

Clarion

**Learning
Clarion**

COPYRIGHT SoftVelocity Inc. All rights reserved.

This publication is protected by copyright and all rights are reserved by SoftVelocity Inc. It may not, in whole or part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from SoftVelocity Inc.

This publication supports Clarion. It is possible that it may contain technical or typographical errors. SoftVelocity Incorporated provides this publication "as is," without warranty of any kind, either expressed or implied.

SoftVelocity Inc

www.softvelocity.com

Trademark Acknowledgements:

SoftVelocity is a trademark of SoftVelocity Inc.

Clarion™ is a trademark of SoftVelocity Inc.

Microsoft®, Windows®, and Visual Basic® are registered trademarks of Microsoft Corporation.

All other products and company names are trademarks of their respective owners.

Table of Contents

Table of Contents	i
Learning Clarion - Introduction	1
Anatomy of a Database	2
Definitions	2
Summary:	3
Table Systems and Table Drivers	3
Data Types	3
Sorting Data: Keys and Indexes	3
Ascending and Descending Sort Orders	4
Using Keys as Range Limits.....	5
Relationships Between Tables	5
Database Summary	6
1 - Planning the Application	7
Defining Application Tasks	7
Designing the Database	7
The Customer Table	8
The Phones Table.....	8
The Orders Table	9
The Detail Table	9
The Product Table.....	9
Referential Integrity.....	9
The Complete Database Schematic	10
Application Interface	10
OK, What Did I Just Do?	11
2 - Creating a Data Dictionary	13
Lesson Files	13
Creating the Dictionary	14
Copying Tables From One Dictionary to Another.....	15
Import Tables from the <i>Getting Started</i> Dictionary	15
Relating the Tables.....	18
Set a default value	18
OK, What Did I Just Do?	19
3 - Adding Tables and Columns	21
Defining New Tables	21
Create the Phones File	21
Name the Detail and Products Tables	23
Defining the Columns	24
Define a Column Pool	24
Define the columns in the Phones Table	27
Define the columns for the Detail Table	30
Define the columns for the Products Table.....	31
Clean up of the Orders table import.....	33
OK, What Did I Just Do?	34
4 - Adding Keys and Relations	35
Examining the Keys for the Orders Table	35
Defining Keys for the Detail Table	36
Define the First Foreign Key	36
Define the Second Foreign Key	37

Defining Keys for the Products Table	37
Create the Primary Key	37
Define an Alphabetical Key	38
Defining a Key for the Phones Table	39
Define the Foreign Key	39
Defining Table Relationships	40
Defining Relationships for the Phones Table	40
Defining Relationships for the Detail Table	42
Defining Relationship-Dependent Validity Checks	43
Define the Validity Check for Order Rows	43
Define the Validity Check for Detail Rows	43
OK, What Did I Just Do?	44
5 - Importing Existing Data	45
Data Conversion	45
Importing a .CSV File Definition	45
Converting a Table	47
OK, What Did I Just Do?	50
6 - Starting the Application	51
Using the Application Generator	51
Creating the application (.APP) file	51
Creating the Main Procedure	53
Editing the Menu	57
Creating the SplashScreen Procedure	61
Adding an Application Toolbar and Toolbar Controls	62
Testing an Application under Development	70
Look at the Generated Source Code	71
OK, What Did I Just Do?	73
7 - Creating a Browse	75
Creating a Browse Window	75
Creating the Customer Browse Window	76
Populating and Formatting a List Box Control	79
Adding the Tabs	82
Hiding the Buttons	88
Setting the Sort Orders	89
Closing the Customer Browse	90
OK, What Did I Just Do?	90
8 - Creating an Update Form	91
Creating an Update Procedure	91
Add a "ToDo" procedure	91
Creating the Update Form Procedure	92
Populating the Columns	95
Moving and Aligning Columns	96
Adding a BrowseBox Control Template	99
Adding the BrowseUpdateButtons Control Template	103
OK, What Did I Just Do?	104
9 - Copying Procedures	107
The Products Table Procedures	107
Copy the Procedures	107
Working with Embed Points	108
Modify the Browse	110
Creating the Form Procedure	114

OK, What Did I Just Do?	116
10 - Control and Extension Templates	117
Creating the Procedure	117
Select the procedure type	117
Placing the BrowseBox Control Template	118
Adding the Browse Update Buttons Template	120
Placing the Second Browse List Box	121
Adding the Close Button Control Template	123
Make the window resizable	123
Set up a Reset Column	124
Close the Procedure Properties dialog and Save the Application	125
OK, What Did I Just Do?	125
11 - Advanced Topics	127
Set Up the UpdateOrder Form	127
Create the Orders table's data entry Form	127
Placing the Detail Table's Control Templates	131
Making it all Work	135
Using the Formula Editor	135
Configuring Edit in Place	137
OK, What Did I Just Do?	146
12 - Creating Reports	148
Overview	148
Updating the Main Menu	148
Creating the Report	149
Populating the Detail	151
An Invoice Report	154
Creating the Report	154
Populating the Report Form Band	155
Populating the Detail Band	156
Adding Group Breaks	157
Populating the Group Header Band	159
Populating the Invoice Group Footer Band	162
Populating the Customer Group Footer Band	162
Adding a Formula	165
A Range Limited Report	167
Creating the Report	167
Modify the new report	167
A Single Invoice Report	169
Creating the Report	169
OK, What Did I Just Do?	172
What's Next?	172
13 - Clarion Language Lesson	176
Clarion—the Programming Language	176
Event-driven Programming	176
Hello Windows	177
Hello Windows with Controls	181
Hello Windows with Event Handling	184
Adding a PROCEDURE	185
Adding Another PROCEDURE	187
Moving Into the Real World—Adding a Menu	189
Really Moving Into the Real World—Adding a Browse and Form	191
ABC Template Generated OOP Code	206

Table of Contents

Load a simple application	206
Look at the Program Source	206
Where to Go From Here?	215
Index	217

Learning Clarion - Introduction

Welcome to Learning Clarion!

The Getting Started document briefly introduced you to Clarion programming at its highest level. Learning Clarion now teaches you how you can use all the rest of the tools Clarion provides to create real-world applications. It contains two parts, on two very different levels:

- A series of **Application Generator** lessons, which familiarizes you with all of the tools in the Clarion development environment.
- A **Clarion Language** lesson, which introduces the Clarion programming language and familiarizes you with the type of code generated for you by the development environment.

What You'll Find in this Document:

Application Generator Lessons

Chapters One through Twelve introduces all of the main Clarion development environment tools. It starts at the application planning stage, walks you through creating the data dictionary with the Dictionary Editor, and then walks you through creating a complete application with the Application Generator. By the end of these lessons you'll have created a complete order-entry application.

You'll use the Application Generator and work with Procedure, Control, and Code templates to produce an Order Entry application. You'll work with the Window Designer to design windows. You'll work with the Report Designer to design reports. You'll use the Text Editor to embed Clarion language source code into the code generated by the templates.

Clarion Language Lesson

Chapter Thirteen introduces the Clarion programming language at the hand-coding level. It starts at Hello World, then walks you through creating simple forms of the most typical types of procedures used in Clarion applications, all while explaining the functionality of the hand-code you're writing and relating it to the type of code you'll see generated for you by the Application Generator.

Anatomy of a Database

This section briefly describes the fundamentals of database design. It is meant only to provide an overview of the subject for those who are not already thoroughly familiar with standard database design concepts and issues. Experienced developers may want to move right on to the next chapter and skip this section.

Definitions

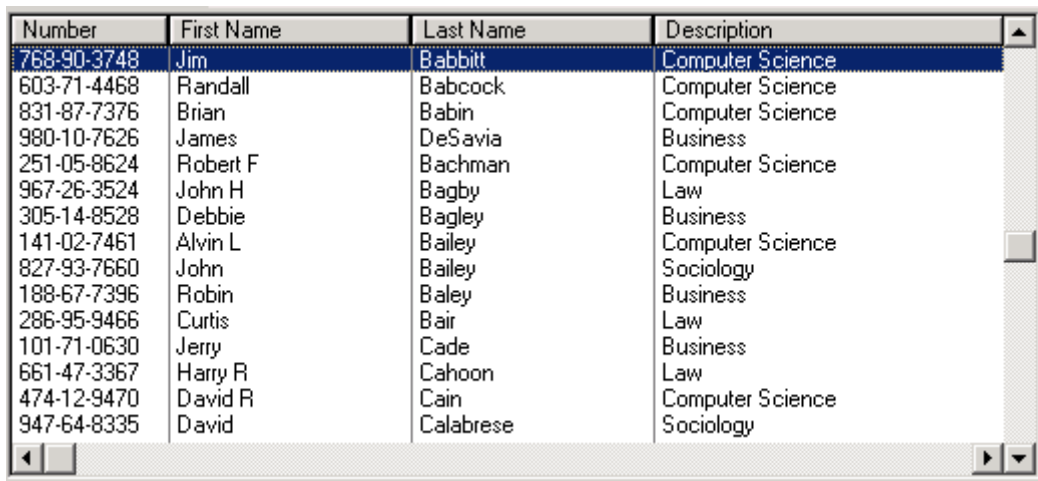
A database is a collection of information (data) in a system of tables, rows, and columns. The database is maintained by one or more computer programs or applications.

The basic unit of data storage is a *column*. A column is a storage place for information of a similar type. For example, one column might store a name and another column might hold a telephone number.

A group of different columns that are logically related make up a *row*. A row contains all the information related to one subject. For example, all the columns containing information concerning one student (name, address, telephone number, student number, etc.) makes up one student's row. This would be similar to a file folder a school might keep for each student.

Collections of logically related rows make up a *table*. Using the same example, a collection of all students' rows makes up the student body table. This would be similar to the file cabinets where students' folders are kept.

Another way of looking at this is as a table or spreadsheet:



Number	First Name	Last Name	Description
768-90-3748	Jim	Babbitt	Computer Science
603-71-4468	Randall	Babcock	Computer Science
831-87-7376	Brian	Babin	Computer Science
980-10-7626	James	DeSavia	Business
251-05-8624	Robert F	Bachman	Computer Science
967-26-3524	John H	Bagby	Law
305-14-8528	Debbie	Bagley	Business
141-02-7461	Alvin L	Bailey	Computer Science
827-93-7660	John	Bailey	Sociology
188-67-7396	Robin	Baley	Business
286-95-9466	Curtis	Bair	Law
101-71-0630	Jerry	Cade	Business
661-47-3367	Harry R	Cahoon	Law
474-12-9470	David R	Cain	Computer Science
947-64-8335	David	Calabrese	Sociology

In this format, the entire table is a file cabinet, with rows (folders) and columns (data about one row).

A *database* is a collection of related tables. This is similar to a bank of file cabinets where the entire school records are kept. One file cabinet might hold the files with students' personal data, another with class enrollment information, and another with faculty information.

A *relational database* is a collection of tables with defined relationships between them. Effective database design breaks the data into related tables that are joined together through linking columns. This will be covered in detail later in this section.

Summary:

- ✓ One or more *columns* combine to form a *row*.
- ✓ One or more rows combine to form a *table*.
- ✓ A collection of related tables is a *database*.

Table Systems and Table Drivers

There are several table formats used on PCs. These are the actual physical storage formats written to disk by programs that maintain the tables. Using TopSpeed's table driver technology, Clarion supports many of them. Table drivers enable Clarion programs to read these different table formats.

The Professional and Enterprise Editions include ADO, TopSpeed, Clarion, ASCII, BASIC, ODBC, Clipper, dBase III & IV, DOS, FoxPro, Pervasive, MSSQL and other SQL table drivers. When you need to read data from another table system, you can add new table drivers to the Professional and Enterprise Editions. Call SoftVelocity's Sales department at (954) 785-4555 to inquire about the availability of any specific table driver you need.

Each table system has its own idiosyncrasies and limitations. See *Database Drivers* in the Language Reference core help for more specific information.

Data Types

Columns can store many different types of data, but each individual column may hold only one type. When a column is defined, its data type is specified. This enables it to efficiently store that type of data. For example, to store a number from 0 to 100, using a column defined as a single BYTE takes less space than one defined as a decimal number column (a byte can hold an unsigned whole number between 0 and 255).

Clarion supports a wide variety of data types. All are fully documented in Chapter 3 of the Language Reference core help.

Sorting Data: Keys and Indexes

One of the most powerful aspects of a computerized database is the ability to sort data in many different ways. To do this manually requires multiple copies of record forms, many file folders, and many file cabinets. It would also require a lot of time spent filing each copy in different places for each sort order.

Sorting computer rows in a database merely requires the definition of keys or indices. Keys and indices declare sort orders other than the physical order of the rows within the table. In some table systems the keys are kept in separate disk tables, in others they are contained within the same table as the data. TopSpeed's table driver technology handles these differences transparently.

Keys and indices are functionally equivalent. The only difference is the way they are maintained by an application.

- A key is dynamically maintained by the application. Every time a row is added, modified, or deleted, the sort sequence is updated, if necessary. Keys are useful for frequently used sort-orders.

- An index is not dynamically maintained, it is only built when needed. Indexes are useful to create sort sequences that are infrequently used, such as month-end or year-end processes.

Using the student table example discussed earlier, suppose you wanted to sort the students' rows two ways: by name alphabetically, and by name within each major. This produces two alternate sorts.

This example uses a one-component key on the student's name.

Number	First Name	Last Name	Description
768-90-3748	Jim	Babbitt	Computer Science
603-71-4468	Randall	Babcock	Computer Science
831-87-7376	Brian	Babin	Computer Science
251-05-8624	Robert F	Bachman	Computer Science
967-26-3524	John H	Bagby	Law
305-14-8528	Debbie	Bagley	Business
141-02-7461	Alvin L	Bailey	Computer Science
827-93-7660	John	Bailey	Sociology
286-95-9466	Curtis	Bair	Law
188-67-7396	Robin	Baley	Business
999-01-9999	Joe	Blix	Business
123-44-9999	Deb	Brown	Computer Science
101-71-0630	Jerry	Cade	Business
661-47-3367	Harry R	Cahoon	Law
474-12-9470	David R	Cain	Computer Science

The next example has two components in the key: major and student name. A key can contain one or more columns, allowing sorts within sorts.

Number	First Name	Last Name	Description
770-99-5150	Bryan	Grace	Computer Science
352-32-2847	Larry	Kaiser	Computer Science
931-82-3266	Brent	Lee	Computer Science
956-25-2995	Mary	Rager	Computer Science
648-27-7686	Mike	Todd	Computer Science
431-40-3937	Vincent	Tracey	Computer Science
360-98-4969	Kevin	Travis	Computer Science
721-34-4763	Tim	Wiley	Computer Science
827-93-7660	John	Bailey	Sociology
947-64-8335	David	Calabrese	Sociology
291-65-7167	Carol	Drake	Sociology
445-28-0353	Jack	Eddings	Sociology
623-67-2121	Clint	Ercanbrack	Sociology
866-78-5927	Tom	Gabardi	Sociology
325-84-3766	Leon	Gaertner	Sociology

Ascending and Descending Sort Orders

Some table systems support both ascending and descending order for keys or components of keys (for example, the TopSpeed system). Other table systems only support ascending order, which means the data can only be sorted from lowest to highest (for example, the Clarion system).

The next example has two components in the key: Graduation Year (descending), and student name (ascending).

Grad Year	First Name	Last Name
1999	Clint	Ercanbrack
1999	Sandra	Fabela
1999	Kent	Hackett
1999	Dan	Headman
1999	Brent	Herbert
1999	Walter	Oakey
1999	Brent	Olsen
1999	Gerardo	Sierra
1999	Gene	Tabor
1999	Tim	Wiley
1998	Brian	Babin
1998	Robert F	Bachman
1998	John H	Bagby
1998	Alvin L	Bailey
1998	Curtis	Bair

Using Keys as Range Limits

Suppose you want to create a class enrollment report from a database for the past fifteen years. All you are interested in (for purposes of this report) is the last three years. You can dramatically reduce processing time if you use a subset of the table that only contains the rows from the last three years. It takes two steps to accomplish this:

- First, define a key to sort the data by the date of each course.
- Next, define the range limits you are interested in. A range limit specifies a subset of the entire table to process. Only those rows that fall within the range limits are considered.

In this example, only one-fifth of the rows are processed (assuming that each year's course offerings are the same). Reducing the number of rows to include by 80% reduces processing time by the same amount.

Relationships Between Tables

One goal of relational database design is reducing data redundancy. The basic rule is that data should be located in only one place. This is beneficial in two ways. First, it reduces storage space requirements. Second, it makes the database easier to maintain. To reach this goal, tables are broken up into separate, related tables through a process called data normalization.

The first step is to move any repeating groups into separate tables. For example, if a student could take a maximum of six classes, you could design the student table to contain six class columns (class1, class2, class3, etc.). But not all of these columns would be used in each row. If one student is taking six courses, all would be used, but if another student only took one class, there would be five empty columns in his row. For that reason, the class columns are moved into a separate table, eliminating the need to reserve space for empty columns. This creates a One to Many relationship (*One* student takes *Many* classes) between the student table and the classes table.

The next step is to move redundant data into separate tables. Every column in the student table must be dependant on the primary key (Student Number). The student's name, address, and phone number remain in the student table. But the student's major description could be moved to a separate table. This eliminates the need to repeat the Major's Description for each student with that Major. To do this, add a Major ID number column to the student table and the Majors table. This creates a Many to One relationship (*Many* students have *One* major) between the student table and the majors table.

Once data storage is "normalized," related information is linked by ensuring a column in one table is identical to a column in the other. These common columns create the links between related tables. A linking column could be a student number, a course code, or a classroom number. Any column (or group of columns) that uniquely identifies a row in the primary table can be used as a link.

Examples of these relations can be found in a school database:

- One teacher teaches many classes (One-to-Many).
- Many students would have one major (Many-to-One).

Database Summary

- ✓ A database is a collection of information (data) in a system of columns, rows, and tables.
- ✓ Columns can store many different types of data, but each individual column is specified to hold only one type.
- ✓ Each data item should be stored in only one place.
- ✓ One or more columns makes up a row. One or more rows make up a table. A collection of related tables make up a database.
- ✓ Clarion programs can access many different table systems through the use of table drivers.
- ✓ Keys and indexes declare sort orders other than the physical order of the rows within the table, and can contain more than one column, allowing sorts within sorts.
- ✓ Range Limits enable you to process a subset of rows, which reduces processing time.
- ✓ Tables are related through common columns containing identical data, which allows you to eliminate redundant data.

1 - Planning the Application

As a general rule, every minute you spend planning your application beforehand saves you ten later. This topic informally describes the planning process for the application you'll create in the subsequent chapters. In the real world, you'll probably create a bona fide functional specification for your important applications. This informal description defines:

- The tasks the application performs.
- The data the application maintains, and how it stores it.

As a starting point, this Application Generator lesson application uses the data dictionary from the applications you created in the Getting Started manual. It extends the concept to a simple Order/Entry system, using the data dictionary for keeping track of customers.

Defining Application Tasks

This application will maintain the customer and billing tables for a manufacturing company. The first task in planning just what the application will do is to assess what the company expects it to do.

For the purpose of the subsequent lessons, the application we'll create is a simple order entry system. Customers typically phone in orders for one or more products at a time. A salesperson takes the order. The billing department prints an invoice that night.

The application therefore must provide:

- Entry dialogs for taking the order, or modifying the data in it later.
- Access to the customer list from within the order entry dialogs. The customer list is the one you created in the Getting Started lessons, stored in the Customer table.
- Access to the list of part numbers (items) that the company manufactures, from the order entry dialogs.
- Browse windows for listing sales transactions.
- Procedures that will maintain the Products list and Customer information.
- Printed reports.

Designing the Database

The first task in planning the table structure is to assess what data the application needs, and how to store it with the minimum amount of duplication.

Good database management maintains separate data tables for each "entity" or group of discrete data elements. The data "entities" this application maintains are:

Customer	Customer name and address data that changes only when a customer moves. Created in the Getting Started lessons, along with its related Orders table.
Phones	In the communication age that we live in, it is probable that a customer may have more than one contact phone number.

Orders	Basic information needed for assembling the data needed to print an invoice. It "looks up" information from the other tables, such as the customer name and address. When a sales person takes a new order, they add a row to this table.
Detail	Product, price, and quantity ordered for an item on a given invoice: the variable information for each order. Though this duplicates price information in the Products table, you must maintain the price at the time of the sale here. Otherwise, when you increase the price in the Products table, it would cause the balance in the Detail table to change.
Products	Information on the products sold by the company, including product number, description and price. This data changes only when a price changes or a new product is added.

The Customer Table

The Customer table stores "constant" data such as customer names and addresses. It's most efficient to store this data in one place, allowing for a single update when the information changes. This also saves space by eliminating redundant customer information in the Orders table; otherwise, if there were 1000 orders for company XYZ, the address information would be repeated 1000 times. Reducing storage requirements by storing the data only once is called *normalization*.

The customer data requires a column to uniquely identify the customer. The company name is unsuitable because there could be duplicates. There may be, for example, multiple rows for a customer called "Widget Depot," if it has multiple locations. The CustNumber column is a good candidate for an auto-number key which automatically creates and stores unique customer numbers.

The CustNumber column also serves as the *primary key* for the data table. Any other data tables which are related to the Customer table must declare the CustNumber column as a *foreign key*. A primary key is a column, or combination of columns, that uniquely identifies each row in a data table. A foreign key is a column, or combination of columns, in one table whose value must match a primary key's value in another related table.

Because there may be many orders for each customer number, the relationship between the Customer table and the Orders table will be a *one to many* (1:Many) relationship. We say the customer data table is the *parent* table, and the Orders data table is the *child* table.

The Phones Table

The Phones table stores telephone numbers—each customer could have several. Each row includes a CustNumber column to relate back to the Customer table.

The Phones table also includes a text column in which we can indicate whether the phone number is an office, fax, mobile or home number. Using the data dictionary, we'll specify that the control for entering data for this column should be a drop-down list with the choices already loaded.

The Orders Table

The Orders data table gathers information for each sales transaction from all the other data tables (such as the customer data). Because much of the basic data in this table prints in the "header" area of the invoice, this is sometimes called the Order Header.

Every sales transaction requires one row in the Orders table. The row relates to the customer information by referencing the unique customer number. Because some order rows may reference one product, and others may reference ten, you'll create a separate Detail table which relates back to a unique order number. This creates a one to many relationship, with the Orders table as parent and Detail as child. The actual products ordered are identified by their product codes, in the Detail table.

The Orders row thus holds a customer number to relate back to the customer data (the foreign key), and a unique order number to relate to the Detail. You'll create a multi-component primary key on the two columns, so that you can easily create a browse sorted by customer and invoice number.

The Detail Table

The Detail table stores the products ordered by their product codes (a foreign key into the Product table), their individual prices, the quantity of each, and the tax rate. An additional column holds an invoice number, which relates back to the Orders table in a many to one relationship.

The Detail table duplicates the price information with the columns in the product table; this is because prices may change. It's important to store the price column within the detail table row because if the price increases in six months, today's paid in full invoice would reflect a balance due.

The Product Table

The Product table stores unique product numbers, descriptions, and prices. When the sales person looks up a product by name, the application inserts the product number into the Detail row. The product code is the primary key—no two items can have the same code, and every product must have a code. An additional column contains the tax rate for the product.

Referential Integrity

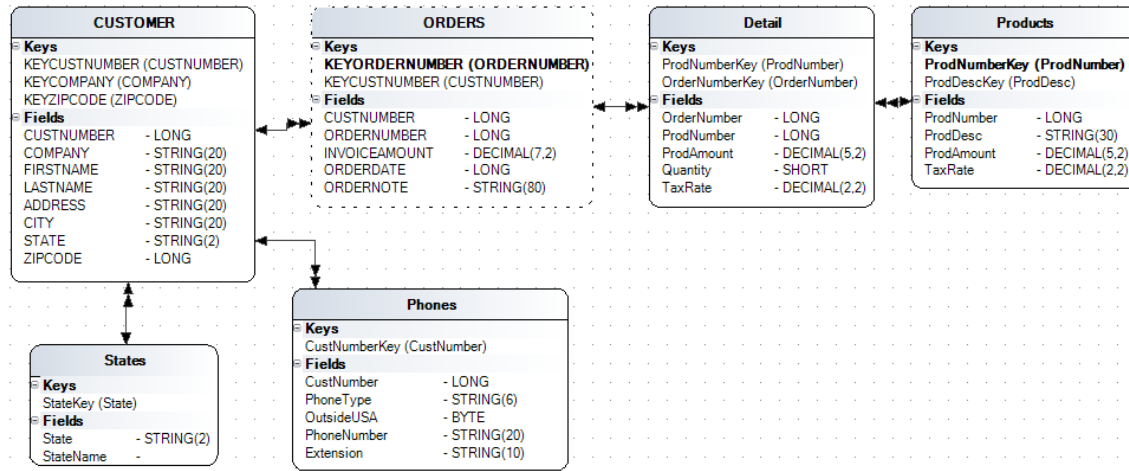
Referential Integrity refers to the process of checking all updates to the key column in a given table, to ensure that the validity of parent-child relationships is correctly maintained. It also refers to ensuring that all child table rows always have associated parent rows so that there are no "orphan" rows in the database.

Because the data for a given transaction resides across several tables, this application must enforce referential integrity. This is critical, yet many database application development tools require you to hand code procedures to enforce this. The Application Generator's templates implement this automatically in your generated source code when you select a few options in the Data Dictionary.

It is essential that the application not allow an update to a row that leaves a blank or duplicate value in a primary key column. For example, we need to restrict the ability of the end user to update a row in a way that could cause a duplicate Customer number. If two different companies shared a duplicate Customer number, you could send a bill to the wrong company.

The Complete Database Schematic

The schematic below provides an overview of the entire database. If you look at it from the point of view of the sales agent taking a phone order, the Orders table stores who's ordering, the Detail stores what they're ordering, and the Customer and Product tables store constant information about the customers and products.



The item code looks up the description and price. The customer code looks up the customer's name and address. Other data, such as the transaction date, fill in automatically (by looking up the system date, for example).

Finally, the following lessons will create a brand new data dictionary, and you will copy and paste the tables that Getting Started defined for you into the new dictionary.

As for the actual application you create, because the lessons are a teaching tool more concerned with showing what Clarion can do for you, it won't create a full-scale order entry system.

However, you will find that some parts of the application will be very "showy," so that you can quickly learn how to do equivalent procedures in your applications.

Application Interface

The next major task before coding is to plan the user interface. For a business application like this, it's crucial that a salesperson quickly locate the data they need, so that they can record the sale and move on to the next phone call. Therefore, the application should put all the important data "up front" by default, and no entry or maintenance dialog should be more than a button or menu command away.

Additionally, the company uses many other Windows applications; therefore, it's especially important that the application have a standard windows "look and feel." End users learn a familiar interface more quickly.

To implement the tasks the application must execute in a consistent manner with our guidelines, we can plan for the items listed below. Though the following is no substitute for a real program spec, it should suit us for the upcoming lessons.

- Because the application will handle the maintenance for the customer, item, and billings tables on different forms, the Multiple Document Interface (MDI) is necessary.
- The application should have a toolbar with buttons to load forms and browse windows, and to control the browsing behavior.

- To maintain a consistent "look and feel," the main menu choices will be File, Edit, View, Window, and Help. The File menu accesses the printing and exit procedures. The Toolbar buttons call the form dialogs for editing a current row (if highlighted in a browse) or adding/deleting rows, and for browsing through the tables. The Browse menu calls the procedures for browsing tables. Window and Help perform standard actions.
- When adding new orders, the sales people should be able to pick customers and products from scrolling lists. Pertinent data in the order dialog—addresses, item descriptions and prices—should automatically "fill in" as appropriate.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- ✓ You defined the tasks the Application must accomplish.
- ✓ You designed the database that will allow the application to accomplish those tasks.
- ✓ You specified the user interface the application will use.

Now that the application description is fairly complete, we're ready to begin work. The first step is to create the data dictionary.

2 - Creating a Data Dictionary

This section teaches you how to:

- Create a new data dictionary.
- Copy and customize table definitions from the Getting Started data dictionary to the new one.
- Relate the tables and specify Referential Integrity constraints.
- Pre-format window controls for the columns.

This lesson assumes that you have completed the Getting Started Lessons.

Lesson Files

We recommend that you complete the entire series of lessons here, regardless of your experience with Clarion. There are many new things you need to know about. As you've already seen from the Getting Started lessons, Clarion's template-driven Application Generator approach to programming is very different from other development environments for 3GL and 4GL languages. If you'll thoroughly immerse yourself in the following lessons, you will get the most out of your new tool.

The completed lesson files reside in the Shared Documents section on your computer.

For example, in Windows XP, the Lessons folder is found in:

C:\Documents and Settings\All Users\Documents\SoftVelocity\Clarion7\Lessons\LearningClarion\Solution

In Vista:

C:\Users\Public\Documents\SoftVelocity\Clarion7\Lessons\LearningClarion\Solution

We provide these so you can see the end result of the lessons and compare your application. There should only be cosmetic differences.

Who should do these lessons?

If you're already an experienced Clarion programmer, you may want to just examine the completed lesson files rather than step through the individual lessons. However, we do recommend that you at least read through the lessons, since there are methods of working in the development environment that the lessons demonstrate that may not be obvious just by "plunging in" and working with the tools—the lessons "show off" some features of the development environment that may not be readily obvious at first glance.

If you are coming from legacy developed applications (Clarion templates) and wish to learn more about ABC based applications, then these lessons are strongly recommended, regardless of Clarion expertise.

Note:

Wherever there are multiple ways to accomplish a single task, the lessons will expose you to several of them to demonstrate the flexibility of the Clarion development environment.

Creating the Dictionary

Whenever you create a new application, you first define the data dictionary (.DCT file). From the data dictionary, the Application Generator obtains all its information about the tables your application uses, their relationships to one another, plus additional information such as predefined formatting for controls.

As a general rule, the more forethought and work you put into designing your data dictionary, the more Clarion's template-driven Application Generator can do for you. And, the more work the Application Generator does for you, the less you have to do yourself!

This chapter continues the examination of the Dictionary Editor that you began in the Getting Started lessons.

Starting Point:

You should have the Clarion development environment open and the Start Page opened.

Fill in the name of the new data dictionary

1. Navigate to the **Dictionaries** section of the Start Page, and press the **New Dictionary** button.

The *Save As* dialog appears.

Name the new dictionary file

1. In the **Save in** drop list, navigate to the ... \Lessons\LearningClarion directory.
2. Type *LCLesson* in the **File name** entry.

Clarion appends the file extension; *LCLesson.DCT* is the full name for the dictionary file.

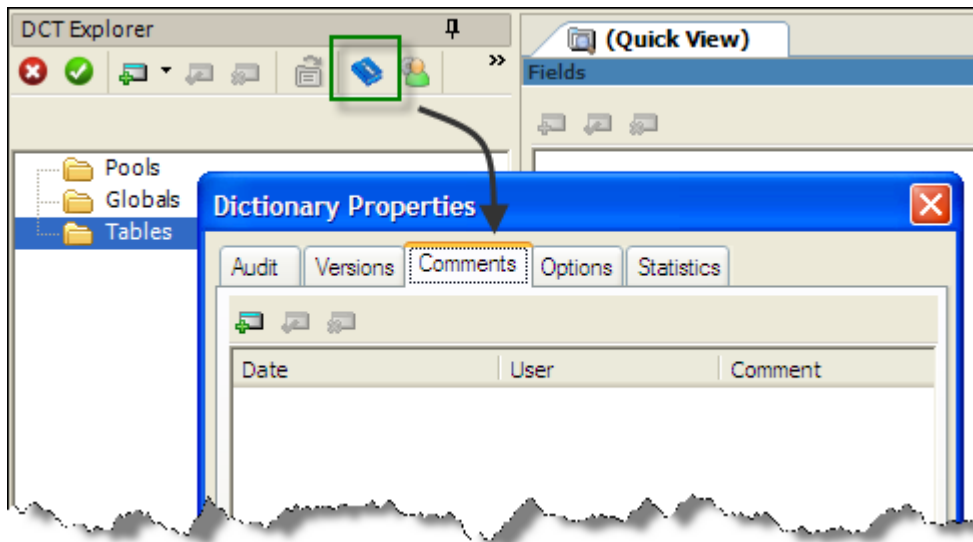
3. Press the **Save** button to create the file.

This creates an empty dictionary file. The tab of the Dictionary Editor shows the file name.

Specify a description for the dictionary

1. Press the Dictionary Properties button (located in the DCT Explorer toolbar).

The *Dictionary Properties* dialog appears.



2. Select the **Comments** tab, press the Add button in the toolbar, and then type *Learning Clarion Lesson Dictionary* in the text field.

The **Comments** tab allows you to write free form text notes regarding the dictionary. It's optional, but extremely useful for programmers who may have to return to a project for maintenance after an interval of months.

3. Close the *Dictionary Properties* dialog by pressing the **OK** button.

Note:

Just to the right of the Dictionary Properties button on the DCT Explorer toolbar, there is a **Users** button which opens a dialog where new users can be added if needed. The dialog also provides a **Password** entry, which allows you to prevent others from using this dictionary. There's no need to fill it in for this lesson, but it's a useful feature to keep in mind.

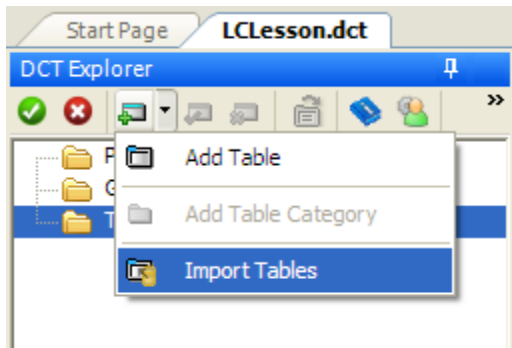
Copying Tables From One Dictionary to Another

You can use the standard Windows copy and paste commands to copy table definitions from another dictionary (or to copy columns from one table to another). In other words, once you've defined it once, why bother to re-define it when you can just copy what you've already done!

