iterations of Order data. Earlier you learned that the default frame of an iterating control block is a down frame. When you nest control blocks, only the innermost block uses a down frame. The other blocks execute one iteration at a time. This default behavior lets the related information group together naturally with the iteration of the blocks.

10.5.3 Reporting Information from Multiple Tables

Each order has one or many order lines, each of which relates to a single inventory item. Accounting needs to pull information from the Order-Line and Item tables to get specific information about what was ordered. One more FOR EACH block, inside the existing block, completes the structure.

Once the procedure gathers all the information, accounting needs the procedure to calculate total prices. Nested FOR EACH blocks create sorted groups of information similar to break groups. Therefore, you can use the aggregate phrase to calculate total prices.

This code fragment comes from `lt-10-08.p` and shows the completed structure of nested FOR EACH blocks:

lt-10-08.p

```
                .
                .
                .
        FOR EACH Customer FIELDS (Balance Credit-Limit Contact Cust-Num)
            WHERE Balance >= (Credit-Limit * .85)
            WITH STREAM-IO:
            DISPLAY Name  FORMAT "x(20)" Contact  FORMAT "x(15)"
                Balance Credit-Limit WITH NO-BOX.

/*1*/       FOR EACH Order FIELDS (Cust-Num Order-Num Order-Date Ship-Date
                Promise-Date) WHERE Order.Cust-Num = Customer.Cust-Num
                WITH STREAM-IO:
                DISPLAY Order-Num Order-Date Ship-Date
                    Promise-Date SKIP(1) WITH 2 COLUMNS.

                FOR EACH Order-Line FIELDS (Order-Num Item-Num Qty)
                    WHERE Order-Line.Order-Num =
                    Order.Order-Num WITH STREAM-IO:
/*2*/               FIND Item WHERE Item.Item-Num = Order-Line.Item-Num.
                    DISPLAY Qty Order-Line.Item-Num
                        Item-Name FORMAT "x(13)"
                        Item.Price LABEL "Unit Price"
/*3*/                    Item.Price * Qty (TOTAL)
/*4*/                  LABEL "Price" FORMAT "$zzz,zz9.99 CR" WITH NO-BOX.
                END.
            END.
        END.
                .
                .
                .
```

These notes explain the code highlights:

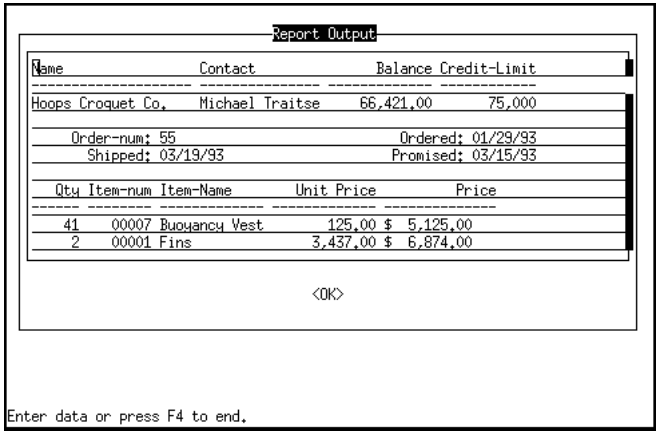
1. The WHERE clause of the third FOR EACH relates the Order-Line table back to the Order table.

2. For each Order-Line, there is a single Item record that contains information about the ordered item. A simple FIND statement retrieves this information.

3. Here, the report totals the result of an expression. Notice the absence of the BY break group syntax.

4. The FORMAT option here specifies a fairly complex format string. The result displays a leading dollar sign ($), suppresses the leading zeroes (z), and displays the credit symbol (CR) when the result is a negative value.

The output of the final version of this report follows:



10.6 Redirecting Output

A report procedure outputs to the screen by default and takes control of the application by building reports and pausing one screen at a time. Since this behavior takes control away from the user, you also learned to create a more user friendly interface for reports based on the OUTPUT TO statement. Essentially, report output gets redirected from the screen to a temporary file, and then the procedure reads the report results into an editor in a dialog box. This section discusses the OUTPUT TO statement and shows you how to send report output to text files and printers. Other uses include sending reports to text tables or UNIX devices. See the *Progress Language Reference* for more information about these uses of the OUTPUT TO statement.

10.6.1 OUTPUT TO and the Default Stream

A *stream* is a path for data movement. In this case, a stream is a path for output to a named destination. Every Progress procedure has one unnamed default output stream that writes data to the screen. The OUTPUT TO statement is your tool for redirecting the default stream to another destination. Once you use OUTPUT TO to change the output destination in a procedure, the output goes to that destination until you close it with the OUTPUT CLOSE statement, or until you name a new output destination.

This is a partial syntax for the OUTPUT TO statement.

SYNTAX

```
OUTPUT [ STREAM stream ]
  TO { PRINTER | opsys-file | opsys-device }
  [ PAGED ]
```

Here is a partial syntax for the OUTPUT CLOSE statement.

SYNTAX

```
OUTPUT [ STREAM stream ] CLOSE
```

Table 10–2 describes the OUTPUT statement syntax components.

Table 10–2: OUTPUT Statement Options

Component	Description
STREAM	Specifies a named output stream. Omit this syntax when using the default stream.
PRINTER	Specifies the default printer as set by your environment.
opsys-file	Specifies a legal file name enclosed in quotes for a text file.
opsys-device	Specifies a legal device name from your environment enclosed in quotes.
PAGED	Directs Progress to break the output into pages. Progress sends a page break control character after every 56 lines. Without this option, Progress sends output in a continuos stream. This option is most often used with printed reports, especially where you want to use page headers and footers (covered later).

10.6.2 Directing Output to a File

To direct output to a file, use the OUTPUT TO statement followed by the name of the destination file enclosed in quotes. The destination can include the path, or just the filename. If you do not provide a path, Progress writes the file to the current directory. If the file does not exist, Progress creates it. If the file does exist, Progress overwrites its contents.

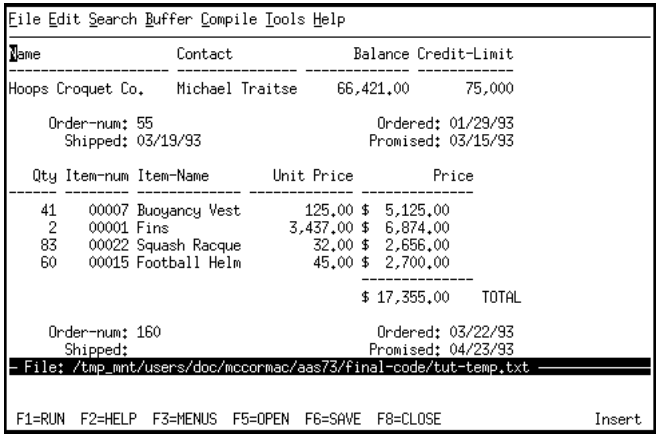
The following code example is simply the report from the last exercise, removed from the interface code:

lt-10-09.p

```
/*1*/  OUTPUT TO "tut-temp.txt".
       FOR EACH Customer FIELDS (Balance Credit-Limit Name Contact Cust-Num)
           WHERE Balance >= (Credit-Limit * .85)
           WITH STREAM-IO:
           DISPLAY Name FORMAT "x(20)" Contact FORMAT "x(15)"
               Balance Credit-Limit WITH NO-BOX STREAM-IO.

           FOR EACH Order FIELDS (Cust-Num Order-Num Order-Date Ship-Date
               Promise-Date) WHERE Order.Cust-Num = Customer.Cust-Num
                 WITH STREAM-IO:
                DISPLAY Order-Num Order-Date Ship-Date
                   Promise-Date SKIP(1) WITH 2 COLUMNS STREAM-IO.

               FOR EACH Order-Line FIELDS (Order-Num Item-Num Qty)
                   WHERE Order-Line.Order-Num =
                   Order.Order-Num WITH STREAM-IO:
                  FIND Item WHERE Item.Item-Num = Order-Line.Item-Num.
                   DISPLAY Qty Order-Line.Item-Num
                        Item-Name FORMAT "x(13)"
                        Item.Price LABEL "Unit Price"
                        Item.Price * Qty (TOTAL)
                            LABEL "Price" FORMAT "$zzz,zz9.99 CR"
                           WITH NO-BOX STREAM-IO.
               END.
           END.
        END.
/*2*/  OUTPUT CLOSE.
```

You can see from the highlighted points that the OUTPUT TO and OUTPUT CLOSE statements control the stream. Also note that the STREAM-IO option must appear in each output frame phrase.

Try running `lt-10-09.p` and then opening `tut-temp.txt` in the procedure editor; as you can see from the figure below, the content of the file matches what the interface showed in the last exercise:



10.6.3 Directing Output to a Printer

To direct output to your default printer, use the OUTPUT TO statement with the PRINTER option. Remember to use the STREAM-IO option in your frame phrases. STREAM-IO removes all graphical characteristics of the widgets, leaving just the textual data.

You probably also want to use the PAGED option of the OUTPUT TO statement to create neat page breaks.

Try modifying `lt-10-09.p` to send the report to a printer, if you have one available. Alternately, add the PAGED option to the OUTPUT TO statement, run the report, and open `tut-temp.txt` in the Procedure Editor.

Notice that Progress redisplays the labels for the current frame after a page break.

To direct output to a printer other than the default printer, use the Printer (-o) startup parameter to designate a specified printer for the Progress session. For more information, see the reference entry for the Printer (-o) startup parameter in the *Progress Startup Command and Parameter Reference*.

10.6.4 Directing Output to Multiple Destinations

Progress lets you specify multiple output destinations in a single procedure. You can use OUTPUT TO several times in a single procedure to direct the output to different destinations. You can also define named streams so you can output to several destinations.

First, you use the DEFINE STREAM statement to create the streams you need. This is a partial syntax for DEFINE STREAM.

SYNTAX

```
DEFINE STREAM stream-name
```

To output to the stream, you reference it in your output statement with the keyword STREAM, as shown in the following example:

```
/*1*/  DEFINE STREAM To-File.

/*2*/  OUTPUT TO PRINTER.
/*3*/  OUTPUT STREAM To-File TO "tut-temp.txt".

      FOR EACH Customer FIELDS (Name Balance):
/*4*/    DISPLAY Name Balance WITH STREAM-IO.
/*5*/    DISPLAY STREAM To-File Name Balance WITH STREAM-IO.
       END.

/*6*/  OUTPUT CLOSE.
/*7*/  OUTPUT STREAM To-File CLOSE.
```

The notes below describe how the code works:

1. When the procedure starts, it has a default unnamed stream that outputs to the screen by default. This statement establishes a second stream that also outputs to the screen by default. Both streams are now available for the life of the procedure.
2. This OUTPUT TO statement redirects the unnamed stream to the the default printer.
3. This OUTPUT TO statement redirects the named stream to output to a file.
4. The first DISPLAY statement outputs to the unnamed stream only.
5. The second DISPLAY statement outputs to the named stream only.
6. This OUTPUT CLOSE statement redirects the default stream back to the screen.
7. This OUTPUT CLOSE statement redirects the named stream back to the screen.

10.7 Designing Frames for Reports

What you've learned so far about reports covers the mechanics of generating, handling, and outputting data. This section covers some basics of report design. Report design, once again, is equivalent to frame design. Since a good portion of interface design also deals with frame design, you've already learned many techniques that can help make your reports look polished. The format phrase and frame phrase options that you relied on for designing interfaces also help with report design. For example, the AT, COLON, TO, SKIP, and SPACE options of the format phrase help you position widgets on the screen. They also let you position data in a report.

All the reports you have seen so far relied heavily on default Progress formatting. You'll want to design your key reports as carefully as you design your interfaces. Good report design means that you'll need the DEFINE FRAME statement to design your report containers. In addition to all the useful format and frame phrase options you learned earlier, there are some new techniques that you'll learn about in this section, including:

- Using base fields as placeholders for variable expressions
- Using page headers
- Using page footers

10.7.1 Using Base Fields with the @ Option

You cannot place an expression in a DEFINE FRAME statement because no values exist at compile time for Progress to format. Since you often need to put derived data in frames, you need a way to handle expressions. One solution is to create variables to hold the values. Another solution provides more flexibility: create variables with the characteristics of the expression results (data type, formatting, and label) as *base fields*. You can include the base fields in the frame definition and then replace the base field with any compatible expression result at run time using the @ option of the DISPLAY statement.

Here is an example:

```
DEFINE Base1 AS CHARACTER NO-LABEL.

DEFINE FRAME Frame1
    Name SKIP
    Base1 SKIP
    Phone SKIP
        WITH NO-BOX.
        .
        .
        .
IF Contact EQ "" THEN
    DISPLAY Name "No Contact" @ Base1 Phone WITH FRAME Frame1.
ELSE
    DISPLAY Name Contact @ Base1 Phone WITH FRAME Frame1.
        .
        .
        .
```

The example above compiles a phone list of customer and contact names. If no contact is on file, “No Contact” appears in the report.

10.7.2 Using the HEADER of a Frame for Running Page Heads

A frame can contain three distinct sections: body, HEADER, and BACKGROUND. Until now, everything you placed in a frame became part of the frame body. Using the keywords HEADER and BACKGROUND, you can also define two additional sections for your frames. The code fragment below shows how a DEFINE FRAME statement with these sections would look:

```
DEFINE FRAME f-example
    body-item1 /* Can be field, variable, constant, image,
    body-item2    or rectangle. */
    body-item3

    HEADER
    header-item1 /* Can be field, variable, constant, EXPRESSION,
    header-item2   image, or rectangle. */
    header-item3

    BACKGROUND
    background-item1 /* Can be field, variable, constant, EXPRESSION,
    background-item2    image, or rectangle. */
    background-item3

        WITH SIDE-LABELS.
```

Frame backgrounds are typically used for placing a logo (image) or other graphic device in the background of a display frame. However, you cannot print graphics from the Progress 4GL. For more information, see the *Progress Language Reference* and the *Progress Report Builder User's Guide*. The rest of this section concentrates on the HEADER part of the frame.

The HEADER has a couple of special properties that allow you to implement running page heads and footers:

- A HEADER section can contain expressions.
- Progress re-evaluates expressions in a HEADER section each time the frame is redisplayed.
- Progress suppresses field and variable labels in a header frame. If you want labels, you supply text strings in the frame definition.

If a HEADER frame contains an expression, field, or variable, the frame definition must take place in the context where Progress can provide new values. In other words, for an iterating report procedure, move the DEFINE FRAME statement from the top of your procedure into the FOR EACH block. Think of a HEADER frame as an executable statement. Just like a DISPLAY statement inside a FOR EACH block, the HEADER section of the DEFINE FRAME statement executes on every iteration of the FOR EACH block.

Also, note that a frame does not have to have a body—it can consist of a header only. You can modularize your report design with three frames: one each for page header, body, and page footer. This approach lets you adopt standard headers and footers.

Assume that All Around Sports wants a standard page header on every page of its reports. This is the information they want to include in the page header:

```
     Date: mm/dd/yy   Title    sales-rep  Page: xxx
```

This code defines the first part of a procedure that implements the three-frame design:

```
/*1*/  OUTPUT TO "tut-temp.txt" PAGE-SIZE 25.
       FOR EACH Customer WHERE Balance >= 1400 BREAK BY Sales-Rep
         WITH STREAM-IO:
                      DEFINE FRAME f-hdr
/*2*/     HEADER
/*3*/       "Date:" TODAY "Customer Report" AT 25
/*4*/       Sales-Rep AT 55
           "Page" AT 65
/*5*/      PAGE-NUMBER FORMAT ">>9" SKIP(1)
/*6*/         WITH PAGE-TOP FRAME f-hdr STREAM-IO.
```

This part of the procedure contains the following language elements and points of interest:

1. The PAGE-SIZE option of the OUTPUT statement sets the default size for a report page.
2. The HEADER option tells Progress to place the specified items in the header section at the top of the frame.
3. The TODAY function returns the current system date, and constitutes an expression.
4. The Sales-rep initials come from the database and represent another part of the HEADER that Progress must evaluate.
5. The PAGE-NUMBER function tracks the current page number (expression).
6. The PAGE-TOP FRAME f-hdr further defines what kind of HEADER the frame is. PAGE-TOP specifies where to place the frame and makes the frame a running page head.

10.7.3 Using the HEADER of a Frame for Running Page Footers

You can also use the HEADER section to create running page footers. The only difference is that you specify PAGE-BOTTOM in the frame phrase instead of PAGE-TOP. Here's how All Around Sports wants their footers to look:

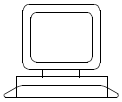
```
Customer Report continued on next page
```

Once again, the report needs a DEFINE FRAME statement with the HEADER option as this code shows:

```
    DEFINE FRAME f-ftr
    HEADER
      "Customer Report"
      "continued on next page"
/*1*/    WITH FRAME f-ftr PAGE-BOTTOM CENTERED STREAM-IO.
```

The PAGE-BOTTOM option tells Progress to display the frame at the bottom of each page. PAGE-TOP and PAGE-BOTTOM frames are activated based on DISPLAY or VIEW statements. They are deactivated when the block to which the frames are scoped iterates or ends, which is why they have to be viewed in every iteration.

10.7.4 Programming Example

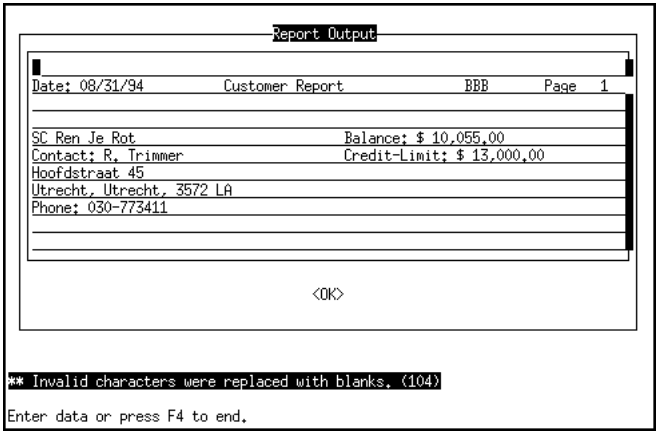


Exercise

The exercise below demonstrates the techniques discussed in this section:

1 ♦ Open `lt-10-10.p` and run it.

2 ♦ Choose Report. The report dialog box appears as shown below:



Notice the header info at the top of each page. As you scroll through, you can see footers as well.

3 ♦ Choose OK, then Exit, and then press **SPACEBAR** to return to the Procedure Editor.

The following code fragment shows the report generating code for this procedure:

lt-10-10.p

```
          .
          .
          .
/*1*/   DEFINE FRAME f-body
           Name NO-LABEL
           Balance AT 40 FORMAT "$zzz,zz9.99 CR" SKIP
           Contact
           Credit-Limit AT 40 FORMAT "$zzz,zz9.99 CR" SKIP
           Address NO-LABEL SKIP
           Holder NO-LABEL SKIP
           Phone SKIP(2)
               WITH SIDE-LABELS STREAM-IO.
/*2*/   OUTPUT TO "tut-temp.txt" PAGE-SIZE 25.
/*3*/   FOR EACH Customer FIELDS (balance Sales-Rep name
           Contact Credit-Limit Address City State Postal-Code Phone)
          WHERE Balance >= 1400 BREAK BY Sales-Rep:
/*4*/        DEFINE FRAME f-hdr
             HEADER
             "Date:" TODAY "Customer Report" AT 25
              sales-rep AT 55
              "Page" AT 65
             PAGE-NUMBER FORMAT ">>9" SKIP(1)
/*5*/               WITH PAGE-TOP FRAME f-hdr STREAM-IO.
/*6*/        DEFINE FRAME f-ftr
             HEADER
              "Customer Report"
              "continued next page"
/*7*/               WITH FRAME f-ftr PAGE-BOTTOM CENTERED STREAM-IO.
/*8*/      VIEW FRAME f-hdr.
           VIEW FRAME f-ftr.
           DISPLAY Name Balance Contact Credit-Limit Address
/*9*/           (City + ", " + St + ", " + Postal-Code) @ Holder Phone
              WITH FRAME f-body.
/*10*/       IF LAST-OF(Sales-Rep) THEN DO:
             HIDE FRAME f-ftr.
             PAGE. END. END.
       OUTPUT CLOSE.
          .
          .
          .
```

The following notes summarize the techniques shown in this chapter:

1. The body frame, which has no HEADER section, appears in its normal position, at the top of the file with other definitions.

2. The PAGE-SIZE option sets the report page size.

3. The use of the control break changes the report output from one report into a series of smaller reports-one for each sales rep.

4. This HEADER frame comprises the running report head.

5. PAGE-TOP places this frame at the top of the report page.

6. This HEADER frame comprises the running page footer.

7. PAGE-BOTTOM places the header frame at the bottom of the page.

8. The VIEW statements force Progress to evaluate the two HEADER frames on each iteration of the block.

9. Here, the report creates an address string and uses the @ option to place the result at the Holder variable.

10. The LAST-OF function is for checking for the end of a break group, allowing you to perform special tasks. In this case, the procedure suppresses the page footer because this break group report is complete. It also uses the PAGE statement to start a new page for the next break group report.

10.8 Using the PUT Statement

The PUT statement is another useful 4GL element for generating reports, especially when you want to customize certain parts of a report. The PUT statement has no default framing services, making it useful for writing data to a file or overriding default framing. Since PUT has no framing defaults, your procedures must contain explicit code for formatting your output.

The PUT statement outputs data one field at a time and uses the format of the field or variable. To include line breaks in the output, you must use the SKIP option. Additionally, the UNFORMATTED option of PUT displays all the data of the field or variable, regardless of format and without spaces between fields.

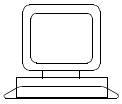
Why would you use PUT instead of DISPLAY? For every DISPLAY statement, Progress needs a frame. To execute a DISPLAY statement, Progress builds a frame capable of handling the expected output, using default services and your explicit instructions. PUT on the other hand, simply outputs data one line at a time, with no default formatting. DISPLAY is most useful when you want automatic formatting. PUT is most useful when you want complete control over output.

This is a partial syntax for the PUT statement.

SYNTAX

```
PUT [ STREAM stream ] [ UNFORMATTED ]
    [   expression  [ FORMAT string ]
          [ AT expression | TO expression ]
      | SKIP [ ( expression ) ]
      | SPACE [ ( expression ) ]
    ] ...
```

One common task that the PUT statement can help with is mailing labels. Since mailing labels must conform to a compact physical layout and be uniform, using PUT is a good idea. Suppose that the All Around Sports accounting department wants to send notices to customers with large balances. They need a procedure that creates mailing labels for the notices.

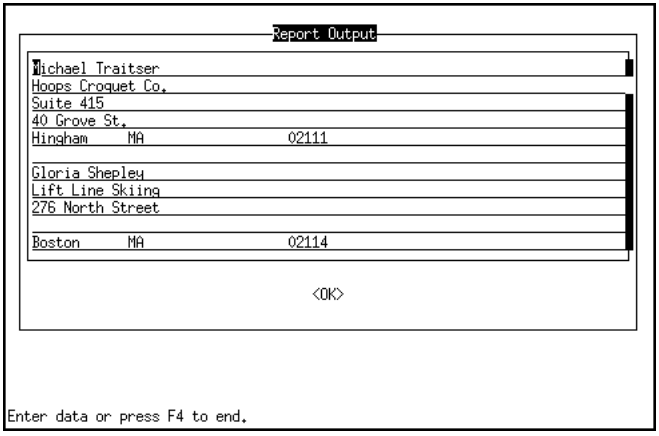


Exercise

Follow these steps to demonstrate the PUT statement:

1 ♦ Open `lt-10-11.p` and run it.

2 ♦ Choose Report. The report dialog box appears as shown below:



The editor shows the mailing list as it appears in a text file. Note that the output is not perfect:

- For simple addresses, there is a blank line between the street address line and the city, state, postal-code line. Some addresses need this blank line for extra information.
- There is too much space between the state and postal code.

3 ♦ Choose OK, then Exit, and then press **SPACEBAR** to return to the Procedure Editor.

Here is the code for this procedure:

lt-10-11.p

```
    {lt-10-in.i} /* Common Interface Setup Code */

    /**********  DEFINE TRIGGERS  **********/
    ON CHOOSE of b-rep
    DO:
/*1*/    OUTPUT TO "tut-temp.txt".
        FOR EACH Customer FIELDS (Balance Postal-Code Contact Name Address
           Address2 City St) WHERE Balance >= 1400 BY Postal-Code:
/*2*/       PUT Contact SKIP
              Name SKIP
              Address SKIP
              Address2 SKIP
              City St Postal-Code SKIP(1).
        END.
        OUTPUT CLOSE.

        ASSIGN Rep-Editor:READ-ONLY IN FRAME Dialog1 = YES
             Rep-Editor:SENSITIVE IN FRAME Dialog1 = YES
             FRAME Dialog1:TITLE = "Report Output"
             Stat = Rep-Editor:READ-FILE("tut-temp.txt") IN FRAME Dialog1.

        IF Stat THEN DO:
            ENABLE Rep-Editor b-ok WITH FRAME Dialog1.
            WAIT-FOR GO OF FRAME Dialog1.
            HIDE FRAME Dialog1.
        END.
    END.

        /**********  MAIN LOGIC  **********/
    ENABLE ALL WITH FRAME Frame1.
    WAIT-FOR CHOOSE OF b-exit.
```

This procedure contains the following points of interest:

1. The OUTPUT TO statement directs the output from this procedure to a text file named `tut-temp.txt`.

2. You can use the PUT statement in addition to the DISPLAY statement when sending data to a file or to a printer (any destination other than the screen).

To improve this procedure, you can:

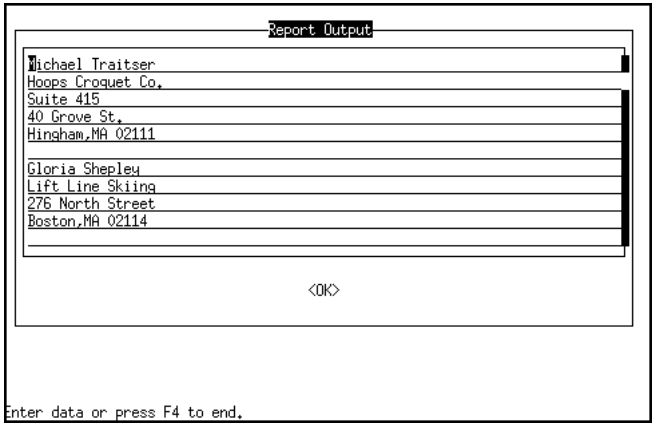
- Remove the blank line from simple addresses that do not use the extra space
- Tighten up the spacing

If you run `lt-10-12.p`, you can see the modified version of this procedure. Here is the code for that version:

lt-10-12.p

```
    {lt-10-in.i} /* Common Interface Setup Code */

    /**********  DEFINE TRIGGERS  **********/
    ON CHOOSE of b-rep
    DO:
        OUTPUT TO "tut-temp.txt".
        FOR EACH Customer FIELDS (Balance Postal-Code Contact Name Address
           Address2 City St) WHERE Balance >= 1400 BY Postal-Code
           WITH STREAM-IO:
            PUT Contact SKIP
                Name SKIP
/*1*/           Address SKIP.

/*2*/       IF Address2 NE "" THEN PUT Address2 SKIP.

/*3*/       PUT City + "," + St + " " + STRING(Postal-Code, "99999")
                FORMAT "x(23)" SKIP(1).

/*4*/       IF Address2 EQ "" THEN PUT SKIP(1).
        END.
        OUTPUT CLOSE.

        ASSIGN Rep-Editor:READ-ONLY IN FRAME Dialog1 = YES
             Rep-Editor:SENSITIVE IN FRAME Dialog1 = YES
             FRAME Dialog1:TITLE = "Report Output"
             Stat = Rep-Editor:READ-FILE("tut-temp.txt") IN FRAME Dialog1.

        IF Stat THEN DO:
            ENABLE Rep-Editor b-ok WITH FRAME Dialog1.
            WAIT-FOR GO OF FRAME Dialog1.
            HIDE FRAME Dialog1.
        END.
    END.

        /**********  MAIN LOGIC  **********/
    ENABLE ALL WITH FRAME Frame1.
    WAIT-FOR CHOOSE OF b-exit.
```

This is the output of the procedure:



The following notes help explain the techniques used in the procedure:

1. The first PUT statement outputs and formats the part of the mailing label that is common to all labels.

2. The first IF statement determines whether the second address line has data. If it does, it outputs the data.

3. When you create a character expression, like the one in this PUT statement, Progress removes trailing blanks from the fields. So this output tightens up the extra white space that showed up in the first mailing list example.

4. Finally, the second IF statement determines whether there is second address line data. If not, the PUT statement sends a blank line at the end of the address. This statement keeps the label data together and keeps the individual labels correctly spaced from each other.

10.8.1 Using PUT for Printer Control

When you send output to a printer, you may want to modify the way the printer generates that output. Many printers have a set of control sequences you can use to specify different print characteristics. You might, for example, want to change the number of printed characters per inch.

When you write a procedure that sends output to a printer, you can include printer control sequences within that procedure. Many control sequences involve special characters that can be represented by their octal (base 8) equivalent. To distinguish these octal codes, you precede the three octal digits by an escape character. Progress then converts the octal number to a single character.

On UNIX, the escape character is a tilde (~) or a backslash (\).

The PUT statement with the CONTROL option allows you to specify a control sequence to send to the printer. This is a partial syntax for this version of the PUT statement.

SYNTAX

```
PUT [ STREAM stream-name ]
  CONTROL "ctrl-sequence" "ctrl-sequence" ...
```

The control sequences you send to the printer have no effect on the current line, page counters, and positions maintained within Progress. Assume you want to print a report on your Brand X printer, using compressed-print mode.

The following procedure implements this task:

lt-10-13.p

```
      /**********  DEFINE WIDGETS  **********/
/*1*/  DEFINE VARIABLE Start-Compress AS CHARACTER
      INITIAL "~033[3w".
/*2*/  DEFINE VARIABLE Stop-Compress AS CHARACTER
        INITIAL "~032[3w".
      DEFINE BUTTON b-normal LABEL "Print Normal Report".
      DEFINE BUTTON b-compress LABEL "Print Compressed Report".
      DEFINE BUTTON b-exit LABEL "Exit".

      /**********  DEFINE FRAMES  **********/
      DEFINE FRAME Frame1
          SKIP(1)
         b-normal b-compress SKIP(1)
          b-exit
            WITH NO-BOX CENTERED THREE-D.

        /**********  DEFINE TRIGGERS  **********/
/*3*/  ON CHOOSE OF b-normal
       DO:
          OUTPUT TO PRINTER.
         FOR EACH Customer WHERE Balance >= 1400
              WITH STREAM-IO:
              DISPLAY Name Phone Balance Sales-rep.
          END.
          OUTPUT CLOSE.
       END.
/*4*/  ON CHOOSE OF b-compress
       DO:
          OUTPUT TO PRINTER.
          PUT CONTROL Start-Compress.
          FOR EACH Customer WHERE Balance >= 1400
              WITH STREAM-IO:
             DISPLAY Name Phone Balance Sales-rep.
           END.
          PUT CONTROL Stop-Compress.
          OUTPUT CLOSE.
      END.

      /**********  MAIN LOGIC  **********/
      ENABLE ALL WITH FRAME Frame1.
      WAIT-FOR CHOOSE OF b-exit.
```

NOTE: This procedure works only if you have a printer connected to your system. Not all printers support compressed printing.

These notes help to explain the code:

1. The start-compress variable contains the four-character sequence that puts the printer into compressed-print mode. These four characters are octal 033 (decimal 27) followed by left bracket ([), 3, and w.

2. The variable stop-compress takes the printer out of compressed print mode.

3. When the user selects btn-normal, Progress runs the report in normal mode.

4. When the user selects btn-compressed, Progress runs the report in compressed mode.

Control sequences are hardware-dependent, thus applications containing hard-coded printer control sequences are not easily portable to other environments. See the *Progress Client Deployment Guide* for more information about using the PUT CONTROL statement with control sequences.

10.9 Summary

This chapter showed how to write Progress procedures that produce both simple and sophisticated reports.

Generating Simple Reports

You can use the DISPLAY statement to generate a simple report output to the screen from a single table or from multiple related tables. You can easily format a report to group records and include calculations with the CONTROL BREAKS option and aggregate functions.

Redirecting Output

The STREAM-IO option allows you to redirect the output of a report-generating procedure to a printer or text file. This option removes the graphical components of your data. You can also redirect output to multiple destinations, and you can define multiple streams.

Generating Reports of Data Represented by Widgets

The widgets that deal with text-fill-in field and text widgets-pose no special problems when you generate reports. However, you must make certain adjustments for widgets that represent data graphically. There are two ways to handle these widgets in reports:

- Use the STREAM-IO option.
- Use the PUT statement instead of DISPLAY. PUT suspends default frame-based formatting.

Customizing Reports

You can customize your reports by:

- Adding headers and footers
- Using advanced printing techniques, such as printer control sequences

11

Building Menus

Menus add flexibility to an interface. A menu provides choices and options for end users that give them more control over how they use an application. Through menus, users can access the different functions of an application in the order that best suits them.

In this chapter, you learn about building and customizing menus. Specifically, you learn about:

- Menu basics
- Creating menu bars, submenus, and menu items
- Menu attributes
- Design conventions

11.1 Menu Basics

Menus are an important part of an event-driven programming model, since menus allow users to control the flow of an application. Progress defines a *menu* as a widget containing a list of commands or options available to users. In Progress, a menu is always associated with another widget. In the tutorial, you learn about the most common type of menu: the menu bar. A *menu bar* is a horizontal bar displayed at the top of a window. The menu bar is always associated with a window widget. To learn about other types of menus, see the *Progress Programming Handbook*.

A menu bar consists of submenus. A *submenu* is a vertical list of commands and options available to a user. So, the menu bar form of menu is a collection of lists of commands and options. The **titles** of the submenus are listed across the menu bar. To a user, a menu bar is a collection of pull-down menus. To a Progress programmer, a menu bar is a collection of submenus—the menu bar is syntactically the menu. This slight distinction in terminology is important to help make the discussion in this chapter clear. The terms **submenu** and **pull-down menu** are analogous, and the terms **menu** and **menu bar** are analogous.

A submenu consists of menu items. A *menu item* is an individual command or option, or even another submenu. When a submenu is a menu item of another submenu, it is known as a *nested submenu*. Nested submenus appear to the side of the main submenu when chosen.

When complete, the menu bar you present to a user is a hierarchical structure consisting of a:

- Menu widget
- One or more submenu widgets
- One or more menu item widgets for each submenu

The widgets that make up the complete menu bar are related to each other. For example, submenus on the menu bar are **children** of the **parent** menu bar. Each of the menu bar submenus are **siblings** to each other. Similarly, menu items are children of a parent submenu, and so on. When you code a complete menu bar, you build it from the bottom up. The lower-level widgets must exist before you can relate them to the higher-level widgets.

The menu widget itself does not have a parent—a menu is always **owned** by one widget. In the case of the menu bar, the window widget owns it. Each Progress window can own one menu bar.

Figure 11–1 shows a menu bar with two menu titles, File and Edit. When you choose Edit, a pull-down menu with three menu items appears below Edit. When you choose the menu item Add, a pull-down menu with four menu items appears to the side of Add. The arrow to the right of Add indicates that there is a nested submenu.

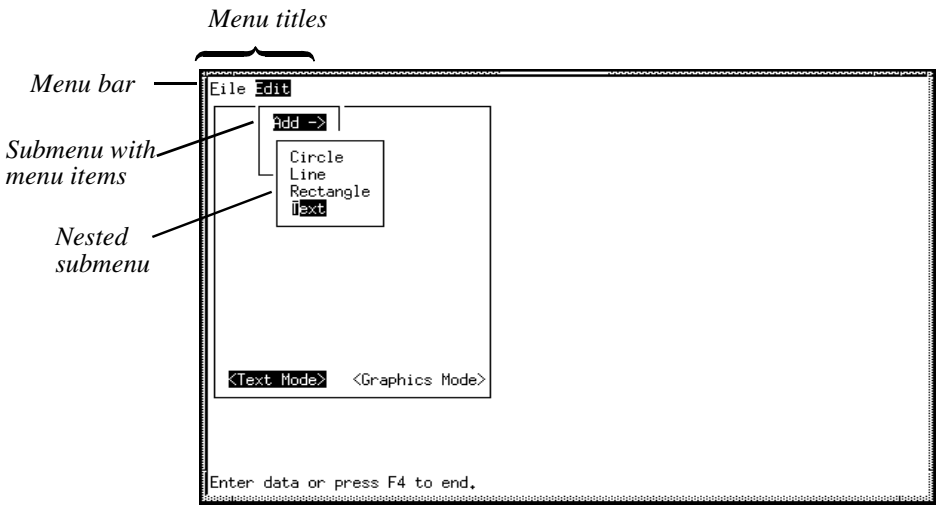


Figure 11–1: Example of a Window with a Menu Bar

11.2 Defining a Menu

A complete menu bar consists of a menu bar, submenus, and menu items. When you define a menu bar in the Progress 4GL, you start at the lowest level, then build the top-level structure. You need to:

1. Define the submenus and the menu items they contain.
2. Define the menu bar.
3. Assign the menu bar to a window.

The following sections explain how to define submenus and menu items, how to add them to a menu bar, and how to assign the menu bar to a window.