There is also a better and recommend way to copy table definitions from one Clarion Dictionary to another, by using the Table Import Wizard. We will use this better method in this exercise.

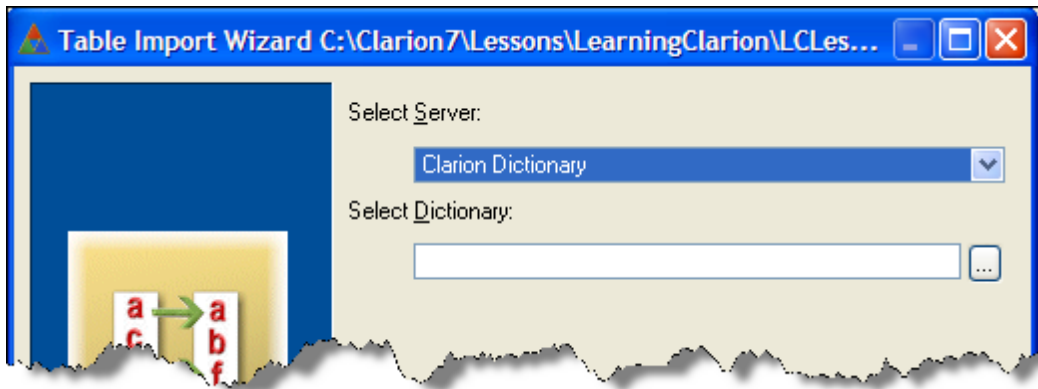
Import Tables from the *Getting Started* Dictionary

1. From the DCT Explorer, select the **Import Tables** option found on the DCT Explorer toolbar. Press **Yes** if you are prompted to save changes to the Dictionary.



The Table Import Wizard appears.

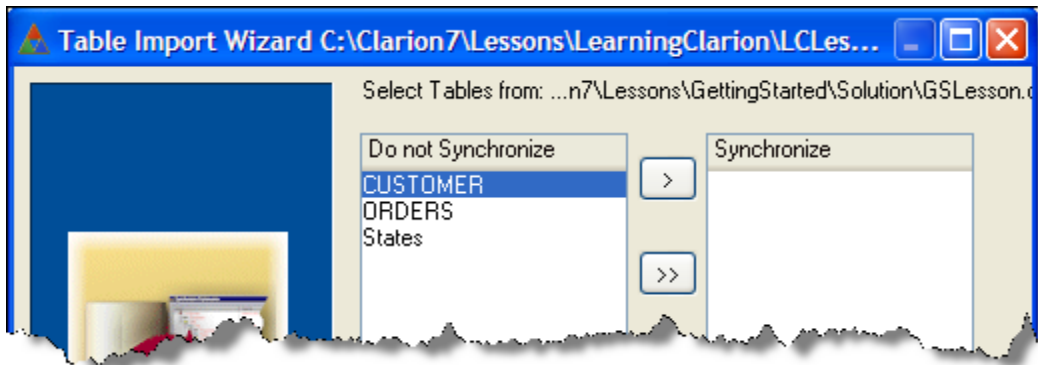
2. Verify that the **Select Server** selection is set to *Clarion Dictionary*. Press the ellipsis button to the right of the **Select Dictionary** prompt (or, you can simply press the **Next** button below) to select the dictionary to import.



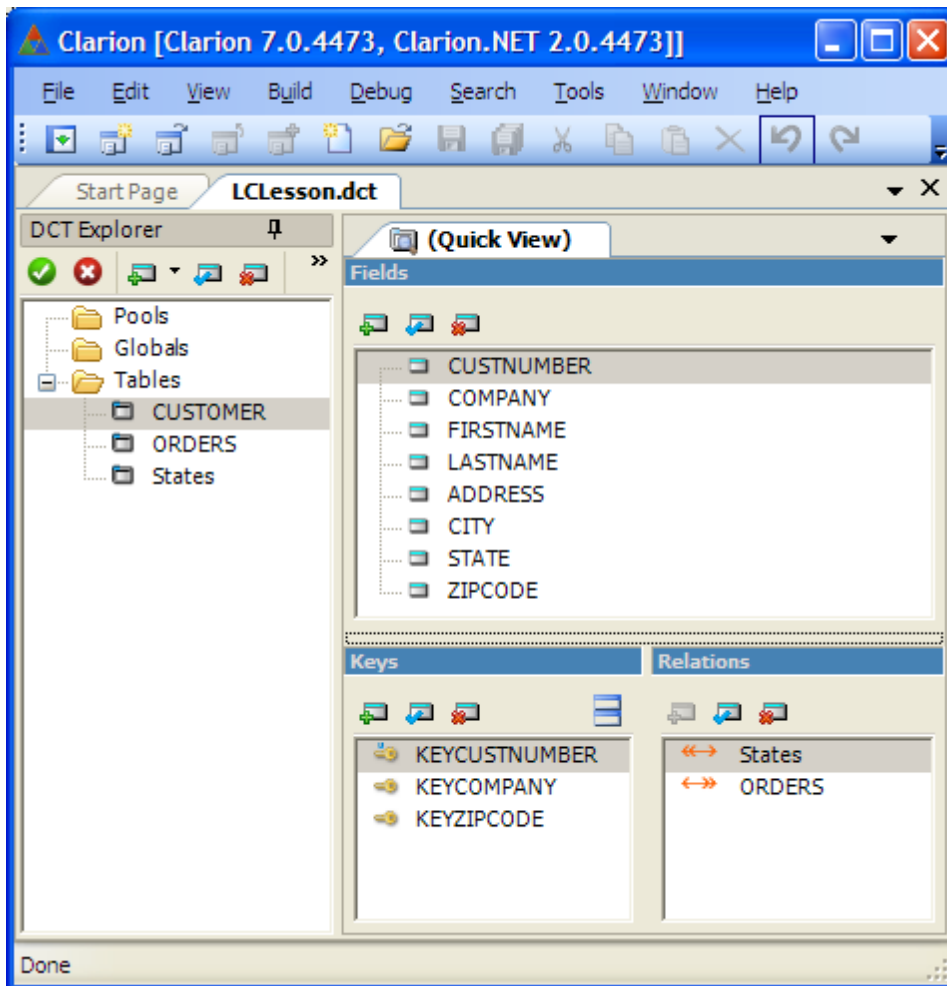
The *Select Dictionary* dialog opens, allowing you to select the other Clarion Dictionary to import from.

3. Select the *GSLESSON.DCT* located in the *CLARION7\GettingStarted\Solution* folder, and press the **Open** button.

The **Select Tables from...** dialog appears:



4. Press the **Add All** button to move all tables in the *GSLesson* dictionary to the new *LCLesson* dictionary, and then press **Finish** to complete the import process.



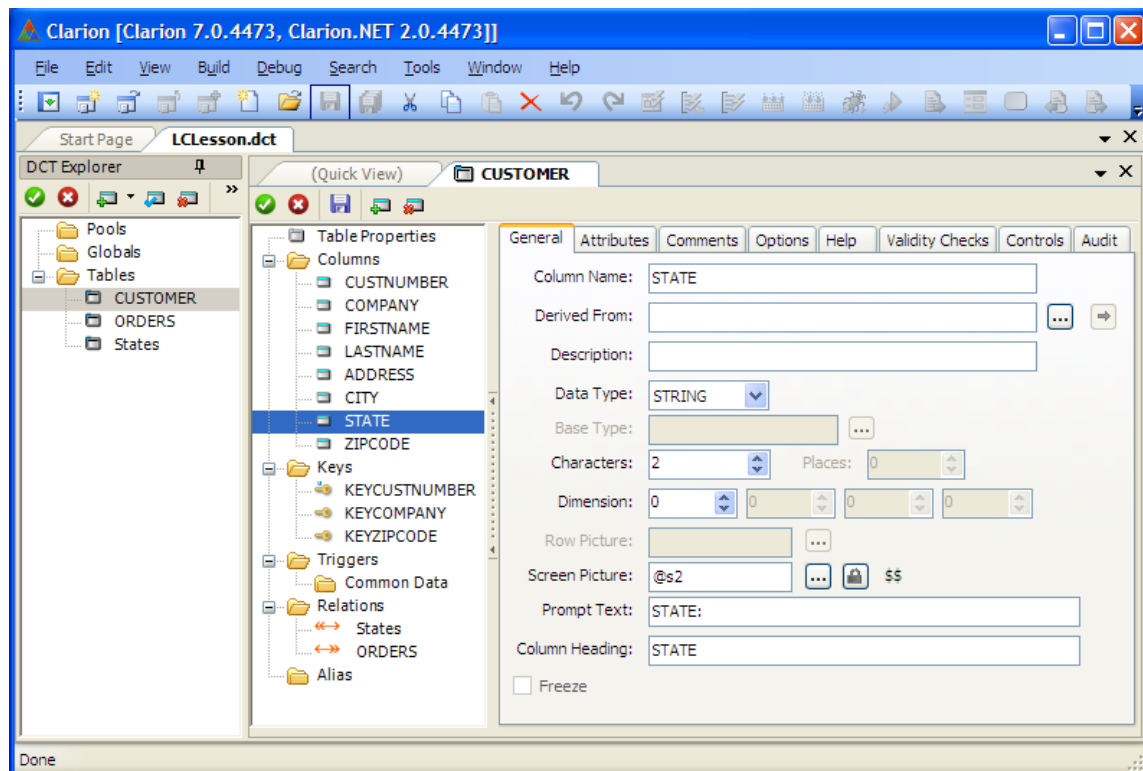
Relating the Tables

The beauty of using the **Table Import Wizard** is that all of the work that you did in that Dictionary, picture tokens, key attributes and relationships is preserved with the import. You can also copy (CTRL + C) table definitions (including their keys) from one DCT to another, but Clarion cannot copy the table relationships from other dictionaries. Therefore, you must re-define the relationship for tables, if you use the Copy/Paste technique.

Before we leave this section, we need to add one other small change to our new LCLesson dictionary.


Set a default value

1. Select the CUSTOMER table in the DCT Explorer, and in the *Fields Quick View*, double-click on the STATE column. The Entity Browser and Column Properties is opened:



2. Select the **Attributes** tab.
3. Type 'FL' in the **Initial Value** entry (including the single-quote marks).

This specifies that anytime the control appears, its default value will be "FL." Initial values can be time savers for the end user; in this case, if most customers were located in "FL," it saves picking it from the list each time a new customer has to be added. The single-quote marks are necessary because you can also name a variable or function as the initial value of a column in a table (be aware that there are slightly different rules for initial values of memory variables). In this case, the initial value is a string constant, as identified by the single quote marks around it.

4. Press the **Save and Close** button in the Entity Browser toolbar to close the *Column Properties* dialog, and the Customer table.
5. Choose **File** ► **Save**, or press the **Save** button  on the IDE tool bar.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- ✓ You created a new, empty data dictionary (.DCT).
- ✓ You imported existing table definitions from one data dictionary to another (the easy way to work—never re-invent the wheel).
- ✓ You added an initial value for one of the columns in the database.

In the next chapter, you'll learn how to add a table to the data dictionary, starting totally "from scratch". You'll see just how quick and easy it is to do even without using a wizard.

3 - Adding Tables and Columns

This section teaches you how to:

- Create new table definitions.
- Create a pool of column definitions from which new column definitions can be easily derived.
- Create additional column definitions for all tables.

Defining New Tables


After copying and modifying the three tables defined in the Getting Started dictionary, you're ready to add a new table from scratch.

Starting Point:

The **LCLesson.DCT** should be open.

Create the Phones File

Specify the label, prefix, and description

1. Make sure that the selection bar is in the DCT Explorer *Tables* section, and press the **Add Table**  button.

The *Add Table* dialog appears.

The 'Add Table' dialog box is shown with the 'General' tab active. It contains several input fields and a set of checkboxes. The 'Driver' dropdown is set to 'TOPSPEED'. The 'Create' and 'Threaded' checkboxes are checked, while 'Reclaim', 'OEM', 'Encrypt', and 'Bindable' are unchecked. The 'OK' and 'Cancel' buttons are at the bottom right.

2. Type *Phones* in the **Label** entry, then press TAB.

The **Label** column only accepts a valid Clarion label, which uniquely identifies the data structure. A label may only contain letters, numbers, and the underscore(_) or colon(:) characters, and must begin with a letter or underscore. Executable code statements use this label to refer to the table.

After pressing the tab key, "Pho" automatically appears in the **Prefix** entry. The prefix is one way to uniquely identify columns of the same name in different tables. For example, Pho:CustNumber is the CustNumber column in the Phones table while Cus:CustNumber is the CustNumber column in the Customers table. You can also uniquely identify columns by using Field Qualification syntax (discussed in the Language Reference).

3. Type *Customer phones table* in the **Description** field.

This description appears next to the table label in the Dictionary dialog list. If you select the **Comments** tab, you can type in a long text description. A description of what the table is for can be very helpful for when you return to the table for maintenance programming.

Choose the table driver

1. Verify that *TOPSPEED* is visible in the **Driver** dropdown list.

This declares the table format for the table as the TopSpeed table format. This is the newer of the two proprietary table formats that SoftVelocity has developed for use in Clarion (the older is "Clarion").

The Database Drivers topic/PDF documents all the available table drivers and provides information about what data types each one supports, plus other useful information such as the default table extensions for data and/or index files. It also provides tips and tricks for choosing the right driver for the job, such as which drivers are best when your application must handle a very


large database which is frequently updated, or which drivers are best when the quickest query time is the foremost concern.

2. Press the **OK** button to close the *Add Table* dialog.

You can accept the defaults for all other options in the dialog. The dialog box closes, and the Dictionary dialog lists the Phones table, with "Customer phones table" listed next to it (If that Dictionary Option is enabled).

Name the Detail and Products Tables

Create the Detail table

1. Make sure that the selection bar is in the DCT Explorer Tables section, and press the **Add Table**  button.
2. Type *Detail* in the **Label** entry.
3. Type *Order detail table* in the **Description** entry.
4. Type *DTL* in the Prefix entry.


By customizing the default prefix (changing it from "DET" TO "DTL"), you can make your code more readable. Three characters is the convention for table prefixes, but you are not limited to that.

5. Choose TOPSPEED from the **Driver** dropdown list.


Accept the defaults for all other options in the dialog.

6. Press the **OK** button.

Create the Products table

1. Make sure that the selection bar is in the DCT Explorer Tables section, and press the **Add Table**  button.
2. Type *Products* in the **Label** entry.
3. Type *Products for sale* in the **Description** entry.
4. Type *PRD* in the **Prefix** entry.
5. Choose *TOPSPEED* from the **Driver** dropdown list.
6. Press the **OK** button.

Save your work

From the IDE menu, choose **File** ► **Save**, or press the **Save** button  on the IDE tool bar.

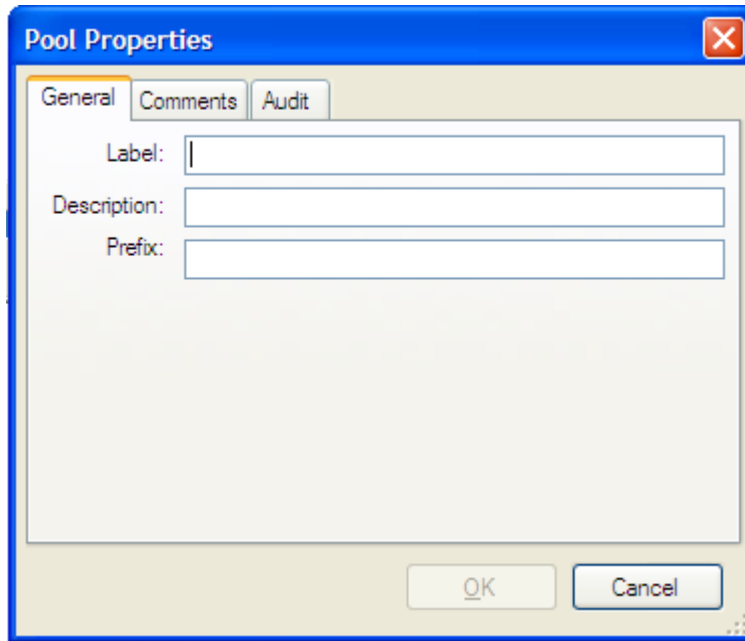
Defining the Columns

Define a Column Pool

At this point, we'll define several columns that will become the linking columns between the tables in our database. We'll use a feature of the Data Dictionary called a "Column Pool" to ensure that all tables that need these columns always define them exactly the same way.

1. **Making sure that the *Pools* section is selected in the DCT Explorer**, press the **Add Table** button.

This time you should see the *Pool Properties* dialog:

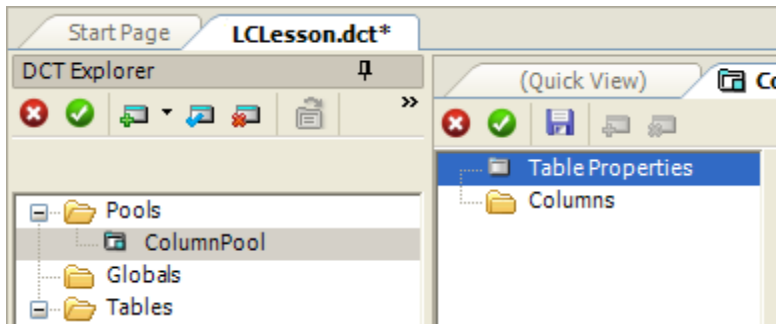


Column Pools are treated just like a table in the Data Dictionary, even though they do not generate any code into applications. You may have as many Column Pools in your data dictionary as you choose, but there is usually no need for more than one.

2. Type *ColumnPool* in the **Label** field.
3. Type *POOL* in the Prefix field.
4. Press the **OK** button.

Open the Column / Key Definition windows

1. Highlight the *ColumnPool* "table" in the **DCT Explorer** list, and double-click to open the Entity Browser:

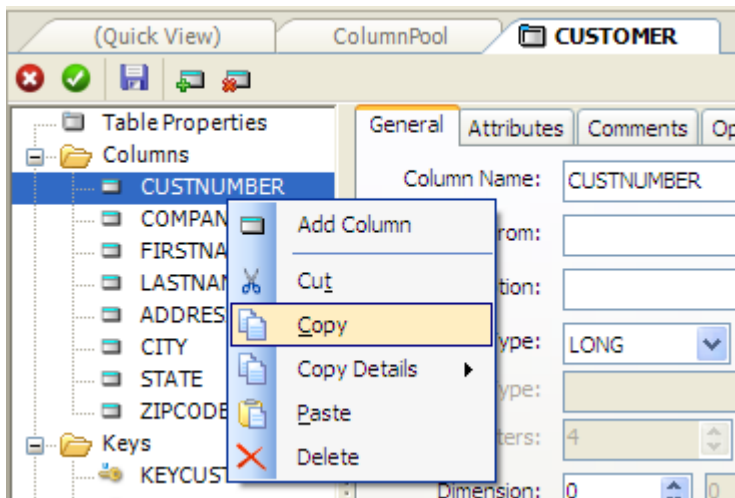


This window displays a list of all the columns defined in the Pool "table." Since this is a new table, there is nothing to display. We could simply start adding columns, but instead, we'll start by copying a column from the Customer table.

2. Highlight the *Customer* table in the **DCT Explorer** , and double-click to open the Entity Browser for this table.

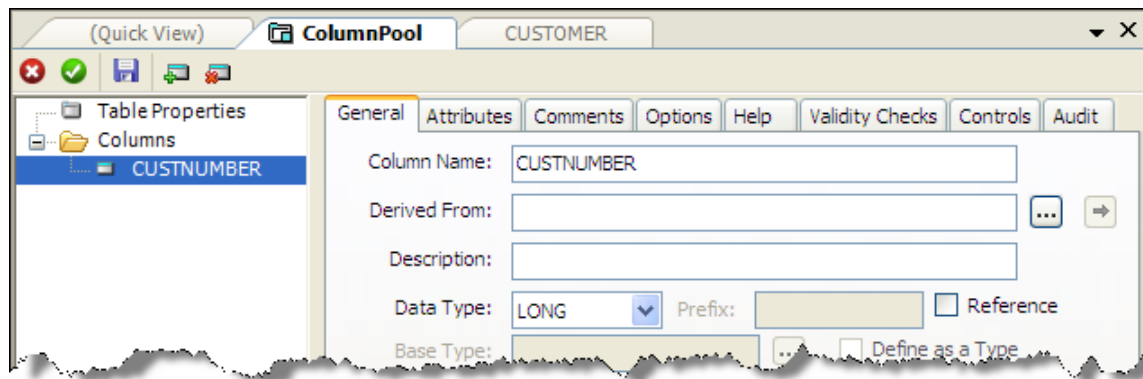
Select the column and copy it

1. In the Entity Browser, locate the *CustNumber* column in the **Columns** section.
2. Choose **Edit ► Copy** (or press CTRL+C), or you can right-click on the column and select **Copy** from the popup menu:



3. Click back on the *ColumnPool* tab to reopen its Entity Browser.
4. In the Columns section of the Entity Browser, select the Columns entry, and choose **Edit ► Paste** (or press CTRL+V).

The *Column Properties* dialog appears.



5. Press the **Save and Close** button to close the *Column Properties* dialog, and the *ColumnsPool* table..

This is the column that will be the linking column for the relationship between the Customer and Orders tables. Linking columns in separate tables are always defined the same, so copying the column definition is one way to get the existing column's definition into the Column Pool.

Derive the existing column

1. The Customer table should still be opened. If the CUSTNUMBER properties is not displayed, double-click on the CUSTNUMBER column to open it.
2. Type POOL:CustNumber in the **Derived From** entry (or, you could press the ellipsis button to the right of the entry, and select the *CustNumber* field from the *ColumnPool* table). Press the TAB key.

The **Set Freeze Checkbox** dialog pops up. Press **Don't Freeze** to allow changes in the Derived Column to cascade changes in the ColumnPool column if they are made.


This means that the CUS:CustNumber column is now *derived from* the POOL:CustNumber column. The term "derived from" means that the POOL:CustNumber column's definition is the "parent" and all "children" columns which are "derived from" that column automatically share all the attributes of the parent.

Deriving column definitions from existing columns gives you the ability to make changes in only one place, then cascade those changes to all derived columns. For example, if the definition of the CustNumber column needs to change in all tables using it, simply make one change to the POOL:CustNumber column definition, then cascade that change to all the derived CustNumber columns in all tables.

3. Press the **Save and Close** button to close the *Column Properties* and Customer table Entity Browser.

Add the rest of the columns to the Column Pool

Just to show you a different approach, let's use the QuickView at this time to complete the ColumnPool table.

1. In the **DCT Explorer**, highlight (select) the *ColumnPool* table.
2. In the Quick View, press the **Add** button  to open the *Column Properties* dialog.

Once you begin the process of defining new columns, an empty *Column Properties* dialog automatically appears after you add each successive column. This speeds up the process of adding multiple columns. After adding your last column, you just have to press the **Cancel** button on an empty *Column Properties* dialog to return to the **Quick View**.

3. Type *OrderNumber* in the **Column Name** entry.

This will be the linking column between the Orders and Detail tables.

4. Choose *LONG* from the **Data Type** dropdown list.

This specifies a four byte signed integer. See the *Language Reference* in the core help for more details.

5. Type @n_6 in the **Screen Picture** entry.
6. Press the **OK** button.

A new *Column Properties* dialog opens again.

7. Type *ProdNumber* in the Column Name entry.

This will link the Detail table to the Products table.

8. Choose *LONG* from the Data Type dropdown list.
9. Type @n_6 in the **Screen Picture** entry.
10. Press the **OK** button.


The *Column Properties* dialog re-appears.

11. On the next *Column Properties* dialog, press the **Cancel** button.
12. Press the **Save** button in the IDE toolbar to save your work to this point.

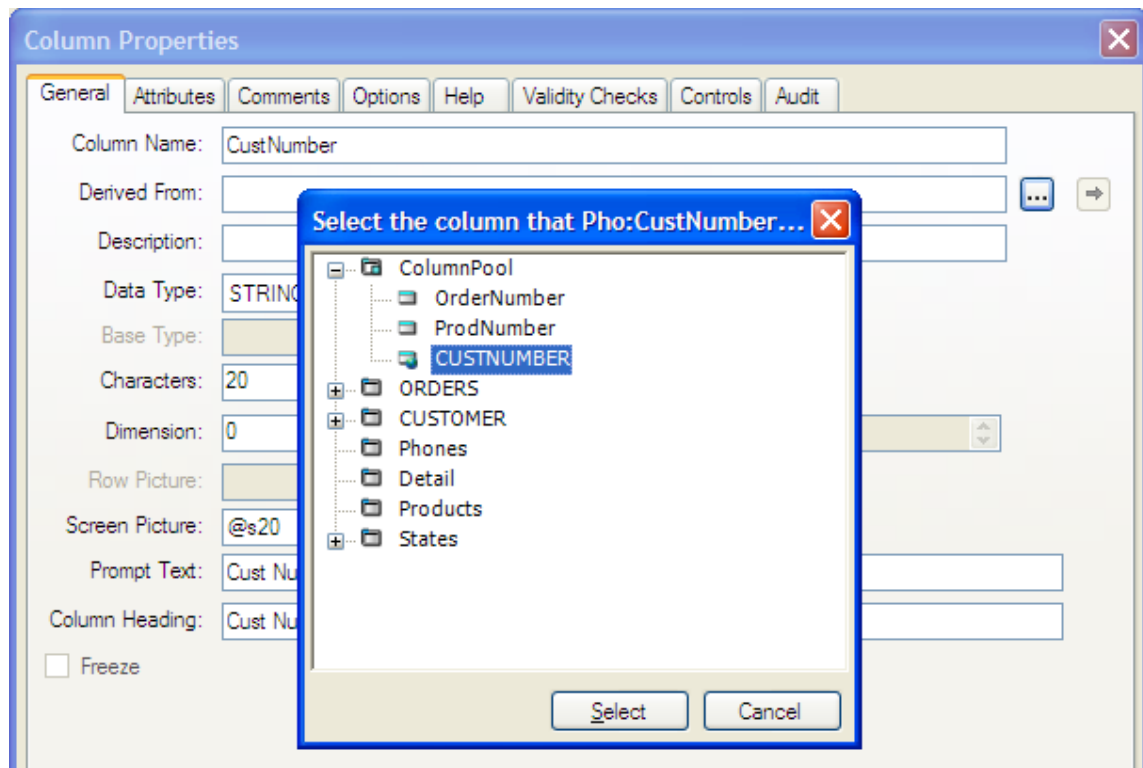
Define the columns in the Phones Table

At this point, go back to the *Phones* table and prepare to define its columns.

Derive the CustNumber Column

1. Highlight the *Phones* table in the **DCT Explorer**.
2. In the **Quick View Fields** pane on the right, press the **Add**  button to open the *Column Properties* dialog.
3. Type *CustNumber* in the **Column Name** entry.
4. Press the ellipsis (...) button to the right of the **Derived From** entry.

A *Select* window appears containing tree lists of all the columns already defined for all the tables in the dictionary. You can derive new columns from any existing column—whether that column is in a table definition, global data, or a column pool.



5. Highlight the *CustNumber* column in the *ColumnPool* table then press the **Select** button. You may need to expand the *ColumnPool* table to see the columns under it.

The **Set Freeze Checkbox** dialog pops up. Press **Don't Freeze** to allow changes in the Derived Column to cascade changes in the *ColumnPool* column if they are made.

The new column automatically becomes a perfect copy of the column from which it was derived—right down to the prompts and window control type. The button with the right arrow icon right next to the ellipsis (...) button allows you to refresh the derived column from its parent's definition.

6. Press the **OK** button to close the *Column Properties* dialog.

This is the column that will provide the link between the *Phones* and *Customer* tables.

A new *Column Properties* dialog now opens.

Define the PhoneType Column, and Add Special Characteristics

This defines the type of phone number (Home, Work, etc.). We will also add a special *Validity Check*, *Default Value*, and *Window Control* to this column.

1. Type *PhoneType* in the **Column Name** entry.
2. Choose *STRING* from the **Data Type** dropdown list.
3. Type 6 in the **Characters** spin control.
4. Select the **Validity Checks** tab.

The *Validity Checks* tab allows you to set numeric ranges for number columns, specify that a column value must match another column in a related value, must be true or false, and in this case, that the column value must be in a list you specify in this dialog.

5. Select the **Must be in List** radio button.
6. Type the following in the **Choices** box:

Home|Work|Fax|Cell|Mobile|Other

A vertical bar (|) must separate each choice.

This defines the actual list of allowable choices. In this case, the dictionary specifies that only these types of phone values are acceptable.

7. Set a default value. Select the **Attributes** tab.
8. Type 'Home' in the **Initial Value** column (including the single-quote marks).

This specifies that anytime the control appears, its default value will be "Home." Initial values can be time savers for the end user; in this case, if most customers call from "Home", it saves picking it from the list each time a new customer phone number has to be added. The single-quote marks are necessary because you can also name a variable or function as the initial value of a column in a table (be aware that there are slightly different rules for initial values of memory variables). In this case, the initial value is a string constant, as identified by the single quote marks around it.

9. Specify a default window control. Select the **Controls** tab.

When you specify a **Must be in List** option, the default window control for the column is an OPTION structure with RADIO buttons. These appear by default in the Window Controls list.

10. Select *LIST* from the **Control Type** list box.

The Window Controls list now updates to show only a PROMPT and a LIST control with a DROP attribute.

11. Press the **OK** button to close the *Column Properties* dialog.

A new *Column Properties* dialog now opens.

Define the OutsideUSA Column, set a Validity Check

This column identifies a phone number as domestic or foreign. This will be used as a flag to control the formatting of the phone number column.

1. Type *OutsideUSA* in the **Column Name** entry.
2. Choose *BYTE* from the **Data Type** dropdown list.
3. Select the **Validity Checks** tab.
4. Select the **Must be True or False** radio button.

This will create a check box for this column by default.

5. Press the **OK** button to close the *Column Properties* dialog.

A new *Column Properties* dialog appears.

Define the PhoneNumber Column

1. Type *PhoneNumber* in the **Column Name** entry.
2. Choose *STRING* from the **Data Type** dropdown list.
3. Type 20 in the **Characters** spin control.

This seems a little large for a normal phone number, but we need to account for a possible international number that could be larger.

4. Type the following in the **Screen Picture** prompt:

@P(###)###-####P

This will set a "mask" for domestic phone numbers. We'll use the *OutsideUSA* column in our application to change the picture later to accommodate international phone numbers.

5. Press the **OK** button to close the *Column Properties* dialog.

A new *Column Properties* dialog appears.

Define the Extension Column

1. Type *Extension* in the **Column Name** entry.
2. Choose *STRING* from the **Data Type** dropdown list.
3. Type *10* in the **Characters** spin control.
4. Press the **OK** button to close the *New Column Properties* dialog.

A new *Column Properties* dialog appears.

Save Your Work!


All the columns are defined, and a blank *Column Properties* dialog should be active.

1. Press the **Cancel** button to close the *Column Properties* dialog.
2. Press the **Save** button in the IDE toolbar to save your work.

Define the columns for the Detail Table

At this point, we'll define the columns for the Detail table.

Derive the linking column definition

1. Highlight the *Detail* table in the **DCT Explorer** list.
2. In the **Quick View Fields** pane on the right, press the **Add**  button to open the *Column Properties* dialog.
3. Type *OrderNumber* in the **Column Name** entry.
4. Press the ellipsis (...) button to the right of the **Derived From** entry.
5. Highlight the *OrderNumber* column in the *ColumnPool* table then press the **Select** button. The **Set Freeze Checkbox** dialog pops up. Press **Don't Freeze** to allow changes in the Derived Column to cascade changes in the *ColumnPool* column if they are made.
6. Press the **OK** button to close the *Column Properties* dialog.

This is the column that will be the link between the *Orders* and *Detail* tables.

A new *Column Properties* dialog appears.

Derive the ProdNumber column

This column allows you to relate this table and the Products table.

1. Type *ProdNumber* in the **Column Name** entry.
2. Press the ellipsis (...) button to the right of the **Derived From** entry.
3. Highlight the *ProdNumber* column in the *ColumnPool* table then press the **Select** button. The **Set Freeze Checkbox** dialog pops up. Press **Don't Freeze** to allow

changes in the Derived Column to cascade changes in the ColumnPool column if they are made.

4. Press the **OK** button to close the *Column Properties* dialog.

A new *Column Properties* dialog appears.

Define the Quantity column

This stores the number of each product ordered.

1. Type *Quantity* in the **Column Name** entry.
2. Choose *SHORT* from the **Data Type** dropdown list.
3. Press the **OK** button.

A new *Column Properties* dialog appears.

Define the ProdAmount column

This stores the unit cost of the product as it was at the time of the order.

1. Type *ProdAmount* in the **Column Name** entry.
2. Choose *DECIMAL* from the **Data Type** dropdown list.
3. Type 5 in the **Characters** spin control.
4. Type 2 in the **Places** spin control.
5. Press the **OK** button.

A new *Column Properties* dialog appears.

Define the TaxRate column

1. Type *TaxRate* in the **Column Name** entry.
2. Choose *DECIMAL* from the **Data Type** dropdown list.
3. Type 2 in the **Characters** spin control.
4. Type 2 in the **Places** spin control.
5. Press the **OK** button.

A new *Column Properties* dialog appears.

Close the dialogs, save your work!