11.2.1 Defining a Submenu

When you assemble a menu, you first define the pull-down submenus and their associated menu items with the DEFINE SUB-MENU statement. This is the basic syntax for the DEFINE SUB-MENU statement.

SYNTAX

```
DEFINE SUB-MENU submenu-name
  { LIKE menu | menu-element-descriptor ... }
```

The LIKE *menu* option is useful when you want to duplicate an already existing menu. The *menu-element-descriptor* allows you to customize the submenu. This is the syntax for the *menu-element-descriptor* phrase.

SYNTAX

```
{ SUB-MENU submenu-name
  [ DISABLED ] [ LABEL label ]
  | RULE
  | SKIP
  | menu-item-phrase
}
```

Table 11–1 explains the relevant components of the syntax.

Table 11–1: MENU Element Options *(1 of 2)*

Component	Description
SUB-MENU	Names a submenu that displays when the user chooses this menu item. The specified submenu must already be defined.
DISABLED	Disables the menu item initially. When you use this option, the user can't choose the menu item.
LABEL	Defines the text descriptor that the user sees on the menu. If you omit LABEL, Progress displays the submenu name by default.
RULE	Inserts a rule (line) at this point on the menu. Use rules to visually group related commands.

Table 11–1: MENU Element Options *(2 of 2)*

Component	Description
SKIP	Inserts a blank line at this position on the menu.
menu-item-phrase	Names a menu-item widget and specifies details about the item. See the following table for more information.

The *menu-item-phrase* lets you customize both how the menu item displays and how the user can access it. This is the syntax for the *menu-item-phrase*.

SYNTAX

```
MENU-ITEM item-name
  [ ACCELERATOR keylabel ]
  [ TOGGLE-BOX ]
  [ DISABLED ]
  [ LABEL label ]
```

Table 11–2 explains the relevant components of the syntax.

Table 11–2: MENU–ITEM Options *(1 of 2)*

Component	Description
MENU-ITEM	Specifies a unique name for the menu item. This name doesn't have to be previously defined.
ACCELERATOR	Specifies the keyboard accelerator for this menu item. A keyboard accelerator is a key that chooses a menu item even if the menu item is not displayed. This option is available only in graphical windowing environments.
TOGGLE-BOX	Displays the menu item with a toggle box that the user can toggle on or off.

Table 11–2: MENU–ITEM Options *(2 of 2)*

Component	Description
DISABLED	Disables the menu item. This means that the user cannot choose this item.
LABEL	Specifies the text descriptor that the user sees in the submenu. If you omit LABEL, Progress displays the item name by default. You can include an ampersand (&) within the label to indicate that the following letter acts as a mnemonic (shortcut key) for the menu item. For example, "E&xit" specifies “x” as the mnemonic.

Note that in this code fragment, which is an example of a complete submenu definition, only the final MENU-ITEM phrase has a period:

```
DEFINE SUB-MENU sm-Reports
    MENU-ITEM mi-Labels      LABEL "Customer Labels"
    MENU-ITEM mi-Names       LABEL "Customer Names"
    RULE
    MENU-ITEM mi-Balances    LABEL "Order Totals"
    MENU-ITEM mi-Today        LABEL "Order Items".
```

The next step in building a menu is to define a menu bar and add submenus and items to it.

11.2.2 Defining a Menu Bar

Once you’ve defined the contents of each submenu, you need to define a menu bar and associate the submenus to it. The DEFINE MENU statement defines a menu, and the MENUBAR phrase makes the menu a menu bar. This is the syntax for the DEFINE MENU statement.

SYNTAX

```
DEFINE
  [ [ NEW ] SHARED ] MENU menu-name
  [ MENUBAR ]
  {   LIKE menu-name
    | menu-element-descriptor ...
  }
```

Table 11–3 explains the relevant components of the syntax.

Table 11–3: MENU Options

Component	Description
NEW SHARED	Defines a menu that you can share among procedures.
SHARED	Specifies a menu created in another procedure with the DEFINE NEW SHARED MENU statement.
MENU	Specifies the menu you are defining.
MENUBAR	Specifies that the menu is a menu bar.
LIKE	Specifies a previously defined menu whose characteristics you want to apply to the new menu.
menu-element-descriptor	Specifies an element to display on the menu. You must specify at least one menu element, unless you use the LIKE option.

This is an example of a complete DEFINE MENU statement:

```
DEFINE MENU mbar MENUBAR
    SUB-MENU sm-Table       LABEL "Tables"
    SUB-MENU sm-Reports     LABEL "Reports"
    SUB-MENU sm-Help        LABEL "Help".
```

First, the DEFINE MENU statement creates the menu bar and names it mbar. Then it assigns each of the previously defined submenus to it. The LABEL option once again lets you provide titles for the menu items. The order in which you list the submenus is important—Progress places the menu titles of the submenus on the menu bar starting with the first listed submenu.

11.2.3 Assigning a Menu Bar to a Window

The last step in building a menu is assigning the menu bar to a window. You accomplish this by setting the window's MENUBAR attribute equal to the menu's widget handle. For example, if you are using the default window, you can refer to it with the DEFAULT-WINDOW system handle, as follows:

```
ASSIGN DEFAULT-WINDOW:MENUBAR = MENU mbar:HANDLE.
```

In character interfaces, you can work only in the single default window. However, you can use alternate menu bars to simulate multiple windows. To invoke a new "window," the simplest approach is to clear all frames from the screen and assign the appropriate menu bar to the default window. (You might choose to manage frames differently.)

11.2.4 Assigning Triggers to Menu Items

To associate functionality with menu items, you use triggers. As with buttons, the key event to attach a trigger to is CHOOSE. Typically, the content of a trigger for a menu item is a RUN statement calling an internal or external procedure, as shown in the example below:

```
/**********  DEFINE TRIGGERS  **********/
ON CHOOSE OF MENU-ITEM mi-Labels IN MENU sm-Reports
DO:
    RUN Display-Labels.
END.
```

Another typical function found on a menu is the Exit command. Throughout the tutorial, you've used a button labeled "Exit" and used the CHOOSE event of the Exit button as the condition that satisfies the WAIT-FOR statement. When the user chooses Exit, the flow of control goes past the WAIT-FOR. If the WAIT-FOR is the last statement in the procedure, the procedure ends.

You can easily replace this functionality with an Exit command on the menu bar. By convention, the Exit command is always the last menu item on the first submenu of a menu bar. If you add this menu item to your menu bar, then you could use the following WAIT-FOR statement to block the application:

```
/**********  MAIN LOGIC  *********/
WAIT FOR CHOOSE OF MENU-ITEM mi-Exit IN MENU mbar.
```

In reality, closing down an application often requires some clean up. While still using the CHOOSE event of the Exit command as your WAIT-FOR condition, you can write a trigger for the same event to do your clean-up. Then, as a last step, you could close the window and end the application, as shown in the following example:

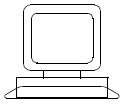
```
/**********  DEFINE TRIGGERS  *********/
ON CHOOSE OF MENU-ITEM mi-Exit IN MENU mbar
DO:
    /* Clean up Code */
    APPLY "CLOSE-WINDOW" TO DEFAULT-WINDOW.
END.
```

CLOSE WINDOW is a Progress event function that executes for any event that equates to dismissing a window.

For a complete discussion of triggers and trigger programming techniques, see the *Progress Programming Handbook*.

Referencing Menus

Each menu and submenu must have a unique name. To reference the menu or submenu, precede the reference with the keyword MENU. To reference a menu item, precede the reference with the keyword MENU-ITEM. Menu items do not have to have unique names. In situations where ambiguity arises, you must extend the menu item reference to include the IN MENU option.

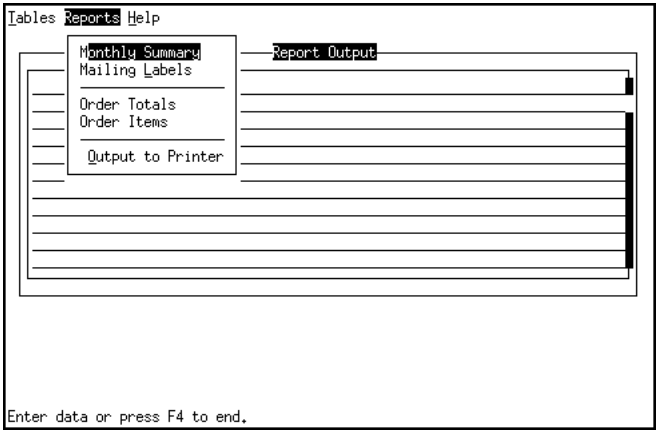


Exercise

11.2.5 Menu Bar Example

You've learned all the basics, so now you can put it all together. This example implements enough of a menu bar to demonstrate what you've learned. Perform these steps:

1 ♦ Open `lt-11-01.p` and run it. The following display appears:



2 ♦ Press **ENTER-MENUBAR** to activate the menu bar.

3 ♦ Choose Tables from the menu bar and browse through the submenu and the nested submenu.

4 ♦ Choose the Reports and Help submenus and browse through them.

5 ♦ Choose Reports→ Mailing Labels. The editor in the window displays the list of customer mailing labels.

6 ♦ Choose Tables→ Exit.

7 ♦ Press **SPACEBAR** to return to the Procedure Editor.

First, look at the code that sets up the menu structure:

lt-11-mn.i

```
      /**********  DEFINE WIDGETS  **********/
/*1*/  DEFINE SUB-MENU sm-Open
          MENU-ITEM mi-Cust        LABEL "&Customer"
          MENU-ITEM mi-Order       LABEL "&Order".

      DEFINE SUB-MENU sm-Table
/*2*/      SUB-MENU sm-Open         LABEL "O&pen"
/*3*/      MENU-ITEM mi-Exit        LABEL "E&xit".

      DEFINE SUB-MENU sm-Reports
          MENU-ITEM mi-Cust      LABEL "&Monthly Summary"
          MENU-ITEM mi-Labels    LABEL "Mailing Labels"
/*4*/      RULE
/*5*/      MENU-ITEM mi-Balances    LABEL "Order Tot&als" DISABLED
           MENU-ITEM mi-Today       LABEL "Order &Items"  DISABLED
           RULE
/*6*/      MENU-ITEM mi-Print       LABEL "&Output to Printer" TOGGLE-BOX.

      DEFINE SUB-MENU sm-Help
          MENU-ITEM mi-Help        LABEL "H&elp".
/*7*/  DEFINE MENU mbar MENUBAR
          SUB-MENU  sm-Table       LABEL "&Tables"
          SUB-MENU  sm-Reports     LABEL "&Reports"
          SUB-MENU  sm-Help        LABEL "&Help".

/*8*/  ASSIGN DEFAULT-WINDOW:MENUBAR = MENU mbar:HANDLE.
```

These notes help to explain the include file code:

1. The DEFINE SUB-MENU statement defines one pull-down menu and the menu items of that pull-down menu. The ampersand (&) in the label, establishes the mnemonic for the menu item (covered later).

2. This submenu appears as a menu item in the next DEFINE SUB-MENU statement, so its definition must come first.

3. The Exit menu item is always the last menu item of the first submenu on the menu bar.

4. Use RULE to provide a graphic divider between groups of menu items. Use SKIP to add space between menu items.

5. The DISABLED option disables this menu item on startup (covered later).

6. The TOGGLE-BOX option makes the menu item a toggle (covered later).

7. The DEFINE MENU statement defines a Progress menu. The MENUBAR phrase makes the menu a menu bar and associates submenus with it.

8. This critical step makes the window the owner of the menu bar. In this case, the owner is the default window.

Next, look at the following code, which uses the previous menu structure to implement the mailing list report you saw in the last section:

lt-11-01.p

```
      /**********  DEFINE WIDGETS  **********/
/*1*/  {lt-11-mn.i} /* Menu definition */
      DEFINE VARIABLE Rep-Editor AS CHARACTER VIEW-AS EDITOR
          SCROLLBAR-VERTICAL SIZE 76 BY 13.
      DEFINE VARIABLE Stat AS LOGICAL.

      /**********  DEFINE FRAMES  **********/
      DEFINE FRAME Frame1
          Rep-Editor WITH NO-LABELS ROW 2 CENTERED TITLE "Report Output"

      /**********  DEFINE TRIGGERS  **********/
/*2*/  ON CHOOSE OF MENU-ITEM mi-Exit
          APPLY "CLOSE-WINDOW" TO DEFAULT-WINDOW.

/*3*/  ON CHOOSE OF MENU-ITEM mi-Labels
          RUN p-Report.

      /**********  MAIN LOGIC  **********/
      ASSIGN Rep-Editor:READ-ONLY IN FRAME Frame1 = YES
             Rep-Editor:FONT = 3.
      ENABLE ALL WITH FRAME Frame1.
/*4*/  WAIT-FOR CHOOSE OF MENU-ITEM mi-Exit.

      /**********  INTERNAL PROCEDURES  **********/
/*5*/  PROCEDURE p-Report:
      OUTPUT TO "tut-temp.txt".
      FOR EACH Customer FIELDS (Balance Postal-Code Contact Name Address
          Address2 City St) WHERE Balance >= 1400 BY Postal-Code:
           PUT Contact SKIP
              Name SKIP
              Address SKIP.
          IF Address2 NE "" THEN PUT Address2 SKIP.
          PUT City + "," + St + " " + STRING(Postal-Code, "99999")
              FORMAT "x(23)" SKIP(1).
          IF Address2 EQ "" THEN PUT SKIP(1).
       END.
      OUTPUT CLOSE.

     ASSIGN Stat = Rep-Editor:READ-FILE("tut-temp.txt") IN FRAME Frame1.
      END PROCEDURE.
```

These notes help explain the main code:

1. This reference includes the menu structure in the procedure.
2. Choosing the Exit menu item ends the application by closing the window.
3. Like most menu items, choosing this item executes a RUN statement to an internal or external procedure. Here, the internal procedure runs a report.
4. The Exit command of the menu bar satisfies the WAIT-FOR condition by closing the application window.
5. The internal procedure executes the customer mailing label report.

11.3 Using Optional Menu Features

This section covers these menu item syntax options:

- Disabled menu items
- Toggle boxes
- Mnemonics
- Accelerators

11.3.1 Disabled Menu Items

You can enable and disable menu items when it is appropriate to do so based on the user's previous action. An enabled item is a valid selection; a disabled item is not, and the user cannot choose it.

This is the syntax for the DISABLED option.

SYNTAX

```
MENU-ITEM menu-item DISABLED
```

11.3.2 Toggle Boxes

Toggle boxes let you represent selectable menu items as logical values. You can then program the menu item to trigger a different event based on the value of the toggle box which changes each time the user selects the menu item. The application can examine the CHECKED attribute for the menu item to determine whether the item is currently checked or unchecked. You can also initialize or change the condition of the menu item by setting the CHECKED attribute in the program.

When the user toggles a toggle-box item, Progress sends the VALUE-CHANGED event to the menu item. Therefore, by defining a trigger on that event, you can take immediate action when the user toggles the value.

This is the syntax for the TOGGLE-BOX option.

SYNTAX

```
MENU-ITEM menu-item TOGGLE-BOX
```

11.3.3 Mnemonics

A *mnemonic* provides a way to access menu items from the keyboard. The user can use mnemonics to navigate through a menu by first activating the menu bar, then typing the mnemonic character. Progress indicates the mnemonic character by underlining it in the menu. For example, suppose the menu bar contained these elements: File, Edit, and Reports. Once the menu bar is active, the mnemonic to access the File menu is **f**, the Edit menu is **e**, and the Reports menu is **r**. By default, Progress assigns the first character in the item label as the mnemonic. However, you can also define it yourself by inserting an ampersand (&) before the letter in the label.

This code fragment defines the mnemonic for the Exit menu item:

```
DEFINE SUB-MENU sm-Open
   SUB-MENU sm-Table LABEL "Open"
   MENU-ITEM mi-Exit LABEL "E&xit".
```

By default, the mnemonic for Open is **o**.

11.3.4 Accelerators

An *accelerator* is a key or key combination that executes an item from a pull-down menu without the user having to pull down the menu. You can define an accelerator for a menu item with the ACCELERATOR option in the menu-item description. This is the syntax for the ACCELERATOR option.

SYNTAX

```
ACCELERATOR keylabel
```

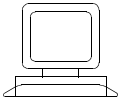
In this syntax, the value `keylabel` must be a character-string constant that evaluates to a valid Progress key label defined for your terminal in the PROTERMCAP file. You can modify the `keylabel` by specifying one or more of these keys: SHIFT, CONTROL, ALT. For example, you can specify "CTRL-N", "PAGE-UP", etc. Progress automatically displays the specified accelerator key label next to the menu item label, so the user knows what key is the accelerator. The user can then choose a menu item by pressing the specified keys.

Note that any key combination that you specify for the accelerator must also yield a recognizable control sequence for the terminal. Otherwise, you might define what appears to be a valid key label, but is rejected by the terminal at run time. A common example is a key label defined with "ESC", such as “ESC-N”.

For more information on key labels, terminals, and how they are specified in the PROTERMCAP file, see the *Progress Client Deployment Guide*.

For more information on keyboard events and keyboard event precedence, see the chapter on handling user input in the *Progress Programming Handbook*.