All the columns are defined, and a blank *Column Properties* dialog should be active.

1. Press the **Cancel** button to close the *Column Properties* dialog.
2. Press the **Save** button in the IDE toolbar to save your work.

Define the columns for the Products Table

At this point, we'll define the columns for the Products table. This is the last table. We'll also show you an alternate way to add columns here.

Derive the linking column definition

1. Highlight the *Products* table in the **DCT Explorer** list.
2. In the **Quick View Fields** pane on the right, press the **Add**  button to open the *Column Properties* dialog.
3. Type *ProdNumber* in the **Column Name** entry.
4. Press the ellipsis (...) button to the right of the **Derived From** entry.
5. Highlight the *ProdNumber* column in the *ColumnPool* table then press the **Select** button. The **Set Freeze Checkbox** dialog pops up. Press **Don't Freeze** to allow changes in the Derived Column to cascade changes in the ColumnPool column if they are made.
6. Press the **OK** button to close the *Column Properties* dialog.

This is the column that will be the linking column for the relationship between the Detail and Products tables.

A new *Column Properties* dialog appears.

Define the ProdDesc column

This allows for a product description.

1. Type *ProdDesc* in the **Column Name** entry.
2. Choose *STRING* from the **Data Type** dropdown list.
3. Type 30 in the **Characters** spin control.
4. Press the **OK** button.

A new *Column Properties* dialog appears.

Define the ProdAmount column

This stores the unit cost of the product.

1. Type *ProdAmount* in the **Column Name** entry.
2. Choose *DECIMAL* from the **Data Type** dropdown list.
3. Type 5 in the **Characters** spin control.
4. Type 2 in the **Places** spin control.
5. Press the **OK** button.

A new *Column Properties* dialog appears.

Define the TaxRate column


1. Type *TaxRate* in the **Column Name** entry.
2. Choose *DECIMAL* from the **Data Type** dropdown list.
3. Type 2 in the **Characters** spin control.
4. Type 2 in the **Places** spin control.

5. Press the **OK** button.

A new *Column Properties* dialog appears.

Close the dialogs and save your work

All the columns are defined, and a blank *Column Properties* dialog should be active.

1. Press the **Cancel** button to close the *Column Properties* dialog.
2. Choose **File ► Save**, or press the **Save** button  on the tool bar to save your work up to this point.

All of the tough work is done! Hang in there, we are almost there!

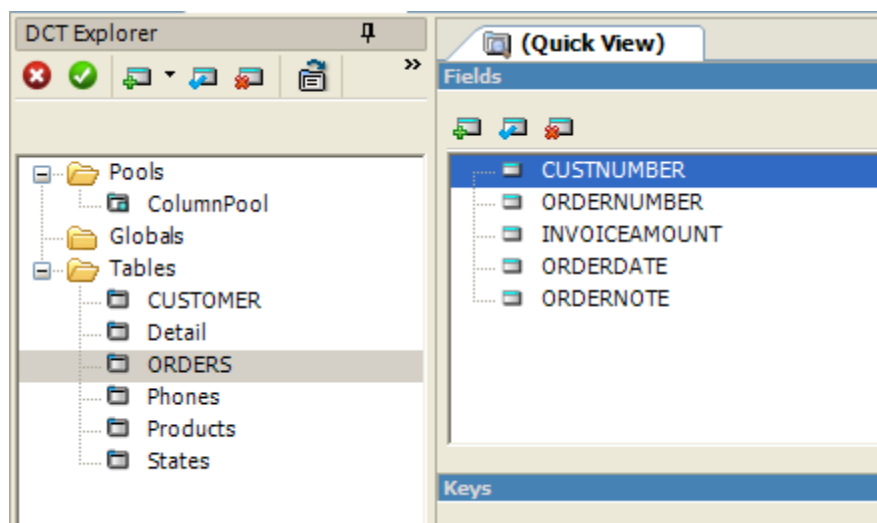
Clean up of the Orders table import

Before we start the next section, let's clean up a couple of items from the *Orders* table that we imported in the *Getting Started* lesson.

Derive the CustNumber column

This provides a unique identifier for each order

1. Highlight the *Orders* table in the **DCT Explorer**, and double-click on the *CustNumber* column in the **Quick View Fields** list.



2. Press the ellipsis (...) button to the right of the **Derived From** entry.
3. Highlight the *CustNumber* column in the *ColumnPool* table then press the **Select** button. The **Set Freeze Checkbox** dialog pops up. Press **Don't Freeze** to allow changes in the Derived Column to cascade changes in the *ColumnPool* column if they are made.
4. Press the **Save and Close** button to close the *Column Properties* dialog.

Derive the OrderNumber column

This provides a unique identifier for each order

1. Highlight the *Orders* table in the **DCT Explorer**, and double-click on the *OrderNumber* column in the **Quick View Fields** list.
2. Press the ellipsis (...) button to the right of the **Derived From** entry.
3. Highlight the *OrderNumber* column in the *ColumnPool* table then press the **Select** button. The **Set Freeze Checkbox** dialog pops up. Press **Don't Freeze** to allow changes in the Derived Column to cascade changes in the ColumnPool column if they are made.
4. Press the **Save and Close** button to close the *Column Properties* dialog.

Close the dialogs

1. Press the **Save** button in the IDE Toolbar to save your work so far.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- ✓ You created new table definitions.
- ✓ You created a pool of column definitions from which new column definitions can be easily derived.
- ✓ You created the column definitions for all the tables.

Now we'll go on to add keys and table relationships.

4 - Adding Keys and Relations

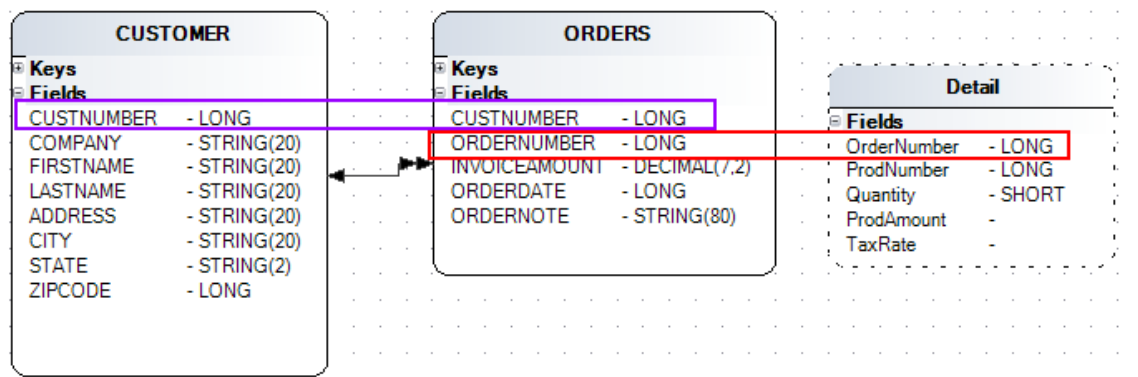
Now that all the tables are defined, we can add keys then specify the table relationships. You already have defined the keys for the two tables you created in the *GSLesson* application in the Getting Started lesson. In this chapter, we'll define keys for the remaining tables.

Starting Point:

The **LCLESSON.DCT** should be open.

Examining the Keys for the Orders Table

The columns in the Orders table that relate to other tables in the database are the OrderNumber and CustNumber columns.



- The OrderNumber column relates to the Detail table.

There should be no duplicate or null order numbers in the Orders table; this is a *primary* key.

There may be multiple Detail rows for a single matching Order Number. Therefore, this is a *One to Many* relationship—the Orders table is the "Parent" of the Detail table.

- The CustNumber column relates to the Customer table.

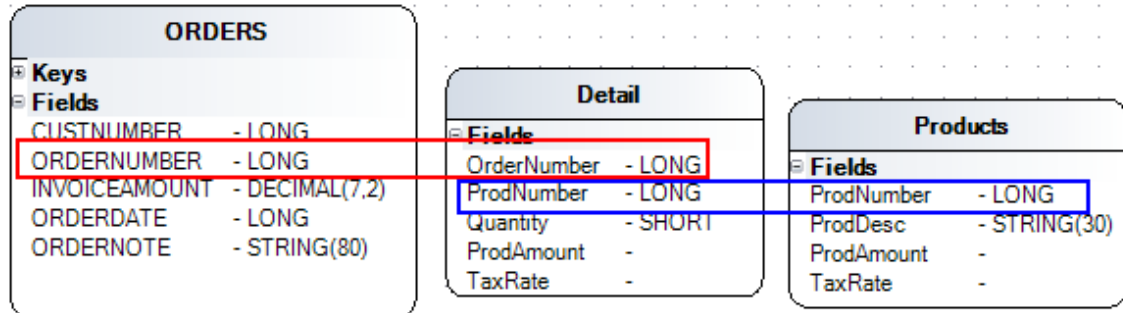
There will be duplicate values in the CustNumber column that relate to rows in the Customers table. The key we define in the Orders table is a *foreign* key. The Customers table key does not allow duplicates and nulls, and was defined as the primary key for that table.

Multiple Order rows can exist for each Customer, making this a *Many to One* relationship—the Orders table is the "Child" of the Customers table.

The Orders table was imported in the Getting Started lesson, and there are two keys already defined that satisfy the above requirements.

Defining Keys for the Detail Table

The columns in the Detail table that relate to other tables in the database are the ProdNumber and OrderNumber columns.



- The OrderNumber column relates to the Orders table.

There will be duplicate values in the OrderNumber column that relate to rows in the Orders table. The key we define in the Detail table is another *foreign* key. The Orders table key does not allow duplicates and nulls, and was defined as a primary key.

There may be more than one Detail row for a single matching Order Number. Therefore, this is a *Many to One* relationship, with the Detail table the "Child" of the Orders table.

- The ProdNumber column relates to the Products table.

There will be duplicate values in the ProdNumber column for the rows in the Detail table. There may be more than one Detail row containing a single Product Number. Therefore, this is another *Many to One* relationship, with the Detail table the "Child" of the Product table.

Define the First Foreign Key

Define *ProdNumberKey* so that there may be duplicate *ProdNumber* values in this table.

- Highlight the *Detail* table in the **DCT Explorer** list.
- In the *Keys* view, press the **Add** button.




- Type *ProdNumberKey* in the **Label** entry.

4. Select the **Columns** tab.
5. Press the **Add** button.
6. Select *ProdNumber* then press the **Select** button.
7. Press the **Cancel** button in the *Select a Column* dialog, and then press **OK** to close the *Key Properties* dialog.

A blank *Key Properties* dialog appears, ready for you to specify another key.

Define the Second Foreign Key

1. Type *OrderNumberKey* in the **Label** entry.
2. Select the **Columns** tab.
3. Press the **Add** button.
4. Select *OrderNumber* then press the **Select** button.
5. Press the **Cancel** button in the *Select a Column* dialog.
6. Press the **OK** button to close the *Key Properties* dialog.
7. Press the **Cancel** button to close the blank *Key Properties* dialog.
8. Choose **File** ► **Save**, or press the **Save** button  on the tool bar.

Defining Keys for the Products Table

Only one column in the Products table relates to another table in the database: the *ProdNumber* column.

- The *ProdNumber* column relates to the Detail table.


Detail		Products	
Fields		Fields	
OrderNumber	- LONG	ProdNumber	- LONG
ProdNumber	- LONG	ProdDesc	- STRING(30)
Quantity	- SHORT	ProdAmount	- DECIMAL(5,2)
ProdAmount	- DECIMAL(5,2)	TaxRate	- DECIMAL(2,2)
TaxRate	- DECIMAL(2,2)		

There should be no duplicate or null order numbers in the Products table; this is a *primary* key.

For each *ProdNumber* in the row there can be many Detail rows. This is a *One to Many* relationship with the Products table a "Parent" to the Detail table.

Create the Primary Key


Name the Key

1. Highlight the *Products* table in the **DCT Explorer** list.
2. In the *Keys* view, press the **Add**  button.
3. Type *ProdNumberKey* in the **Label** entry.
4. In the *Attributes* group, check the **Require Unique Value** box, then check the **Primary Key** box.
5. Check the **Auto Number** box.
6. Select the **Columns** tab.
7. Press the **Add** button.
8. In the *Select a Column* window, select *ProdNumber* then press the **Select** button.
9. Press the **Cancel** button to close the *Select a Column* window.
10. Press the **OK** button to close the *Key Properties* dialog.

A blank *Key Properties* dialog appears, ready for you to specify another key.

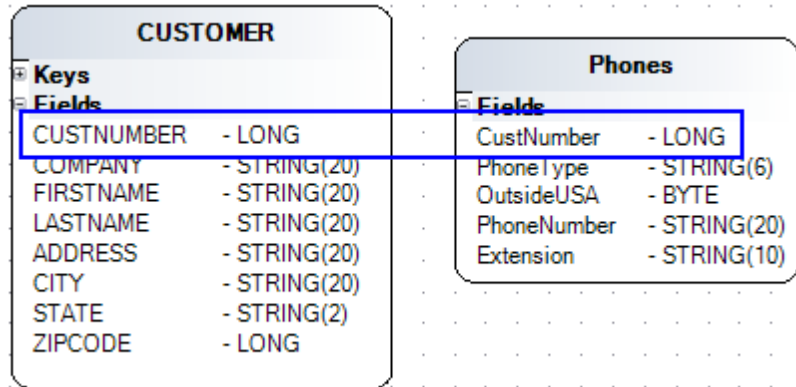
Define an Alphabetical Key

Users will probably want to see the list of *Products* in alphabetical order, so we'll add a key for that.

1. Type *ProdDescKey* in the **Label** entry.
2. Select the **Columns** tab.
3. Press the **Add** button.
4. Select *ProdDesc* then press the **Select** button.
5. Press the **Cancel** button to close the *Select a Column* window.
6. Press the **OK** button to close the *Key Properties* dialog.
7. Press the **Cancel** button to close the blank *Key Properties* dialog.
8. Choose **File** ► **Save**, or press the **Save** button  on the tool bar.

Defining a Key for the Phones Table

The column in the Phones table that relate to the Customer table in the database is the CustNumber column.





- The CustNumber column relates to the Customer table.

There will be duplicate values in the CustNumber column that relate to rows in the Customer table. The key we define in the Phones table is another *foreign* key. The Customer table key does not allow duplicates and nulls, and was defined as a primary key.

There may be more than one Phones row for a single matching Customer Number. Therefore, this is a *Many to One* relationship, with the Phones table the "Child" of the Customer table.

Define the Foreign Key

Define *CustNumberKey* so that there may be duplicate *CustNumber* values in this table.

1. Highlight the *Phones* table in the **DCT Explorer** list.
2. In the *Keys* view, press the **Add**  button.
3. Type *CustNumberKey* in the **Label** entry.
4. Select the **Columns** tab.
5. Press the **Add** button.
6. Select *CustNumber* then press the **Select** button.
7. Press the **Cancel** button to close the *Select a Column* window.
8. Press the **OK** button to close the *Key Properties* dialog.
9. Press the **Cancel** button to close the blank *Key Properties* dialog.
10. Choose **File** ► **Save**, or press the **Save** button  on the tool bar.

Defining Table Relationships

Note:


The relationships for the *Customer*, *Orders*, and *States* tables were defined in the Getting Started lessons. Refer back to that section in the Getting Started if you would like to review those definitions.

Defining Relationships for the Phones Table

Now that all the keys are defined, we can add the relations. Once you have defined relationships, you can add Validity Checks for the columns that should only contain values that exist in another table. These are the last steps to completing the data dictionary.

- CustNumberKey relates the Phones table to the Customer table in a *Many to One* relationship.

Define the relationship

1. Highlight the *Phones* table in the **DCT Explorer** if not already highlighted.
2. In the lower right pane, press the **Add** button  located just above the **Relations** list.

The *Relationship Properties* dialog appears:

Relationship Properties

General | Comments | Options | Audit

Relationship for Phones

Type: **1:MANY** Primary Key: **None**

Child

Related Table: **None** Foreign Key: **None**

Column Mappings

Phones -> Related Key

Related Table:-> Key

Map By Name Map By Order

Referential Integrity Constraints

On Update: **None** On Delete: **None**

OK Cancel

- Choose *MANY:1* from the **Type** dropdown list.

Notice that, when you chose *MANY:1*, the prompts for the Primary Key and Foreign Key columns switched places. This happens because we are now defining this relationship from the "Child" table's viewpoint; the opposite side of the relationship to what we did previously. A Primary Key is always in the Parent table, while a Foreign Key is always in the Child table.

- Choose *Pho:CustNumberKey* from the **Foreign Key** dropdown list.
- Choose *Customer* from the **Related Table** dropdown list.

This establishes the Customer table as the "Parent" in this relationship.


- Choose *CUS:KeyCustNumber* from the **Primary Key** dropdown list.
- Press the **Map by Name** button.

Set up the Referential Integrity constraints

- Choose *Cascade* from the **On Update** dropdown list.

Although we are defining this relationship from the "Child" table's viewpoint, the Referential Integrity constraints are still set on the "Parent" table actions.

- Choose *Cascade* from the **On Delete** dropdown list.

3. Press the **OK** button.
4. Press the **Cancel** button to close the new *Relationship Properties* window.
5. Choose **File ► Save**, or press the **Save** button  on the tool bar.


Defining Relationships for the Detail Table

Each time you define a relationship in the Dictionary Editor, you define it for both tables at the same time.

The relationships for the Detail table:

- *ProdNumberKey* relates the Detail table to the Products table in a Many to One relationship.
- *OrderNumberKey* relates the Orders table to the Detail table in a One to Many relationship. From the Detail table, you can also look at it as "Many detail records relate to a single Order record".


Define the Detail-Product relationship

1. Highlight the *Detail* table in the **DCT Explorer** list.
2. In the lower right pane, press the **Add** button  located just above the **Relations** list. The *Relationship Properties* dialog appears:
3. Choose *MANY:1* from the **Type** dropdown list.
4. Choose *DTL:ProdNumberKey* from the **Foreign Key** dropdown list.
5. Choose *Products* from the **Related Table** dropdown list.
6. Choose *PRD:ProdNumberKey* from the **Primary Key** dropdown list.
7. Press the **Map by Name** button.


Set up the Referential Integrity constraints

1. Choose *Restrict* from the On Update dropdown list.

We won't allow any changes to the product numbers.


2. Choose *Restrict* from the On Delete dropdown list.
3. Press the **OK** button.
4. Press the **Cancel** button to close the new *Relationship Properties* window.
5. Choose **File ► Save**, or press the **Save** button  on the tool bar.

Define the Detail-Order relationship

1. Highlight the *Detail* table in the **Tables** list.
2. In the lower right pane, press the **Add** button  located just above the **Relations** list. The *Relationship Properties* dialog appears:

3. Choose *MANY:1* from the **Type** dropdown list.
4. Choose *DTL:OrderNumberKey* from the **Foreign Key** dropdown list.
5. Choose *Orders* from the **Related Table** dropdown list.
6. Choose *ORD:KeyOrderNumber* from the **Primary Key** dropdown list.
7. Press the **Map by Name** button.

Set up the Referential Integrity constraints


1. Choose *Restrict* from the On Update dropdown list.
2. We won't allow any changes to the product numbers.
3. Choose *Restrict* from the On Delete dropdown list.
4. Press the **OK** button.
5. Press the **Cancel** button to close the new *Relationship Properties* window.
6. Choose **File** ► **Save**, or press the **Save** button  on the tool bar.

Defining Relationship-Dependent Validity Checks

Now that all the table relationships are defined, we can set the Validity Checks for two columns that we expect to use on update forms.

- When entering a new *Orders* row, we can specify that the *CustNumber* must match an existing row in the *Customer* table.
- When entering a new *Detail* row, we can specify that the *ProdNumber* must match an existing row in the *Products* table.

Define the Validity Check for Order Rows

1. Highlight the *Orders* table in the **DCT Explorer** list.
2. In the **Fields Quick View**, highlight *CustNumber* and press the **Change**  button.
3. Select the **Validity Checks** tab.
4. Select the **Must Be In Table** radio button.
5. Choose *Customer* from the **Table Label** dropdown list. Actually, the IDE detects that it is the only valid table here and selects it for you.

This requires that the column can only contain values verified by getting a matching row from the *Customer* table. This is validated using the table relationship information, which is why this Validity Check cannot be set until the relationships have been defined.

6. Press the **Save and Close** button in the Entity Browser to close the *Orders* table.

Define the Validity Check for Detail Rows

1. Highlight the *Detail* table in the **DCT Explorer** list.
2. In the **Fields Quick View**, highlight *ProdNumber* and press the **Change**  button.

3. Select the **Validity Checks** tab.
4. Select the **Must Be In Table** radio button.
5. Choose *Products* from the **Table Label** dropdown list. Again, this is auto-selected for you.
6. Press the **Save and Close** button in the Entity Browser to close the Orders table.
7. Press the **Save** button to save your work so far.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- ✓ You created the keys for all the new table definitions.
- ✓ You defined the relationships between the new tables.
- ✓ You defined two relationship-dependent validity checks to require that foreign key column values always have related primary key rows in a parent table.

The data dictionary is now complete to this point. In the next chapter, we will import some existing data from another application, to show you just how simple it is to accomplish.

5 - Importing Existing Data

Data Conversion

You may have existing data from legacy applications that you want to save and use in your Clarion applications. Therefore, this chapter shows you:

- How to import a table definition from an existing table.
- How to browse and edit a table using the Database Manager.
- How to convert data from one table format to another.

Starting Point:

The **LCLESSON.DCT** should be open.

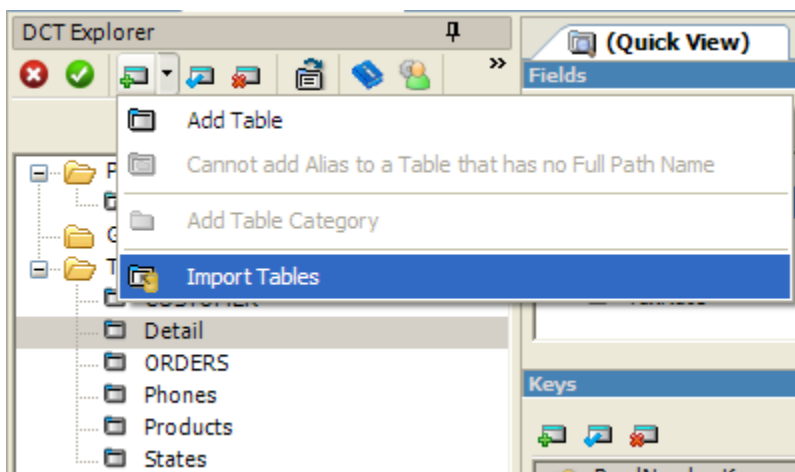
Importing a .CSV File Definition

One easy way to convert tables is to export your old data from the previous application to Comma Separated Values (.CSV) files. This is the file format originally used by the Basic language—the data is contained in DOUBLE-quotes, commas separate fields, and a Carriage Return/Line Feed separates records. Clarion's BASIC file driver will easily read from and write to these .CSV files.

We will import the definition of an existing .CSV file containing Customer data, then generate a simple table conversion program (to show you just how easy it is to do) to place the data into a TopSpeed table.

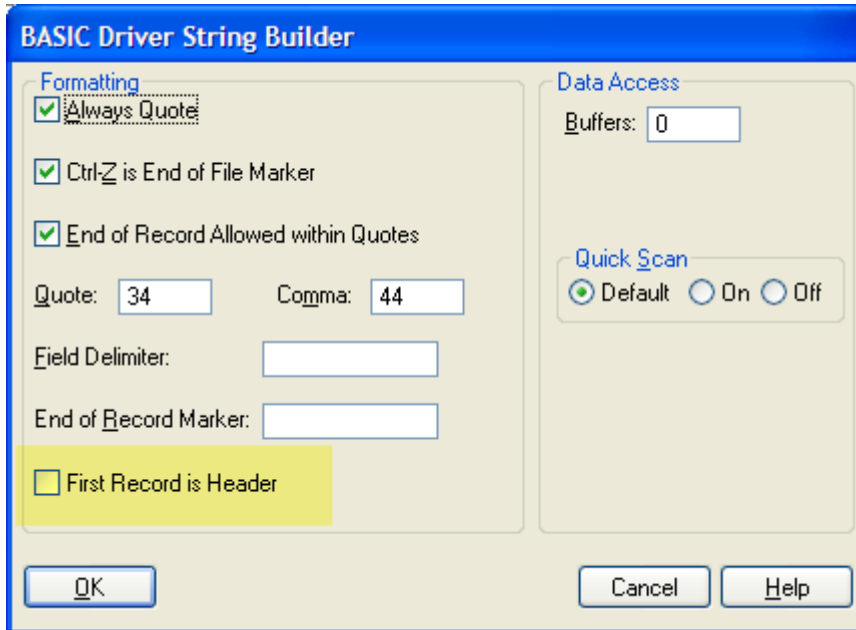
Import the file definition

1. In the **DCT Explorer**, press the **Add** button drop list, and select the **Import Table** option.



2. In the **Select Server** entry choose *Comma Delimited Files (BASIC)* from the dropdown list.

3. Press the ellipsis button to the right of the **Select Dictionary** entry to open the Select Database dialog. Alternatively, you can also press the **Next** button.
4. In the **Filename** field, press the ellipsis button to select ...\\Lessons\\LearningClarion\\import1.csv then press the **Open** button.
5. In the **Driver Options** field, press the ellipsis button to open the *BASIC Driver String Builder* dialog.




6. Locate and check the **First Record is Header** check box.
7. Press the **OK** button to close the *BASIC Driver String Builder* dialog, and then press the **OK** (or **Next**) button to close the *Select Database* dialog.

The import process completes, and you should now see a new *Import1* table in the **DCT Explorer**.

Now you have the IMPORT1.CSV file's definition. The next step will be to look at the data in the Database Manager.

View the imported data

RIGHT-CLICK on the *IMPORT1* table, highlight **Browse Table** then CLICK to call the Database

Browser. Or press the Browse button  on the **DCT Explorer** toolbar. Press the **Apply** button when the *Dataset Parameters* dialog is opened. This will open the Database Browser.

IMPORT1.CSV								
	CUSTNUMBER	COMPANY	FIRSTNAME	LASTNAME	ADDRESS	CITY	STATE	ZIPCODE
▶	1	Technology T...	Willam	Fast	1000 Capitol ...	Chicago	SC	33000
	2	Acme Fastners	David	Smith	111 Circle Str...	Durham	SC	27715
	3	A & A Exports	Susan	White	2222 First Ave...	Grand Prairie	SC	75052
	4	K Processing	Mike	Smith	3122 2nd Str...	Forest Hills	MS	11375
	5	Fidelity America	Rodger	Wiles	5060 Third Av...	Glen Burnie	FL	21061
	6	Easi Everything	Frank	Forest	1000 First Str...	Louisville	FL	40220
	7	Have Everything	Richie	Rich	2222 Expensiv...	Evanston	GA	60202
	8	SVFutures	Bob	Barnett	1298 S Main S...	Miami	FL	34002
*								

The Clarion Database Browser allows you to directly edit the data in your tables. This is a programmer's tool, designed to allow you to do whatever is necessary to change the actual data contained in your tables. This means that there are no safeguards against violating your database's Referential Integrity or Data Integrity rules. Therefore, you must take care when you use this tool.

In this lesson, we do not need to make any changes at this time, and simply wanted to identify this IDE tool if needed.

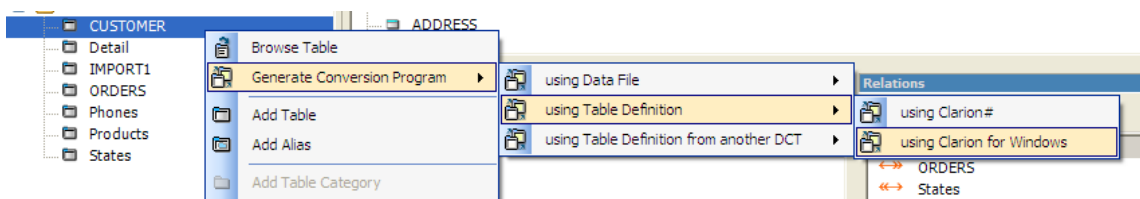
Converting a Table

At this point, you're looking at the .CSV file's data in Clarion's Database Browser utility. Next, you need to move that data into a TopSpeed table so your Clarion programs can use it.

Close the Database Browser at this time and return to the DCT Explorer in the already opened GSLesson dictionary.

Generate a table conversion program

1. In the DCT Explorer, right click on the CUSTOMER table, and select the following conversion option:



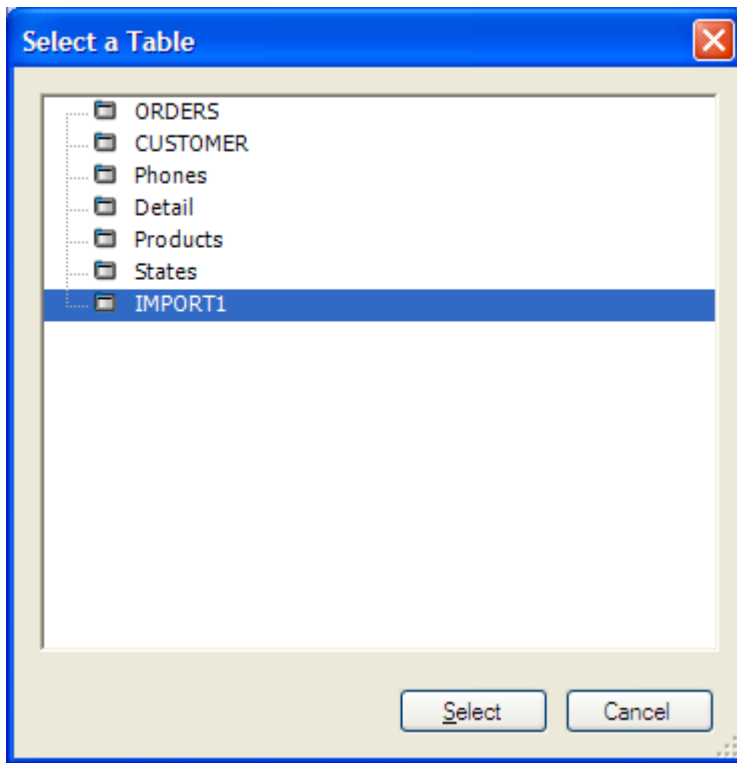
You can specify a table to convert from three different sources. The target can be directly on disk (*using Data File*), a table definition in the current dictionary (*using Table Definition*), or a table definition in another dictionary (*using Table Definition from another DCT*). We have chosen a table from our current dictionary.

Note:

If you have Clarion.NET installed, you also have the option of generating a conversion program in standard Clarion language format or Clarion# to generate a conversion program that runs with the .NET Framework.

The table that you select *first* in the conversion program process is always the *target*. In our example, the CUSTOMER table is the target.

2. In the Select a Table dialog, highlight IMPORT1 and press the **Select** button.



3. The *Select New Project File* dialog appears. Accept the default name (*convert.cwproj*), and the default *LearningClarion* folder name, by pressing the **Save** button.

Project files determine what source file to compile, and how to build (link) it. The default extension of project files is **.cwproj*.


When you press the **Save** button, this will generate all the Clarion source code necessary to take the data in the **Source Filename**, and copy it into a new **Target Filename**, using the file format specified by the **Target Structure**.

4. The *Select Destination Data File* dialog appears. Highlight the CUSTOMER.TPS file located in the Learning Clarion folder, and press the **Open** button.
5. Another dialog pops up that asks you if you would like to load the data conversion program. Press the **Yes** button. We will return to the conversion program shortly.

The best reason to generate Clarion source code for the data conversion is to provide you the opportunity to modify the code before you compile and execute it to handle any special data conversion needs you may have. This makes the conversion process completely flexible to handle any situation that can occur.

Delete the IMPORT1 file definition

The only purpose this file definition served was to allow the Dictionary Editor to generate file conversion source code for you. Therefore, we can delete it from the Data Dictionary right now.

1. Highlight the *IMPORT1* table in the **DCT Explorer**, and press the **Delete**  button.
2. Press the **Yes** button when asked to confirm the deletion.
3. Press the **Save and Close** button to exit the Data Dictionary Editor, and press the **Yes** button to save your changes as you exit.

Load the conversion program

1. Choose **View ► Solution Explorer** (or press CTRL+ALT+L).
2. Expand the Convert project node, and highlight the *Convert.clw* file, then press the **Open** button in the Solution Explorer toolbar, or you can right-click and select Open from the popup menu, or simply double-click on the file to open it.

Clarion's Text Editor appears with the file loaded, ready to edit.

The Dictionary Editor created the conversion program code in this file. This contains all the Clarion language source code necessary to read the data from the BASIC (.CSV) file and copy it to the TopSpeed table.

Compile/Build and run the conversion program

The Dictionary Editor generated the *Convert.cwproj* file for you at the same time it created the *Convert.clw* file.

Every Clarion program has a Project that controls the options for compiling the source code and linking to create the resulting .EXE file. For hand-coded programs (and conversion programs generated by the Database Manager), these settings are contained in a .cwproj file.

At this point, you could modify the generated source code to perform any special data conversion you require (see the How to Convert a File—Generate Source and How to Make a Field Assignment topics for more information on customizing table conversion code). However, there is nothing we need to do in this project, so you simply compile and run the program.

1. Choose **Debug ► Start Without Debugger** from the IDE Menu (or press CTRL+F5).

You can also press the  button on the IDE toolbar.

This compiles the program, links it into an .EXE, then runs the resulting executable to perform the file conversion. A status window appears as the program runs, letting you know the progress of the file conversion. Since there are only a few records to convert in this case, you probably won't be able to read it (it'll go by too fast).

2. Choose **File ► Close ► File** to close the program source file and exit the Text Editor.
3. Choose **File ► Close ► Solution** to close the conversion project. Close the Solution Explorer by clicking on the "X" in the upper right corner of the Solution Explorer.

Check it out

Now you can check the data in the new file by opening it with the Database Browser and browsing through the records.



1. Choose **Tools ► Browse Table** from the IDE Menu.

2. In the *Select Driver...* dialog window, highlight *TopSpeed (TPS)* and press the **Select** button.
3. In the *Open TopSpeed File* dialog, press the ellipsis button, and select the *CUSTOMER.TPS* file, then press the **OK** button.

A *Dataset Parameters* dialog opens that allows you to specify how many rows you want to read. Press the **Apply** button to open the Database Browser:

CUSTOMER.TPS								
1) Record Order ()								
	CUS:CUSTNUMB	CUS:COMPANY	CUS:FIRSTNAME	CUS:LASTNAME	CUS:ADDRESS	CUS:CITY	CUS:STATE	CUS:ZIPCODE
▶	0	Company	FirstName	LastName	Address	City	St	0
	1	Technology T...	William	Fast	1000 Capitol ...	Chicago	SC	33000
	2	Acme Fastners	David	Smith	111 Circle Str...	Durham	SC	27715
	3	A & A Exports	Susan	White	2222 First Ave...	Grand Prairie	SC	75052
	4	K Processing	Mike	Smith	3122 2nd Str...	Forest Hills	MS	11375
	5	Fidelity America	Rodger	Wiles	5060 Third Av...	Glen Burnie	FL	21061
	6	Easi Everything	Frank	Forest	1000 First Str...	Louisville	FL	40220
	7	Have Everything	Richie	Rich	2222 Expensiv...	Evanston	GA	60202
	8	SVFutures	Bob	Barnett	1298 S Main S...	Miami	FL	34002
*								

This demonstrates another way to open the Database Browser, other than from within the Dictionary Editor.

4. Press the **Save** button  in the Database Browser toolbar to save the changes to the TopSpeed table.
5. Press the **Exit** button  to close the Browser, or you can simply select **File ▶ Close ▶ File** from the IDE Menu.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- ✓ You imported a table definition from an existing .CSV file.
- ✓ You used Clarion's Database Browser utility to examine the contents of the .CSV table.
- ✓ You used Clarion's Conversion Program utility to generate code to create a table conversion program.
- ✓ You compiled and executed a table conversion program to import data from a .CSV table into a TopSpeed table.

Now you've converted some valuable existing data to the TopSpeed table format so your Clarion applications can use it. In the next chapter, we will begin building an application "from scratch" using the Application Generator.

6 - Starting the Application

Using the Application Generator

With the Data Dictionary complete, you now can use the Application Generator to create your application. This chapter shows you:

- How to create the .APP file, which stores all your work for the project.
- How to define the first (Main) procedure to create an MDI application frame, and how to call procedures from the application's menu.

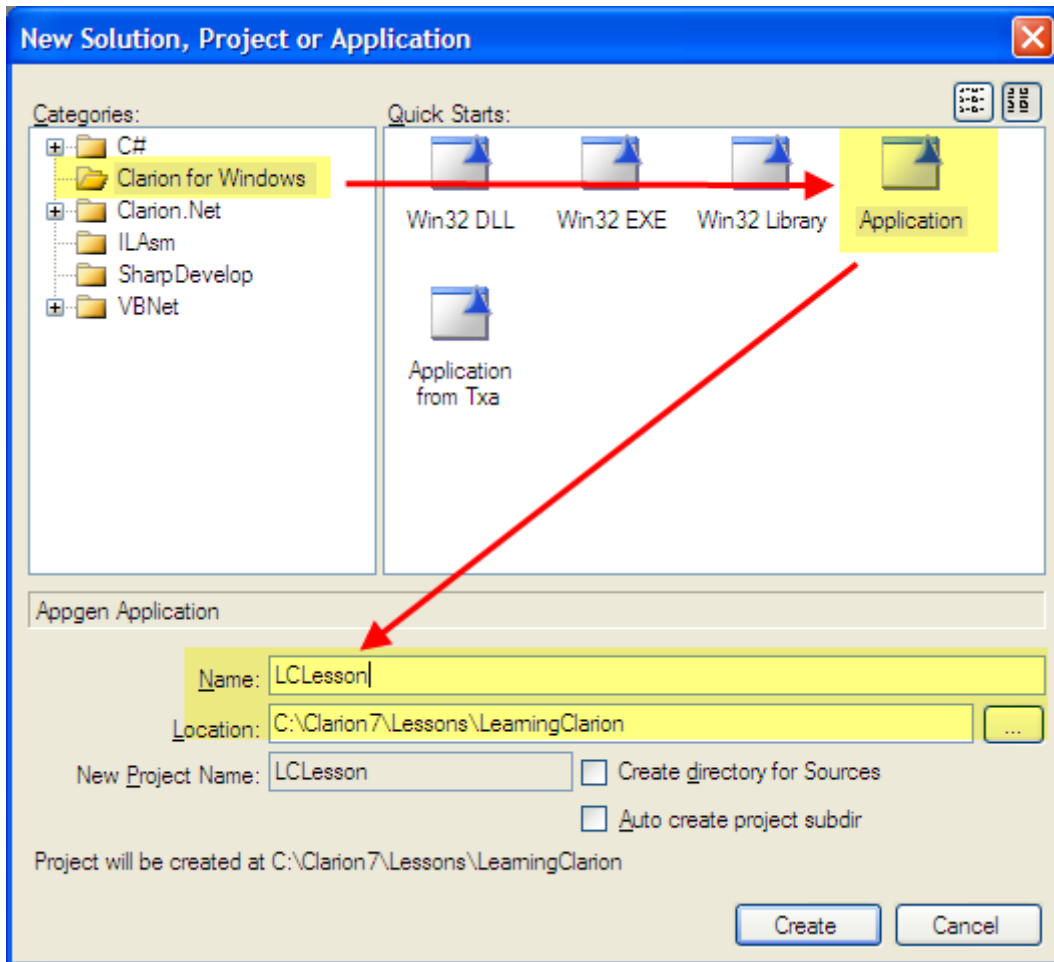
Starting Point:

The Clarion environment should be open, and all dialog windows closed.

Creating the application (.APP) file

1. From the IDE Menu, choose **File ► New ► Solution, Project or Application**.

The New Solution, Project or Application dialog appears.

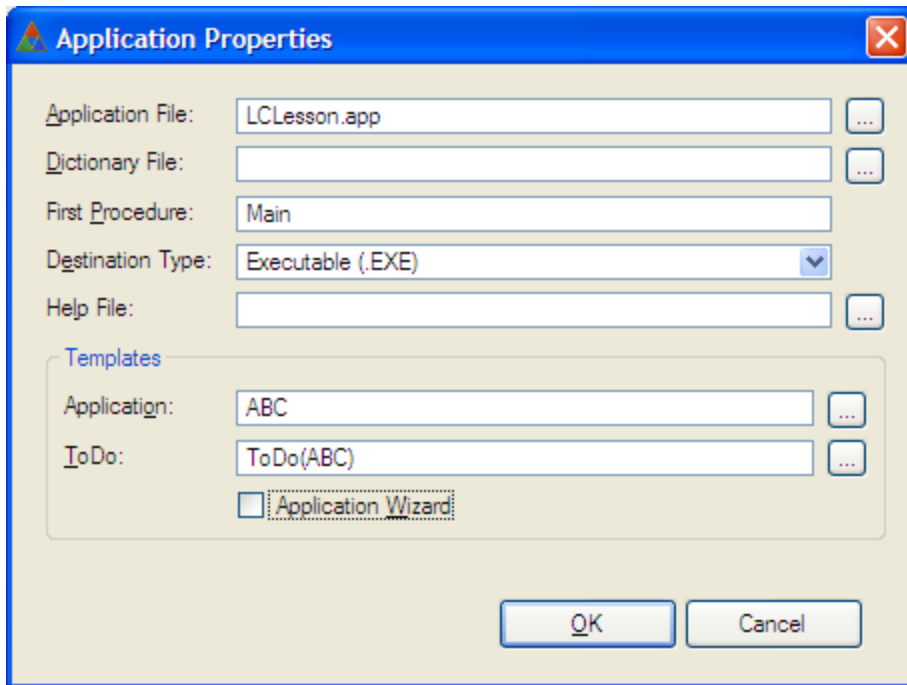


2. Select the *Clarion for Windows* item in the left **Categories** pane, and click or select the **Application Quick Starts** as shown above.
3. Verify that the **Location** is set to the ... \Lessons\LearningClarion directory.
4. Type *LCLesson* in the **Name** field.
5. Uncheck the **Auto create project subdir** checkbox.
6. Press the **Create** button.

The *Application Properties* dialog appears.

1. Type *LCLESSON.DCT* in the **Dictionary File** entry, or select it from the ellipsis button.
2. Clear the **Application Wizard** check box.

This Application Generator lesson will not use any Wizards so that it can demonstrate how to use all the other tools that Clarion provides. We will also use the ABC template chain in this lesson.



3. Press **OK** to close the *Application Properties* dialog.

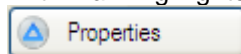
Creating the Main Procedure

The *Application Tree* dialog appears. This lists all the procedures for your application in a logical procedure call tree, which provides a visual guide to show the order by which one procedure calls another. You previously saw it in the Getting Started lessons.

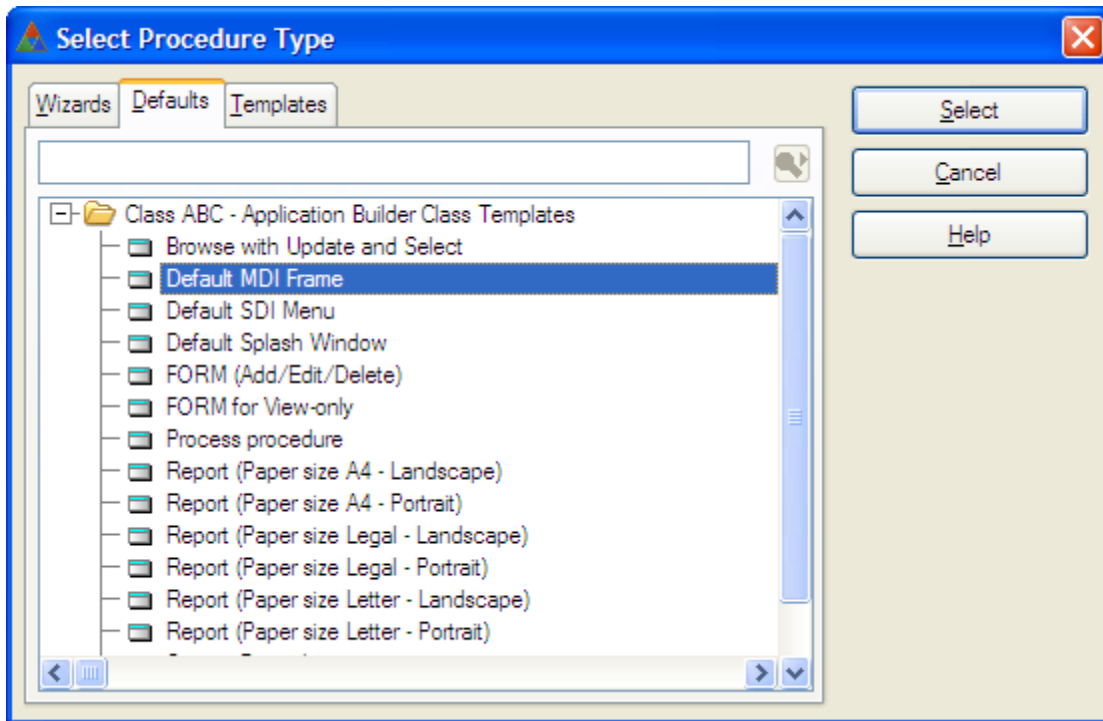
The *Main* procedure is the starting point. The lesson application will be an MDI (Multiple Document Interface) program. Therefore the natural starting point is to define the Main procedure using the Frame Procedure template to create an application frame. A frame is a type of visual program-wide starting point, providing menus, toolbars, and other options.

Select the procedure type for Main

1. With *Main* highlighted in the *Application Tree* dialog, press the **Properties**

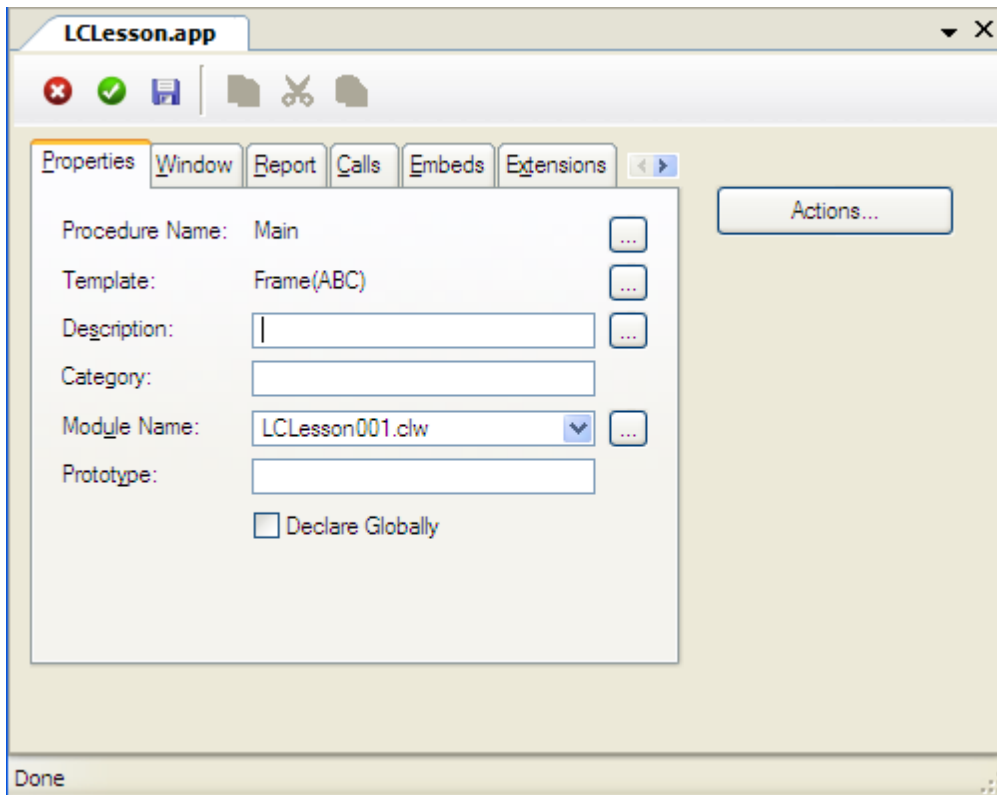


button.



2. Choose the **Defaults** tab and highlight *Default MDI Frame* in the *Select Procedure Type* dialog, then press the **Select** button.

The *Procedure Properties* dialog appears. It defines the functionality and data structures for the procedure.



3. Press the **Actions** button
4. In the *Main – Properties* dialog, type *SplashScreen* in the **Splash Procedure** field.
5. Press **OK** to close the *Main – Properties* dialog.

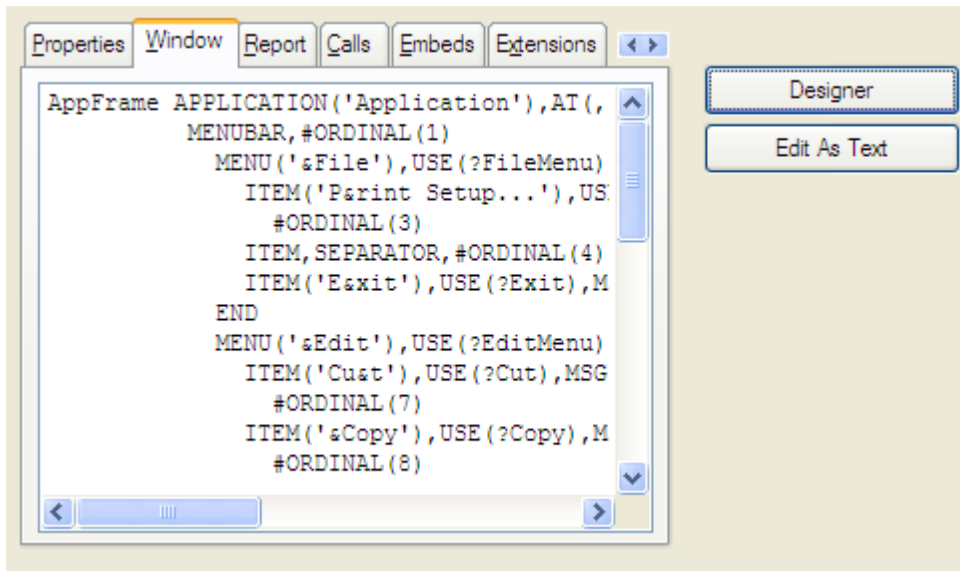
This names the procedure containing an opening screen that will appear for just a brief period when the user first opens the application.

Usually the first task when creating a procedure is to edit the main window. You can place controls, or if the procedure template has controls already, you can customize them.

The application frame itself never has controls. Windows doesn't allow it. We will, however, customize the window caption (the text that appears on its title bar). Then we will add items to the predefined menu, which is also built into the *Frame Procedure* template, and create a toolbar for the application (a toolbar can have controls).

Edit the Main window

1. Press the **Window** tab. The Window Editor appears:

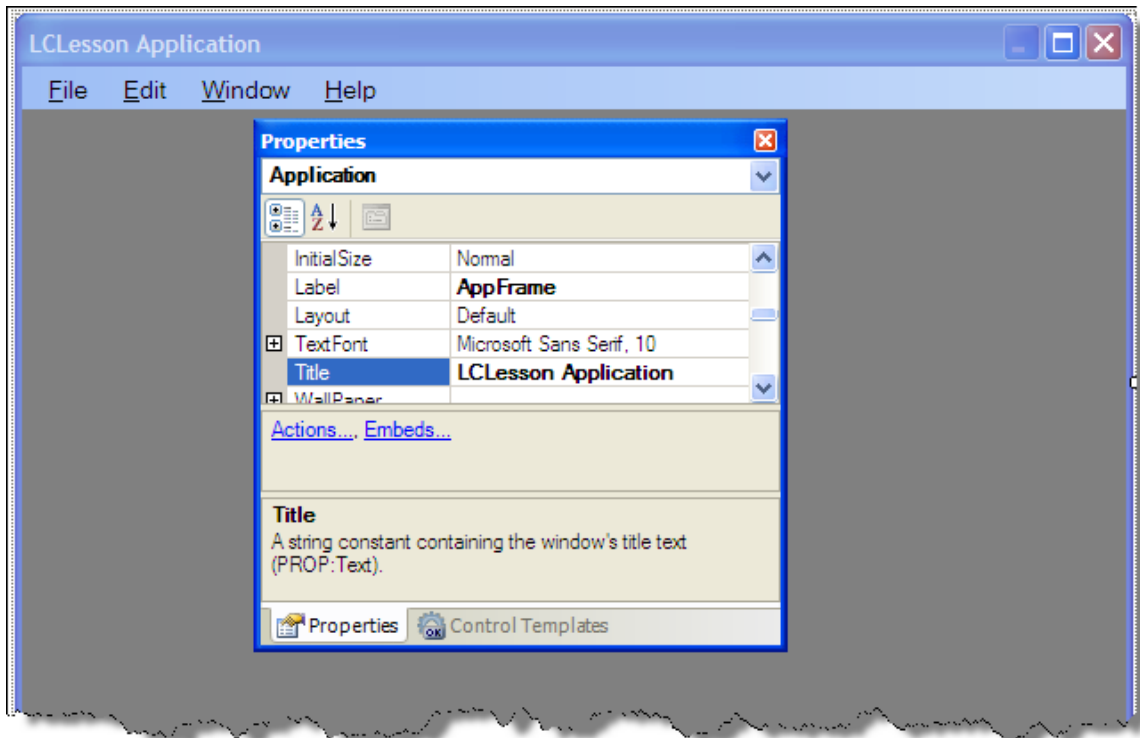


2. Press the **Designer** button shown above. The Window Designer appears. Here are all the tools that allow you to visually edit the window and its controls.
3. Choose **View ► Properties** to display the Property Pad (if it is not already present).

You'll notice under the View menu that there are several toolboxes, or pads, that are available—all of these are fully dockable and resizable. This means you can configure your workspace, as you want. Because of this configurability, throughout the rest of the lessons some screen shots taken within the Window Designer may not appear the same as on your computer.

4. CLICK on the window's title bar to make sure it has focus (that is, the white "handles" appear to the right and bottom edge of the window).
5. Type *LCLesson Application* in the **Title** entry of the Property Pad, then press TAB.

This updates the caption bar text in the sample window.



Editing the Menu

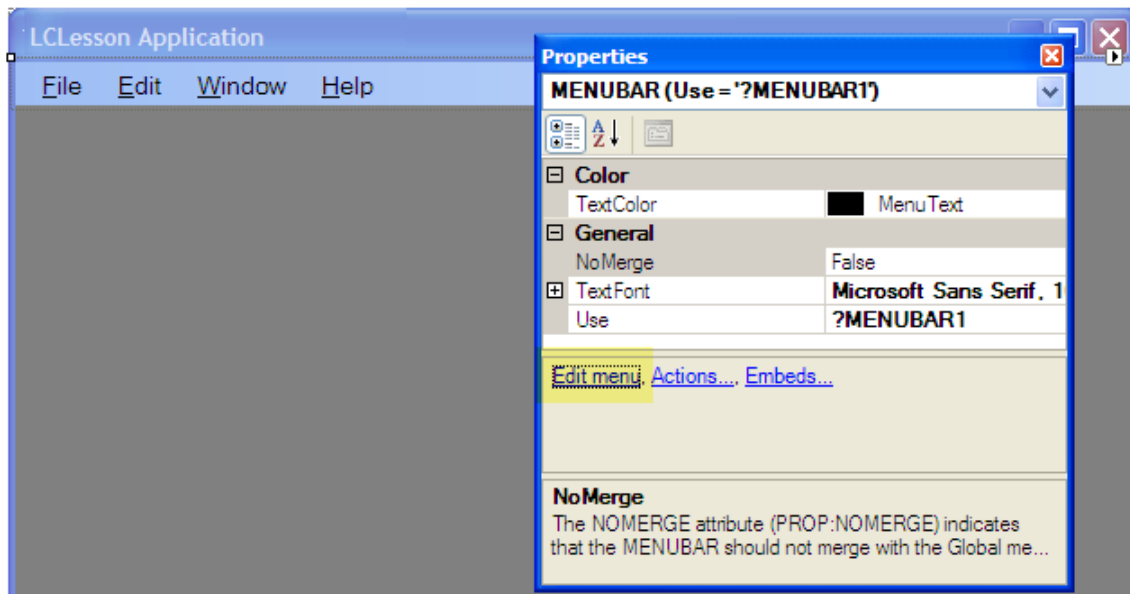
From the Window Designer or Properties Pad, you can call the **Menu Editor**, which allows you to add or edit menu items for the application frame window. As you add each menu item, you can access the **Actions** dialog to name the procedure to call when the user chooses that menu item.

For each new procedure you name for the menu to call, the Application Generator automatically adds a "ToDo" procedure to the Application Tree. You can then define that procedure's functionality, just as you are now defining the application frame procedure's functionality.

When the Application Generator generates the source code for your application, it automatically starts a new execution thread for each procedure you call from the main menu (this is required for an MDI application).

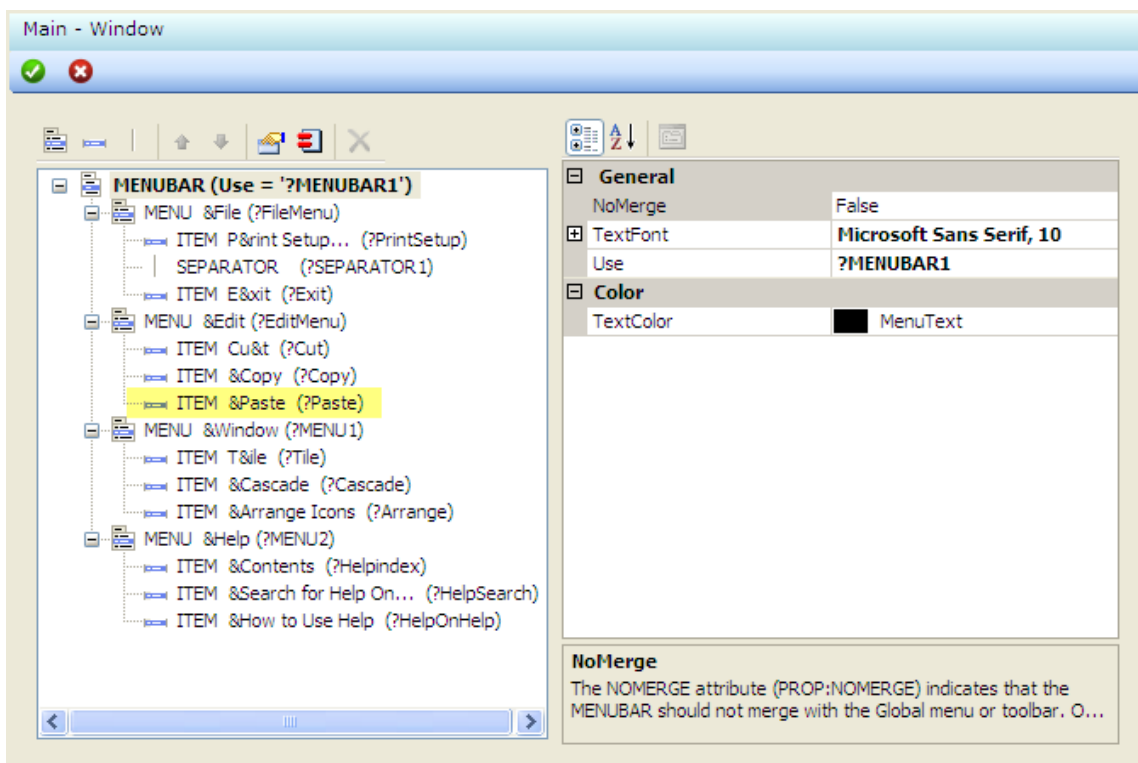
Add menu items

1. From the Window Designer, click on the menu action bar in your window design. In the Property Pad, click on the **Edit Menu** link as shown here:




The Menu Editor appears. It displays the menu in hierarchical form in a list box at the left. The fields at the right allow you to name and customize the dropdown menus and menu items.

This template already provides you with a "standard" menu. It contains basic window commands such as an **Exit** command on a **File** menu, the standard editing **Cut**, **Copy**, and **Paste** commands, and the standard window management commands commonly found in an MDI application.




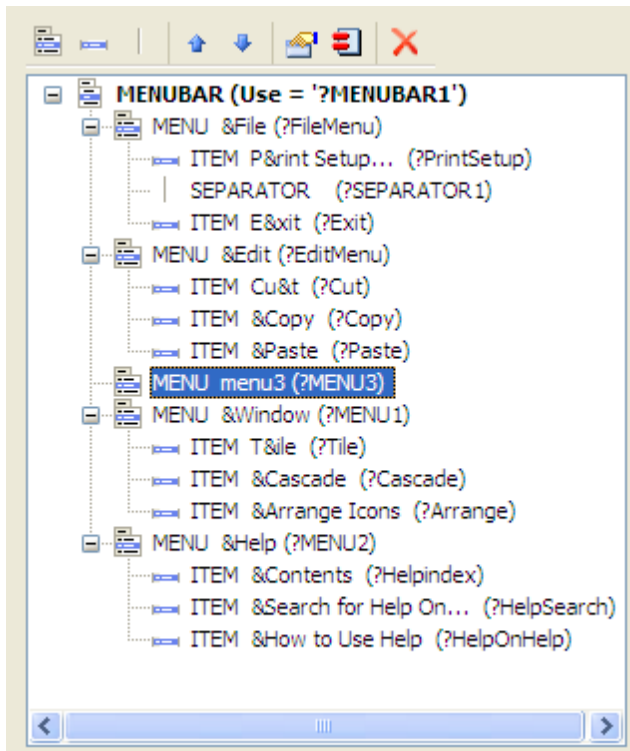
2. Highlight the ?Paste menu item (see the illustration above).

The Menu Editor inserts new items immediately below the currently highlighted selection. The menu you'll add will be called **Browse**. It will contain three items: **Products**, **Customers**, and **Orders**. It will appear on the menu bar just before the Window menu.

3. Press the **Add New Menu** button  (or press SHIFT+INSERT).

This inserts a new MENU statement, and its corresponding END statement.


4. Press the Menu Down button  to position the MENU inline with the other MENUs.



5. In the **Text** field in the right hand pane, type **&Browse** in the field then press TAB.

This defines the text that appears on the menu to the end user. The ampersand (&) indicates that the following character (B) is underlined and provides keyboard access (the user can press ALT+B to drop down this menu).

Add the first menu item

1. Press the **New Item** button  (or press INSERT).

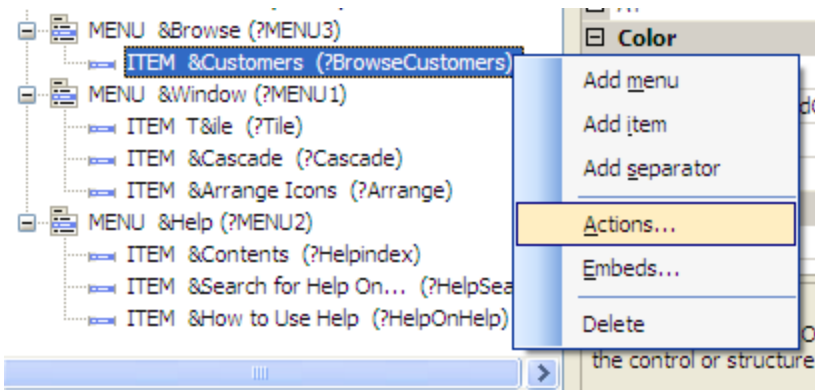
This updates the list on the left side of the dialog, changing the text of the menu you just added to "&Browse." It adds a new menu ITEM—a command on the dropdown menu—under &Browse.

2. Type **&Customers** in the **Text** field then press TAB.

3. Type *?BrowseCustomers* in the **Use** field. This is an equate for the menu item so code statements can reference it. The leading question mark (?) indicates it is a field equate label.

Text	&Customers
TextFont	Microsoft Sans Serif, 10
Use	?BrowseCustomers

4. Right-click on the ITEM just added, and select the **Actions** item from the popup menu.



The prompts allow you to name a procedure to execute when the end user selects the **Browse Customers** menu item.

5. Choose *Call a Procedure* from the **When Pressed** drop list.

New prompts appear to allow you to name the procedure to call and choose options.

6. Type *BrowseCustomers* in the **Procedure Name** field.

This names the "ToDo" procedure for the Application Tree.

7. Check the **Initiate Thread** box.

The *BrowseCustomers* procedure will display an MDI "child" window, and you must always start a new execution thread for any MDI window called directly from the application frame. The **Thread Stack** field defaults to the minimum recommended value.


8. Press **OK** to close the *?BrowseCustomers Prompts* dialog

Add the second menu item

1. Press the **New Item**  button.
2. Type *&Products* in the **Text** field and press TAB.
3. Type *?BrowseProducts* in the **Use** field.

Normally, the next step is to define the action for the menu item—what happens when the end user executes it from the menu. We'll skip over this step for now, for this menu item only. Later, you'll create a procedure by copying it, then attaching it to this menu, just to show you this capability of the Clarion environment.

Add the third menu item

1. Press the **New Item**  button.
2. Type `&Orders` in the **Text** field and press TAB.
3. Type `?BrowseOrders` in the **Use** field.
4. Right-click on the ITEM just added, and select the **Actions** item from the popup menu.
5. In the *Actions* dialog, choose *Call a Procedure* from the **When Pressed** drop list.
6. Type `BrowseOrders` in the **Procedure Name** field.
7. Check the **Initiate Thread** box.
8. Press **OK** to close the *?BrowseOrders Prompts* dialog

Close the Menu Editor and Window Designer and save your work

1. Press the **Save and Close**  button to close the Menu Editor.

This returns you to the Window Designer.

2. Press the **Save and Close** button.

This returns you to the *Window Designer Editor* dialog.


3. Press the **Save and Close** button to close the Window Editor and the *Procedure Properties* dialog.

This returns you to the Application Tree dialog. There are now three new procedures marked as "(ToDo)": *BrowseCustomers*, *BrowseOrders*, and *SplashScreen*. These were the procedures you named in the Menu Editor.

4. Choose **File ► Save**, or press the **Save** button  on the toolbar.

Creating the SplashScreen Procedure

We named a Splash procedure, and now we'll create it.

1. Highlight the *SplashScreen (ToDo)* procedure and press the **Properties**  button.
2. Choose the **Defaults** tab, highlight *Default Splash Window* in the *Select Procedure Type* dialog, then press the **Select** button.

The *Procedure Properties* dialog appears. There's nothing we really have to do to this procedure except accepting all of the defaults.