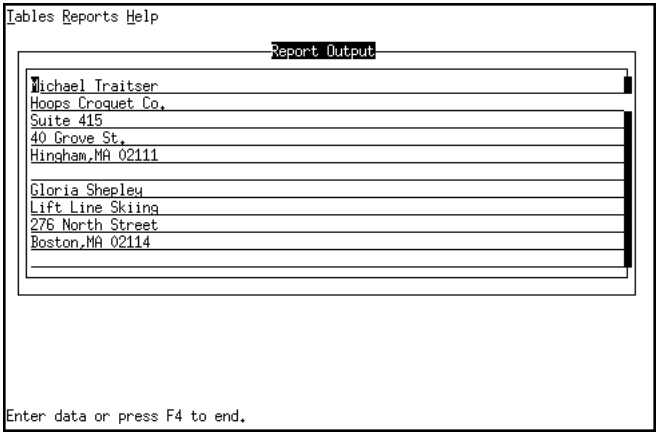
11.3.5 Programming Example



Exercise

Follow these steps to view the enhanced menu:

1 ♦ Open `lt-11-02.p` and run it. The display shown below appears:



2 ♦ Display the Reports pull-down menu. The third and fourth items are disabled, so you can't choose them.

3 ♦ Choose Mailing Labels. The mailing label frame appears.

4 ♦ Choose Output to Printer from the Reports menu.

5 ♦ Choose Mailing Labels again. An alert box appears informing you that the report printed. Choose OK to dismiss the alert box.

6 ♦ Browse through the various submenus. Notice that a letter in each submenu or menu item is underlined. The underlined letter is a mnemonic.

7 ♦ Choose Tables→ Exit, then press **SPACEBAR** to return to the Procedure Editor.

Following is the code for the enhanced version of the procedure:

lt-11-02.p

```
      /**********  DEFINE WIDGETS  **********/
      {lt-11-mn.i} /* Menu definitions */
      DEFINE VARIABLE Rep-Editor AS CHARACTER VIEW-AS EDITOR
          SCROLLBAR-VERTICAL SIZE 76 BY 13 .
      DEFINE VARIABLE Stat AS LOGICAL.

          /**********  DEFINE FRAMES  **********/
      DEFINE FRAME Frame1
          Rep-Editor WITH NO-LABELS ROW 2 CENTERED TITLE "Report Output"

          /**********  DEFINE TRIGGERS  **********/
      ON CHOOSE OF MENU-ITEM mi-Exit
          APPLY "CLOSE-WINDOW" TO DEFAULT-WINDOW.
      ON CHOOSE OF MENU-ITEM mi-Labels IN MENU sm-Reports
          RUN p-Report.

          /**********  MAIN LOGIC  **********/
      ASSIGN Rep-Editor:READ-ONLY IN FRAME Frame1 = YES
             Rep-Editor:FONT = 3.
      ENABLE ALL WITH FRAME Frame1.
      WAIT-FOR CHOOSE OF MENU-ITEM mi-Exit.

          /**********  INTERNAL PROCEDURES  **********/
      PROCEDURE p-Report:
/*1*/  IF MENU-ITEM mi-Print:CHECKED IN MENU mbar = NO THEN
           OUTPUT TO "tut-temp.txt".
      ELSE OUTPUT TO PRINTER.

      FOR EACH Customer FIELDS (Balance Postal-Code Contact Name Address
          Address2 City St) WHERE Balance >= 1400 BY Postal-Code
          WITH STREAM-IO:
          PUT Contact SKIP
              Name SKIP
              Address SKIP.
          IF Address2 NE "" THEN PUT Address2 SKIP.
          PUT City + "," + St + " " + STRING(Postal-Code, "99999")
              FORMAT "x(23)" SKIP(1).
          IF Address2 EQ "" THEN PUT SKIP(1).
      END.
      OUTPUT CLOSE.
/*2*/  IF MENU-ITEM mi-Print:CHECKED IN MENU mbar = NO THEN
        ASSIGN Stat = Rep-Editor:READ-FILE("tut-temp.txt")
            IN FRAME Frame1.
      ELSE
         MESSAGE "Report Printed" VIEW-AS ALERT-BOX MESSAGE BUTTONS OK.
      END PROCEDURE.
```

These notes help explain the code:

1. The CHECKED attribute applies only to toggle boxes and toggle box menu items. You can check it at any time to determine the on screen state of the menu item. If you wanted to write a trigger for the toggle box menu item, you would normally use the VALUE-CHANGED event function.

2. Here, the CHECKED attribute determines whether to read the report output file into the editor or to let the user know that the output went to the default printer.

11.4 Menu Attributes

Progress maintains attributes for menu widgets. Table 11–4, Table 11–5 and Table 11–6 list the specific attributes for menus, submenus, and menu items. For more information about these attributes, see the *Progress Language Reference*.

Table 11–4: Menu Attributes

Attribute	Type	Default	Readable	Setable
FIRST-CHILD	WIDGET-HANDLE	–	√	–
LAST-CHILD	WIDGET-HANDLE	–	√	–
OWNER	WIDGET-HANDLE	–	√	–
TITLE	CHARACTER	–	√	–
TYPE	CHARACTER	MENU	√	–

Table 11–5: Submenu Attributes

Attribute	Type	Default	Readable	Setable
FIRST-CHILD	WIDGET-HANDLE	–	√	–
LAST-CHILD	WIDGET-HANDLE	–	√	–
PARENT	WIDGET-HANDLE	–	√	√
TYPE	CHARACTER	SUB-MENU	√	–

Table 11–6: Menu Item Attributes

Attribute	Type	Default	Read-able	Setable
CHECKBOX	LOGICAL	FALSE	√	√
CHECKED	LOGICAL	?	√	√
LABEL	CHARACTER	–	√	√
SUBTYPE	CHARACTER	–	√	√
TYPE	CHARACTER	MENU-ITEM	√	–

Here are some examples of using menu attributes:

```
DISPLAY MENU sm-Reports:LABEL. /* Outputs a menu or submenu label * /

DISPLAY MENU-ITEM mi-Exit:LABEL. /* Outputs a menu item label * /

DISPLAY MENU-ITEM mi-Customer:LABEL IN MENU sm-Table.
                                        /* Outputs a menu item label.* /
```

11.5 Design Conventions

There are common practices to consider when designing menus. Following these guidelines gives your menus a common look and makes them predictable so that users can navigate unfamiliar applications more quickly. Here are some useful design conventions:

- Group menu items in submenus by task. Order the tasks by frequency of use, as well as the order in which they are used. Place the most frequently used menu items where users can easily access them—in general on the left and close to the top.
- Keep any destructive menu item (like delete) away from frequently used menu items.
- Disable menu items when they are not appropriate.
- When possible, provide alternate methods for accessing menus, like mnemonics and accelerator keys.
- Use an ellipsis (...) to indicate menu items that require further user input.
- Each menu bar should contain at minimum these three items: File, Edit, and Help.

- Place the Exit option as the last menu item on the first menu.
- Limit submenus to three levels or less, because multiple levels of submenus clutter the screen and make the menu difficult to use.

Practice Problems

To practice defining menu bars, submenus, and menu items, design a basic menu system for an application you plan to develop.

1. Begin planning at the lowest level with the menu items.
2. Design the submenus.
3. Associate submenus to a menu bar.

Now that you have a basic menu worked out, experiment by adding some advanced features, such as disabled menu items, mnemonics, and toggle boxes.

11.6 Summary

Menus create an interface that allows your application to take full advantage of the event-driven programming model. The user controls which tasks are accomplished and the order in which they're accomplished.

A menu consists of three parts:

- A menu bar at the top of the window is the highest level of the menu.
- Submenus open below the menu bar and offer the user additional submenus or items to choose from.
- Menu items are the individual options available to the user.

When you build a menu, you start with the lowest-level components. Building a menu involves three steps:

1. Define the submenus and the menu items they contain.
2. Define the menu bar.
3. Assign the menu bar to a window.

Once you've assembled the menu, you can assign functionality to it by adding triggers that execute when the user chooses the menu items.

Advanced Menu Features

Progress menus include the following features to make the menus easier to navigate:

- Disabled menu items let the user know which options are not available to them at certain points in the application.
- Toggle boxes can keep track of the user's decision regarding a certain option.
- Mnemonics provide keys that the user can use as shortcuts to choosing a menu item.

12

Managing Your Application

When you begin your own application development cycle, you'll reach a point where you have all your functionality and interface code stored in procedure files. Together, these procedure files make up the complete application. Your final task will be to make sure it all works smoothly together.

This chapter discusses the issues that arise when putting together a large-scale application. It summarizes many techniques you already know about in the context of application management. The chapter also covers several new techniques and language elements that are important in managing your application.

Specifically, the chapter covers:

- Managing procedure files
- Sharing resources
- Managing interfaces
- Managing application control

12.1 Managing Procedures

Organizing your procedure files and how they work together is the first step in managing your application development. This section describes the typical organization of an event-driven application and how to work around some common problems with larger applications.

12.1.1 Structuring an Event-driven Application

An event-driven application typically has a flat structure. There is a main procedure that creates the main user interface and presents the user with functionality choices in the form of buttons and menu commands. Normally, each button or menu command links to a trigger in the main procedure that runs an external procedure file. The procedure files are all very modular, performing a single task.

As your application grows, your tendency will be to define all resources that the individual procedures use in the main procedure file, including:

- Shared user interface elements, like frames and widgets
- Variables and other data structures
- Triggers
- Internal procedures that represent functions—pieces of common code that more than one external procedure relies on.

Figure 12–1 illustrates the flat structure of an event-driven interface.

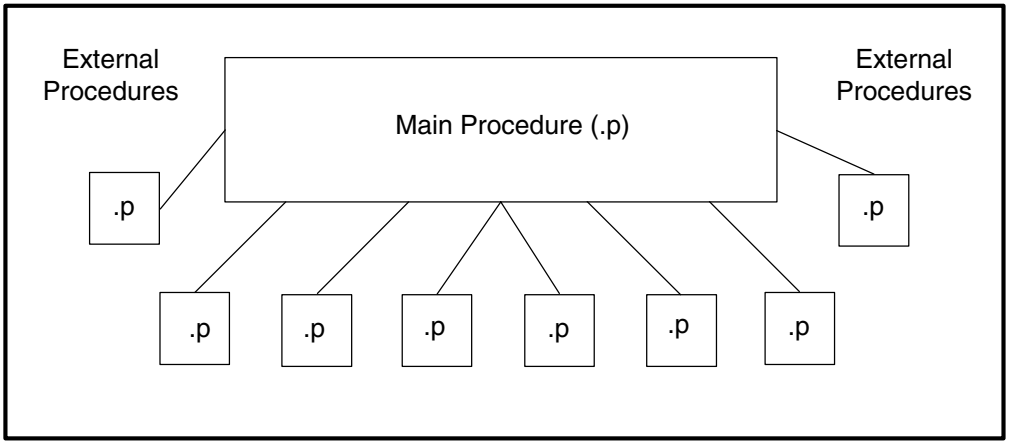


Figure 12–1: Flat Structure of an Event-driven Application

Because the event-driven model promotes giving the user access to most functionality at the top level of an application's interface, as opposed to a tree structure of screen menus, the main procedure that defines the top level of the application can get very large.

The tendency to grow the main procedure and keep the external procedures atomic modules can lead to problems. While you achieve modularity and ease of maintenance in the external procedures, the main procedure can become difficult to read and may tax the execution limits of your system.

One way to more easily maintain the main procedure might be to use include files to group related code. This may make the code easier to understand, but does not reduce the resource burden of having an overly large module of executable code.

12.1.2 Using Persistent Procedures

What you need is a technique for modularizing the content of the main procedure. For example, suppose you have a collection of internal procedures in the main procedure that acts as a function library for the external procedures. If you move those internal procedures into a separate procedure file and then call that procedure from the main procedure during startup, what happens?

Progress scopes all resources defined in a procedure to that procedure. The sum of the interface objects, data structures, and internal procedures is called the *context* of the procedure. The context of the procedure is only available during the life of the procedure. So, when the procedure of internal procedures mentioned above finishes executing, all the resources it defines are unscoped.

Progress has another type of procedure that creates a context that lasts beyond the execution of the procedure. This type of procedure is known as a *persistent procedure*. The context of a persistent procedure lasts until you explicitly delete it.

Persistent procedures promote modularity of application functionality. For example, you could have several persistent procedures that contained related groups of internal procedures. This technique essentially lets you have loadable and unloadable libraries of functions. This ability gives you the modularity you need and allows you to manage your system resources.

Later in the chapter, you'll learn the specifics of programming persistent procedures. For now, here's a quick description of how they work. First, you instantiate a procedure context by using the PERSISTENT option of the RUN statement. You can provide the RUN statement with a HANDLE variable to hold a pointer to the context. A procedure context created in this way is a self-contained context. Your code manages the communication between this module and any other modules currently running. The HANDLE variable is the bridge between contexts.

The Progress SESSION handle contains two attributes named FIRST-PROCEDURE and LAST-PROCEDURE. These two attributes allow you to access the beginning and end of the chain of persistent procedures. Together with the NEXT-SIBLING and PREV-SIBLING attributes of the procedure HANDLE variable, you can move through the chain of persistent contexts.

The DELETE PROCEDURE statement lets you delete persistent contexts.

Finally, you should know that you can run each procedure persistently many times. Each separate call with the RUN statement creates a separate context. For situations where you choose to use this technique, you'll need to create a management system so that you can differentiate between the different instances of the same procedure.

12.2 Sharing Resources

A large application has many resources that are potentially reusable by several modules of the application. This section describes different approaches to sharing resources with modules.

12.2.1 Using SHARED Resources

In the tutorial, you learned about SHARED variables. SHARED variables allow you to set aside an area in memory that is accessible by each procedure that defines the SHARED variable. The SHARED variable is available to all of these procedures, as long as the procedure context that initially created the SHARED variable is still available.

Progress provides the SHARED option for most resources. You can have SHARED variables, frames, buffers, queries, and streams. To implement a SHARED resource, see the *Progress Language Reference* for the appropriate syntax.

The advantage of SHARED resources is that they promote sharing resources and communicating data between application modules. SHARED resources also take a little less time to resolve at startup.

The disadvantage of SHARED resources is that they are not explicit and therefore not very portable. Since SHARED resources are available for both input and output operations, it may not be immediately clear to another programmer how a particular module interacts with the shared resource.

12.2.2 Using Input and Output Parameters

In the tutorial, you also saw how to define input and output parameters and supply values when calling runnable modules. Progress supports these types of data structures as parameters: variables, buffers, and handles.

Parameters are more precise in their use of resources and more readily understandable since they specify whether they are input, output, or both.

For most applications, and especially where resource concerns are an issue, parameters are the most common and important technique for sharing among modules.

12.2.3 Sharing with Persistent Procedures

The persistent procedure is different from parameters and SHARED resources, in that it makes an entire procedure context available to other modules.

It is important to remember that persistent procedures instantiate separate, individual contexts—not one large context in which each context becomes a part of the whole. By using the procedure HANDLE your code can make the separate procedures interact with each other.

The point of the persistent procedure is that what is available to be shared between modules is now under your control. The resources in a persistent procedure can be loaded and unloaded in bulk.

Persistent Procedure Syntax

The first step in creating a persistent procedure, is to define a HANDLE variable to hold the context reference as the following code fragment shows:

```
DEFINE VARIABLE proc-handle AS HANDLE.
```

The next step is to run an external procedure with the PERSISTENT option of the RUN statement. The diagram below shows the relevant RUN statement syntax.

SYNTAX

```
RUN extern-procedure
  [ PERSISTENT [ SET handle ] ] [ runtime-parameters ]
```

Progress sets the HANDLE variable you supply to the handle for the procedure context.

This code fragment shows a valid persistent procedure call:

```
RUN lt-12-02.p PERSISTENT SET proc-handle.
```

When you are through with a procedure, you need to delete it. This is the syntax for the DELETE PROCEDURE statement.

SYNTAX

```
DELETE PROCEDURE proc-handle
```

Here is an example of the delete procedure statement:

```
DELETE proc-handle NO-ERROR.
```

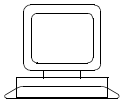
There are some other 4GL language elements that help you manage persistent procedures. Table 12–1 describes them.

Table 12–1: Language Elements Used with Persistent Procedures *(1 of 2)*

Element	Description
SESSION:FIRST-PROCEDURE	Use the FIRST-PROCEDURE attribute of the SESSION system handle to find the first persistent procedure in the current chain.
SESSION:LAST-PROCEDURE	Use the LAST-PROCEDURE attribute of the SESSION system handle to find the last persistent procedure in the current chain.
handle:FILE-NAME	Use the FILE-NAME attribute of a HANDLE variable to access the name of the external file that was used to create the procedure's context. FILE-NAME can be one way to identify a particular context if only one copy of the context can be running at a time.
handle:PRIVATE-DATA	Use the PRIVATE-DATA attribute of a HANDLE variable to store additional information about the procedure. For example, PRIVATE-DATA can be used in conjunction with FILE-NAME for procedures that can be run several times to uniquely identify each instance (context) of that procedure.

Table 12–1: Language Elements Used with Persistent Procedures *(2 of 2)*

Element	Description
handle:NEXT-SIBLING	Use the NEXT-SIBLING attribute of a HANDLE variable to find the next persistent procedure in the current chain. If NEXT-SIBLING is null, then the current procedure is the last procedure.
handle:PREV-SIBLING	Use the PREV-SIBLING attribute of a HANDLE variable to find the previous persistent procedure in the current chain. If PREV-SIBLING is null, then the current procedure is the first procedure.
VALID-HANDLE	Use the VALID-HANDLE function to check if a HANDLE variable is a valid reference to a current persistent context.
THIS-PROCEDURE	Use the THIS-PROCEDURE system handle to access the handle of the current context.

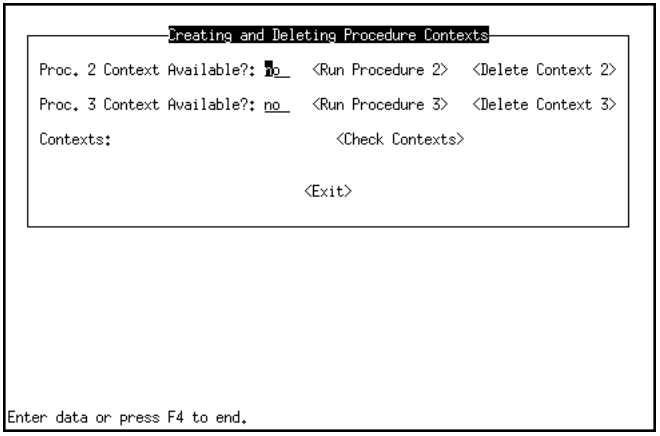


Exercise

Persistent Procedure Programming Example

Follow the steps below for a demonstration of persistent procedures.