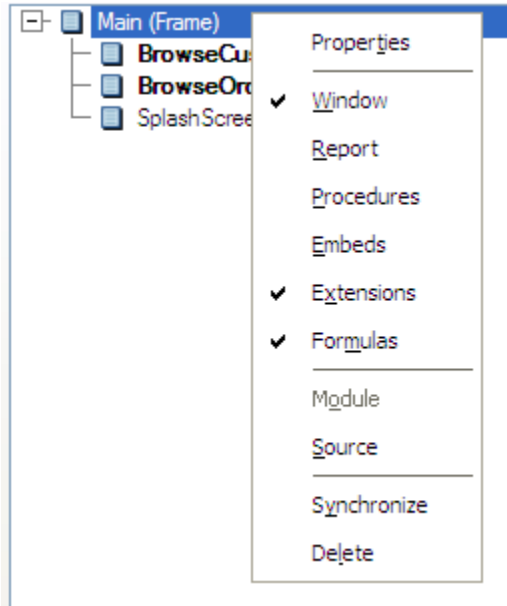
3. Press the **Save and Close** button.

4. In the Application Tree toolbar, press the **Save** button  to save your work.

Adding an Application Toolbar and Toolbar Controls

Call the Window Designer and create the tool bar

1. Highlight the *Main* procedure.
2. RIGHT-CLICK to display the popup menu.



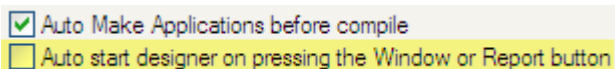
Notice that this popup menu contains a set of menu selections that match the set of tabs on the top of every *Procedure Properties* window. It also aligns closely with the right hand pane.

This popup menu provides you with direct access to all the Clarion tools that you use to modify existing procedures, so that you don't have to go through the Procedure Properties window every time.

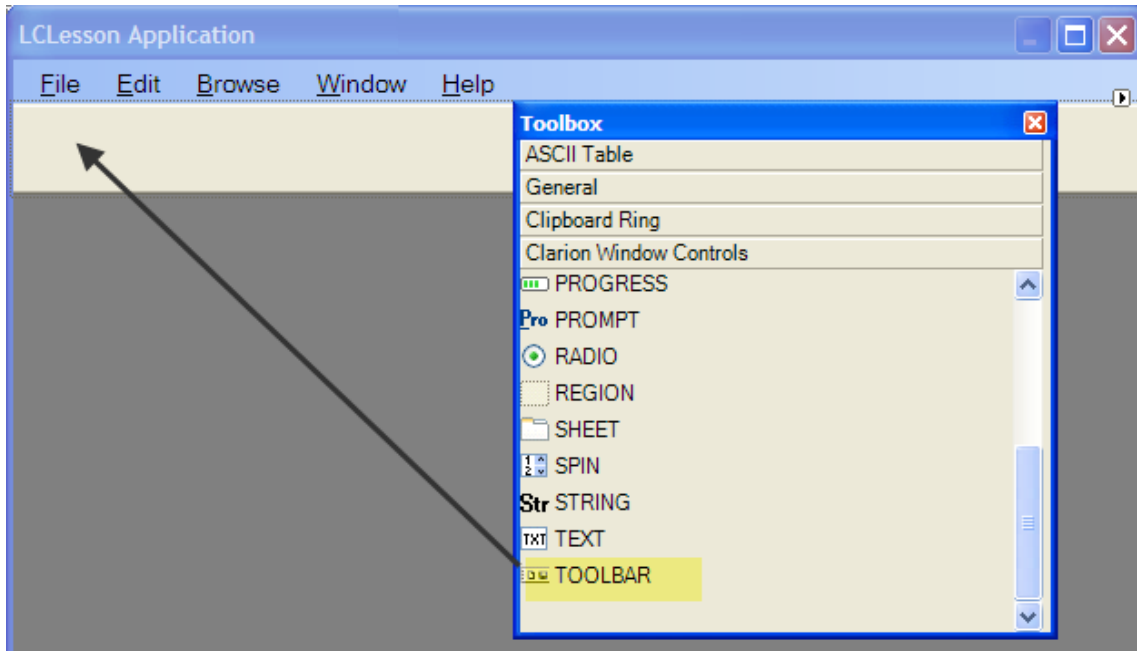
3. Choose the **Window** menu item. The Window Designer Editor opens. If you are not automatically taken there, press the **Designer** button to access the Window Designer.

Note:

You can bypass the Window Designer Editor and load Designer directly, by setting the following Application Options (this dialog can be found in the IDE Tools menu):



4. With the Window Designer active, choose **View ► Toolbox** from the main IDE Menu (or press CTRL + ALT + X).
5. Locate and highlight the TOOLBAR control in the Toolbox, and DRAG and DROP to the window:

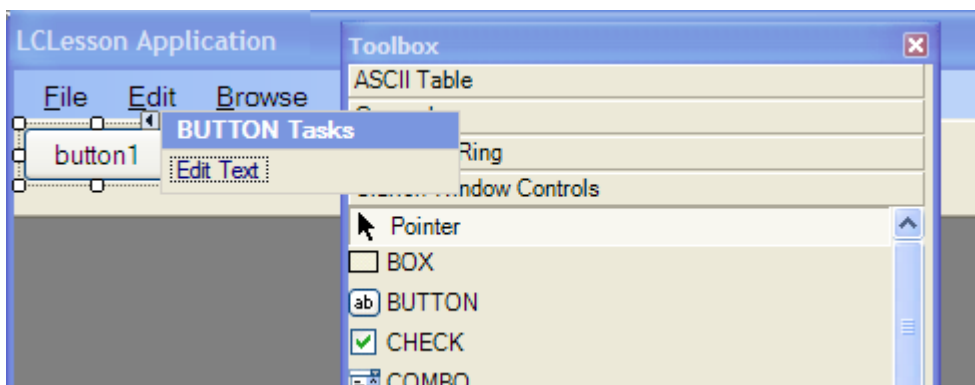


This adds the toolbar—always immediately below the menu—to the sample window. You can add any control to your toolbar by CLICKing on a control in the Controls toolbox, then CLICKing in your toolbar.

Place the first button

If the control toolbox is not present, choose **View ► Toolbox** from the Window Designer menu.

1. CLICK on the BUTTON control, drag the mouse to the toolbar design area (not the Designer's toolbar!), just below the upper left corner, and drop to place the button.



2. RIGHT-CLICK on the button you just placed, then choose **Properties** from the popup menu. Or you can press the F4 key.

The *Properties* dialog appears.

3. Clear the **Text** property.

Note:

Do not use the spacebar to do this! Use the DELETE or BACKSPACE keys.

We'll place images on these buttons, instead of text, to give the application a more modern look.

4. Type `?CustomerButton` in the **Use** property.

This is the field equate label for referencing the button in code. We included the word "Button" for code readability.

5. Set the **Flat** property to TRUE.

Tip

You can simply click on the Flat property, and it will toggle the TRUE/FALSE values!

Options	
DefaultButton	False
Flat	True
Immediate	False

6. Drop down the **Icon** property, scroll to the bottom and CLICK on *Select File.....*

The *Select Icon File* dialog appears.

7. Select *GIF Files* from the **Files of type** drop list.
8. Select the *CUSTOMER.GIF* file, then press the **Open** button.

Extra	
Delay	0
DropID	
Icon	CUSTOMER.GIF ▼
Repeat	0
STN	

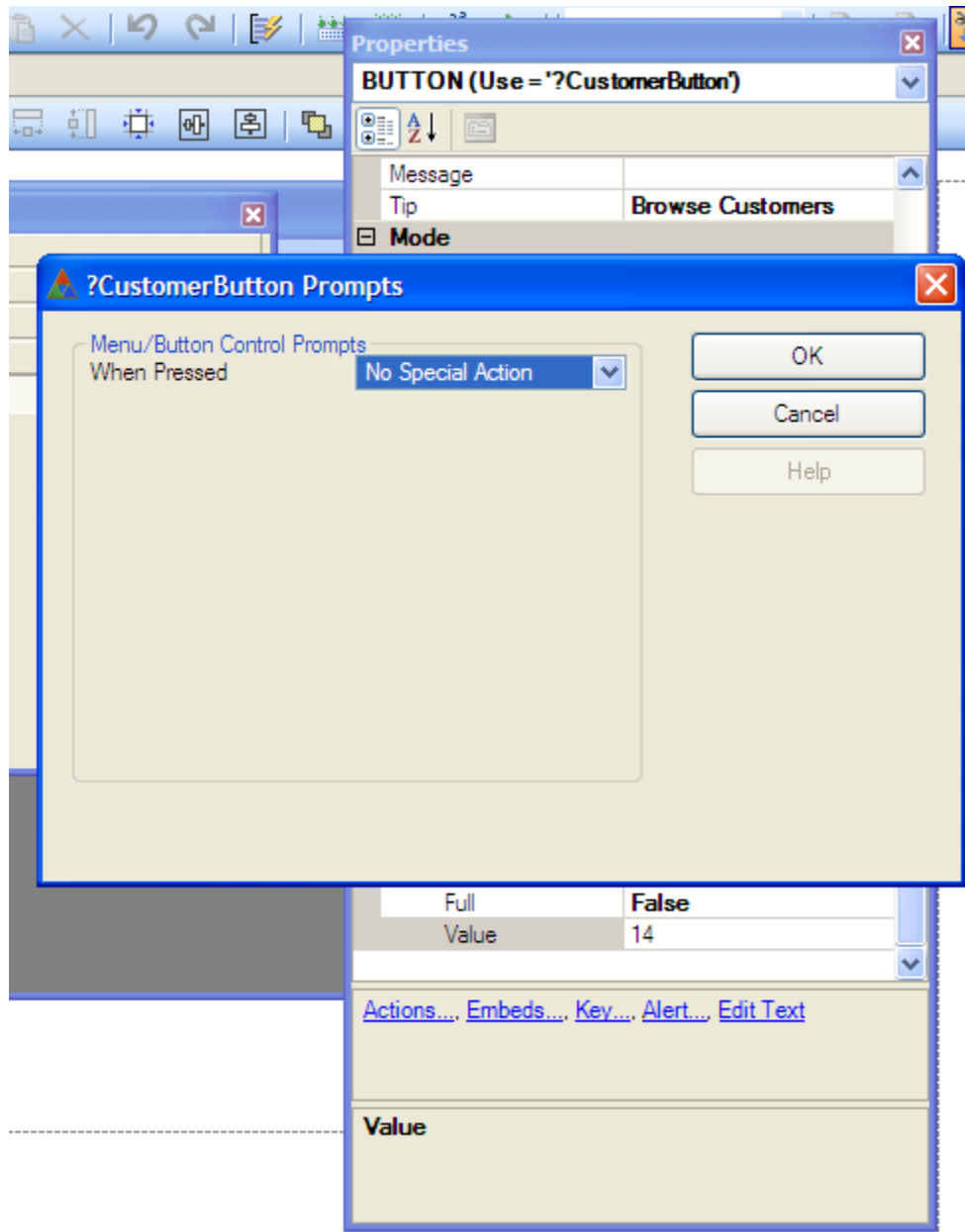
9. Type *Browse Customers* into the **Tip** property.

This adds a tool tip to the button that will display whenever the mouse cursor hovers over the button.

10. Expand the AT property in the Properties Pad, and then set the **Default** property to *False* for both **Width** and **Height**.
11. Set the **Width** to *16* and **Height** to *14*.

[-] Position	
[-] AT	AT(2,2,16,14)
X	2
Y	2
[-] Width	16
Default	False
Full	False
Value	16
[-] Height	14
Default	False
Full	False
Value	14

12. Near the bottom of the Property Pad, click on the Actions link:



13. Choose *Call a Procedure* from the **When Pressed** drop list.
14. Choose *BrowseCustomers* from the Procedure Name drop list.

This is the procedure name you typed for the Browse ► Customers menu item. Pressing the button will call the same procedure. Often, a command button on a toolbar serves as a quick way to execute a menu command.

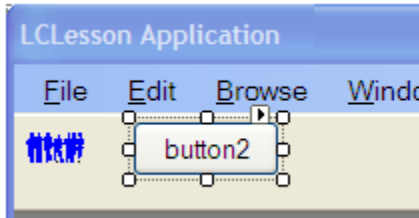
15. Check the **Initiate Thread** box.
16. Press the **OK** button.

Place the second button

If the control toolbox is not present, choose **View ► Toolbox** from the Window Designer menu.

1. CLICK on the **BUTTON** control, drag the mouse to the toolbar design area (not the Designer's toolbar!), right next to the first button, and drop to place the button.

A button appears, labeled "button2."



2. RIGHT-CLICK the button you just placed, then choose **Properties** from the popup menu.
3. Clear the **Text** property.
4. Type *?ProductsButton* in the **Use** property.
5. Set the **Flat** property to TRUE.
6. Drop down the **Icon** property, scroll to the bottom and CLICK on *Select File.....*
7. Select *GIF Files* from the **Files of type** drop list.
8. Select the *PRODUCTS.GIF* file, then press the **Open** button.
9. Type *Browse Products* into the **Tip** property.

Normally you attach an action to the button at this point. Skip this step for now, for this button only. Later, we'll copy a procedure, then call it at the point in the generated source code which handles what to do when the end user presses the button to demonstrate using embed points.

Place the third button

If the control toolbox is not present, choose **View ► Toolbox** from the Window Designer menu.

1. CLICK on the **BUTTON** control, drag the mouse to the toolbar design area (not the Designer's toolbar!), right next to the second button, and drop to place the button.
2. RIGHT-CLICK on the button you just placed, then choose **Properties** from the popup menu.
3. Clear the **Text** property.
4. Type *?OrdersButton* in the **Use** property.
5. Set the **Flat** property to TRUE.
6. Drop down the **Icon** property, scroll to the bottom and CLICK on *Select File.....*
7. Select *GIF Files* from the **Files of type** drop list.
8. Select the *ORDERS.GIF* file, then press the **Open** button.
9. Type *Browse Orders* into the **Tip** property.
10. Near the bottom of the Property Pad, click on the **Actions** link.
11. Choose *Call a Procedure* from the **When Pressed** drop list.

12. Choose *BrowseOrders* from the **Procedure Name** drop list.

This is the procedure name you typed for the **Browse ► Orders** menu item.

13. Check the **Initiate Thread** box.
14. Press the **OK** button to close the *Actions* dialog.

Position and align the buttons

The Window Designer has a set of alignment tools that easily allow you to line up and resize your window controls.

1. With the *Orders* button still selected, CTRL+CLICK on the *Customers* button.

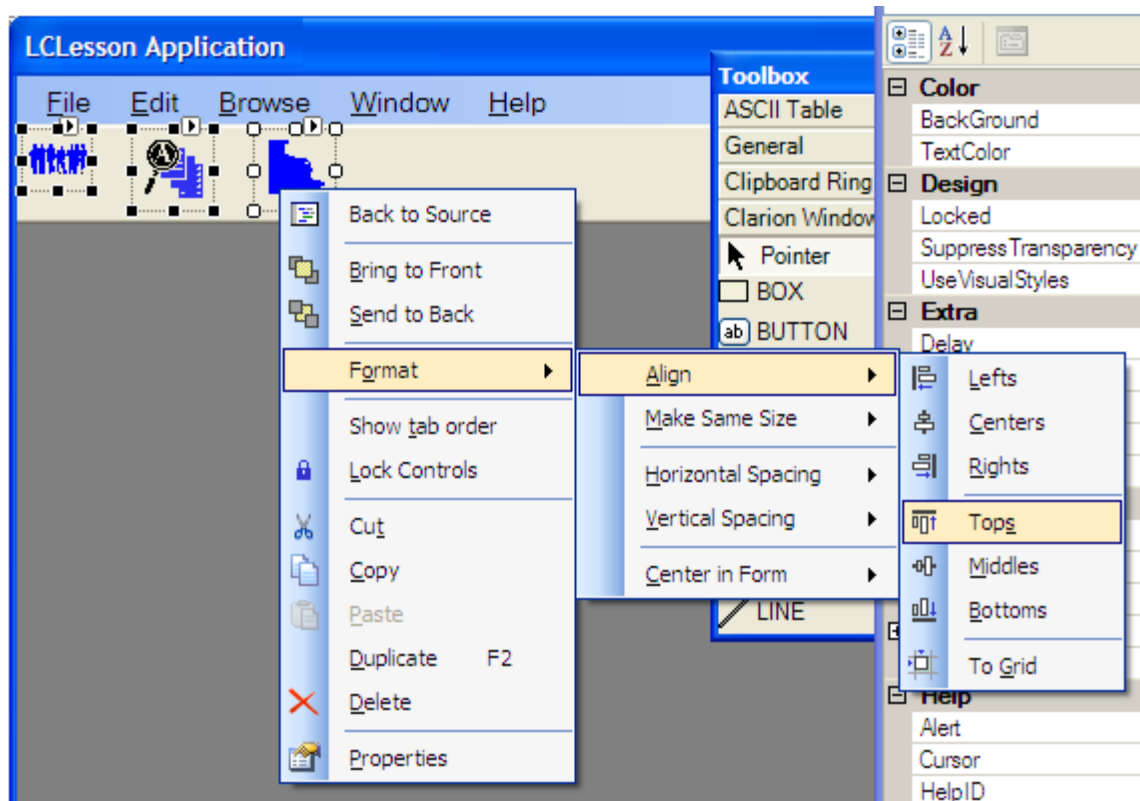
This gives both buttons "handles" and the Customers button has the Red handles that indicate it has focus.

CTRL+CLICK is the "multi-select" keystroke that allows you to perform actions on several controls at once. Once multiple controls are selected, you can move them all by dragging on any one of the selected controls, or you can use any of the Alignment menu's tools on the entire group.

2. With both buttons still selected, CTRL+CLICK on the *Products* button.

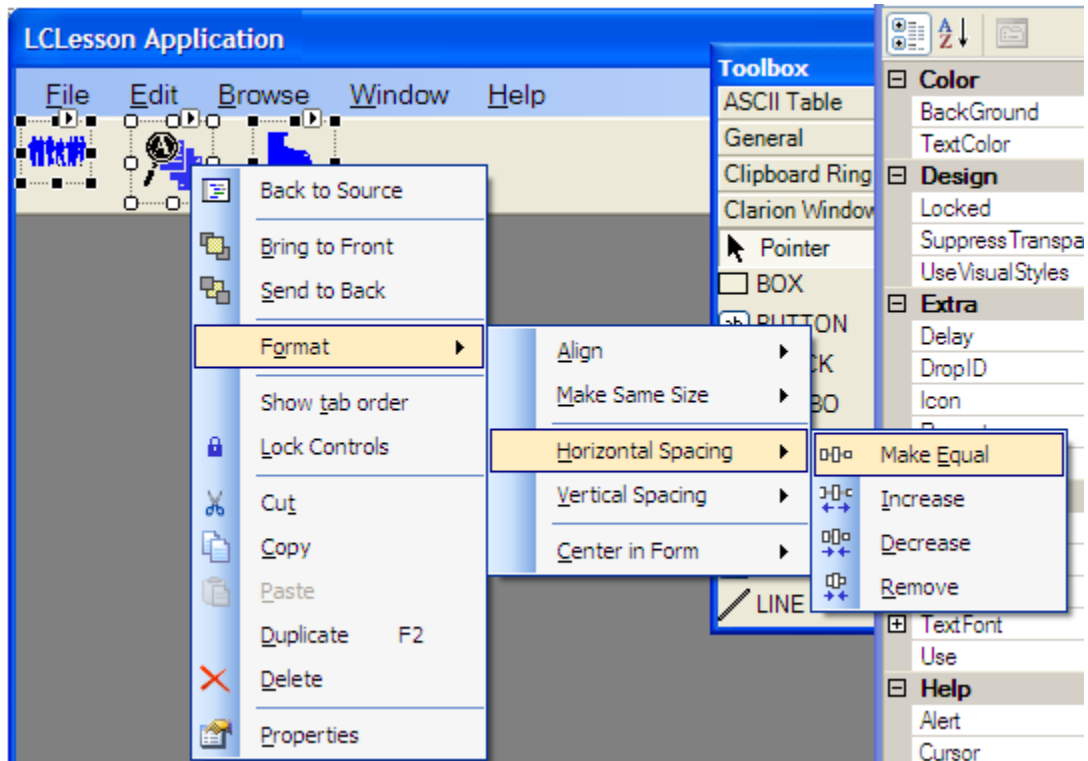
Now all three buttons have "handles", and the Orders button has the white handles that indicate it has focus and is the "key control" for the alignment actions.

3. RIGHT-CLICK and choose **Format > Align > Tops** from the popup menu.



When you have multiple controls selected, RIGHT-CLICK displays an alignment popup menu instead of a single control's popup menu. This action aligns all three buttons with the top of the *Products* button.

4. RIGHT-CLICK and choose **Format > Horizontally Spacing > Make Equal** from the popup menu.



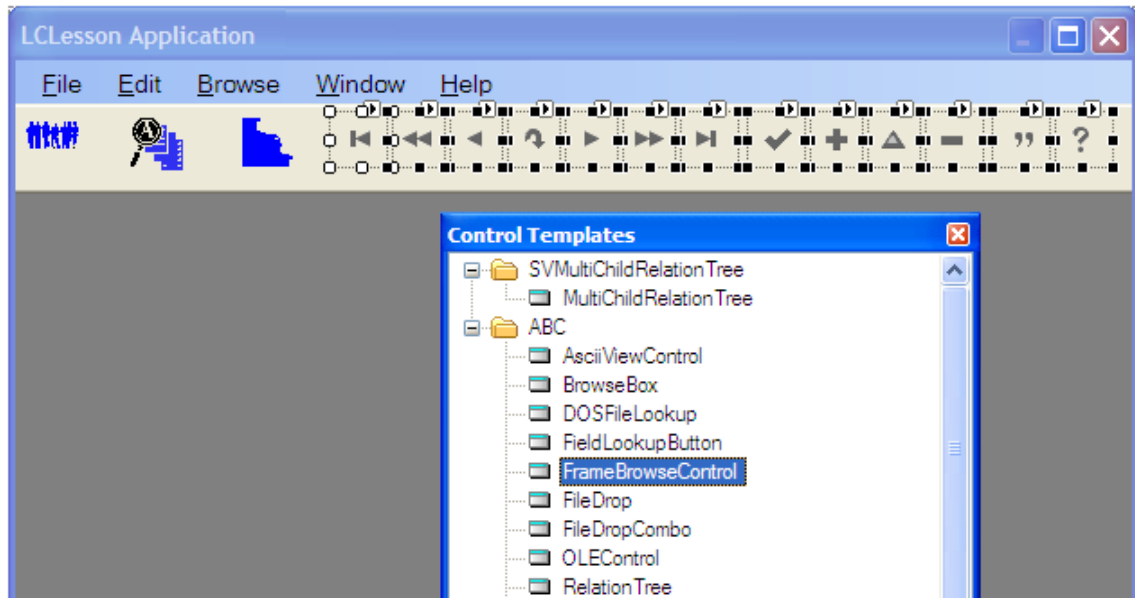
This spaces all three buttons apart equally. Make sure they're close together, so there's room for what's coming next!

Add the Frame Browse Control buttons

1. Close all opened pads at this time, so you have plenty of room to work.
2. From the IDE Menu, choose **View > Control Template**.

The *Control Template* pad appears.

3. Highlight (click on) the *FrameBrowseControl* template, then drag and drop just to the right of the 3rd toolbar button.




The thirteen Browse control buttons appear on the toolbar. You've already seen these buttons on the applications you created in the Getting Started topic. These buttons can control the scrolling and update procedure call behavior of the Browse procedures you'll create (later in the lessons).

Close the Window Designer and save your work

1. Press the **Save and Close**  button.

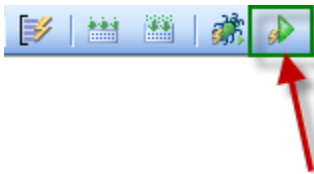
This returns you to the *Window Designer Editor* dialog.

2. Press the **Save and Close** button to close the *Procedure Properties* dialog.

3. Choose **File ► Save**, or press the **Save** button  on the toolbar.

Testing an Application under Development

1. With the Application Tree dialog open, **Generate Source**, and **Build** and **Run** your application by pressing the **Start without Debugger** button (shown below):

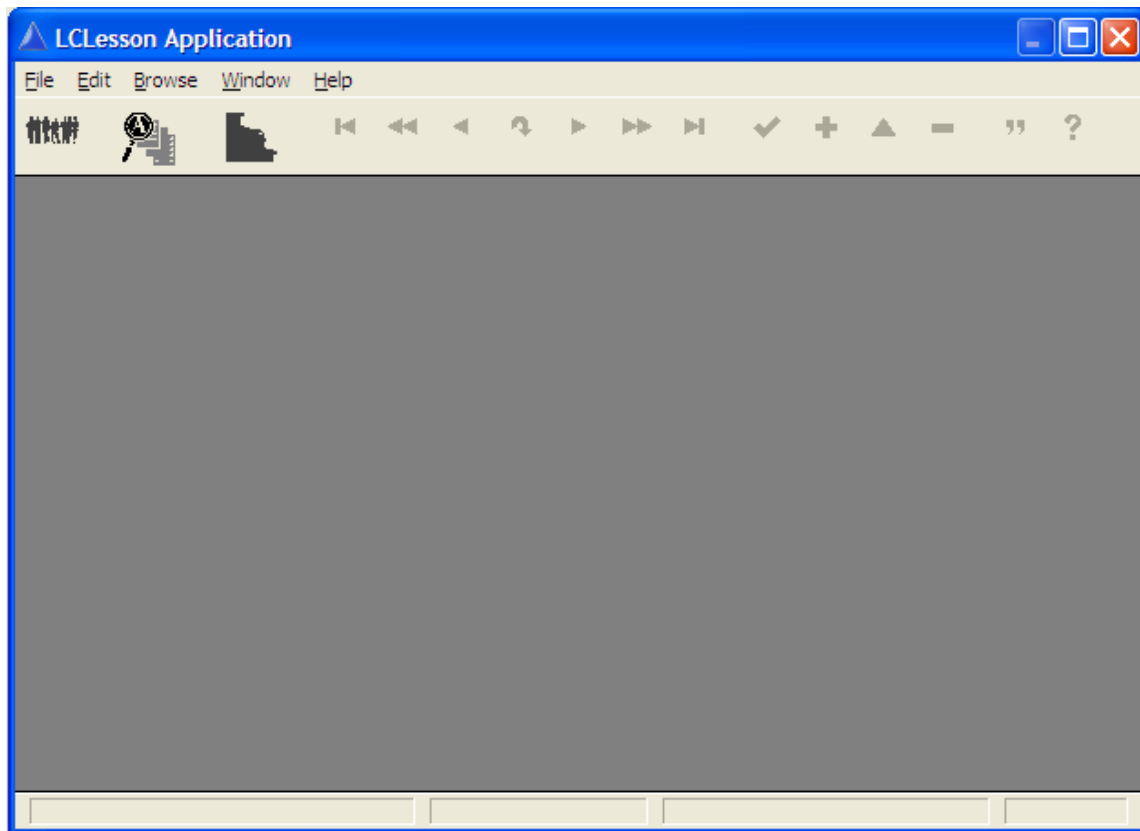


:

The Application Generator generates the source code, displaying its progress in a message window, procedure by procedure.

Next, the **Output** window appears, showing you the progress of the build as the compiler and linker do their work.

Then your Application window appears.



2. Press one of the buttons on the toolbar, or choose one of the items on the **Browse** menu.

A message box appears that says, "The Procedure (*procedure name*) has not been defined":

This capability allows you to incrementally test your application, whether you have designed all the procedures or not.

You'll fill in their functionality, starting in the next topic (Creating a Browse).

3. Press the **OK** button to close the message box.
4. Choose **File ► Exit** to close the *LCLesson* application.

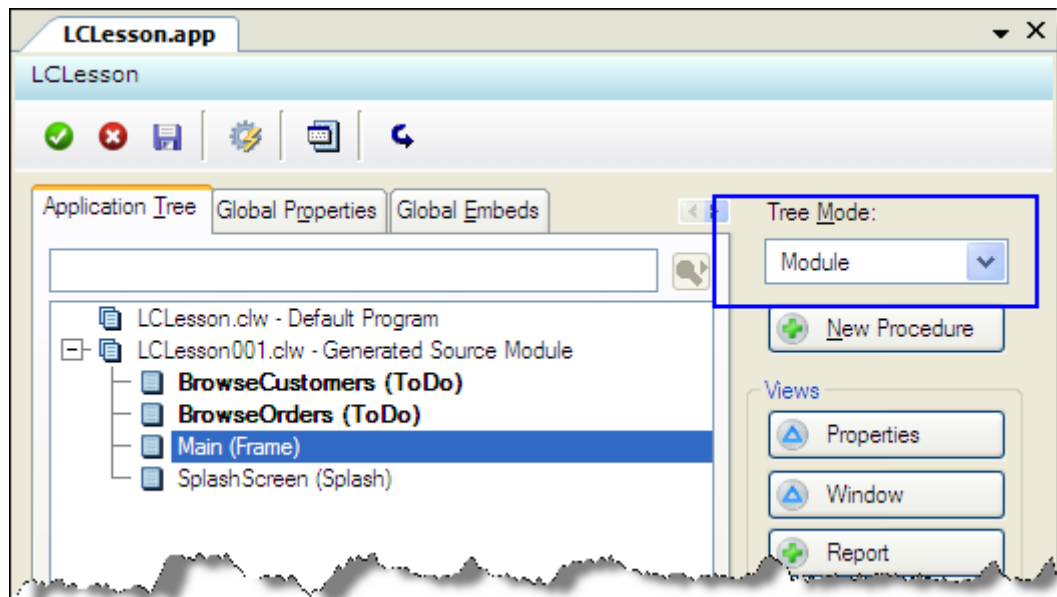
Throughout the rest of this lesson, feel free to Make and Run the developing application whenever the lesson instructs you to save the file.

Look at the Generated Source Code

Let's take a quick look at what the Application Generator has done for you. The whole purpose of the Application Generator (and its Templates) is to write Clarion language source code for you. There is no "magic" to what the Clarion toolset does to create applications—it all goes back to the Clarion programming language.

1. With the Application Tree dialog open, Change the **Tree Mode** drop list to *Module*.


This changes your view of the application from the logical procedure call tree to the actual source code modules generated for the application.



2. Highlight the *LCLesson.CLW* module, RIGHT-CLICK to display the popup menu then CLICK on **Module**.

This takes you right into the Text Editor, looking at the last source code you generated (the last time you pressed the Run button). Any changes you made since the last time you generated code will not be visible here.

The *LCLesson.CLW* file is the main program module for this application, containing the Global data declarations and code. Don't be intimidated looking at all this code. After you've finished this lesson, you can go on to the *Introduction to the Clarion Language* lesson at the end of the Learning Clarion topics and become more familiar with the Clarion Language (it's actually very straight-forward).

3. When you have finished looking at the code, choose **File ► Close ► File** to exit the Text Editor and return to the Application Generator (You can also simply press the Close button  in the source window).

Note:

Be sure NOT to choose **File ► Exit**, otherwise you'll exit Clarion for Windows completely.

4. With the Application Tree dialog open, Change the **Tree Mode** drop list to *Procedure*.


This changes your view of the application back to the logical procedure call tree.

5. Highlight the *Main (Frame)* procedure, RIGHT-CLICK to display the popup menu, then CLICK on **Module**.

This takes you into the Text Editor again, looking at the last source code you generated for the Main procedure. Again, any changes you made in the Application Generator since the last time you generated code will not show up in this code.

You may have noticed that right below the *Module* selection was another called *Source*. Do not confuse these two, they do very different things. We will demonstrate the *Source* selection later in the lessons.

If you do make any changes to this code, you actually can compile and run the program to see what effect the changes make, however, your changes will be lost the next time you generate source code. Therefore, it is not a good idea to make any changes here.

6. When you have finished looking at the code, choose **File ▶ Close ▶ File** to exit the Text Editor and return to the Application Generator. You can also simply press the **Close** button  in the source window.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- ✓ You created a new .APP file—without using a wizard.
- ✓ You created an application Frame procedure—without using a wizard.
- ✓ You created a menu in your application frame.
- ✓ You created a splash screen procedure for your application.
- ✓ You created a toolbar under your application's main menu and placed iconized, flat buttons on it.
- ✓ You used Clarion's resize and alignment tools to adjust your screen design.
- ✓ You used a Control Template to populate your toolbar with a set of standard navigation buttons.
- ✓ You compiled and ran your work-in-progress to test its functionality.

In the next lesson, we'll add a Browse procedure to the application.

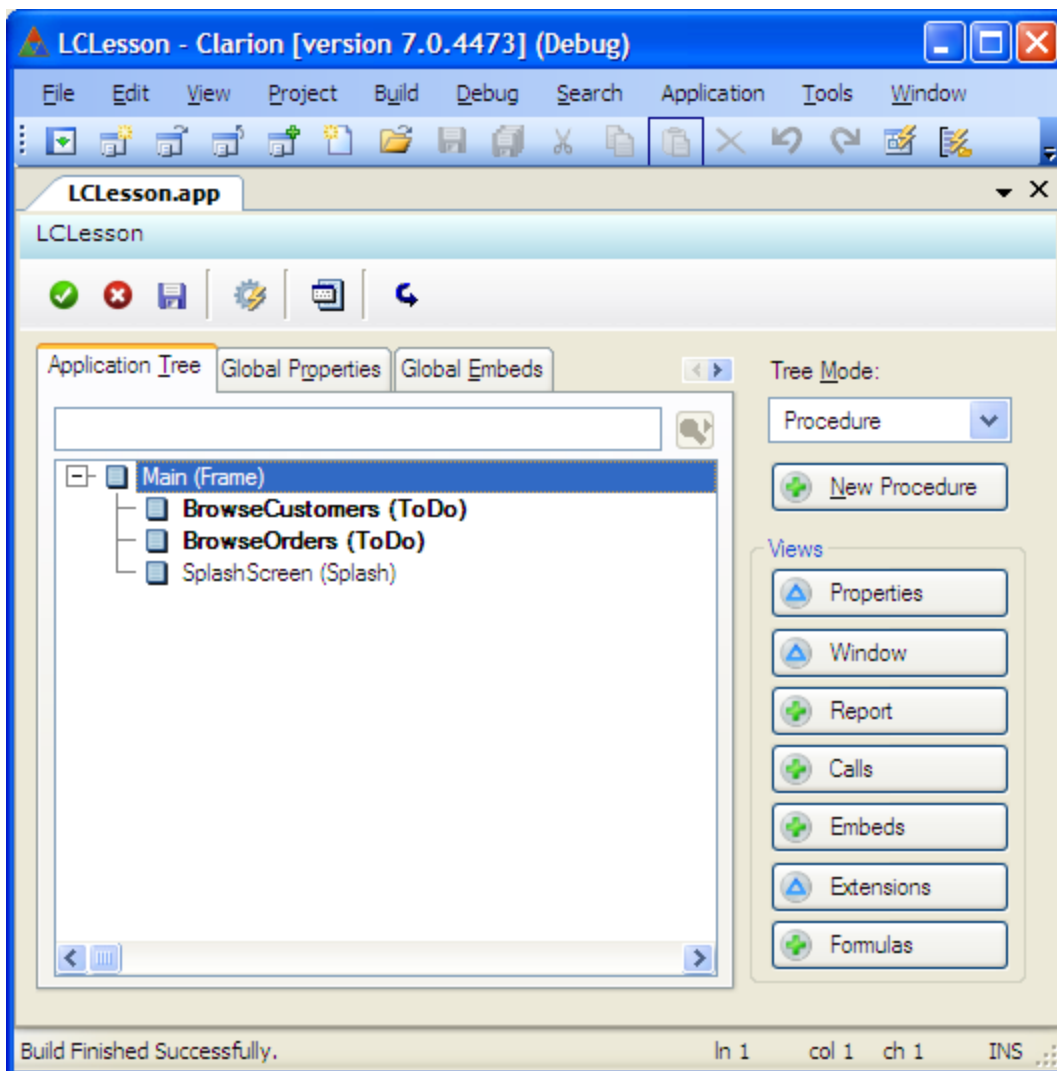
7 - Creating a Browse

Creating a Browse Window

In this chapter, you'll create a browse window similar to the one created for you by the Application Wizard. The Application Generator uses the same templates, which generate the same basic code—but doing it this way, you'll have a chance to "do it from scratch." This shows you just how much the Wizards do for you, and how you can do it all yourself, too. You'll start with the Customer browse window.

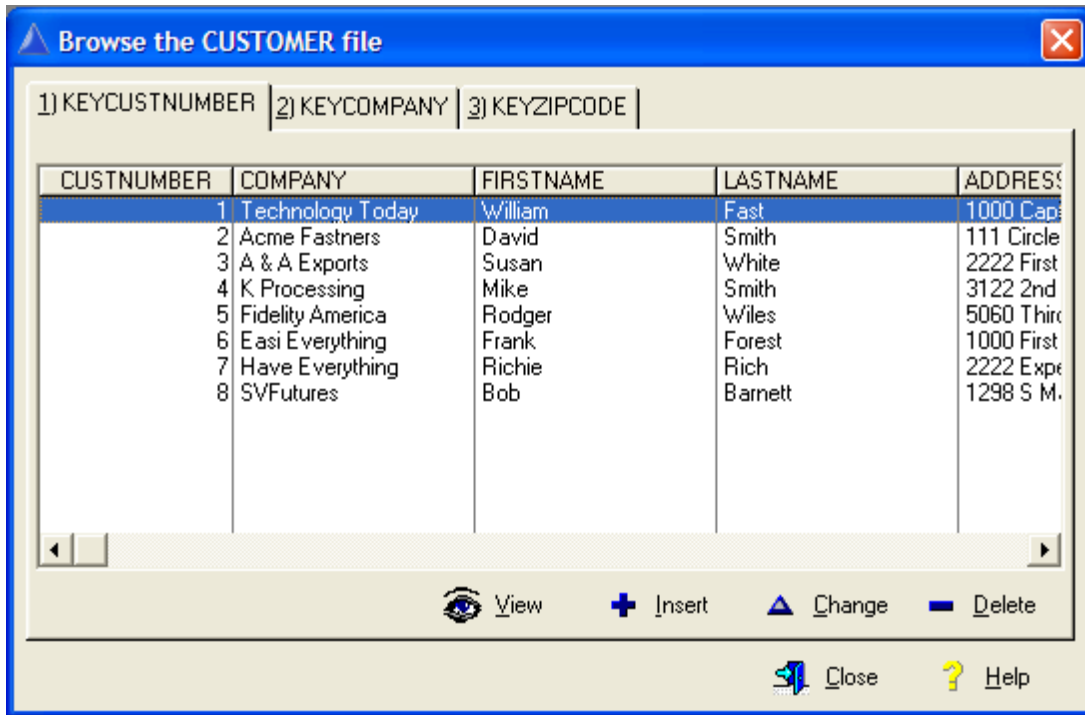
Starting Point:

The **LCLESSON.APP** should be open in the Application Editor (Tree).



Creating the Customer Browse Window

You recall from the Getting Started lessons that the Application Wizard created a window for the *Customer* table Browse procedure that looked like this:

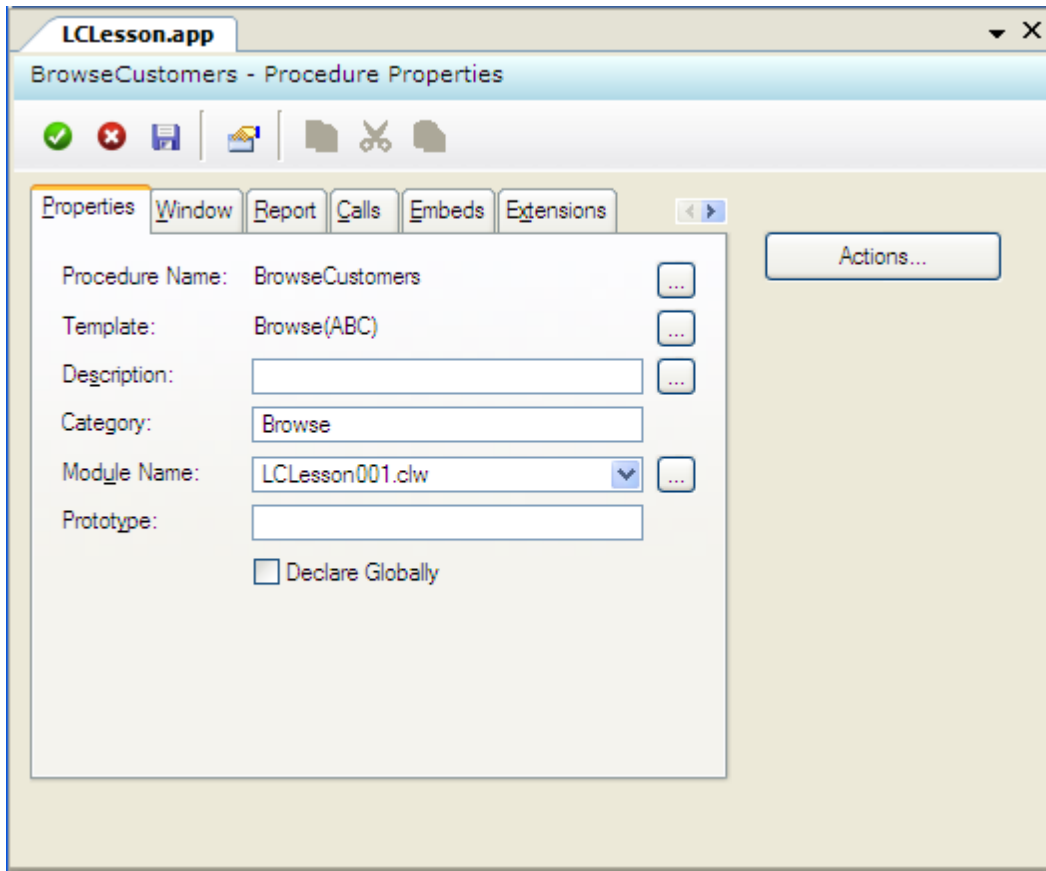


Now you'll create a similar one using the *Browse Procedure* template:

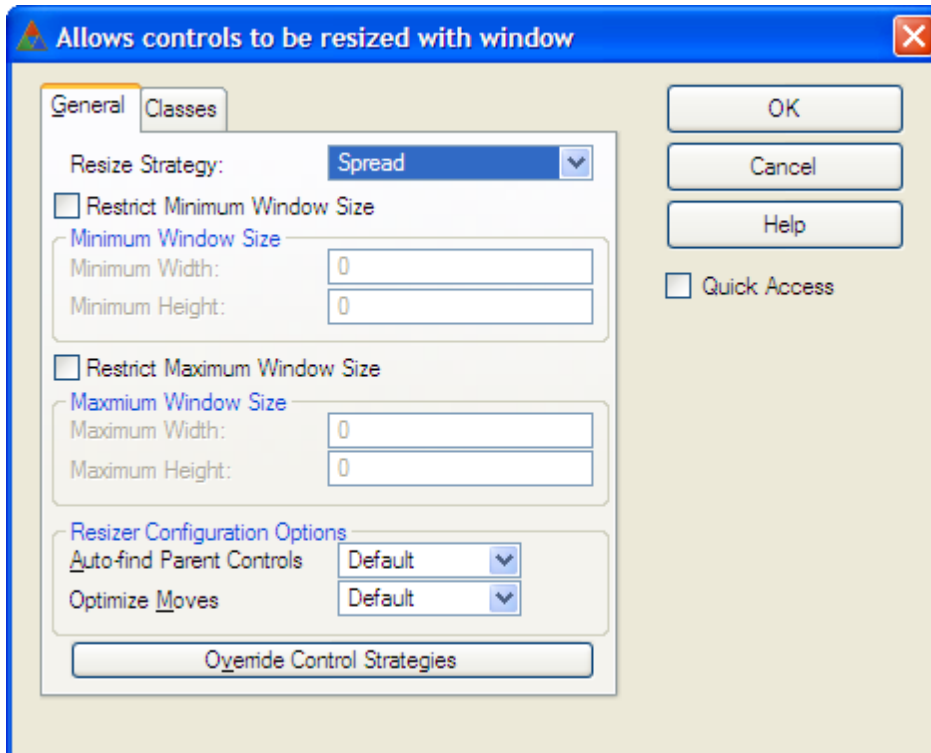
Select the procedure type for the BrowseCustomers procedure

1. DOUBLE-CLICK on *BrowseCustomers* in the Application Tree.
2. Select the **Defaults** tab and highlight the *Browse with Update and Select Procedure* template in the *Select Procedure Type* dialog, then press the **Select** button.

The *Procedure Properties* dialog appears.

**Make the window resizable**

1. In the *Procedure Properties* dialog, select the **Extensions** tab.
2. In the *Extensions* dialog, press the **Insert** button.
3. Highlight *WindowResize* in the *Select Extension* dialog, then press the **Select** button.

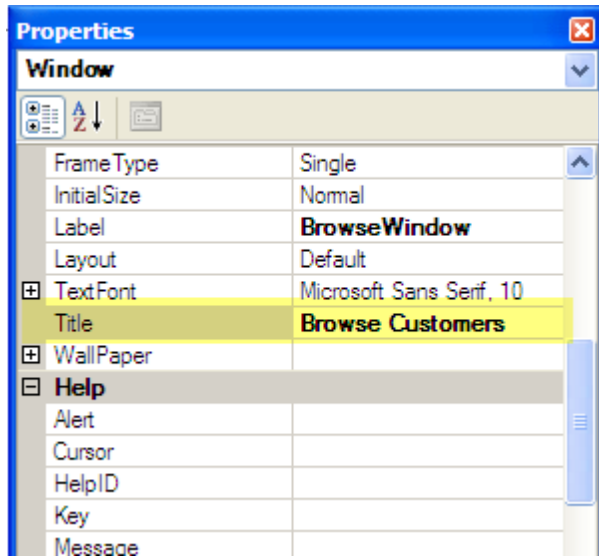


This Extension template generates code to automatically handle resizing and re-positioning all the controls in the window when the user resizes the window, either by resizing the window frame, or by pressing the Maximize/Restore buttons.

4. Press the **OK** button to close the *Window Resize* template dialog.

Edit the Browse procedure window

1. In the *Procedure Properties* dialog, select the **Window** tab.
2. Press the **Designer** button to load the Window Designer.
3. CLICK on the window's title bar, and open the *Properties Pad* (if not already opened). If not opened press the F4 key to open it.
4. In the *Properties Pad* dialog, type *Browse Customers* in the **Title** property:




5. Select *Resizable* from the **FrameType** drop list property.
6. Locate the **MaximizeBox** property and set to TRUE.

These last two steps allow the user to resize the window at runtime.

Populating and Formatting a List Box Control


The List Box Formatter allows you to format the data in the list.

Prepare to format the list box

1. RIGHT-CLICK on the list box in the window, then choose **List Box Format...** from the popup menu. Press the *Add Field* button .

The *Select Column* dialog appears. This provides access to the tables defined in the data dictionary. The *Tables* list displays all the tables selected for use in this procedure in a hierarchical arrangement (the Table Schematic), which includes the browse list box control.

Note:

You can call this dialog at any time by pressing the **Add Field**  button.

Select the table and columns to place in the browse list box control

1. Highlight the *<ToDo>* item below the *File-Browsing List Box* and press the **Add** button.
2. Highlight the *Customer* table in the *Select* dialog, then press the **Select** button.

This adds the table to the Table Schematic in the *Select Column* dialog, which now lists the table and its columns.

3. With the *Customer* table that you just highlighted selected, press the **Change** button.

4. Highlight *KeyCustNumber* in the *Change Access Key* dialog and press the **Select** button.

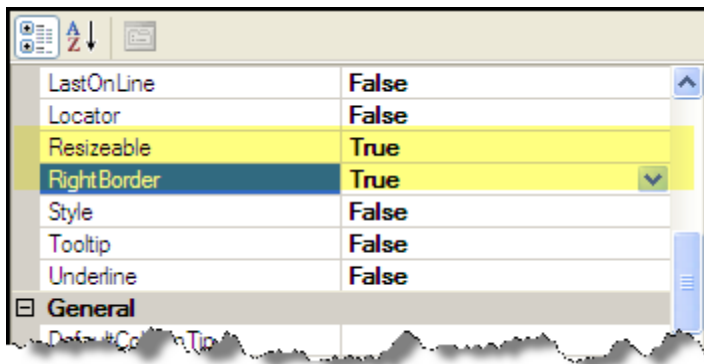
This is important, because it sets the display order for the rows in the list. If you don't specify a key, the rows appear in (sort of) whatever order they were added to the table (also called "Row Order").

5. Highlight *COMPANY* in the **Columns** list, then press the **Select** button.

This brings you into the List Box Formatter with the selected column added to the list. The tabs on the right allow you to format the appearance of the column highlighted in the list on the left.

Apply special formatting to the first column


1. In the Property list in the lower right pane, locate the **Right Border** and **Resizable** properties and set them to TRUE:



This adds a resizable right vertical border to the column at runtime.


2. Set the Header **Indent** property to 2 to slightly indent the heading text and verify that the **DataIndent** property is already set at 2.

Populate the second column

1. Press the **Add Field**  button.
2. Highlight *FIRSTNAME* in the *Select Columns* list, and press the **Select** button.
3. In the Property list in the lower right pane, locate the **Right Border** and **Resizable** properties and set them to TRUE if not already set.

The List Box Formatter automatically "carries forward" these formatting options from the last column you added, making it very simple to add multiple columns with similar formatting options. In this case, clearing these check boxes deletes the column divider between this and the next column, which will be the *LastName* column.


Populate the third column

1. Press the **Add Field**  button.
2. Highlight *LASTNAME* in the *Select Columns* list, and press the **Select** button.


3. In the Property list in the lower right pane, locate the **Right Border** and **Resizable** properties and set them to TRUE.

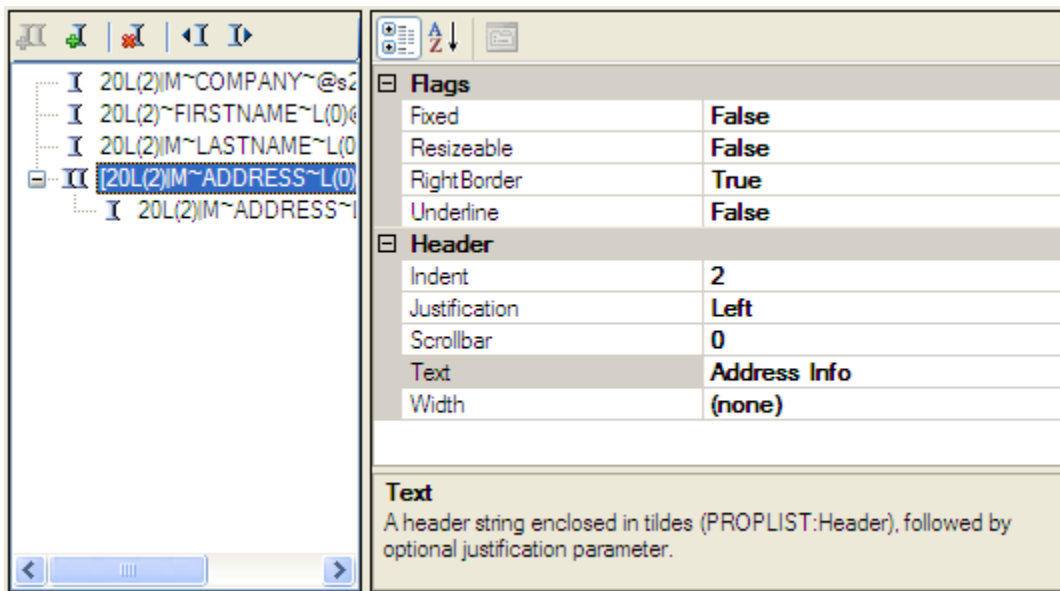
This once again adds the resizable column divider between this and the next column.

Populate the fourth column

1. Press the **Add Field**  button.
2. Highlight *ADDRESS* in the *Select Columns* list, and press the **Select** button.
3. In the Property list in the lower right pane, locate the **Right Border** and **Resizable** properties and set them to TRUE.

Group some columns

1. Press the **Add Group**  button.
2. Highlight the new group as shown here:



Note:

As the list box begins to grow in size, you can resize the List Box Formatter as needed to view additional elements.

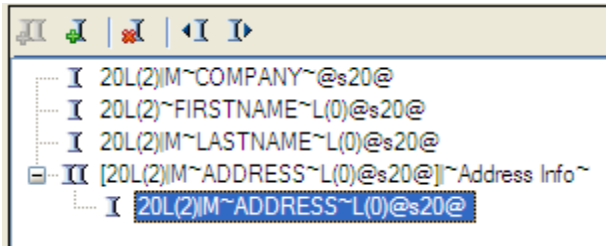
By creating a new group, in which you'll place the address information, you can add a group header. This appears above the column headers, and visually links the data in the columns beneath. Notice that, as you add columns and make changes you can see the effects of your changes in the sample list box at the top of the List Box Formatter dialog.

3. Type *Address Info* in the **Text** property.


This provides the text for the group header. Any columns appearing to the right of this one will be included in the group, until you define another group.

As you add columns, the List Box Formatter continually updates its sample window (at the top) to show you how your list will appear.


4. Highlight the *ADDRESS* column:



Populate the fifth column

1. Press the **Add Field**  button.
2. Highlight *CITY* in the **Columns** list, and press the **Select** button.
3. In the Property list in the lower right pane, locate the **Right Border** and **Resizable** properties and set them to TRUE.

Populate the sixth column

1. Press the **Add Field**  button.
2. Highlight *STATE* in the **Columns** list, and press the **Select** button.
3. In the Property list in the lower right pane, locate the **Right Border** and **Resizable** properties and set them to TRUE.

Populate the seventh column, and exit the List Box Formatter

1. Press the **Add Field**  button.
2. Highlight *ZIPCODE* in the **Columns** list, and press the **Select** button.

At this time, feel free to use the WYSIWYG display at the top to resize any of the columns as needed.

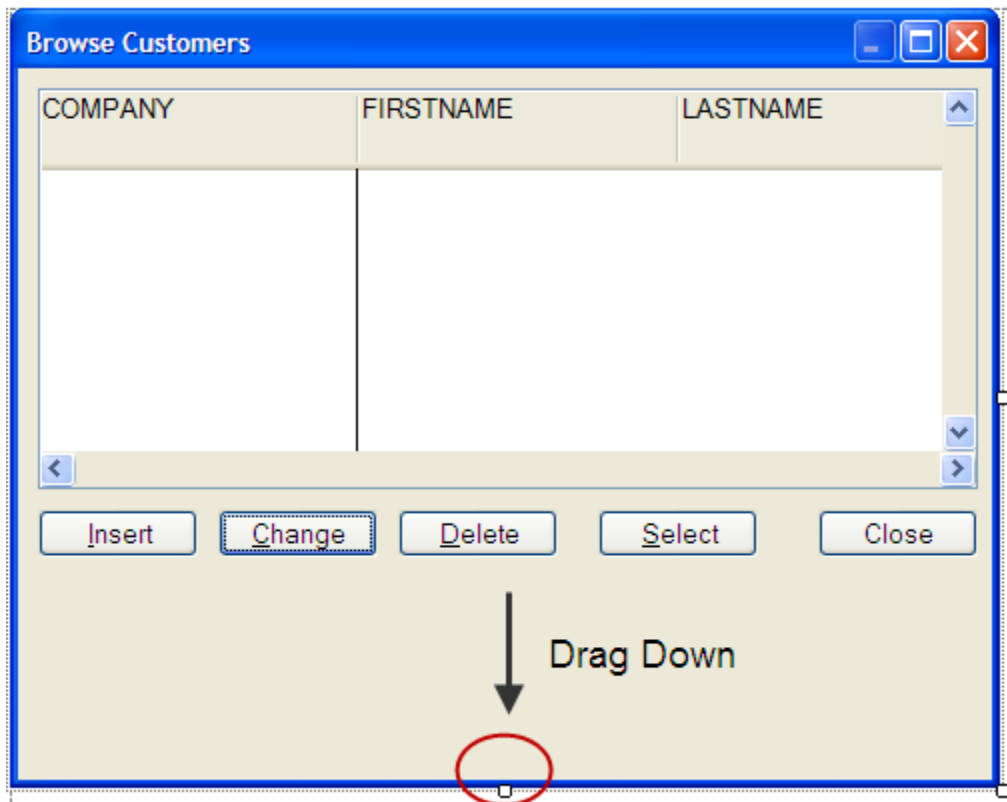
3. Press the **OK** button to close the **List Box Formatter**.

Adding the Tabs

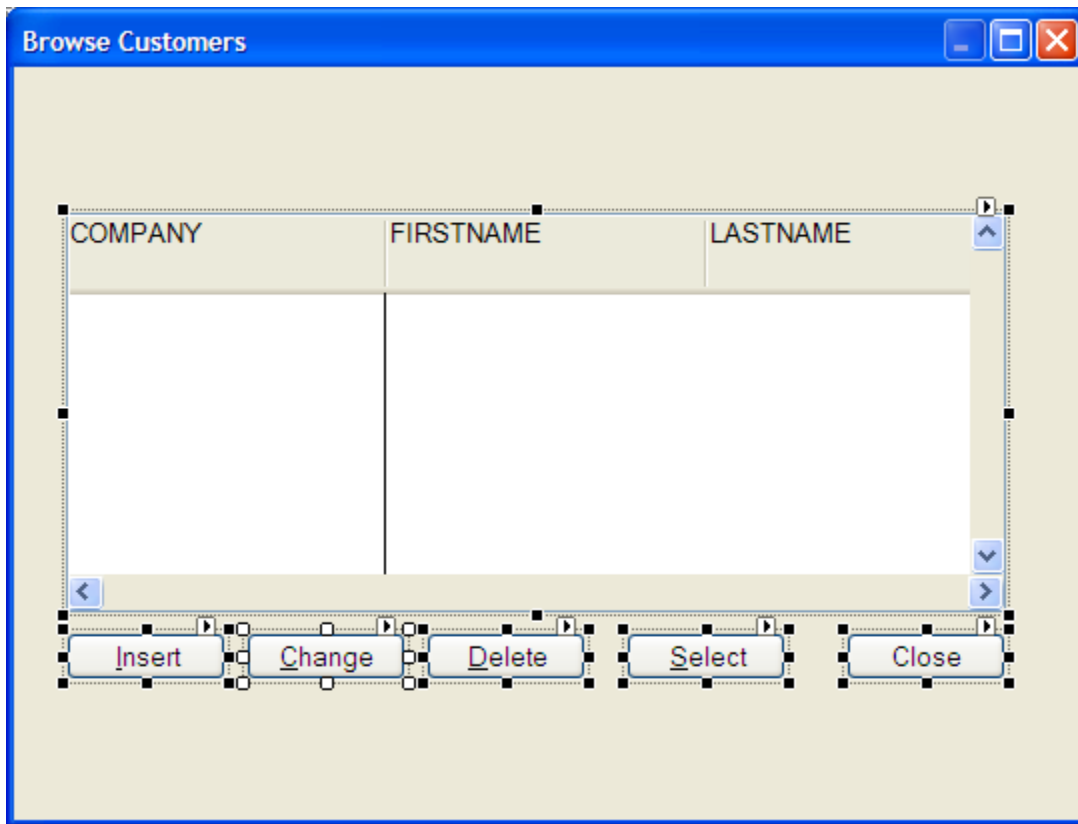
When the Application Wizard created this procedure it had tab controls that changed the list's sort order depending on which tab was selected. Therefore, we'll add this functionality right now to show how easy it is to accomplish!

Add the Property Sheet and the first tab

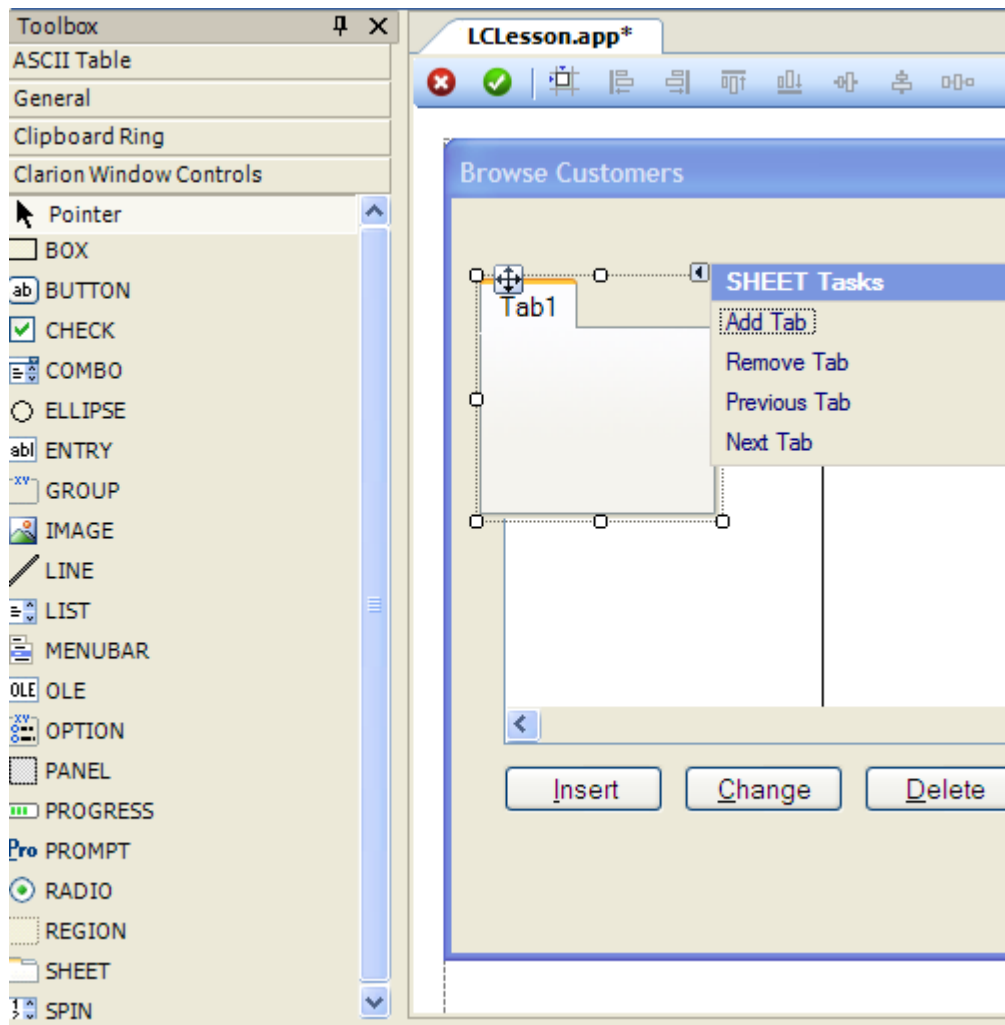
1. CLICK on your window's title bar to place the white "handles" on your window design.
2. Place the mouse cursor directly over the handle on the lower edge, and DRAG it down to resize the window.



3. CLICK and drag the LIST and BUTTONS to select all of them, then use your mouse or arrow keys to move all controls down as shown here:



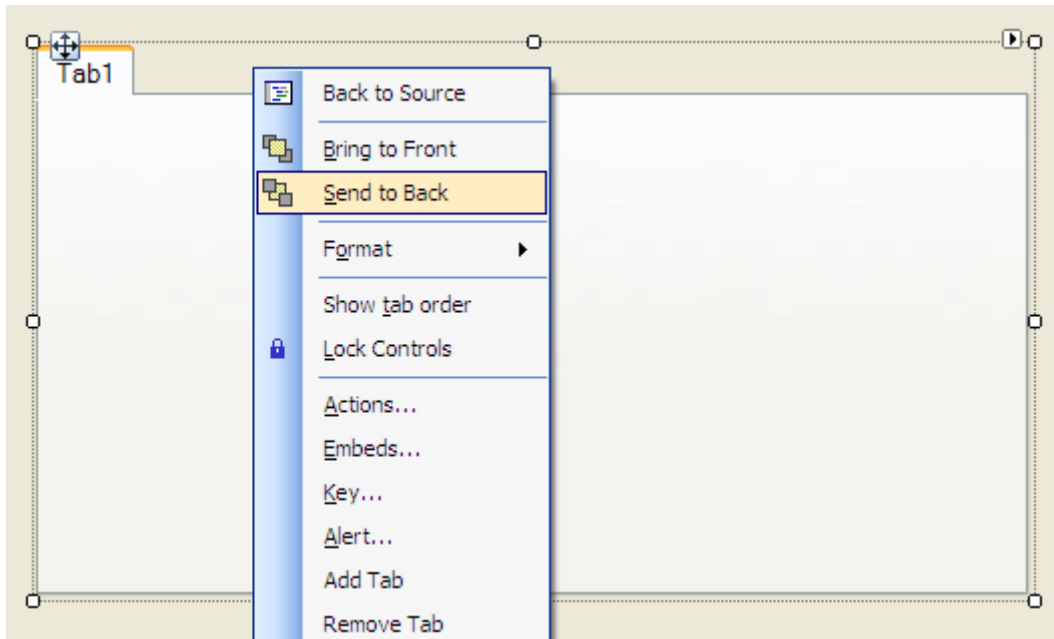
4. If the Controls **Toolbox** is not already opened, open it now by selecting **View ► Toolbox** from the IDE Menu (or simply press CTRL + ALT + X)
5. CLICK and DRAG the **SHEET** control in the *Controls* toolbox above and to the left of the List box to place the property sheet and one Tab control.



6. DRAG the "white handle" at the bottom *left*-hand corner so that it appears just below and to the left of the Insert button.
7. DRAG the "white handle" at the bottom *right*-hand corner so that it appears just below and to the right of the Close button.

This resizes the property sheet so that it appears as though the list box and buttons are on the tab. In fact, they are not, and we don't want them to be, since we want all these controls to be visible no matter which tab the user selects.

8. The list and button controls appear to be hidden. RIGHT-CLICK on the SHEET control JUST TO THE RIGHT OF THE TAB CONTROL, and select Send To Back from the popup menu.