1 ♦ Open `lt-12-01.p` and run it. The display shown below appears:



2 ♦ Choose Run Procedure 2. An alert box appears informing you that the context for procedure 2 is available. This alert box represents the body of an internal procedure stored in an external procedure file. The internal procedure was accessed and run from the main procedure after the execution of the external procedure was already complete.

3 ♦ Choose Check Contexts. The filenames of all persistent procedures appear in the display.

4 ♦ Choose Delete Context 2 and then choose Check Contexts again. The deleted context disappears from the list.

5 ♦ Choose Exit, and then press **SPACEBAR** to return to the Procedure Editor.

The code for the main procedure is on the next page. The two external procedures are simply MESSAGE statements inside internal procedures:

lt-12-01.p

```
      /**********  DEFINE WIDGETS  **********/
/*1*/ DEFINE VARIABLE h-pp2 AS HANDLE.
      DEFINE VARIABLE h-pp3 AS HANDLE.
      DEFINE VARIABLE h-proc AS HANDLE.
            .
            .
            .
      /**********  DEFINE TRIGGERS  **********/
      ON CHOOSE OF b-pp2-on DO:
/*2*/      IF NOT pp2-cnxt THEN DO:
/*3*/          RUN lt-12-02.p PERSISTENT SET h-pp2.
/*4*/          pp2-cnxt = VALID-HANDLE(h-pp2).
               DISPLAY pp2-cnxt WITH FRAME Frame1.
           END.
/*5*/        RUN p-message IN h-pp2.
       END.
      ON CHOOSE OF b-pp2-off DO:
/*6*/      DELETE PROCEDURE h-pp2 NO-ERROR.
          pp2-cnxt = VALID-HANDLE(h-pp2).
           DISPLAY pp2-cnxt WITH FRAME Frame1.
       END.
              .
              .
              .
/*7*/    ON CHOOSE OF b-num-cnxt DO:
         File-list = "".
         h-proc = SESSION:FIRST-PROCEDURE.
         REPEAT WHILE VALID-HANDLE(h-proc).
              File-list = File-list + " " + h-proc:FILE-NAME.
              h-proc = h-proc:NEXT-SIBLING.
         END.
        DISPLAY File-list WITH FRAME Frame1.
      END.
/*8*/    ON CHOOSE OF b-exit DO:
          IF pp2-cnxt THEN DELETE PROCEDURE h-pp2.
          IF pp3-cnxt THEN DELETE PROCEDURE h-pp3.
          APPLY "WINDOW-CLOSE" TO CURRENT-WINDOW.
        END.
       /**********  MAIN LOGIC  **********/
      DISPLAY pp2-cnxt pp3-cnxt WITH FRAME Frame1.
      ENABLE ALL WITH FRAME Frame1.
      WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

The points below help explain the code:

1. The HANDLE variables hold the pointers to the persistent procedures.
2. This IF statement prevents the user from establishing many separate contexts for each procedure.
3. The PERSISTENT option asks Progress to save the procedure, and the SET option provides the handle to the procedure.
4. The VALID-HANDLE function takes a HANDLE variable and determines if it is a legal reference to a current persistent procedure.
5. This RUN statement uses the IN `handle` syntax to access and run an internal procedure in the persistent procedure context.
6. The DELETE PROCEDURE statement removes the procedure.
7. This trigger runs through the current chain of persistent procedures and records the filenames of each in a string.
8. The exit trigger makes sure the procedures are destroyed before exiting the procedure. Because you stored the procedure HANDLE when you ran the procedure, you have programmatic control of that procedure. In this example, you use that HANDLE to explicitly delete any surviving procedure contexts before exiting the main procedure.

12.3 Managing Interfaces

The strategies you employ for creating interfaces impacts the usability of your application. The topics in this section present different interface designs, including:

- Single-frame interfaces
- Multi-frame interfaces
- Dialog boxes
- Alert boxes
- Simulated multi-window interfaces

12.3.1 Using Single-frame Interfaces

All of the interfaces in this tutorial have been either single-frame interfaces or single-frames with dialog boxes as secondary interfaces.

If your main interface is a single frame in a window, it can still be dynamic. If you rely on one frame, you can manipulate the widgets inside the frame to adapt the frame to the user's current task. For example, if your application is a database maintenance program, you could show a selection list populated with the table names for the selected database. Using techniques you learned in this tutorial, you could either repopulate the same selection list with valid data each time the user selects a different database, or you could have a selection list for each database and hide the others when appropriate.

Either of these techniques can work, but bear in mind that hidden widgets do use system resources—having a large number of them might be a programming and resource burden.

12.3.2 Using Multi-frame Interfaces

Multi-frame can mean two different things. First, it could mean that you have more than one active frame in a window. Second, it could mean that a window has several different frames, each one appropriate for a different mode or task of the main interface.

Also, you can choose from two basic methods of organizing multiple active frames in a window:

- Defining all the frames as independent siblings that are child widgets of the window.
- Defining frame families (see Chapter 3, "Programming the Progress Way"), where at least one frame is a parent that contains all the other frames as its child widgets.

In general, frame families solve a number of problems related to navigation and control in multiple frame interfaces. Among these are tabbing and conditional display. However, some code management is equally necessary for frame families and independent frames.

Tabbing in Multi-frame Windows

While there are several keyboard functions available to help a user navigate through an interface, most users rely on the TAB function. From a user's viewpoint, TAB moves input focus to the next widget in the interface. From a technical viewpoint, TAB moves input focus to the next widget in the **frame**. So, if you have two independent frames on screen, the user cannot get to the second frame by using the TAB key. Input focus cycles within the first frame. The user needs to invoke the NEXT-FRAME key function to move focus to another frame.

One way to avoid this problem is to write a trigger that simulates the response the user expects. When the user presses TAB in the last widget of a frame, you can suppress the default behavior and apply the NEXT-FRAME event function to the frame. This moves input focus to the first widget of the next frame. Here is an example:

```
ON TAB OF last-widget1 IN FRAME Frame1, last-widget2 IN FRAME Frame2
DO:
    APPLY "NEXT-FRAME" TO SELF. /* Move input focus to next frame  */
    RETURN NO-APPLY. /* Suppress normal TAB behavior. */
END.
```

However, a simpler method is to include all the frames in a single frame family. With this approach, all tabbing behavior is built in. The user can tab among all the frames contained by the parent frame. Focus automatically moves to the first or last field-level widget in the next or previous frame, respectively.

Storing Many Frames

You can design several frames, one for each task that the user can accomplish from the main interface. It might be helpful to store each of these frames in a separate file. They can be stored as include files or as runnable modules (if you included the executable code that goes with the frame). The Progress AppBuilder stores runnable interface code in procedure files with a .w extension. You can adopt this convention to help keep your files organized.

Frame Scoping

Just like records, frames have a life span, and that life span is referred to as the *frame scope*. The general rules for frame scope are:

- Frame scope begins from the first reference of a frame.
- Frame scope ends when the block that contained the first reference to the frame ends.
- Frames are available to any nested blocks within the block that contained the first reference.

If you violate these rules, Progress will notify you at compile time. In general, you won't need to worry too much about frame scope because the code techniques used with the event-driven programming model tend to scope all frames to a procedure block. Because the frames are scoped to the procedure block, all references to the frame within the procedure block are legal.

One of the more common scoping problems involves trying to reference a frame used by a control block from the main procedure. Here's an example:

```
DEFINE FRAME Frame2.
        .
        .
        .
FOR EACH Customer WITH FRAME Frame2:
  DISPLAY Customer.
END.
        .
        .
        .
DISPLAY Field1 WITH FRAME Frame2.
```

This is invalid code because of the attempt to reference Frame2, which is scoped to the FOR EACH block, outside of that block. If you need to reference the frame in both blocks, you have to make sure that the first reference is in the block that contains the FOR EACH block. The DEFINE FRAME statement is not considered a reference.

For more information on frame scoping, see the *Progress Programming Handbook*.

Accessing Frame Attributes

Like other widgets, frames have attributes that you can access, but the syntax for accessing a frame attribute is a little different. This is the normal widget attribute syntax.

SYNTAX

```
widget-name:attribute-name
```

For frames, this is the syntax.

SYNTAX

```
FRAME frame-name:attribute-name
```

Table 12–2 describes some of the basic frame attributes that you may want to reference.

Table 12–2: Frame Attributes

Attribute	Description
BOX	A LOGICAL value where TRUE indicates the default frame border is visible.
CENTERED	A LOGICAL value where TRUE indicates that the frame is centered in the window.
COLUMN	For frames, the COLUMN attribute holds the column position of the frame within the window.
HIDDEN	A LOGICAL value that determines whether or not Progress can display the frame implicitly. See the next section for a complete discussion.
ROW	For frames, the ROW attribute holds the row position of the frame within the window.
SIDE-LABELS	A LOGICAL value where TRUE indicates that the frame is in side-label mode and FALSE indicates that the frame is in column-label mode.
VISIBLE	A LOGICAL value that determines whether the frame is currently visible on the display.

Hiding Frames

In Chapter 3, "Programming the Progress Way," you learned about using the VIEW and HIDE statements. These two statements operate on the VISIBLE attribute, which all widgets have. The VISIBLE attribute is TRUE if the widget is currently visible in the display, and FALSE if it is not. It is sometimes useful to access the VISIBLE attribute directly to make widgets appear and disappear. VISIBLE does not give you complete control over a widget visibility, however, because Progress has default behaviors that can make a widget visible, even if you made it invisible. For example, displaying a frame makes all the widgets in that frame visible, including any child frames.

Progress provides the HIDDEN attribute to allow you to suppress implicit viewing of a widget. So to make a widget completely invisible, set VISIBLE to FALSE and HIDDEN to TRUE.

The HIDDEN attribute can be used to give your application a significant performance boost. When you start an application, or switch to a new interface within an application, the computer periodically redraws the screen. If the computer redraws the screen several times while you are still setting up the interface, you slow down the appearance of that interface to the user.

Use the HIDDEN attribute on a frame (or window) to make the frame immune to Progress implicit viewing behaviors. If the frame is invisible, so are all the widgets in the frame. With HIDDEN set to TRUE, you can populate a frame with widgets, display data in the frame, and enable the widgets without the frame becoming visible. Then when the frame is ready, use the VIEW FRAME statement to make the interface visible.

Here is an example of the HIDDEN attribute:

```
DEFINE FRAME Frame1
        .
        .
ASSIGN FRAME Frame1:HIDDEN = TRUE.
        .
        .
DISPLAY a b c WITH FRAME Frame1.
ENABLE ALL WITH FRAME Frame1.
VIEW FRAME Frame1.
```

In the first case, if the frames are stored separately as include files, then those frames need to be scoped to the main procedure so they are available. As mentioned earlier, in large applications this can make the main procedure overly large.

For runnable code modules, you can use external procedures that use the main interface (as opposed to a dialog box).

Enabling Input in Frame Families

Typically, to enable all field-level widgets in a frame for input, you only have to execute a single ENABLE statement as shown in the following code fragment:

```
ENABLE ALL WITH FRAME Frame1.
```

However, if this frame contains child frames that own some of the input widgets you want enabled, these input widgets do not become enabled by this single statement. You must enable input for each frame separately, even though they are in a single frame family:

```
DEFINE FRAME Frame1
        .
        .
DEFINE FRAME Frame2
        .
        .
FRAME Frame2:PARENT = FRAME Frame1:HANDLE.

ENABLE ALL WITH FRAME Frame1.
ENABLE ALL WITH FRAME Frame2.
```

While this might seem like unnecessarily redundant code, it does afford a measure of control where you choose to group input fields within child frames by common function. You might then enable and disable each child frame (group of fields) under the parent frame according to input received by fields owned by the parent frame itself.

12.3.3 Using Dialog Boxes

The dialog box is your main tool for implementing *modal* interface modules. Modal interfaces require the user to complete or cancel the current operation before moving on. Modal interfaces focus the user's attention on the current task, helping to ensure valid data and minimizing the time that Progress holds onto individual records. This behavior makes modal interfaces a natural choice for working with records in a database application.

Typically, a dialog box is the interface to some functionality that the user launches by choosing a button or menu command. This means that a dialog box is usually associated with an external procedure. The Progress AppBuilder stores these secondary interface modules with a .w file extension.

In the tutorial, you first saw the dialog box when you learned about the UPDATE statement. A dialog box overlays the main interface and blocks interaction with the main interface until the user dismisses the dialog box. In the case of the UPDATE statement, the main interface presents data for the user to **view and change**. Without an UPDATE statement, the user can perform an explicit action (choosing a button) to bring up the dialog box. The dialog box then gives the user the ability to change the data.

This example illustrates the concept behind a dialog box: a dialog box is an event-driven tool for implementing *modes*. Normally, event-driven programming strives to let users work with an application in any way that suits them. They can begin one task, start another, go back to the first, start a third—whatever makes sense to them. Event-driven programming attempts to be *modeless*, which means that a user can work with several parts (modes) of an application simultaneously.

In many instances, however, it is clearly desirable, or even necessary, to have a user start and complete a particular task before starting another. That's where the dialog box comes in. When you use a dialog box, you are defining a mode—you are being *modal*. To go back to the UPDATE example, that technique for working with data had two clearly defined modes: viewing data and changing data. By using the dialog box, you enforced the distinction between the two modes. In event-driven programming, it's also possible to use the same interface to view data and allow changes to the data. This technique is a modeless one.

Figure 12–2 illustrates a modeless technique.

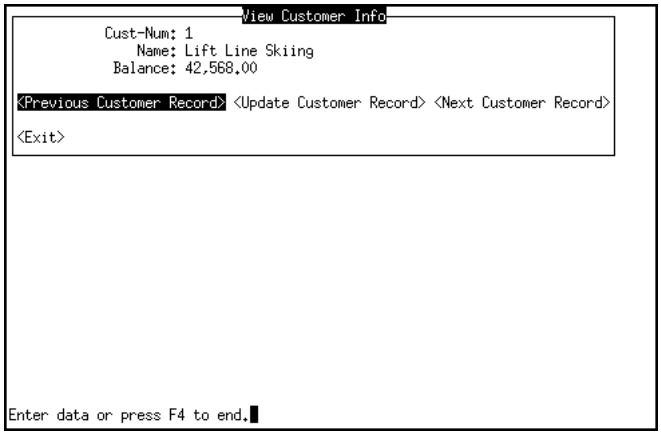


Figure 12–2: A Modeless Interface

Figure 12–3 illustrates a modal technique.

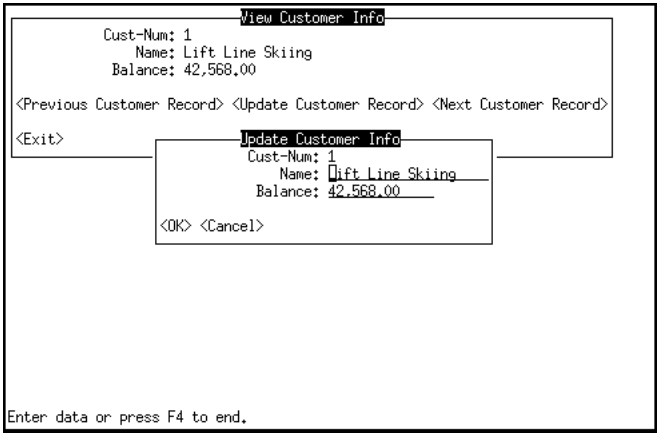


Figure 12–3: A Modal Interface

Syntactically, the difference between a frame and a dialog box is just the VIEW-AS DIALOG-BOX option on the frame phrase. Also, while a dialog box can contain a frame family, a frame family cannot contain a dialog box. The dialog box must be the top-level parent.

The idea of a dialog box is common to all event-driven programming, and there are some widely implemented standards that go with dialog boxes. The first is the OK and Cancel buttons that you see in Figure 12–3. These buttons implement the idea that a user must dismiss a dialog box before the user can continue:

- A user who chooses the OK button is saying "**dismiss** the dialog box **and save** any changes made to data in the dialog box."
- A user who chooses the Cancel button is saying "**dismiss** the dialog box **and discard** any changes made to the data."

Fortunately, the Progress programming model contains these two concepts, which apply to all Progress interfaces, not just dialog boxes. The GO event function maps to the idea of "dismiss and save," while the ENDKEY event function maps to the idea of "dismiss and discard." Using the AUTO-GO and AUTO-ENDKEY options on the DEFINE BUTTON statement for OK and Cancel maps the GO and END-ERROR event functions to the buttons.

12.3.4 Using Alert Boxes

An alert box is a dialog box for the message area of a window widget. An alert box is like a dialog box in that an alert box blocks the rest of the interface until the user acknowledges the alert box. When a message is important enough to interrupt the user, then the alert box is the right interface component to use. Here's the alert box syntax, which is included in the MESSAGE statement.

SYNTAX

```
[  VIEW-AS ALERT-BOX alert-type BUTTONS button-set
   [ TITLE title-string ]
]
```

Table 12–3 describes the alert box options.

Table 12–3: Alert Box Options

Component	Description
alert-type	Different environments define different types of alert boxes for use in certain circumstances. This ensures that all applications running in the environment use the same graphical cues for similar situations. This kind of uniformity helps the user recognize the seriousness of the message. The three most common types are: • **MESSAGE** — Specify this keyword when you want to communicate important information about the normal operations of an application. For example, "The new records have been added to the database." • **QUESTION** — Specify this keyword when you need to have the user answer a yes/no question before the application can proceed or you want the user to confirm a decision. For example, "Do you really want to delete the record?" • **INFORMATION** — Specify this keyword when you want to give the user information. Use this type to provide the user with information they specifically request. For example, you might have a button named Version. When the user chooses the button, an alert box comes up displaying the version number of your software.
button-set	While the alert-type syntax lets you specify the type of message to send, the BUTTON-SET syntax lets you choose the method for user response to the alert box. The most common button sets are: • **OK** — Use the OK button set with both the MESSAGE and INFORMATION alert box types. The user chooses the OK button to dismiss the alert box. • **YES-NO** — Use this button set with the QUESTION alert box type. The alert box will return the logical value YES or NO depending on the choice of the user to a LOGICAL field specified in the MESSAGE statement. See below for a description of this technique.
TITLE	Like a frame or dialog box, an alert box can have a title. Good style dictates that you always include a descriptive title.

Returning a Logical Value from an Alert Box

Earlier, you ran a code example that deleted records. In that example, you saw an alert box used to make the user confirm a deletion request. Figure 12–4 shows that alert box.

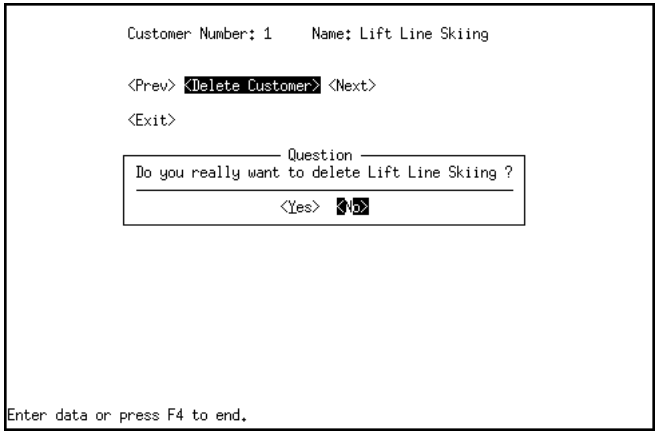


Figure 12–4: QUESTION Alert Box

This is the QUESTION alert box type and it uses the YES-NO button set.

The code fragment below shows the relevant part of procedure `lt-08-06.p`:

```
              .
              .
              .
/*1*/  DEFINE VARIABLE Answer AS LOGICAL.
              .
              .
              .
      ON CHOOSE OF btn-Delete
      DO:
/*2*/      MESSAGE "Do you really want to delete" Customer.Name "?"
             VIEW-AS ALERT-BOX QUESTION BUTTONS YES-NO
/*3*/      UPDATE Answer.
/*4*/        IF Answer THEN DO:
             DELETE Customer.
             GET NEXT Cust-Query.
            IF NOT AVAILABLE(Customer) THEN GET FIRST Cust-Query.
            DISPLAY Customer.Cust-Num Customer.Name WITH FRAME Frame1.
          END. /* IF Answer */
      END. /* ON CHOOSE OF btn-Delete */
              .
              .
              .
```

These notes help explain the code:

1. You need to define a variable before you use the alert box to store the user's response.
2. The special VIEW-AS syntax on a MESSAGE statement re-routes a message from the status area to an alert box.
3. This UPDATE is an **option** of the MESSAGE statement (note that there is no period on the line before). Use this option to specify the variable that will hold the user's response.
4. If the user answers YES, then the DELETE statement executes.

12.3.5 Referencing Windows

Progress defines a *window widget* as the workspace of your application. Each time you start a Progress session, it automatically creates a default window. In character mode, the default window is the only window you use and is basically equivalent to your screen.

When you are working with the default window, you reference it by using the DEFAULT-WINDOW system handle. This system handle holds the widget handle of the default window.

The default window has message and status areas, but you can override these.

Although you have only one window widget in character mode, you can partially simulate multiple windows by assigning alternate menu bars to the default window. For more information, see Chapter 11, "Building Menus."

For a complete discussion of windows and window programming techniques, see the *Progress Programming Handbook*.

12.4 Managing Application Control

Managing what choices a user has available is an important part of an event-driven application. This section discusses some techniques to help you provide the right choices.

12.4.1 Using Basic Control Mechanisms

Basic control is the ability to put up a functioning interface and allow your users to interact with it. The WAIT–FOR statement is the statement that blocks the execution of an application and specifies an end condition that signals Progress to end the current module and continue processing or exit.

Only one WAIT-FOR statement should be active at once. Multiple WAIT-FOR statements can interfere with each other, unless you take explicit control over enabling and disabling all widgets to prevent conflicts.

It is also true that the implied execution blocking that comes with the UPDATE statement and related statements should not be mixed with a WAIT-FOR in the same interface, for the same reasons. UPDATE statements with dialog boxes work well together because the disabling of the main interface by the dialog box allows the UPDATE to function without interference from the main WAIT-FOR.

In most cases, an application should have only a single active WAIT-FOR. Secondary interfaces should use another technique to block execution.

12.4.2 Using Dialog Boxes with WAIT–FOR Statements

In Chapter 10, "Creating Reports," the tutorial used a technique that violated the advice from the last section. The programming examples in that chapter used a dialog box to display static report output and blocked the interface with a nested WAIT-FOR statement.

In this case, the nested WAIT-FOR works, because:

- The dialog box has no updatable widgets and therefore has no need for an UPDATE statement.
- The dialog box disables the main interface, avoiding conflict with the main WAIT-FOR.
- The code took care of explicitly dismissing the dialog box. Without the HIDE statement immediately following the WAIT-FOR, the code would yield an error message.

12.5 Summary

Event-driven programs strive to make as much functionality available from the top-level of an application as possible. This leads to a characteristic flat application structure consisting of one large main procedure and several atomic modules that reside in separate procedure files. Having a large main procedure may tax the execution limits of your system. To alleviate this problem, you can use persistent procedures to create loadable and unloadable modules of resources.

SHARED resources, including variables, frames, buffers, queries, and streams, allow you to create individual resources that can be available to many procedures.

Persistent procedures create resources that do not disappear when the creating procedure ends. Persistent procedure contexts must be destroyed programmatically when the application no longer needs them.

Progress supports a variety of interface designs, including:

- Single-frame interfaces
- Multi-frame interfaces
- Dialog boxes

Like other widgets, frames and windows have attributes that you can manipulate to fine tune your interface.

A successful application that mingles more than one interface needs to manage application control by using WAIT-FOR statements, implied application blocking, and persistent procedures in such a way as to give the user freedom to move through the application without conflict.

13

Where to Go from Here

The tutorial introduced and developed several broad programming themes. The themes and coverage were chosen to give you a solid foundation in the Progress 4GL. With what you've learned, you could certainly begin to develop Progress applications. However, Progress contains so many features and language options that there's a lot more to learn. This chapter provides a conceptual map for learning more about the programming areas that are important to you. The topics covered are:

- Progress Tools
- Designing and implementing databases
- Learning the language
- Multi-user programming
- Data integrity
- Record reading
- Working with large data sets
- Multiple window applications
- Menus
- Dynamic widgets and direct manipulation
- Graphical features

13.1 Progress Tools

In the tutorial, you learned just enough about the Procedure Editor and Data Dictionary to help you write procedures. These two basic tools have many more features that you'll want to use. These tools are covered in the *Progress Basic Development Tools* on-line help manual.

In addition to the two basic tools, there are two more that you should be aware of: the AppBuilder and the the Debugger. Whether or not these tools are available to you depends on the specific package you purchased and your development environment. Here's a quick summary of these tools and their documentation.

AppBuilder

The AppBuilder is a tool for graphical environments that allows you to draw an interface and the AppBuilder codes the interface for you. Of course, this is only the beginning of what the AppBuilder can do. The tool takes full advantage of the graphical features of a GUI to make creating Progress applications easier. The tool includes a section editor where you can write code, including triggers.

The AppBuilder includes an implementation of encapsulated objects called SmartObjects. These contain a variety of objects that you can combine to quickly build graphical applications. When you use SmartObjects to build your application, you create an easily maintainable body of code based on reusable objects. You can change a SmartObject and affect that change for every instance of that SmartObject.

If you program in a graphical environment, the AppBuilder could easily become the hub of your development environment. However, keep in mind that the AppBuilder can also be used to design character environments, and that SmartObjects can be used in character environments. For an introduction to the the AppBuilder and to SmartObjects, see the *Progress Application Development Environment — Getting Started* manual. For an in-depth treatment of the AppBuilder, see the *Progress AppBuilder Developer's Guide*. For information on porting Progress applications, especially those containing SmartObjects, between Windows and character interfaces, see the *Progress Portability Guide*.

Debugger

When your application grows and consists of many procedure files, debugging may take up more of your time. The Progress Debugger can reduce that burden. The Debugger is documented in the *Progress Debugger Guide*.

Other Tools

Progress has a number of other tools that serve very specific purposes. If you need to use one of these tools, the Progress documentation set will point you towards it.

13.2 Designing Databases

As the tutorial pointed out early on, designing effective databases is an extensive topic in itself. Chances are, if you have experience with other database systems, that you already know about database design. In this case, you can probably gain the knowledge you need to implement your database ideas by reading the Data Dictionary documentation found in the *Progress Basic Development Tools* on-line help manual.

On the other hand, if you want to expand your knowledge of what makes a good database, then you'll want to read the *Progress Database Design Guide*.

13.3 Learning the Language

Just about everything you learned in the tutorial, from conceptual discussions to language syntax diagrams, covers only a portion of the features and options that Progress offers. When you begin to develop your application and start to encounter programming issues not covered specifically in the tutorial, you have two choices. First, if you want a thorough discussion of the topic, you can turn to the *Progress Programming Handbook*. If you're interested in specific usage of a particular syntax, turn to the *Progress Language Reference*.

13.4 Multi-user Programming

Perhaps the most significant topic that the tutorial does not cover is programming applications that can be accessed by more than one user at a time. The whole topic of database access needs to be revisited if you are programming for multi-user access. The central concept behind this discussion is the idea of record locking. What should Progress do if more than one user wants to access the same record? Basically, Progress can lock a record from being accessed by a user while it is in use by another user. Progress has default locking behaviors designed to prevent the kind of database corruption that can come from many users all trying to write data at the same time. However, you can also take control of locking behavior to give your application the amount of protection it needs.

For a discussion of locking and other multi-user programming issues, see the *Progress Programming Handbook*.

13.5 Data Integrity

Protecting your data is a vital concern, and this topic encompasses different concepts. First, you don't want users to enter illegal values. So, you want to check or *validate* new data before it gets to the database. The tutorial touched on some of the ways you can validate data, like validation expressions in the database or database triggers. On a lower level, you can use triggers in your interface to check data as the user enters it. To learn more about validation, see both the *Progress Database Design Guide* and the *Progress Programming Handbook*.

Next, if you begin to write a set of changes to a database, you need to make sure that either the whole set is written to the database or the whole set is not written. Incomplete database writes can cause database corruption, incomplete data, or conflicting data.

A set of data slated for writing to the database is known as a *transaction*. A transaction may be interrupted by some user actions or system errors. Progress has many valuable default behaviors that commit or roll back transactions in response to interruptions. Understanding when Progress starts and ends transactions and how Progress decides to commit or roll back transactions is very important for mission critical applications. See the *Progress Programming Handbook* for this information.

13.6 Record Reading

Two chapters of the tutorial were devoted to data handling and record reading. In addition to this material, multi-user programming and data integrity issues affect the techniques you use to access records. Once you understand this material and the data-access material in the *Progress Programming Handbook*, you can apply the Progress features that work best for your programming needs.

Developing a sophisticated understanding of which statements and which options provide the right amount of flexibility, the right amount of security, and the optimum efficiency is an effort that can only pay off with better applications.

13.7 Working with Large Data Sets

When you learned about working with values, you learned about variables, shared variables, and extent variables. Variables, however, are tools for working with just a few values at a time. Some parts of your application may need to work with larger data sets—either temporary data you create in your application or temporary copies of database tables.

Progress has two important components for working with large data sets: temporary tables and work files. Understanding when to use one or the other is a topic covered in the *Progress Programming Handbook*.

You also learned about buffers and how Progress manages them for you. However, Progress also has the following features:

- Named buffers (DEFINE BUFFER)
- Shared buffers (DEFINE NEW SHARED BUFFER)
- Buffer parameters (DEFINE BUFFER PARAMETER)

You can find information on these language statements in the *Progress Language Reference*.

13.8 Multiple-window Applications

Multiple-window applications are possible with Progress in the graphical environments that support them. Using more than one window adds another complete set of programming techniques and interface conventions that are covered in the *Progress Programming Handbook*.

13.9 Menus

You learned how to define the menu-bar type of menu in the tutorial. Another type of menu that you might want to use is known as a pop-up menu. A popup menu is associated with a widget and provides options that affect that widget. For example, you might want to have a pop-up widget on a frame. See the *Progress Programming Handbook* section on menus for a complete description of pop-up menus.

13.10 Dynamic Widgets and Direct Manipulation

Two advanced topics that the tutorial did not go into are dynamic widgets and direct manipulation. Both of these concepts are covered in the *Progress Programming Handbook*.

All of the widget syntax you learned about in the tutorial produces static widgets. A *static widget* is one that is defined before use. Static syntax always includes the keyword DEFINE. Static widgets are most frequently used because Progress handles most of the work of displaying and updating these widgets. However, you may have a programming situation where you cannot predict the number of widgets you may need. Then you can use the dynamic syntax of a widget, which includes the keyword CREATE, to create the widgets at run time. *Dynamic widgets* require more explicit maintenance than static widgets.

Another advanced use of widgets available in graphical environments is to allow users to directly change physical attributes of widgets. For example, you may want to let your users rearrange the positions of buttons on a particular interface. Implementing direct manipulation involves working with widget attributes and special events that you have not yet learned about. The *Progress Programming Handbook* is your first step in learning about direct manipulation.

13.11 Graphical Features

If you are programming for a graphical environment, you have language options for using:

- Graphical images
- Colors
- Fonts

Once you've created a functionally sound application that offers an intuitive interface, you can begin to add images, colors, and fonts to create a unique and pleasant style.

The image widget allows you to place graphic files into your application. The image widget also has syntax options that allow some interesting uses.

Most interface components support syntax options that give you some, or total, control over color. Interface components that include text may also have options for specifying font choices and font styles.

For pursuing any of these programming issues, your best bet is to look up the specific syntax in the *Progress Language Reference*.

13.12 Internationalization and Localization

Finally, if you are programming for users of different nations or cultures, Progress offers some of the most complete feature sets for internationalizing or localizing your applications:

- *Internationalization* (I18N) is the process by which you design and program for a non-specific linguistic and cultural user base.
- *Localization* (L10N) is the process by which you tailor your internationalized application for a specific linguistic and cultural user base.

A *localized application* is an application that you have customized for a specific region. The region need not be another country; it can be a region characterized by a language, such as the French-speaking part of Switzerland, or a region characterized by different business practices, such as a province that has a different tax structure.

Progress facilitates both these processes by allowing you to program multiple display formats for:

- Dates
- Time of day
- Numeric values
- Currency values
- Addresss and phone numbers
- Measurements
- Form sizes
- Interface designs
- Labels, titles, and messages

Progress also provides access to numerous code pages for many different natural languages throughout the world. This includes double-byte support for languages such as Japanese and Chinese. Progress also provides a collation table mechanism that allows you to specify a sort order for the code pages you access.

You can specify different code pages for different functional areas of an application, such as database, memory, and stream I/O. In addition, you can code string functions that translate between code pages in memory and that properly index characters in single or double-byte strings.

You can also manage the translation of strings in your application independently of the programming by using the Progress Translation Manager and the Progress Visual Translator. These tools allow you to translate all the strings that you specify as translatable into alternate languages. After compiling your application for each language translation, you can run under each language without recompiling.

For more information on the general 4GL features that promote I18N and L10N, see the *Progress Programming Handbook*. For more information on the Progress Translation Manager and Visual Translator, see the *Progress Translation Manager Guide* and the *Progress Visual Translator Guide*.

Glossary

Abbreviated Index

Uses just the first few characters of a field, if the field is a character data field. Indexes not comprised of character data require an exact match.

Accelerator Keys

Function and special key combinations that you can press to choose a menu option.

Active Database

The database that is currently in use. If you have more than one database, the Data Dictionary requires you to select an active, working database.

Aggregate Functions

A set of functions that allow you to evaluate groups of values.

Aggregate Phrase

A Progress language element that identifies one or more values to be calculated based on a change in a break group.

Alert Box

A window that appears on the screen requesting user response. The user must dismiss the alert box before the application can continue.

Application

A set of programming language instructions that accomplishes a specific task. An application can be created from Progress procedures.

Argument

A piece of data that a calling procedure gives to a called procedure. Progress evaluates the passed data at compilation time.

Array

A field or variable with multiple elements. Each element in an array has the same data type.

Array Extent

The number of elements contained in the array.

Assignment Operator

The equal sign (=).

Attribute

A value associated with a widget or system handle. The value represents an aspect (e.g. size or color), state (e.g. visibility), or capability (e.g. to allow scrolling).

Block

A series of statements that Progress treats as a single unit. Typically, each block begins with a block header statement and concludes with an END statement.