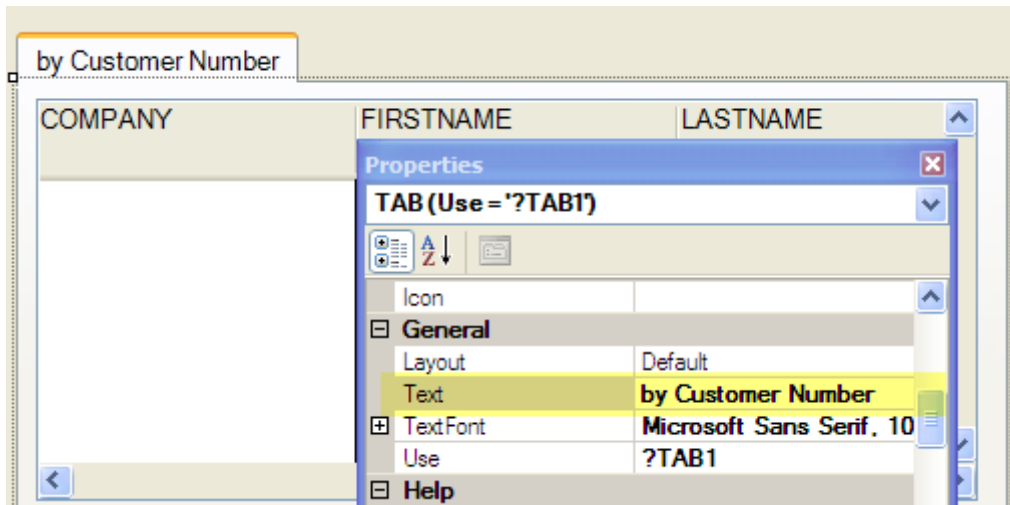


9. CLICK just below the "Tab1" text for the first tab in the sheet.



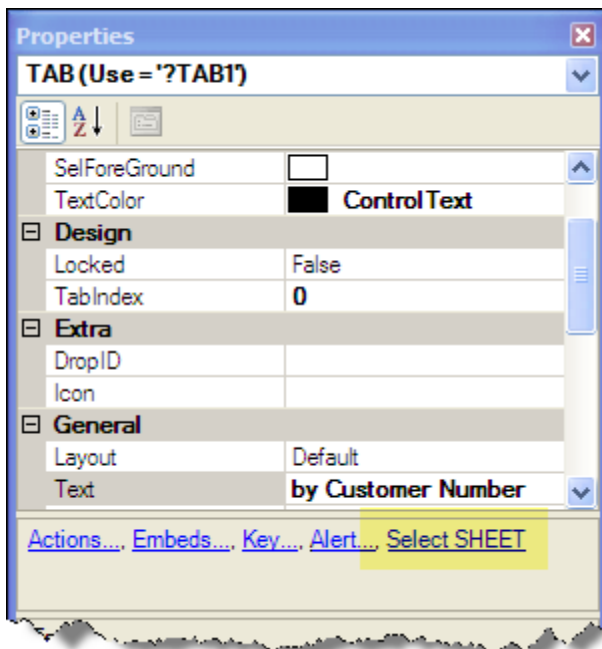
10. In the **Properties Pad**, type *by Customer Number* in the **Text** property, and then press the TAB key.

This changes the tab's text. This tab text can be anything, but naming the key also names the sort order it will display.

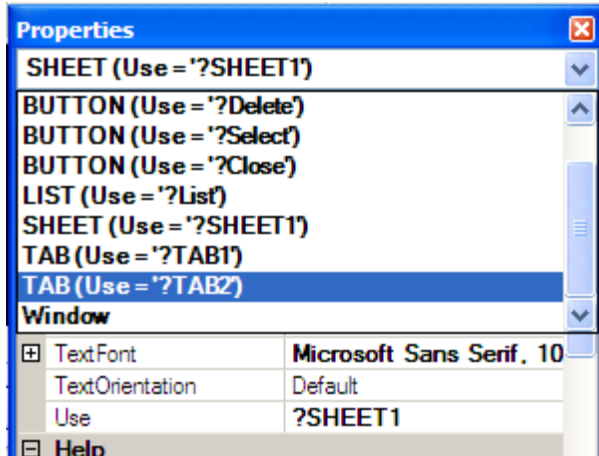


Add the rest of the tabs

1. With the *Properties Pad* still opened for the TAB control, locate and click on the **Select Sheet** link shown below:



2. With the SHEET control selected and active in the *Properties Pad*, press the **Add Tab** link.
3. Here's another way to select a control in your active window. In the *Properties Pad*, select the ?TAB2 control in the drop list as shown here:



4. Type *by Company Name* in the **Text** entry of the *Property* toolbox, then press TAB.
5. With the *Property Pad* still opened for the ?TAB2 control, locate and click on the **Select Sheet** link.
6. With the SHEET control selected and active in the *Properties Pad*, press the **Add Tab** link.
7. In the *Property Pad*, select the ?TAB3 control in the drop list
8. Type *by Zip Code* in the **Text** entry of the *Property* toolbox, then press TAB.

Hiding the Buttons

The toolbar buttons defined on the Frame procedure actually just tell hidden buttons in the Browse procedure to do what they normally do. Therefore, when you are designing a Browse procedure without using the Wizards, you do need to have the update buttons on the screen, but the user does not have to see them at runtime.

1. RIGHT-CLICK on the *Close* button in the sample window then choose **Properties...** from the popup menu.
2. Set the **Hide** property to TRUE.

This adds the HIDE attribute to the control so you won't see it on screen at runtime. Of course, you can still see it in the Window Designer.

3. RIGHT-CLICK on the *Select* button then choose **Properties....**
4. Set the **Hide** property to TRUE.
5. RIGHT-CLICK on the *Delete* button then choose **Properties....**
6. Set the **Hide** property to TRUE.
7. RIGHT-CLICK on the *Change* button then choose **Properties....**
8. Set the **Hide** property to TRUE.
9. RIGHT-CLICK on the *Insert* button then choose **Properties....**
10. Set the **Hide** property to TRUE.

Note:

Alternatively, you can select all of the buttons by pressing CTRL + CLICK, and then you can set the **Hide** property to TRUE for all selected button controls.

Move the buttons and resize the list

There's no need to waste the space these buttons (which the user won't see) occupy on the window, so we'll move them out of the way.

1. CLICK on the *Select* button then CTRL+CLICK on the *Close*, *Insert*, *Change*, and *Delete* buttons to select them all, then CLICK and drag the buttons up into the list box.

DRAGGING multiple controls at once allows you to move the controls while maintaining the relative positions of the controls within the group. Now we'll use the space we just gained to make the list longer.

2. CLICK on the list box then DRAG its bottom-center handle down to make the list longer.

Setting the Sort Orders

Now that the tabs are there, we need to tell the list box what alternate sort orders to use and when.

1. RIGHT-CLICK on the list box then choose **Actions...** from the popup menu.
2. Click on the **Browse Box Behavior** button.

The list box is actually a *BrowseBox* Control template that has been placed in the *Browse* Procedure template's default window design in the Template Registry. This means that it has associated prompts that tell it how to populate the list and what actions to perform.

The prompts that appear on the Actions tab come directly from the templates (in this case, the *BrowseBox* Control template). This is how you communicate to the templates exactly what code they need to generate to give you the behavior you ask for (and nothing else). These prompts, their meanings and uses, are all covered in this User's Guide and in the on-line help for each window in which they appear.

3. Select the **Conditional Behavior** tab.
4. Press the **Insert** button.
5. Type *CHOICE(?Sheet1) = 2* in the **Condition** entry.

This sets the condition under which the alternate sort order will be used. This expression uses the Clarion language CHOICE function to detect when the user has selected the second tab on the sheet. The generated code will use this expression in a conditional statement that will change the sort order at runtime.


6. Press the ellipsis button (...) next to the **Key to Use** entry.
7. Highlight *CUS:KeyCompany* then press the **Select** button on the *Select Key* dialog.

Now, when the user selects the second tab, the *BrowseBox* Control template will generate code to switch to the key on the *Company* column. It doesn't need to know what to do for the first tab, because that always uses the Access Key we set in the Table Schematic.

8. Press the **OK** button.
9. Press the **Insert** button.

10. Type `CHOICE(?Sheet1) = 3` in the **Condition** entry.
11. Press the ellipsis button (...) next to the **Key to Use** entry.
12. Highlight `CUS:KeyZipCode` then press the **Select** button on the *Select Key* dialog.
13. Press the **OK** button.
14. Press the **OK** button to close the *?List Prompts* dialog.

Closing the Customer Browse

1. Choose **Save and Close** on the Window Designer's menu bar, and save your window changes if prompted to do so.
2. Press the **Save and Close** button in the Window Designer Editor dialog to close it.
3. Choose **File ► Save**, or press the Save button  on the toolbar to save your work.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- ✓ You created a new Browse Procedure—without using a wizard.
- ✓ You added an Extension Template to automatically make the new procedure's window resizable.
- ✓ You used the List Box Formatter tool to design a scrolling list of rows.
- ✓ You used the List Box Formatter tool to design a scrolling list of rows.
- ✓ You added a Property Sheet and several Tabs to your screen design.
- ✓ You hid and moved buttons to provide a "cleaner" screen design.
- ✓ You used the Window Designer's Preview mode to see your window design in action.
- ✓ You set dynamic sort orders for the user based on which Tab control they select.

Now that the first Browse procedure is complete, we'll go on and create its associated update Form procedure.

8 - Creating an Update Form


Creating an Update Procedure

In the last chapter, we formatted the Customer Browse procedure's list box and added tab controls to change the sort order. To finish the basic procedure, we name the Update procedure. This is the procedure that handles the action for the Insert, Change, and Delete buttons.

Starting Point:

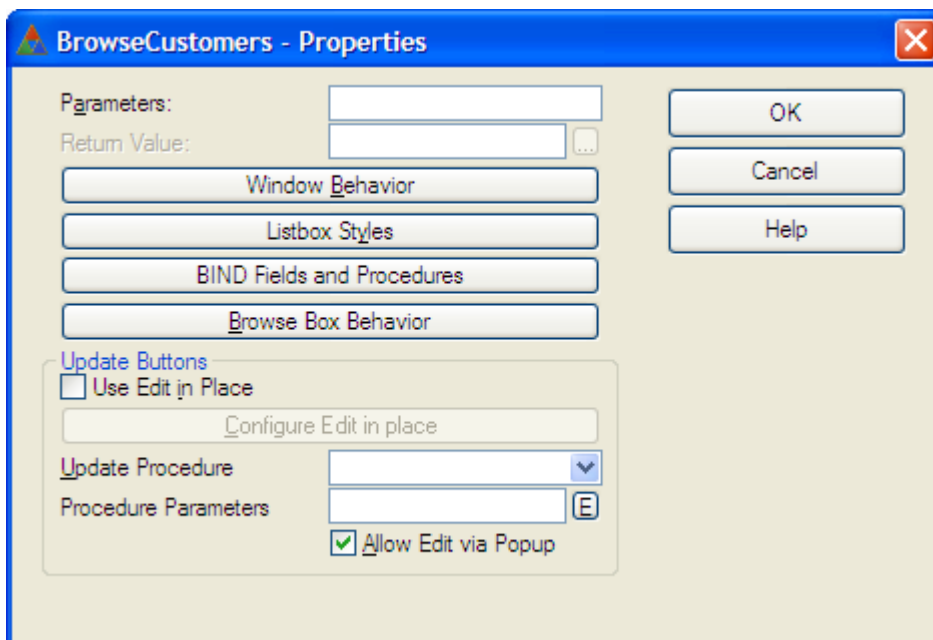
The **LCLESSON.APP** file should be open.

Add a "ToDo" procedure

1. Highlight *BrowseCustomers* in the Application Tree dialog, then press the **Properties**  button.

The *Procedure Properties* dialog appears. There are actually three ways to get to this dialog (use the method that suits the way you work):

- Highlight the procedure then press the **Properties** button.
 - DOUBLE-CLICK the procedure in the Application Tree dialog.
 - RIGHT-CLICK the procedure and select **Properties** from the popup menu.
2. Click on the **Actions** button to open the *Browse Customers – Properties* dialog.



3. Type *UpdateCustomer* in the **Update Procedure** entry box at the bottom of the *Procedure Properties* dialog.

This names the procedure to update the rows displayed in the browse. The new procedure appears in the Application Tree as a "ToDo."

4. Press the **OK** button to close the *Browse Customers – Properties* dialog, and then press the **Save and Close** button to close the *Procedure Properties* dialog.

Notice that you didn't have to start a new execution thread for the update procedure. You want it to run on the same thread as the browse, so that the end user can't open a form window to change a row, then activate the browse window again, and open another form on the same row. In other words, you don't want a user trying to change the same row twice at the same time!

Creating the Update Form Procedure

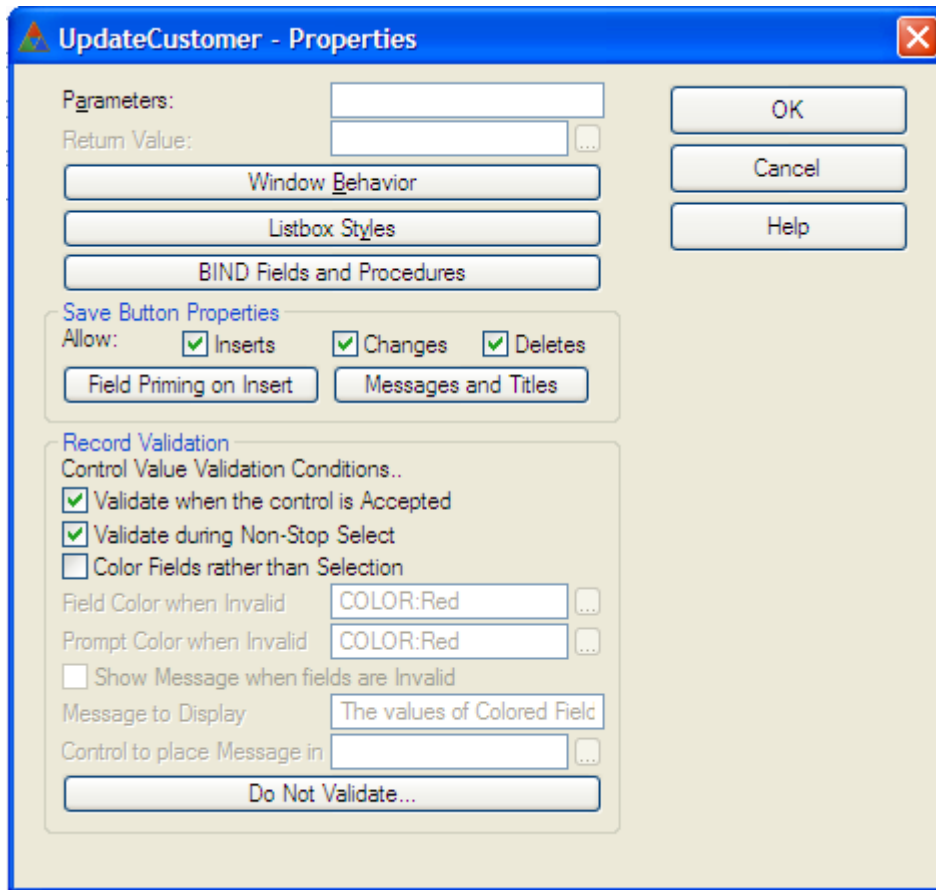
The Update Procedure should use the Form Procedure template to create a procedure that the end user can use to maintain a row. It should provide a prompt and entry control for each column in the row.

Select the procedure type for UpdateCustomer.

1. DOUBLE-CLICK on *UpdateCustomer* in the Application Tree dialog.
2. Choose the **Defaults** tab, Highlight the *FORM (Add/Edit/Delete)* template, then press the **Select** button.

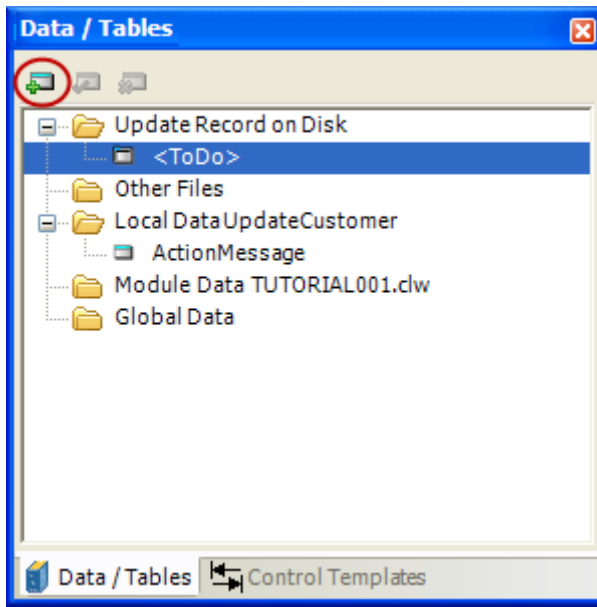
The *Procedure Properties* Window appears.

3. Press the **Actions** button.



Notice that this dialog looks different than the *Splash*, *Frame*, or *Browse Procedure Properties* dialogs, because the prompts vary for each type of Procedure template. The User's Guide and on-line help describe the customization options available on each *Procedure Properties* dialog.

4. Press **OK** to close the *Update Customer – Properties* dialog.
5. Open the **Data / Tables Pad** to name the table the Form will update. If not already opened, press the F12 key, or select **View ► Data / Tables Pad** from the IDE Menu.
6. Highlight the <ToDo> just under the *Update Record on Disk* entry and press the **Add** button.



7. The *Select a Table* dialog appears. Highlight the *Customer* table in the *Select* dialog, then press the **Select** button.

Make the window resizable

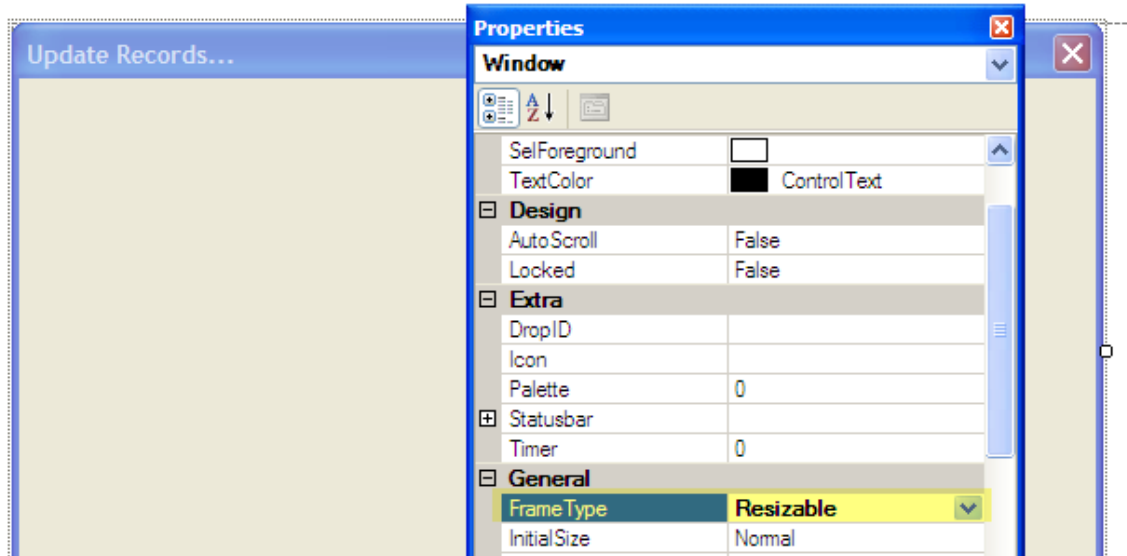
1. In the *Procedure Properties* dialog, select the **Extensions** tab.
2. In the *Extensions* dialog, press the **Insert** button.
3. Highlight *WindowResize* in the *Select Extension* dialog, then press the **Select** button.

This Extension template generates code to automatically handle resizing and re-positioning all the controls in the window when the user resizes the window, either by resizing the window frame, or by pressing the Maximize/Restore buttons.

4. Press the **OK** button to close the *Allow controls to be resized with window* dialog.

Edit the Window Properties

1. In the *Procedure Properties* dialog, press the **Window** tab.
2. Press the **Designer** button to load the *Window Designer*.
3. CLICK in the window's title bar, then open the *Properties Pad* (if not already opened – remember the F4 key?).
4. Select *Resizable* from the **FrameType** property.



5. Set the **MaximizeBox** property to *TRUE*.

These last two items allow the user to resize the window at runtime.

Populating the Columns

The default window design contains three controls for you already. The "OK" button will close the dialog, accepting the end user's input and writes the Customer table row to disk. The "Cancel" button closes the form without updating. The string control provides an action message to inform the end user what action they are taking on the row.

Placing the columns in a window is called populating it.

1. Choose **Window Designer ► Populate ► Multiple Columns**.

This displays a dialog containing all the columns from all the tables specified in your procedure's Table Schematic. These columns are all ready to populate onto your window design.

2. First, select the CUSTOMER table, and then DOUBLE-CLICK on *CustNumber* in the *Select Column* dialog then move the cursor over the window design.

The cursor changes to a crosshair and a "little book" which indicates the column comes from the data dictionary.

3. CLICK near the upper left corner of your window design.

This places both the data entry control and its associated prompt. These controls default to whatever you specified in the data dictionary for the column.

4. DOUBLE-CLICK on *Company* in the *Select Column* toolbox, and then move the cursor to where you want your control to appear, and CLICK.
5. DOUBLE-CLICK on *FirstName* in the *Select Column* toolbox, and then move the cursor to where you want your control to appear, and CLICK.
6. DOUBLE-CLICK on *LastName* in the *Select Column* toolbox, and then move the cursor to where you want your control to appear, and CLICK.
7. DOUBLE-CLICK on *Address* in the *Select Column* toolbox, and then move the cursor to where you want your control to appear, and CLICK.

8. DOUBLE-CLICK on *City* in the *Select Column* toolbox, and then move the cursor to where you want your control to appear, and CLICK.
9. DOUBLE-CLICK on *State* in the *Select Column* toolbox, and then move the cursor to where you want your control to appear, and CLICK.
10. DOUBLE-CLICK on *ZipCode* in the *Select Column* toolbox, and then move the cursor to where you want your control to appear, and CLICK.
11. Press **Cancel** in the *Select Column* toolbox to return to the Window Designer.

Tip

The sequence of steps above can also be accomplished directly via the Data / Tables Pad! Simply DRAG a control from the Pad and DROP it on to the window. It's that easy, try it!

Moving and Aligning Columns

For a professional look, we need to move these columns around and align the sides and bottoms of all the columns in the screen.

Move the columns to their approximate positions.

1. CLICK on the *State* entry control.
2. SHIFT+DRAG the *State* entry control to the left, closer to its prompt.

SHIFT+DRAG "constrains" the control's movement to the plane of its first movement (either horizontal or vertical).

3. CTRL+CLICK on the *State* prompt.

When you CTRL+CLICK on a control, the previously selected control's handles turn blue while the newly selected control's handles are red. You now have two controls selected.

4. Move the *State* entry control and prompt (CLICK and DRAG on either of the two selected controls) up and to the right of the *City* controls.

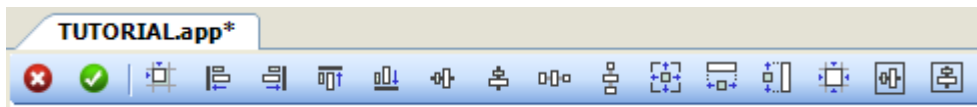
Once multiple controls are selected, you can move them as a group the same way you would move one individual control.

5. CLICK on the *ZipCode* entry box.
6. SHIFT+DRAG the *ZipCode* entry box to the left, closer to its prompt.
7. CTRL+CLICK on the *ZipCode* prompt.
8. Move the *ZipCode* entry box and prompt up and to the right of the *City* and *State* controls.

You may need to make the window a little wider to accomplish this.

9. CLICK on the *LastName* entry box.
10. SHIFT+DRAG the *LastName* entry box to the left, closer to its prompt.
11. CTRL+CLICK on the *LastName* prompt.
12. Move the *LastName* entry box and prompt up and to the right of the *FirstName* controls.

Align the columns to their final positions



The Window Designer has an *Alignment* toolbar (shown above) that contains the same set of alignment tools that are also available through the **Window Designer > Format** menu.

1. CLICK on the first prompt in the upper left corner.

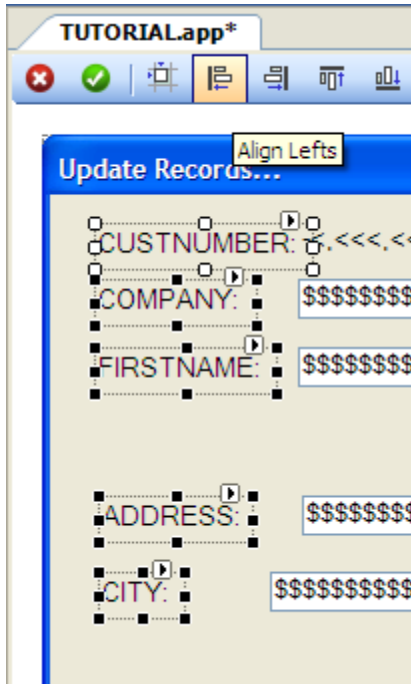
This should be the *CustNumber* prompt. Its handles should appear when you CLICK it.

2. CTRL+CLICK on the four prompts immediately below the first.

As you CTRL+CLICK on each control in turn, the previously selected controls' handles turn blue while the newest selected control's handles are red. The control with the red handles provides the "base point" for the alignment operation. All the other selected controls are aligned in relation to the control with the red handles.


3. Press the **Align Left** button  (the top left button) in the *Alignment* toolbar.

The controls all line up along their left edges, based on the position of the last item selected (the one with the red handles).



Tip

To identify the controls in the Alignment toolbar, simply place the mouse cursor over the control and wait half a second for the tool tip to appear.

4. Press the **Spread Vertically** button  in the floating *Align* toolbox.

The controls all evenly spread themselves between the top and bottom controls selected.

5. CLICK on the first entry control (this should be the *CustNumber* string control).
6. CTRL+CLICK on the entry controls immediately below it to select them all.
7. RIGHT-CLICK and choose **Align Left** from the popup menu that appears.

Here is yet another way to get to the alignment tools. The alignment popup menu appears only when you have multiple controls selected.

8. RIGHT-CLICK and choose **Spread Vertically** from the popup menu that appears.

This should align all the data entry controls with their respective prompts that we already spread vertically.

9. CLICK on the *LastName* entry control.
10. CTRL+CLICK on the three controls to its left (its prompt, the *FirstName* entry control and prompt).
11. Choose **Align Middles** from either the Alignment toolbar or the RIGHT-CLICK popup alignment menu.

This aligns the controls in a neat row.

There is one more way to select multiple controls in the Window Designer: Lasso them.

12. Place the mouse cursor slightly above and to the left of the first control in the bottom row (this should be the *City* prompt).
13. CTRL+CLICK and drag slightly down and to the right until the box outline surrounds all five controls to the right (the prompts and controls for *City*, *State* and *ZipCode*) then release the mouse button.

The red "handles" appear on the *ZipCode* entry control and the blue "handles" on the other controls. This is the "lasso" technique.

14. Choose **Align Middles** from either the floating Alignment toolbar or the RIGHT-CLICK popup alignment menu.
15. Use the **Align Middles** tool to align the *CustNumber*, *Company*, and *Address* entry boxes with their respective prompts.

The window should now look something like this:

The form window is almost done. Now we will add a browse list box for the related Phones table rows.

Adding a BrowseBox Control Template

Control templates generate all the source code required to create and maintain a specific type of control (or set of controls) on your window. All the entry controls we just placed on this window are simple controls, not Control templates, because they do not need any extra code to perform their normal function. Control templates are only used when a specific control needs extra functionality that the "bare" control itself does not provide. For example, the OK and Cancel buttons are both Control templates—the OK button's Control template saves the row to disk, while the Cancel button's Control template has all the "cleanup" code necessary to cancel the current operation.

Now you will place a *BrowseBox* Control template that displays all the rows from the *Phones* table that are related to the current *Customer* row.

Place the Control Template

1. If not already opened, select **View ► Control Templates** from the IDE Menu.
2. In the *Control Templates* dialog, highlight the *BrowseBox* Control template, then DRAG to the window area.

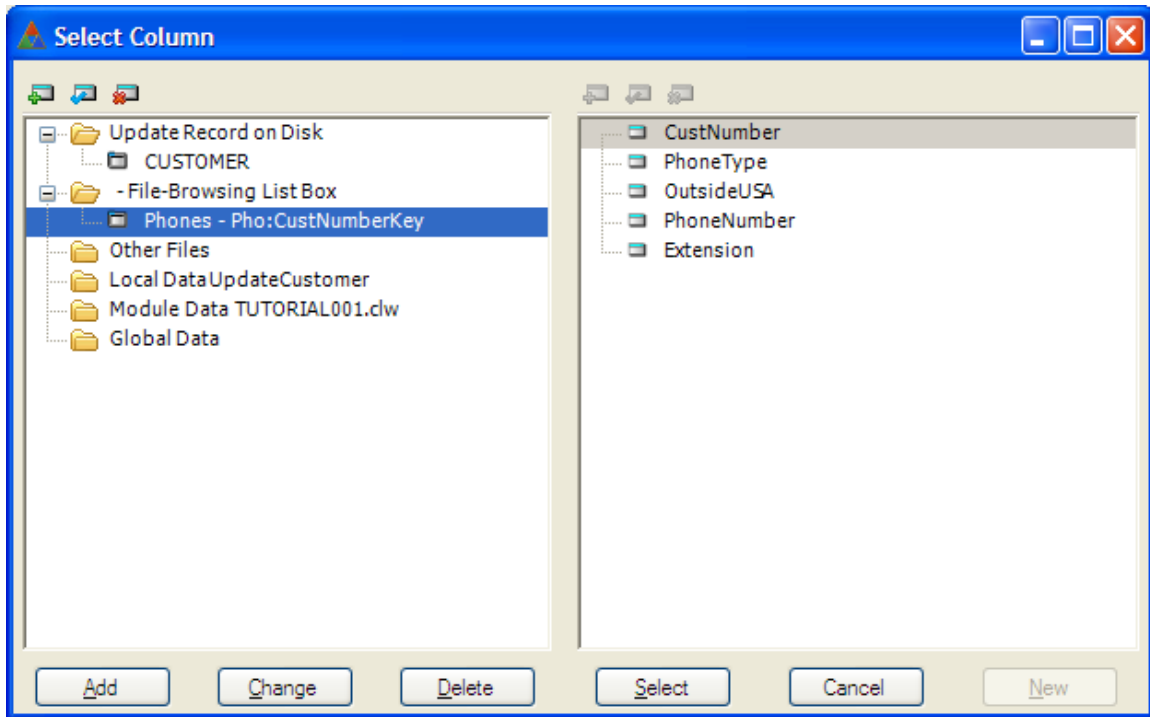
The cursor changes to a DROP cursor.

3. CLICK just below the City entry box to place the control.

The *Select Column* dialog appears, ready for you to choose the table this BrowseBox will display.

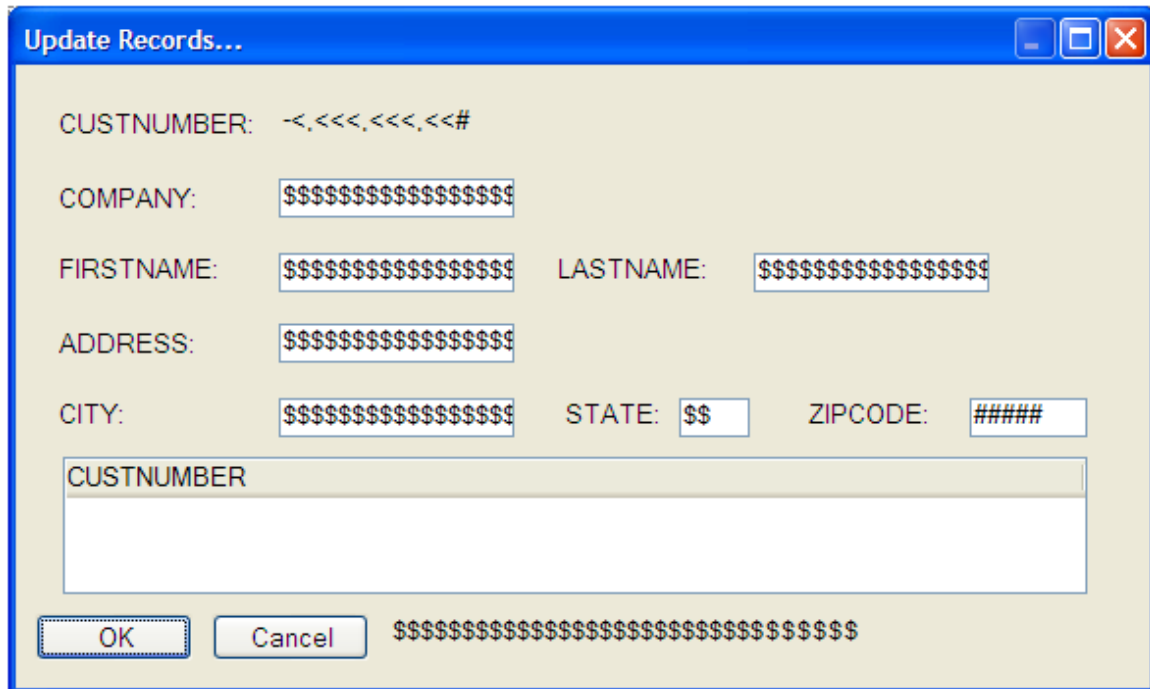
4. Select the <ToDo> item below the *File-Browsing List Box* and press the **Add** button.

5. Highlight the *Phones* table in the *Select a Table* dialog, then press the **Select** button.
6. Highlight the *Phones* table in the *Select Column* dialog, then press the **Change** button.
7. Highlight *Pho:CustNumberKey* in the *Change Access Key* dialog, then press the **Select** button.



Place the Phones table columns in the List Box Formatter

1. Highlight *CustNumber* in the *Select Columns* list, then press the **Select** button.
2. Resize the Browse Box to fit between the entry column and OK button as shown here:



Update Records...

CUSTNUMBER: -<,<<<,<<<,<<#

COMPANY: \$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$

FIRSTNAME: \$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$ LASTNAME: \$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$

ADDRESS: \$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$

CITY: \$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$ STATE: \$\$ ZIPCODE: #####

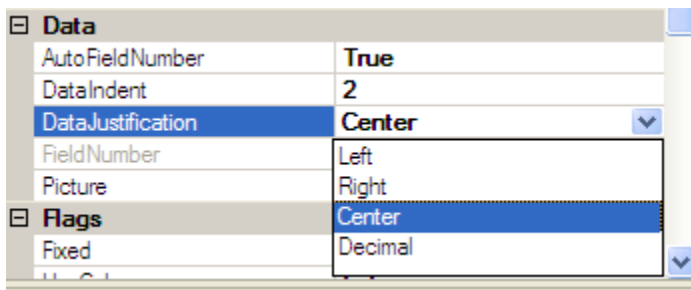
CUSTNUMBER

OK Cancel \$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$

3. RIGHT-CLICK on the Browse Box, and select **List Box Format** from the popup menu.

The *List Box Formatter* now appears, ready for you to choose the rest of the columns to display.