Block Header

The statement that begins a block. It is different from other kinds of statements in two ways: it ends with a colon (:) (all other statements end with a period) and it can have a label (the label also ends with a colon).

Block Label

Text in a procedure block that identifies the block.

Box

In a frame, the four lines around the outside of the frame that designate its boundary. By default, frames have boxes; however, an application can turn off the box.

Break Group

A set of records having a common value in a certain database field. Break groups are used in reports to display file and record relationships.

Browse Widget

A widget that displays the results of a database query. There are two types of browse widget: updateable and read-only. An updateable browse lets the user view, update, add, and delete data in records. A read-only browse is used for display only. It shares properties of a container and field-level widget.

Buffer

A small amount of memory used as a temporary storage area for data during input or output operations. See also **data buffer**, **record buffer**, **screen buffer**. See also **edit buffer**.

Button Widget

A field-level widget that the user can choose to direct an application to execute a trigger or control the interface.

Character Client

A combination of hardware and software components that support a character-based interface. The following platforms support Progress running in character mode: UNIX, Windows NT, and Windows 95. Windows NT and Windows 95 only support the character mode by enabling a character client to run in DOS only when DOS is running under one of these 32-bit Windows operating systems.

Character Constant

A value made up of non-numeric or character data, or a combination of numeric and non-numeric data that remains unchanged during a procedure. Character constants must be enclosed in quotation marks when used in a procedure.

Character Data Type

A property of a field or variable that determines that the data stored there can be of the character data type.

Character Field

A field having a character data type.

Child Frame

A frame parented to (contained by) another frame. See also **parent frame**.

Child Window

A window parented to (owned by) another window. See also **parent window**.

Child Window Family

A child window and all of its descendant windows.

Column

1.) A component of a record, a field. 2.) A vertically aligned set of character cells in the character coordinate system.

Column Label

A label displayed above a column of data (field values). This label is useful for creating lists of values, like columnar reports. See also **side labels**.

Combo Box Widget

A field-level widget that presents a scrolling list of character strings. The list is only visible when the user chooses the button next to the combo box. When the user selects a value, the combo box closes and displays only the selected value.

Comparison Expression

A combination of constants, variables, operators, and parentheses used to compare values.

Concatenation Operator

The plus sign (+) used to link or join two or more character strings into a single character string.

Conditional Processing

A means of processing based on one or more logical expressions.

Connected Database

A database that is accessible and can be worked on in the Data Dictionary.

Constant

A value that remains unchanged during the execution of a program. A constant can be of any data type: character, date, decimal, integer or logical.

Container Widget

A widget that can contain other widgets. For example, a frame is a container widget.

Current Buffer

The contents displayed in the procedure area of the Procedure Editor.

Current Window

1.) The window in which one is working and so indicated by a filled-in window title bar.
2.) The window that contains the current widget.

Current Widget

The widget that has input focus.

Cycling Sequence

A sequence that begins at an initial value and increments in one direction until it reaches the designated limit.

Database

A collection of data organized in logically related tables or fields that can be accessed or retrieved.

Database Event

An action performed against a database, such as finding or writing to a record.

Database Object

A component of the database that is defined in the database schema such as a field or a table.

Database Properties

The schema definitions that define the database, such as the database name, the database type, and so on.

Database Trigger

A piece of code that an application associates with a component of a database and a database action (such as writing a record). When the action occurs on the component, Progress locates this piece of code and executes it on behalf of the application.

Data Dictionary

An interactive tool that enables you to create, modify, and delete Progress database objects. You can also use it to generate database reports.

Data Type

A property of a field or variable that determines the nature of data that can be stored there (integers as opposed to characters, for example). Progress supports five data types: character, integer, decimal, date, and logical.

DATE Data Type

A property of a field or variable that determines that the data stored there can be a date. In Progress, a date data type may contain a date and time from 1/1/32768 B.C. through 12/31/32767 A.D.

Date Field

A field having a date data type. You can specify dates in this century with either a two-digit year, such as 8/9/90, or a four-digit year (8/9/1990). Dates in other centuries require a four-digit year.

Decimal Data Type

A property of a field or variable that determines that the data stored there can be a decimal.

Decimal Field

A field having a decimal data type. A decimal field can contain decimal numbers up to 50 digits in length. Progress allows up to 10 digits to the right of the decimal point.

Dialog Box Widget

A container widget that either notifies the user of important information or requests more information. The main interface is disabled until the user is done working with the dialog box. A dialog box is a frame, but it has some properties of a window. Dialog boxes have modal properties.

Directory

A file-system object that lists or contains files and, possibly, other directories.

Display Format

The way data appears on screen and in printed reports. Progress automatically supplies a default display format for each data type, but you can change that default format.

Down Frame

A frame that displays multiple records, one per line, one after another. See also **frame**.

Edit Buffer

A temporary work area for procedures under construction in the Procedure Editor. The Procedure Editor allows you to open several buffers simultaneously. If a buffer has no name assigned, it appears as “Untitled” plus a number that makes it unique (Untitled:1).

Editor Widget

A field-level widget that allows editing of large character variables or database fields. By default, editor widgets support features such as cut, copy, paste, and word-wrap.

Event Action

Any simple user interaction, like a single keystroke.

Event Function

An abstraction of one or more event actions.

Event-driven Programming

Think of event-driven programming as *bottom-up programming*, because you define the smaller components before you define the larger components.

Expression

In a program, a combination of constants, variables, operations, and parentheses used to perform a desired computation.

External Procedure

A group of Progress 4GL statements that you store as an operating-system file.

Field

1.) The “atomic particle” of a database-that is, the smallest unit of useful information.
2.) A component of a record that holds a data value. Also called a column.

Field-level Widget

Any widget that can be placed in a frame.

Field Label

A label on the screen or printed reports that identifies the field. If you do not supply a label, Progress uses the field name as the label.

Field Properties

The schema definitions that define the field, such as the field name, the field type, and so on.

Fill-in Field Widget

A field-level widget that handles text entry and display. The fill-in field is the default widget for displaying a variable or database field. Also known as a **fill-in**.

Footer

Text, such as a page number, placed at the bottom of each page of a report.

Format Phrase

A Progress language description of the display characteristics for a field, variable, or expression.

Frame

A container widget composed of a display area within a window that Progress uses to display field-level widgets.

Frame Family

A related collection of parent and child frames.

Frame Scope

The range or extent of a frame’s availability within a procedure.

Function

A prepackaged solution to a task that can be used in an expression-a "shortcut" to handling tasks. Progress supports two types of functions: pre-defined function that the Progress 4GL supplies, and user-defined function that allows you to create functions based on specific programming tasks you want to solve.

Function Keys

Special keys to which Progress assigns certain actions. In procedures, you can reassign these actions to different keys.

Graphical User Interface (GUI)

Features high-resolution graphics and an additional user input device called a mouse.

Handle

An internal identifier for a widget. See also **system handle** and **widget handle**.

Header

Text, such as a page number, title, or date, placed at the top of each page of a report.

Icon

A small, rectangular graphic image that represents a window when the user has "minimized" (or "iconified") it.

Image

A field-level widget used to display an image from a bitmap file.

Include File

A separate file containing Progress code that you can call from other procedures by placing the filename in braces within the procedure. Include files have a .i extension.

Index

A directory or table containing 1) the field or fields identifying the records in a file, and 2) the locations where the records are stored.

Index Properties

The schema definitions that define the index, such as the index name, the fields it comprises, and so on.

Initial Value

The value to which a field is set when it's created.

Inner Join

A join where records from the first (left-hand) table (or join) are only returned to the results list that also return a corresponding record from the second (right-hand) table.

Input Focus

The ability for a widget to receive user-interface events.

Integer Data Type

A property of a field or variable that determines that the data stored there can be an integer.

Integer Field

A field having an integer data type.

Internal Procedure

Code blocks that you write and compile as part of a larger containing procedure.

Iteration

A repetition of a procedure block.

Join

A type of query operation that requests information from two tables or joins at a time. The output is a results list where each row contains a record from each joined table. A join can only occur when the two queried tables or joins have a common field.

Join Condition

The criteria used to relate the fields of one table in a join with those of another.

Key Functions

Tool functions that map to a keystroke.

Key Index

A primary, unique index field in a table.

Label

Text that appears with a field or variable when it is displayed.

Left Outer Join

A join that returns all the records of the first (left-hand) table, including the records from the second (right-hand) table for an inner join and null field values for all records in the right-hand table that have no corresponding records in the left-hand table.

Local Variable

A variable that you can use only within the procedure in which you defined it. The data in the variable is no longer available when you exit the procedure.

Literal

An alphanumeric constant. Literals can be character strings, numerics, dates, or logical data.

Logical Data Type

Can be one of two values consisting of yes/no, true/false, or a logical value pair.

Logical Database Name

The name of a physically connected database. The logical database name is used to resolve ambiguous database references.

Logical Expression

An expression that, when evaluated, yields true or false.

Logical Field

A field having a logical data type.

Logical Operator

An operator such as AND, OR, or NOT used in an expression that yields a true or false value.

Menu

A widget containing a list of commands that users frequently use. The two kinds of menu widgets available are menu bars and submenus.

Menu Accelerator Keys

A key or key combination which when pressed invokes an associated menu item. See also **accelerator key**.

Menu Bar

A horizontal bar displayed at the top of a window. The menu bar contains menu labels arranged horizontally. When you select a menu label, a pull-down menu containing a vertically arranged list of items is displayed.

Menu Item

A menu component that the user can choose to direct an application to perform some action or task.

Menu Titles

The text strings in a menu bar.

Menu Owner

The widget a menu is associated with. For menu bars, the owner is always a window.

Message Area of Screen

Three lines at the bottom of a Progress screen. The first two lines are for procedure-specific messages. The third line is for Progress system and help messages.

Method

A function associated with a widget or system handle that performs a specific action and returns a value (like a function) representing the result of that action.

Mnemonic

A single letter that you can press to choose a menu option. The mnemonic for each option is underlined when it is displayed.

Modal Application

An application or part of an application that forces you to perform a specific action before you can go on to other tasks.

Modeless Application

An application that gives you control over which tasks you perform when. It allows you a number of processing choices and then reacts depending on the event you trigger.

Mouse

A device for communicating with a user interface.

Nonterminating Sequence

A sequence that begins at an initial value and increments in one direction with no limit. See also **terminating sequence**.

One-to-one Relationship

Occurs when one record can relate to only one instance of a record in another table.

One-to-many Relationship

Occurs when one record can relate to many records in another table.

Operating System

Software that controls and manages the execution of computer programs and services.

Operator

The symbol you use to perform numeric calculations, date calculations, character string manipulations, or data comparisons (+, /, and GT, for example).

Options

Similar to a phrase, only smaller. Options also modify the way Progress executes a statement. Options usually consist of a single keyword and a possible accompanying value.

Output Destination

A file or device to which a program sends output. Output destinations can include a terminal, a printer, an ASCII file, or a printer queue.

Outer Join

See **left outer join**.

Parameter

A variable or constant passed between a subroutine and the main program. Also, a Progress startup option.

Parent

The widget that contains or owns other widgets.

Parent Frame

A frame that contains another frame. See also **root frame**, **child frame**.

Parent Window

A window that owns another window. See also **root window**, **child window**.

Pathname

Specifies the complete name of a directory or file by starting at the root directory or disk volume and tracing the hierarchy of the file.

Phrase

A collection of keywords and values that modifies the way Progress executes a statement.

Physical Database Name

The actual name of the database on a disk.

Predefined Function

A Progress-supplied function that provides short cuts for accomplishing common programming tasks. See also **Function**.

Primary Index

Usually the most frequently used index. Progress allows you to set one index as primary and uses it by default when searching for a record.

Procedure

The largest Progress unit of execution, consisting of one or more Progress source or r-code statements in a single, outer block.

Procedure Area

The visible part of the current edit buffer. This is where you type and edit Progress procedures. It is also called a *text pane*.

Procedure Block

A series of statements that Progress treats as a single unit.

Procedure Editor

An interactive utility that provides operations for editing code. You can use it to create, compile, and run Progress procedure files.

Procedure File (.p file)

A source file containing Progress 4GL code which can be compiled into r-code. Procedure files normally have a .p or .w (window procedure files) file extension.

Progress 4GL

An application development language which has a highly readable syntax employing a default behavior while performing the work of multiple 3GL statements.

Pull-down Menu

A vertically displayed list of menu elements (menu items and submenus) appearing when the user selects a menu title in a menu bar.

Query

1.) A request for information from a database. 2.) A set of database records that you create with your procedures. A query can involve one or more tables and can consist of all the records in a particular table or just a subset. The browse widget is a ready-made interface to a query.

Radio Button

One of the components of a radio set; it represents one of the possible values that the radio set can take on.

Radio-set Widget

A field-level widget composed of a series of buttons that represent a single field or variable, with each button representing a different possible value. Derived from the preset tuning buttons on many automobile radios-when one button is selected, any previously selected button is deselected.

Readable

Means that your application can read the current value and assign that value to a variable.

Record

A single occurrence of the data contained in a table. Records are treated as single units. Records are made up of fields.

Record Buffer

A temporary storage area in data memory for a record, field, or variable. When you write a record to the database, Progress gets that record from the record buffer.

Record Scope

The life span of a record in a record buffer. When the scope of a record ends, Progress releases the record and writes the changes made to the record buffer to the database.

Rectangle Widget

A decorative parallelogram (all of whose angles equal 90 degrees) displayed in a frame or in a frame background. Rectangles are display-only and cannot receive user input.

Related Tables

Tables that have at least one field in common.

Relational Database Management System (RDBMS)

A collection of hardware and software that organizes and provides access to a relational database.

Relational Operator

A symbol such as = (equal to), < (less than), or > (greater than) that is used to compare two values. It specifies a condition that can be either true or false.

Report

An organized display of data from a database.

Root Frame

A parent frame that is owned by a window and is the top parent of a frame family. See also **parent frame**, **child frame**.

Root Window

A parent window that is parented only by the window system and is the top parent of a window family. See also **parent window**, **child window**.

Row

1.) A collection of related items. Also called a record. 2.) A horizontally aligned set of character cells in the character coordinate system.

Schema

A description of a database's structure-the tables it contains, the fields within the tables, views, etc. In addition to database structure, Progress database schemas contain items such as validation expression and validation message.

Screen Buffer

A display area for a field, variable, or the result of a calculation. When you prompt for information or display information for the user, Progress places that information in the screen buffer.

Selection Criteria

Conditional expressions used to select particular records from a database table or tables.

Selection-list Widget

A scrolling list of character strings. The strings are the possible values for an underlying field or variable.

Self

In a trigger, SELF is a system-handle whose value is the widget handle of the widget that received the event that fired the trigger. See also **system handle** and **widget handle**.

Sensitive Widget

A widget that reacts to user actions.

Sequence

A database object that provides incremental values to an application. Sequences can generate sequential values within any integer range with a positive or negative value.

Sequence Properties

The schema definitions that define the sequence, such as the sequence name, the initial value, and so on.

Setable Attribute

An attribute to which the application can assign a new value.

Shared Variable

A variable used to pass information from one procedure to another.

Side Labels

Field labels to the left of the data and separated from the data by a colon and a space. See also **column labels**.

Slider Widget

A field-level widget composed of a rectangular scale that represents a range of INTEGER values. A marker within the scale indicates the current value. Only values of type INTEGER can be viewed as sliders.

Sports Database

A sample Progress database containing files, fields, and indexes (including some data) that procedures in Progress documentation use to acquaint you with the way Progress operates.

Statement

An instruction written in the Progress language.

Stream

A sequence of characters or items.

String

A character expression (a constant, field name, variable name, or any combination of these that results in a character string value).

Submenu

A vertically displayed list of menu items and submenus that appears when the user selects a menu title in a menu bar.

Subprocedure

A procedure that runs from the main procedure. A subprocedure can be either an internal or external procedure.

System Handle

A pseudo widget handle that provides a range of system capabilities sometimes associated with a visual widget. System handles that reference instances of visual widgets (for example, FOCUS) adopt all the attributes, methods, and events of the widget. Other system handles provide monitoring and control capabilities (for example, CLIPBOARD). System handles have their own attributes and methods, but do not respond to Progress events.

Table

A collection of logically related records organized into rows and columns. A table can contain a program, data, or text. A table is made up of records. It is also called a file.

Table Properties

The schema definitions that define the table, such as the table name, the table type, and so on.

Terminating Sequence

A sequence which begins at an initial value and increments in one direction until it reaches the designated limit. See also **nonterminating sequence**.

Text Widget

A field-level widget that displays any kind of data. It is read-only, and does not receive normal user events. Text widgets cannot have input focus.

Toggle Box Widget

A field-level widget composed of a small box that represents a logical value. Only values of type LOGICAL can be viewed as toggle boxes. The presence (TRUE) or absence (FALSE) of filling in the toggle box indicates the current value. A toggle box can be a field-level widget or a special type of menu item.

Top-down Programming

Traditional programming where you define larger structures first and then define smaller and smaller components.

Trigger

A piece of code that Progress executes on behalf of an application when a certain database action occurs or a certain user-interface event occurs. There are two types of triggers: database triggers and user-interface triggers.

Unique Index

An indexed field where every index key must be different. For example, Social Security number could be a unique index.

User-defined Function

A function that you can create based on a programming task that you want to solve. You can reuse this code once you create it. See also **function**.

User Interface Triggers

A piece of code that an application associates with one or more event-widget pairs. When the widget receives the event, Progress locates this piece of code and executes it on behalf of the application.

Validation Expression

A test to make sure that the user does not enter invalid data in a field.

Validation Message

A user-defined message the Dictionary displays when the user attempts to delete the selected table without meeting the table-validation criteria.

Variable

A temporary field for storing data.

Widget

An object that provides visual and interactive capabilities for a Progress application. A widget is a 4GL-aware control. That is, it is a control that shares common capabilities with other controls of the same type defined in the 4GL. There are several widget types that form a hierarchy in the 4GL, starting with windows. Windows can own other windows. Windows also contain frames and dialog boxes. Frames and dialog boxes contain field groups, and field groups contain field-level widgets or other frames. Widgets have attributes and methods and respond to events. Widgets serve as receptors for user input and also display application output.

Widget Attributes

Characteristics of the widget such as its type, screen location, size, color, font, and relationship to other widgets.

Widget Handle

To allow you to reference widgets within an application, Progress assigns each widget a unique *widget handle*. A widget handle is a pointer to the widget. Progress provides the WIDGET-HANDLE data type, to support handle values.

Window Family

A related collection of parent and child windows.

Window Widget

The workspace of your application. Each time you start a Progress session, it automatically creates a default window.

Word Index

Contains all the words from a character field or array of character fields. Searching with this index makes it very easy to locate records that contain specific words.

Working Directory

The directory you are in when you issue a command.

Working Database

The connected database that is currently in use. Multiple databases can be connected at any time, but only one is the current, working database.

Index

Numbers

3GL. see also Third generation languages

4GL conventions 2–19

4GL. see also Fourth generation languages

4GL/RDBMS 1–5

A

Accelerator keys 11–5
  defining 11–16

Action widgets 3–16

Aggregate functions
  TOTAL BY 10–16

Alert boxes 8–45, 12–19

ALL option 3–37, 3–44

Application defaults 1–10, 4–2, 4–15

Application development language 1–10

Applications 1–5

APPLY statement 6–23

Arguments
  passing 6–17
  using with include files 6–18

Arrays 5–5

ASC function 6–28

ASSIGN statement 5–18, 8–24, 8–27

Assignment operator 5–18

Attributes 3–28

Audience xvii

B

Base fields 10–29

BEGINS operator 9–4

Block header 6–21

Block processing 2–19

Blocking execution 3–8, 3–37, 6–22

Blocks 2–19, 6–21
  DO 6–24
  FOR EACH 6–29
  procedure 6–22
  properties 6–21, 6–31
  REPEAT 6–26

trigger 3–11, 6–23

Bold typeface
as typographical convention xx

Bottom-up programming 3–33

BOX attribute 12–14

Browse widget 3–15, 8–52, 9–8
control keys 8–57
events 8–60
read-only 8–52, 8–56
updateable 8–56

BUFFER-CHARS option 7–59

BUFFER-LINES option 7–59

Buffers
record 8–2
screen 8–2

Button widget 3–15, 3–19

BY option 9–9

C

CASE statement 6–33

CENTERED attribute 3–7, 12–14

CHARACTER data type
combo boxes 7–53

Character operators 5–13

Checking widget values 7–17

CHOOSE statement 3–23, 3–32

CHR function 6–28

CLOSE key 2–7

CLOSE QUERY statement 8–19

Code 1–5

COL attribute 3–30

COLUMN attribute 3–36, 12–14

Column events
browse widget 8–60

Combo box widget 3–14, 7–52
item selection 7–53
TOOLTIP option 7–54

Comparison operators 5–12

Compound indexes 1–9, 4–12

Concatenation operator 5–13

Conditional processing 6–32
CASE 6–33
IF 6–32

Constants 5–2, 5–10

Container widgets 3–16

CONTAINS operator 9–5

Context 12–3

Control blocks 6–24

Control sequences 10–41

Conventions
4GL 2–19
menus 2–11, 11–20
typographical 2–12

CREATE statement 8–40, 12–23

Creating records 8–40

D

Data 1–4

Data conversion functions
ASC 6–28
CHR 6–28

Data Dictionary 1–10
using 4–7

Data locations 8–2

Data types 3–17, 3–29, 4–11, 5–3
  character 4–11
  constants 5–10
  date 4–12
  decimal 4–11
  integer 4–11
  logical 4–12

Data widgets 3–16

Database events
  ASSIGN 4–17
  CREATE 4–17
  DELETE 4–17
  FIND 4–17
  WRITE 4–17

Databases 1–4
  connecting 4–3, 4–5
  creating 4–4
  defining objects 4–10
  disconnecting 4–6
  events 4–17
  fields 5–8
  labels 4–21
  logical names 4–6
  normalizing 9–16
  physical names 4–6
  reports 4–21
  schema 4–2

DATE data type
  combo boxes 7–53

Date operators 5–13

DECIMAL data type
  combo boxes 7–53

Default behavior 1–11, 3–8
  applications 4–15

DEFINE BROWSE statement 8–55

DEFINE MENU statement 11–6

DEFINE QUERY statement 8–19
  field lists 8–8

DEFINE SUB-MENU statement 11–4

DEFINE VARIABLE statement 3–7, 3–17, 5–3, 5–5
  NEW SHARED 6–10

DELETE PROCEDURE statement 12–4, 12–6

DELETE statement 8–44

Deleting records 8–44

DELIMITER option 7–45, 7–48

Dialog box widget 3–15, 12–16

DISABLE statement 3–41

DISPLAY statement 3–37, 8–24

Displaying data 10–11

DO block 6–24

DO PRESELECT statement
  field list 8–8

E

EDGE-CHARS option 3–23

Edit buffers 2–13, 2–17
  multiple 2–17

Editing keys 2–13, 2–16

Editor widget 3–14, 7–58
  BUFFER-CHARS 7–59
  BUFFER-LINES 7–59
  LARGE option 7–59
  MAX-CHARS 7–59
  NO-WORD-WRAP 7–60
  SCROLLBAR-HORIZONTAL 7–60
  SCROLLBAR-VERTICAL 7–60
  TOOLTIP option 7–60

ELSE option 6–32

ENABLE statement 3–7, 3–37, 3–41, 8–24,

8–26
syntax 8–27

Enabling input
frame families 12–15

END event 8–60

END statement 2–20, 6–21

END-ERROR key 2–7

ENDKEY event 12–19

END-SEARCH event 8–60

ENTER-MENUBAR key 2–7, 2–10

ENTRY event 3–32, 8–60

ENTRY event function 7–24

ENTRY statement 3–10

Error messages
displaying descriptions xxvi

Event actions 3–30

Event functions 3–30
ENDKEY 12–19
ENTRY 7–24
GO 12–19
LEAVE 7–24
VALUE-CHANGED 7–29, 7–41
WINDOW-CLOSE 12–24
WINDOW-MAXIMIZED 12–24
WINDOW-MINIMIZED 12–24
WINDOW-RESTORED 12–24

Event-driven programming 3–7, 12–2

Events 3–4, 3–11, 3–30
browse column 8–60
browse row 8–60
database 4–17
high-level 3–32
responding 3–8
VALUE-CHANGED 7–53

Example procedures xxiv

EXCEPT option 8–27

Exit key 2–7

Expressions 5–1, 5–18, 8–15

External procedures 6–3

F

Field lists 8–8

Field-editing key functions 3–32

Fields 1–8, 4–10
formatting 4–18

FIELDS option 8–8

FILE-NAME attribute 12–6

Fill-in field widget 3–4, 3–14, 7–22

FIND statement 8–12

FIRST-CHILD attribute 3–30

FIRST-PROCEDURE attribute 12–4

Focus *see Input focus*

Focus. see also Input focus

FOR EACH block 5–20, 6–29
selection options 9–18

FOR statement
field lists 8–8
inner joins 9–25

FORMAT option 3–35

Format strings 4–18

Fourth generation languages 1–10

FRAME attribute
frame families 3–28

Frame backgrounds 10–30

Frame families 3–27

enabling input 12–15
rules 3–28
tabbing 12–11

Frame headers 10–30

Frame scope 12–12

Frame widget 3–15
attributes 12–13
default 3–23
defining 3–23, 3–36
frame phrase 3–26, 3–37, 3–44
IN 3–49
WITH FRAME 3–37

Frames
frame phrase 6–28
WITH FRAME 8–15

FREQUENCY option 7–40

FUNCTION statement 6–6
defining parameters 6–13

Functions 5–2, 5–14
arithmetic
SQRT 5–15
data conversion
ASC 6–28
CHR 6–28
event
ENDKEY 12–19
GO 12–19
VALUE-CHANGED 7–29, 7–41
WINDOW-CLOSED 12–24
WINDOW-MAXIMIZED 12–24
WINDOW-MINIMIZED 12–24
WINDOW-RESTORED 12–24
FUNCTION statement 5–15, 6–6
PAGE-NUMBER 10–32
pre-defined 5–14
user-defined 5–15, 6–6
VALID-HANDLE 12–7

G

GET key 2–7

GO event 3–32, 12–19

GO key 2–7

Graphic widgets 3–16

Graphical user interfaces 3–3

H

HANDLE attribute 3–29, 12–3

Help
Progress messages xxvi

HIDDEN attribute 3–29, 12–14

HIDE MESSAGE statement 3–36

HIDE statement 3–44

High-level event functions 3–32

HOME event 8–60

HORIZONTAL option 7–33, 7–39

I

IF statement 6–24

IF THEN ELSE statement 6–32

Image widget 3–15

IN FRAME option 3–45, 3–49

Include files 6–8
using arguments 6–18

Indexes 1–8, 4–12, 9–5, 9–12
compound 1–9, 4–12
primary 4–13
properties 4–13
simple 1–9, 4–12
sorting by 9–5
word 4–13

INITIAL option 3–7, 3–17

Inner joins 9–25, 9–26

INNER-CHARS option 7–46

INNER-LINES option 7–46

Input focus 3–5, 3–7

INPUT option 6–12

Input. see also User input

INPUT-OUTPUT option 6–12

INSERT statement 8–51

INTEGER data type
  combo boxes 7–53

Interfaces. see also User interfaces

Internal procedures 6–3

Italic typeface
  as typographical convention xx

J

Joins
  conditions 9–24
  table 9–24

K

Key functions
  CLOSE 2–7
  END-ERROR 2–7
  ENTER-MENUBAR 2–7, 2–10
  EXIT 2–7
  field-editing 3–32
  GET 2–7
  GO 2–7
  navigation 3–32
  PUT 2–7
  universal 3–32

Keyboard mapping
  operating system considerations 2–7

Keys 9–19

Keystrokes xx

Keywords 2–19

L

LABEL option 3–17, 11–4

LARGE option 7–59

LARGE-TO-SMALL option 7–39

LAST-PROCEDURE attribute 12–4

LEAVE event 3–32, 8–60

LEAVE event function 7–24

LEAVE statement 6–26

Left outer joins 9–25

LIKE option 5–3, 5–4

LIST-ITEMS option 7–45

LIST-ITEMS( ) method 7–48

Literals
  passing 6–17

LOGICAL data type
  combo boxes 7–53

Logical database names 4–6

Logical operators 5–14

M

Main logic 6–22

Main procedures 6–2

Manual
  organization of xvii
  syntax notation xxi

MATCHES operator 9–4

MAX-CHARS option 7–59

MAX-VALUE option 7–39

Menu bar widget 3–15, 11–2, 11–6
  menu title 11–2

Menu conventions 2–11

Menu item widget 3–15, 11–5
  disabling 11–14

Menu titles 11–2

Menu widget
  attributes 11–19
  conventions 11–20
  toggle box 11–15

Menus 2–8
  symbols 2–12
  using 4–7

MESSAGE statement 3–10, 3–45

Messages 3–36
  displaying descriptions xxvi

Methods 3–49

MIN-VALUE option 7–39

Mnemonics 2–11, 11–6
  defining 11–15

Modes 12–17

Modules 6–2

Monospaced typeface
  as typographical convention xx

Multi-frame interfaces 12–11

MULTIPLE option 7–45

N

Navigation key functions 3–32

Nested submenus 11–2

NEW SHARED option 6–10

NEXT-SIBLING attribute 3–30, 12–7

NO-BOX option 3–7

NO-CURRENT-VALUE option 7–39

NO-FILL option 3–23

NO-PAUSE option 3–45

NO-PAUSE statement 3–36

Normalization 9–16

NO-WORD-WRAP option 7–60

Numeric operators 5–11

NUM-ITEMS() method 7–48

O

OF option 9–19

ON statement 3–9, 3–36

On-line Widget Tutorial 2–6

OPEN QUERY statement 8–19
  inner joins 9–25
  left outer joins 9–25
  relation options 9–18

Operands 5–2

Operations 5–2

Operator precedence 5–17

Operators 5–10
  assignment 5–18
  character 5–13
  comparison 5–12
  concatenation 5–13
  date 5–13
  logical 5–14
  numeric 5–11

Options 2–19
SKIP 8–15
SPACE 8–15

OR operator 3–11

Outer joins 9–27

Output destinations 10–24

OUTPUT option 6–12

OUTPUT TO statement 10–24, 10–38

Output, redirecting 10–24

P

PAGE-BOTTOM option 10–32

PAGE-NUMBER function 10–32

Parameters 6–11, 12–5
INPUT 6–12
INPUT-OUTPUT 6–12
OUTPUT 6–12

PARENT attribute 12–26

Passing parameters 6–11

Passing values 6–11

Persistent procedures 12–3, 12–5

Phrases 2–19

Physical database names 4–6

Platform considerations
basic editing keys 2–13
basic key functions 2–7
text block manipulation 2–16
using the browse widget 8–54

Precedence 5–15

Pre-defined function 5–14

PREV-SIBLING attribute 12–7

Primary indexes 4–13

PRIVATE-DATA attribute 12–6

Procedure Editor 1–12, 2–3
edit buffers 2–17
editing keys 2–13, 2–16
insertion point 2–4
menu bar 2–4, 2–8
procedure area 2–4
text editing 2–13

Procedure files 2–17, 6–2

Procedure handle 12–3

PROCEDURE statement 6–4

Procedures 1–4, 2–3
examples of xxiv
managing 12–2
parts 3–7
persistent 12–3, 12–5
procedure block 6–22
tutorial examples 2–2

Programmed responses 3–9

Programming models 3–2

PROMPT-FOR statement 8–49

Property sheets 4–8

Pull-down menu widget 3–15, 11–4

PUT key 2–7

PUT statement 10–36

Q

Queries 8–11, 8–18, 8–53

R

Radio set widget 3–15, 7–32
HORIZONTAL 7–33
VERTICAL 7–33

Readable attributes 3–29

Read-only browse widget 8–52, 8–56

Record buffers 8–2

Records 1–7
  creating 8–40
  deleting 8–44
  releasing 8–38
  selecting 9–2
  sorting 9–9

Rectangle widget 3–15, 3–21
  defining 3–21

Related tables 9–16

Relational database management systems 1–6

Relational databases 1–4

RELEASE statement 8–39

REPEAT blocks 6–26

REPEAT PRESELECT statement
  field list 8–8

REPEAT statement
  inner joins 9–25

Reports 10–2
  aggregate values 10–15
  control blocks 10–4
  editors 10–12
  multiple destinations 10–28
  multiple-table 10–19
  output statements 10–4
  printing 10–27, 10–41
  sending to other devices 10–24
  text widgets 10–3
  viewing on screen 10–7

RETURN statement
  NO-APPLY option 3–12

ROW attribute 3–30, 3–36, 12–14

Row events
  browse widget 8–60

Row IDs 8–26

ROW-DISPLAY event 8–60

ROW-ENTRY event 8–60

ROWID function 8–26

ROW-LEAVE event 8–60

RUN statement 6–2, 6–12
  parameters 6–13

S

Sample procedures 2–2

Saving widget values 7–17

Schema 4–2

Schema triggers 4–17

Screen buffers 8–2

SCREEN-VALUE attribute 3–29

SCROLLBAR-HORIZONTAL option 7–46, 7–60

SCROLLBAR-VERTICAL option 7–46, 7–60

SCROLL-NOTIFY event 8–60

Selecting records 9–2

Selection criteria
  WHERE option 5–19

Selection list widget 3–14, 7–44
  DELIMITER 7–48
  LIST-ITEMS 7–45, 7–48
  MULTIPLE 7–45
  NUM-ITEMS() 7–48
  SCROLLBAR-HORIZONTAL 7–46
  SCROLLBAR-VERTICAL 7–46
  SINGLE 7–45
  SORT 7–46

SELF handle 5–8

SENSITIVE attribute 3–29

Sequences 4–13
  cycling 4–15
  nonterminating 4–14
  properties 4–14
  terminating 4–15

Session triggers 4–17

SET statement 8–50

Setable attribute 3–29

SHARED option 6–10

Shared resources 12–4

Shared variables 6–10

SIDE-LABELS attribute 12–14

SIDE-LABELS option 3–7

Simple indexes 1–9, 4–12

SINGLE option 7–45

Size phrase 7–33, 7–40, 7–46, 7–59

SKIP option 8–15

Slider widget 3–15, 7–38, 7–40, 7–46
  FREQUENCY option 7–40
  HORIZONTAL 7–39
  LARGE-TO-SMALL 7–39
  MAX-VALUE 7–39
  MIN-VALUE 7–39
  NO-CURRENT-VALUE 7–39
  TIC-MARKS option 7–40
  VERTICAL 7–39

SORT option 7–46

Sorting records 9–9

SPACE option 8–15

sports database 4–9

SQRT function 5–15

Starting Progress 2–2
  on UNIX 2–3
  on Windows 95 and Windows NT 4.0 2–2
  on Windows NT 3.51 2–2

START-SEARCH event 8–60

Statements 2–19
  DEFINE VARIABLE 3–7, 3–17

Status area 2–4

STREAM-IO option 10–11

Submenu widget 3–15, 11–4

Subprocedures 6–3
  parameters 6–11
  passing data 6–11
  shared variables 6–10

Syntax 2–19
  block 2–19

Syntax notation xxi

System handles 5–8
  CURRENT-WINDOW 11–8, 12–23
  DEFAULT-WINDOW 11–8, 12–23
  SELF 5–8
  THIS-PROCEDURE 12–7

T

Tabbing
  between frames 12–11

Table joins 9–15

Table triggers 4–17

Tables 1–6
  keys 9–19
  related 9–16

Text blocks 2–15

Text widgets 3–14, 3–35, 7–26

THEN option 6–32

Third generation languages 1–10

THIS-PROCEDURE handle 12–7

TIC-MARKS option 7–40

Toggle box widget 3–15, 7–28, 11–15

ToolTip attribute 7–10

TOOLTIP option 7–40, 7–46, 7–54, 7–60

Top-down programming 3–33

Trigger block 3–11

Trigger blocks 6–23

Triggers 3–9, 3–19, 3–51
  defining 3–36
  schema 4–17
  session 4–17
  table 4–17

Typographical conventions xx, 2–12

U

Universal key functions 3–32

UPDATE statement 8–32

Updateable browse widget 8–52, 8–56

Updating data 8–14

USE-INDEX option 9–12

User input 3–5

User interfaces 2–5, 3–2
  character 3–3
  graphical 3–3
  managing 12–10
  multi-window 12–22

User-defined function 5–15

USING option 9–5

V

ValExp option 4–15

Validation 4–2
  Data Dictionary 8–28
  expressions 4–15
  messages 4–15

VALID-HANDLE function 12–7

ValMsg option 4–15

VALUE-CHANGED event 7–29, 7–41, 7–53, 8–60

Variables 3–4, 3–37, 3–49, 5–2, 5–3
  arrays 5–5
  ASSIGN statement 5–18
  data types 3–17, 5–3
  defining 3–7, 3–17, 3–35, 5–3
  initializing 3–7, 3–17
  LIKE option 5–3, 5–4
  naming 3–17
  shared 6–10

VERTICAL option 7–33, 7–39

VIEW statement 3–44

VIEW-AS phrase 3–17, 4–20, 7–22

VISIBLE attribute 3–29, 3–51, 12–14

W

WAIT-FOR statement 3–8, 3–31, 3–37, 12–27

WHERE option 5–19, 9–22
  selecting records 9–4

WHILE option 6–25

Widget attributes 3–29, 3–49

Widgets 2–5, 3–5, 3–14

action 3–16
attributes
  readable 3–29
  setable 3–29
browse 3–15, 9–8
button 3–15, 3–19
categories 3–16
checking values 7–17
combo box 3–14, 7–52
  item selection 7–53
containers 3–16
data 3–16
default response 3–8
defining 3–7, 3–16
dialog box 3–15
disabled 3–41
editor 3–14, 7–58
enabled 3–5, 3–41
field-level 3–23, 3–35, 3–52
fill-in field 3–4, 3–7, 3–14, 7–22
frame 3–15
graphic 3–16
HANDLE attribute 11–8
hiding 3–44
image 3–15
labels 3–7
menu bar 3–15, 11–2, 11–6
menu item 3–15, 11–5
methods
  accessing 3–49
pull-down menu 3–15, 11–4
radio set 3–15, 7–32
rectangle 3–15, 3–21
saving values 7–17
selection list 3–14, 7–44
size phrase 7–33, 7–40, 7–46, 7–59
slider 3–15, 7–38
submenu 3–15, 11–4
text 3–14, 3–35, 7–26
toggle box 3–15, 7–28
types 3–14, 3–19
using attributes 5–9
viewing 3–44
window 3–16, 11–2, 12–22

Window families 12–24
  coordination 12–25

Window widget 12–22

Window widgets 3–16, 11–2, 12–22
  alert box 8–45
  attributes 11–8, 12–22
  families 12–24

Word indexes 4–13