4. Select *Center* from the Data property list's **DataJustification** drop list:






Data	
AutoFieldNumber	True
DataIndent	2
DataJustification	Center
FieldNumber	Left
Picture	Right
Flags	
Fixed	Center
	Decimal

This changes the data justification from the default value (which is Right justification for numeric values).

5. Set the **Right Border** and **Resizable** properties to *TRUE*.

This adds a right border to the column and allows the user to resize the column width at runtime.

6. Press the **Add Field**  button.
7. Highlight *PhoneNumber* in the *Columns* list, then press the **Select** button.
8. Set the **Right Border** and **Resizable** properties to *TRUE*. Set the **Width** property in the **Header** property list to 52.

9. Press the **Add Field**  button.
10. Highlight *Extension* in the *Columns* list, then press the **Select** button.
11. Set the **Right Border** and **Resizable** properties to *TRUE*. Set the **Width** property in the **Header** property list to 56.
12. Press the **Add Field**  button.
13. Highlight *PhoneType* in the *Columns* list, then press the **Select** button.
14. Set the **Width** property in the **Header** property list to 56.

You may also want to give the left align columns an offset of 2, so the data is not touching the left border.

15. Press the **OK** button to close the List Box Formatter.

Set up the control template's row range limits

1. RIGHT-CLICK the list box you just placed, and select **Actions** from the popup menu.
2. Press the **Browse Box Behavior** button.
3. Press the ellipsis (...) button next to the **Range Limit Field**. The *PHO:CustNumber* column is auto-selected, as it is the only component in the list.
4. Choose *File Relationship* from the **Range Limit Type** drop list.
5. Press the ellipsis (...) button next to the **Related file** column then select *Customer* from the *Select Table* dialog.

This identifies the Customer table as the related table. These steps limit the rows displayed in the list box to only those rows related to the currently displayed Customer table row.

Default Behavior	Conditional Behavior	Hot Fields
<p>Table Schematics Description</p> <div></div>		
<input type="checkbox"/> Quick-Scan Records (buffered reads)		
<p>Loading Method</p>	<p>Page</p>	
<input checked="" type="checkbox"/> Accept browse control from ToolBar		
<p>LIST Line Height:</p>	<p>0</p>	
<p>Locator Behavior</p>		
<p>Record Filter:</p>	<p></p>	
<p>Range Limit Field:</p>	<p>Pho:CustNumber</p>	
<p>Range Limit Type:</p>	<p>File Relationship</p>	
<p>Range limiting file</p>		
<p>Related file:</p>	<p>CUSTOMER</p>	
<p>Additional Sort Fields</p>		
<p>Reset Fields</p>	<p>Scroll Bar Behavior</p>	

- Press **OK** to close the *Browse Box Behavior* dialog. Press **OK** again to close the *Prompts* dialog.

Adding the BrowseUpdateButtons Control Template

Next we'll add the standard Insert, Change and Delete buttons for the list box control.

Place another type of Control Template

1. First, resize your window again to make it a little taller, and make some room between the list box and buttons at the bottom:

CUSTNUMBER	Phone Number	Extens	Phone Type

\$

2. In the **Control Templates** Pad, highlight the *BrowseUpdateButtons* Control template, then DRAG to the window area.

The cursor changes to a DROP cursor.

3. CLICK below the left corner of the list box.


The **Insert**, **Change**, and **Delete** buttons appear. Remember, these are the buttons that will allow the toolbar buttons to function, so they must be present in the window design. They do not have to be visible to the end-user, so you can hide them if you choose. However, since this *BrowseBox* is placed on an update Form procedure, for this application we'll leave this set of *BrowseUpdateButtons* visible. This will allow the user to use either set of buttons. The toolbar update buttons will only function for this list when the list box has focus—not when the user is inputting data into any other control—so keeping these buttons visible will ensure that the user can easily maintain the Phones table rows.

You may wish to center these buttons under the list box.

Specify Edit in Place for Phones Update

1. RIGHT-CLICK on the **Delete** button and choose **Actions** from the popup menu.
2. Check the **Use Edit in place** box.

Setting the Actions for one button sets them for all three buttons in the set, because they all belong to the same Control Template. Since the Phones table is a small table with just a couple of columns, there's no need for a separate Update Procedure.

3. Press the **OK** button.
4. Press the **Save and Close** button to close the Window Designer.
5. Press the **Save and Close** button to close the Window Designer Editor.
6. Choose **File ► Save**, or press the **Save** button  on the tool bar.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- ✓ You created a new Form Procedure—without using a wizard.
- ✓ You learned just how quickly you can populate data entry controls onto a Form by using the Columnbox toolbox.
- ✓ You learned to use the Window Designer's tools to move and align controls.
- ✓ You used the List Box Formatter again and created a range limited list box.
- ✓ You learned how to implement edit-in-place for simple updates which don't require a Form procedure.

Now you've created all the procedures necessary to maintain both the *Customer* and *Phones* tables. Next, we'll create the procedures that will maintain the *Products* data table.

9 - Copying Procedures

The Products Table Procedures

Now that we've created the Customer Browse procedure, we can reuse much of that work for the next procedure by copying the procedure, then changing its columns. In this chapter, you'll copy the *BrowseCustomers* procedure to create the *BrowseProducts* procedure.

You will also use "Embed points" to write "embedded source code" to call the *BrowseProducts* procedure from your application's menu and toolbar. This will introduce you to the numerous points at which you can add a few (or many) lines of your own source code to add functionality to any procedure.

Starting Point:

The **LCLESSON.APP** file should be open, and the **Application Editor (tree)**.

Copy the Procedures

As you recall, when you created your **Browse ► Products** menu item, and the toolbar button labeled "Products," you didn't specify a procedure to call when the end user executed them. We'll start by creating the procedure to call.

1. Highlight the *BrowseCustomers* procedure in the *Application Tree* dialog.

This is the procedure you will copy.

2. Choose **Application ► Copy Procedure** from the main IDE menu.

The *New Procedure* dialog appears.

3. Type *BrowseProducts* in the entry control then press the **OK** button.

Because the *UpdateCustomer* procedure is nested under the *BrowseCustomers* procedure (the one you are copying), the *Procedure name clash* dialog appears. This offers you options on how to handle the clashing procedures.

4. Press the **Prompt** button.

By pressing the **Prompt** button, you tell the Application Generator to let you have the opportunity to rename all the clashing procedures, or not.

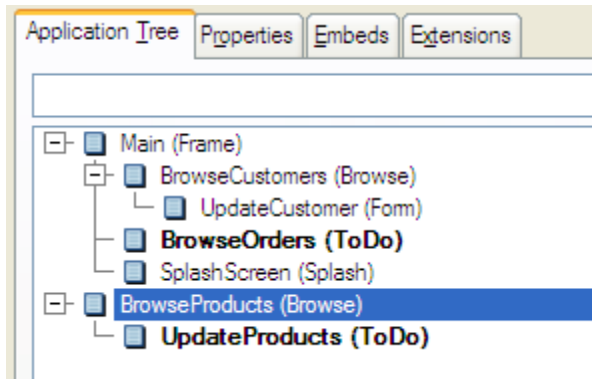
Another warning message box appears to inform you of a specific duplicate procedure name.

5. Press the **Rename** button.

The *Alternative Procedure* dialog appears.

6. Type *UpdateProduct* in the entry control, then press the **OK** button.

The *BrowseProducts* and *UpdateProduct* procedures appear in the Application Tree. They look "disconnected" from the other procedures because no other procedure calls them (yet). We'll do that next.



Working with Embed Points

The Clarion templates allow you to add your own customized code to many predefined points inside the standard code that the templates generate. It's a very efficient way to achieve maximum code reusability and flexibility. The point at which your code is inserted is called an *Embed Point*. Embed points are available at all the standard events for the window and each control, and many other logical positions within the generated code.

In this example, you add embedded source code—using a Code template that will write the actual source for you—at the points where the end user chooses the **Browse ► Products** menu item, and at the point where the end user presses the Products button on the application's toolbar.

Name the procedure to call to Browse the Products

1. RIGHT-CLICK on the *Main* procedure in the Application Tree.

There are several ways to access the embedded source code points within a procedure. Two of them appear on the popup menu that you now see.

The first is the **Embeds** selection, which calls the *Embedded Source* dialog to show a list of all the embed points within the procedure.

The second is the **Source** selection, which actually generates source code for the procedure and calls the "Embeditor" (the Text Editor in embed point edit mode) to allow you to directly edit all the embed points within the context of generated source. The generated source code is "grayed out" to indicate that you cannot edit it, and every possible embed point in the procedure is identified by comments, following which you may type your code.

There are advantages to each method of working in embed points, so we'll cover both methods during the course of this lesson. First, we'll use the *Embedded Source* dialog.

2. Choose **Embeds** from the popup menu.

The *Embeds Tree* dialog appears, allowing access to all the embed points in the procedure. You can also get here from the **Embeds** button on the *Procedure Properties* window, but the popup menu is quicker. This list is either sorted alphabetically or in the order in which they appear in the generated source, depending on whether you have the **Sort Embeds Alphabetically** box checked in **Setup ► Application Options**.

3. Press the **Contract All** button on the right of the window.

This will make it easier to locate the specific embed point you need.

4. Locate the *Control Events* folder, then CLICK on its + sign to expand its contents. You can easily locate this by entering *BrowseProducts* in the entry locator above the embed tree. It works, even when the tree is totally collapsed.

The menu selection is a control, just as an entry box on the window is.

You'll notice that there are some up and down buttons and a spin box at the right side of the window that allow you to select a **Priority**—these are important. The templates generate much of the code they write for you into these same embed points. Sometimes, the code you want to write should execute *before* any template-generated code, and sometimes it should execute *after*, and sometimes it should execute somewhere between various bits of generated code. The exact placement of your code within the embed point is determined by the **Priority** number. This provides you with as much flexibility in placing your embed code as possible. The **Priority** numbers themselves do not matter, but the logical position within the generated code does, and that's why this dialog also shows comments which identify the embed priorities. Don't worry, there's more coming on this issue later that'll help make it clearer.

5. Locate the *?BrowseProducts* folder, then CLICK on it to expand it.
6. Highlight *Accepted* then press the **Insert** button.



The "Accepted" event for this menu selection marks the point in the generated code that executes when the user chooses the menu command.

The Select embed type dialog appears to list all your options for embedding code. You may simply Call a Procedure, write your own Clarion language Source in the Text Editor, or use a Code template to write the source code for you. This is one advantage to editing embed points from within the Embedded Source dialog—you can use Code templates to write the code for you instead of writing it yourself.

7. Select the *InitiateThread* Code template, then press the **Select** button.

A Code template usually provides just a few prompts and instructions on its use. It gathers the information it needs from you to write its executable code, which it then inserts into the standard generated code produced by the Procedure template directly into this embed point. This Code template is designed to start a new execution thread by calling the procedure you name using the START procedure.


8. Choose *BrowseProducts* from the **Procedure Name** dropdown list.

This names the procedure to call when the user chooses the menu item. This is the name of the procedure you previously copied.


9. Press the **OK** button.

Name the procedure to call for the Products toolbar button

At this point, you could do the same thing to call the *BrowseProducts* procedure from the Product button. However, there's an easier way to write this code again—just Copy and Paste it from one embed point to another!

1. CLICK on the **Copy** button  located on the toolbar.

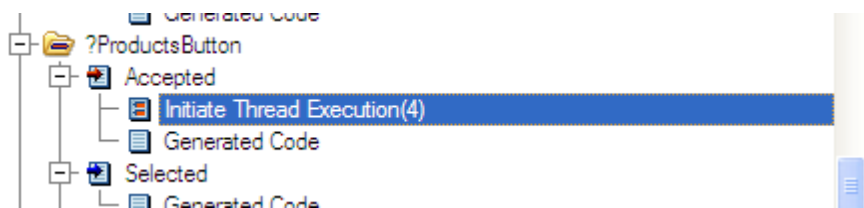
The Code template you just added should still be highlighted, so this will copy it to the Windows clipboard.

2. Locate the *?ProductsButton* folder (in the same Control Event Handling folder you are already in), then CLICK on it to expand it.
3. Highlight *Accepted* then press the **Paste** button  located on the toolbar.

The **Procedure name clash** dialog appears again to warn you that you've already called this procedure once.

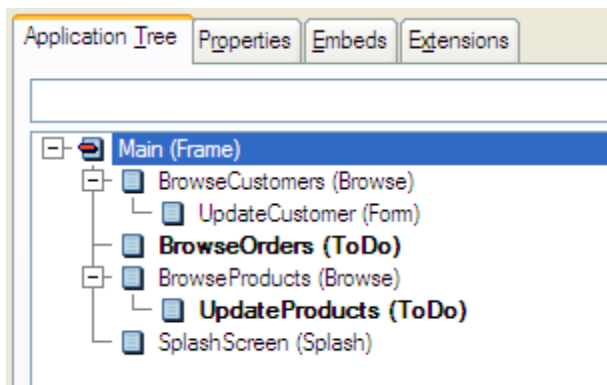
4. Press the **Same** button.

Now this embed point will generate the same code as the previous one.



5. Press the **Save and Close** button.



The *BrowseProducts* procedure now "connects" to the *Main* procedure. Now you can customize the copied procedures for the Products table.



Modify the Browse

Change the table for the browse list control

1. RIGHT-CLICK the *BrowseProducts* procedure and choose Window from the popup menu.




2. RIGHT-CLICK on the list box control and choose **List Box Format...** from the popup menu.
3. In the *List Box Formatter* press the **Delete** button  repeatedly until all the columns are removed.
4. Press the **Add Field**  button.
5. Highlight the *Customer* table in the *Tables* list, then press the **Delete** button.
6. Highlight the *<ToDo>* which replaces the *Customer* table, then press the **Add** button.
7. Highlight the *Products* table then press the **Select** button.

The *Select Column* dialog now lists the correct table and columns for this procedure.

8. Press the **Change** button, then select *ProdDescKey* from the *Change Access Key* dialog.

The *Select Column* dialog now lists the correct table and columns.

Re-populate the columns

1. Highlight *ProdNumber* in the *Columns* list, then press the **Select** button.
2. Set the **RightBorder** and **Resizable** properties to *TRUE*, and verify that the **Width** property is set to 24.
3. Press the **Add Field**  button.
4. Highlight *ProdDesc* in the *Columns* list, then press the **Select** button.
5. Set the **RightBorder** and **Resizable** properties to *TRUE*, and verify that the **Width** property is set to 120..
6. Press the **Add Field**  button.
7. Highlight *ProdAmount* in the *Columns* list, then press the **Select** button.
8. Set the **RightBorder** and **Resizable** properties to *TRUE*, and verify that the **Width** property is set to 32..
9. Press the **Add Field**  button.
10. Highlight *TaxRate* in the *Columns* list, then press the **Select** button.

Notice the default Justification is *Decimal*, and the Data group's Indent spin box is still set to twelve (12) from the previous column.

11. Press the **OK** button to close the *List Box Formatter*.

Don't worry about the buttons on top of the list box. Remember, these are just the "hidden" buttons that the toolbar update buttons call.

Change the name of the window

1. CLICK on the sample window caption bar.
2. Type *Browse Products* in the **Title** property of the *Property Pad*, then press TAB.

Remove all tab controls

1. CLICK immediately to the right of the *KeyZipCode* tab to select the entire property sheet.

To be sure that you have CLICKed in the right place, look at the *Property Pad* and make sure that its Use entry displays *?Sheet1*. If it does not, try again until it does.


2. Press DELETE on your keyboard.

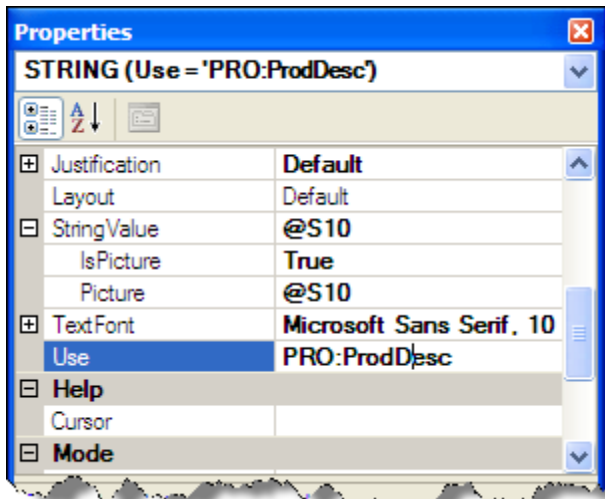
All of the tabs disappear.

Add an incremental locator

If the list of Products to display is very long, the user can do a lot of scrolling before finding the specific Product they want. By default, all BrowseBox Control Templates have a "Step" row locator that allows the user to press the first letter of the value in the sort key column to get to the first row that begins with that letter.

Sometimes with large databases however, the user needs to enter the first several letters to get close to the row they want. An Incremental locator provides that functionality by specifying an string control for the user to see the information they type. As they type, the list scrolls to the first row matching the data the user entered thus far. This works best with STRING keys.

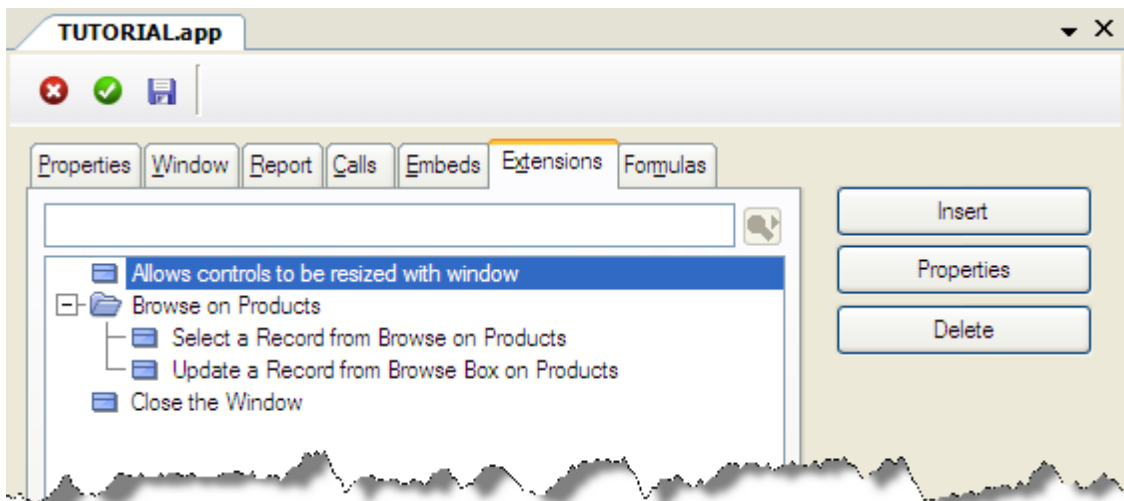
1. If the **Control Toolbox** is not opened, open it now by selecting **View ► Toolbox** from the IDE Menu.
2. CLICK on the **Str STRING** button on the control toolbox.
3. The mouse cursor changes as you move it across the window. Drop the string above the list box on the left side of the window. You should now see *String 1*.
4. RIGHT-CLICK on the string control and choose **Properties**.
5. In the Properties Pad, set the **IsPicture** property to *TRUE*.
6. In the **Use** property, type *PRO:ProdDesc*, or press the ellipsis button to select it from the column list.
7. Press the **Save and Close** button to exit the Window Designer.
8. Press the **Save and Close** button to exit the Window Designer Editor.
9. Choose **File ► Save**, or press the **Save** button  on the tool bar.



Clean up the alternate sort orders

1. RIGHT-CLICK the *BrowseProducts* procedure and choose **Extensions** from the popup menu.

The *Extension and Control Templates* dialog appears. This dialog lists all the Control templates in the procedure and their Actions prompts.



This dialog also allows you to add and maintain Extension templates to the procedure. Extension templates are very similar to Control templates, in that they add specific functionality to the procedure, but an Extension template's functionality is not directly associated with any control(s) on the window. In other words, Extension templates add "behind the scenes" functionality to a procedure that don't directly affect the user interface.

A very good example of Extension templates comes in Clarion's ASP (Active Server Pages) product. Clarion ASP contains (among its other development tools) a set of Extension templates, which automatically "translate" a Clarion procedure into dynamic ASP pages.

2. Highlight *Browse on Products* then press the **Properties** button.

3. In the *Browse on Products* dialog, press the **Locator Behavior** button.
4. Select *Incremental* from the **Locator** drop list then press the **OK** button.

This completes the requirements for the Incremental Locator. The key column of the sort order (in this case *PRO:ProdDesc*) is the default locator control.

5. Select the **Conditional Behavior** tab.
6. Press the **Delete** button twice.



This removes the two conditional expressions we entered for the *BrowseCustomers* procedure.

7. Press the **OK** button.
8. Press the **Save and Close** button to return to the Application Tree.

Creating the Form Procedure

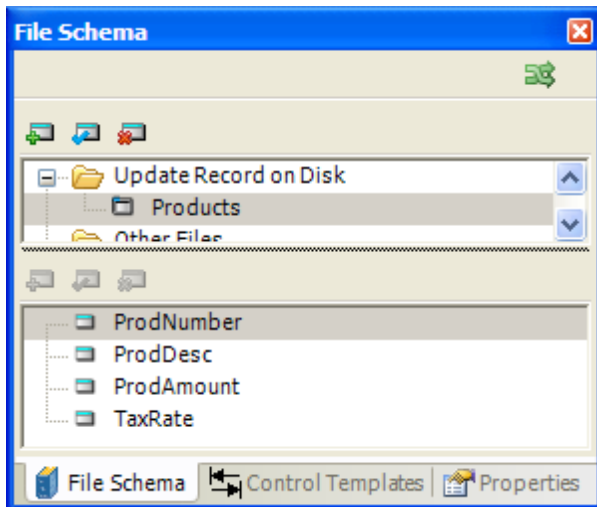
When you renamed the reference to the *UpdateCustomer* procedure while copying *BrowseCustomer* to *BrowseProducts*, it made the *UpdateProduct* procedure a "ToDo" procedure. Therefore, we need to create a form to update the Products table.

Select the procedure type for UpdateProduct

1. Highlight *UpdateProduct*, then press the **Properties**  button.
2. Select the **Defaults** tab, choose *FORM (Add/Edit/Delete)* and then press the **Select** button.
3. Open the **Data / Tables Pad** if not already opened.
4. In the **Data / Tables Pad**, highlight the *<ToDo>* table, then press the **Add**  button.
5. Highlight the *Products* table then press the **Select** button.
6. Press the **Window** tab and press the **Designer** button to enter the *Window Designer* to design your form.

Populate the columns

1. We will use the **Data / Tables Pad** in this section to populate our Form.
2. In the **Data / Tables Pad**, select the Products table as shown below:



3. DRAG *ProdNumber* to the upper left of the window, and DROP.

This automatically places both the prompt and entry box for the column near the top left corner of the window.

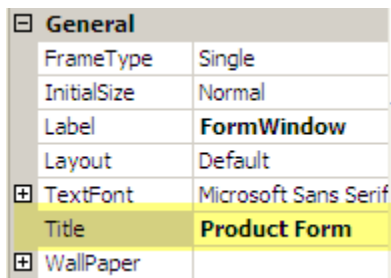
4. DRAG *ProdDesc* just below the *ProdNumber* prompt, and DROP.

This automatically places both the prompt and entry box for the column immediately below the last column that was placed.


5. DRAG *ProdAmount* just below the *ProdDesc* prompt, and DROP.
6. DRAG *TaxRate* just below the *ProdAmount* prompt, and DROP.

Change the form window caption

1. CLICK on the caption bar of the sample window.
2. Type *Product Form* in the **Title** property of the *Property Pad* then press TAB.



Exit the Window Designer, and save your work

1. Press the **Save and Close** button on the toolbar to close the Window Designer (save your changes).
2. Press the **Save and Close** button in the *Window Designer Editor* to close it.
3. Choose **File ► Save**, or press the **Save** button  on the toolbar to save your work.

The Products table update form window is completed.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- ✓ You copied an existing procedure, renaming the subsequent procedures it called.
- ✓ You used a Code Template in the Embedded Source dialog to call the new procedure from the main menu.
- ✓ You modified the copied procedure to display from another table.
- ✓ You added an Incremental Locator to the Browse procedure.
- ✓ You created an entire Form procedure very quickly by just using the Populate Column toolbox.

Now that you're thoroughly familiar with Procedure Templates, we'll go on to use some Control and Extension Templates.

10 - Control and Extension Templates

For the `BrowseOrders` procedure, you'll create a window with two synchronized scrolling list boxes. One will display the contents of the `Orders` table, and the other will display the related rows in the `Detail` table. You'll use a generic `Window` procedure, and populate it using `Control` templates. The only reason for doing it this way instead of starting with a `Browse Procedure Template` is to demonstrate another way of building a procedure—using `Control` templates placed in a generic `Window` (the same way the `Browse Procedure` template itself was created).

`Control` templates generate all the source code for creating and maintaining a single control or related group of controls. In this case, placing a `BrowseBox` `Control` template allows the `Application Generator` to produce the code that opens the tables and puts the necessary data into the structures which hold the data for display in the list box.

Starting Point:

The `LCLESSON.APP` file and `Application Editor (tree)` should be open.

Creating the Procedure

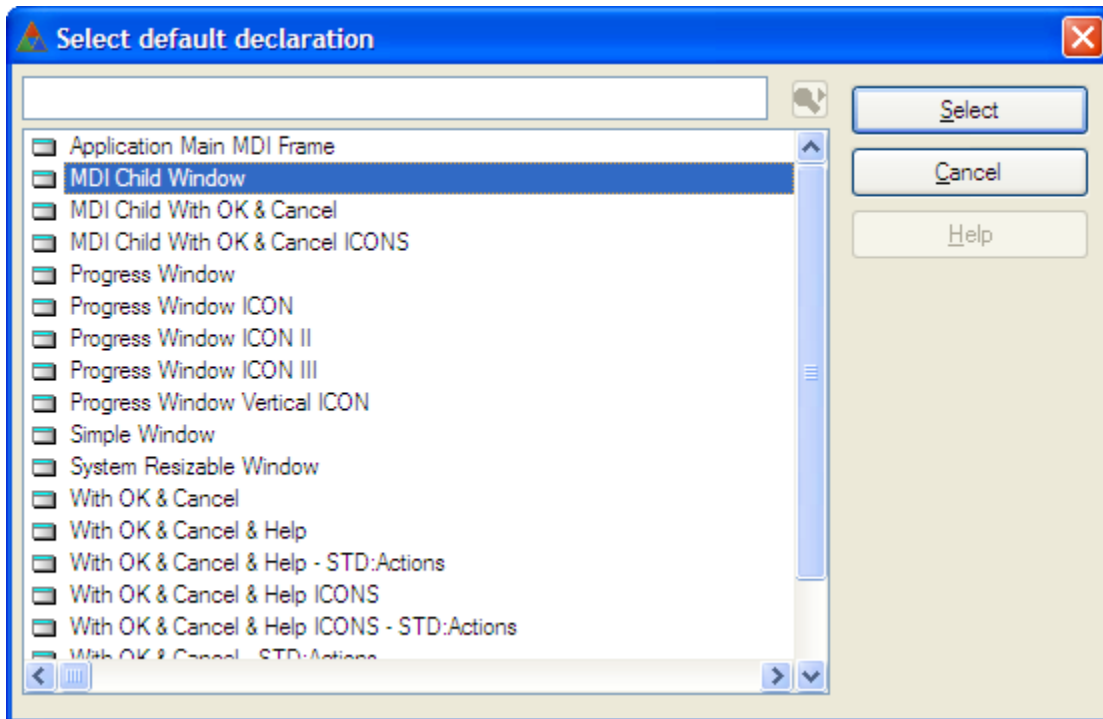
Select the procedure type

1. Highlight `BrowseOrders` then press the **Properties** button.
2. In the *Select Procedure Type* dialog, select the **Template** tab, choose *Window – Generic Window Handler*, then press the **Select** button.

Edit the Window

1. In the *Procedure Properties* dialog, press the **Window** tab.
2. In the *Window Designer Editor*, press the **Designer** button.

The *Select default declaration* dialog appears. The generic window procedure is like a blank slate in which you define your own window. Since the procedure has no predefined window, you choose the type of window for your starting point. In this case, you need an *MDI child window*.



3. Highlight *MDI Child Window*, then press the **Select** button.

The Window Designer appears.


4. Resize the window, making it more than twice its original size (in both directions).
5. CLICK in the window's title bar, then open the **Properties** Pad if not already opened.
6. In the **Properties** Pad, type *Orders* in the **Title** property .
7. Verify that the **FrameType** property is set to *Resizable*.
8. Set the **MaximizeBox** property to *TRUE*.

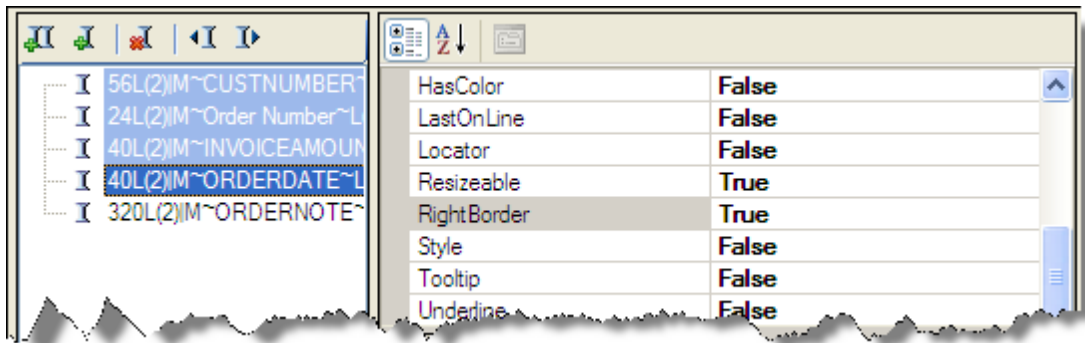
Placing the BrowseBox Control Template

1. Verify that the **Control Templates** Pad is now opened.
2. Highlight the *BrowseBox* Control template, then DRAG near the upper left corner of the sample window and DROP.

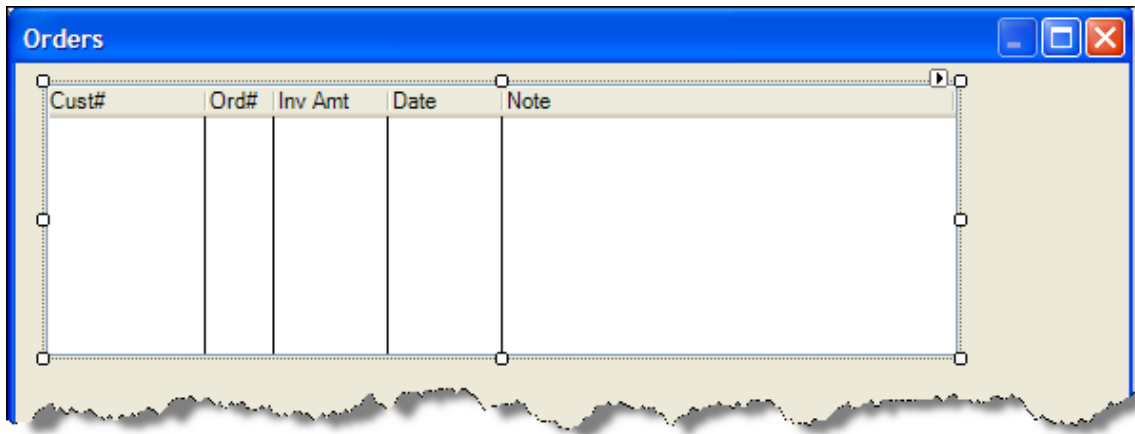
Place the Orders table columns in the List Box Formatter

1. Highlight the *<ToDo>* item below the *File-Browsing List Box* and press the **Add** button.
2. Highlight the *Orders* table in the *Select* dialog, then press the **Select** button.
3. Highlight the *Orders* table, press the **Change** button and select *KeyOrderNumber* from the *Change Access Key* dialog.
4. Highlight *CustNumber* in the *Columns* list, then press the **Select** button.

5. RIGHT-CLICK on the Browse box just populated, and select **List Box format...** from the popup menu.
6. Press the **Add Field**  button.
7. Highlight *OrderNumber* in the *Columns* list, then press the **Select** button.
8. Press the **Add Field** button.
9. Highlight *InvoiceAmount* in the *Columns* list, then press the **Select** button.
10. Press the **Add Field** button.
11. Highlight *OrderDate* in the *Columns* list, then press the **Select** button.
12. Press the **Add Field** button.
13. Highlight *OrderNote* in the *Columns* list, then press the **Select** button.
14. Resize the columns if needed by adjusting the column's **Width** property.
15. CTRL+CLICK to multi select the first four columns, and set the **Resizable** and **RightBorder** properties to *TRUE*.



16. Finally, examine the **Header** text property of each column, and modify as follows:
 Cust#
 Ord#
 Inv Amt
 Date
 Note
17. Press the **OK** button to close the List Box Formatter.
18. Resize the browse list box control to make it wider by dragging the handle in the middle of the right side (almost as wide as the window).



Format the list box appearance

1. RIGHT-CLICK the list box, and select **Properties** from the popup menu.
2. Set the **Vertical** and **Horizontal** properties to *TRUE*.



This adds vertical and horizontal scroll bars to the list.

3. Press the ellipsis button located just to the right of the **TextFont** property.

Because one column (the description column) is long, you can specify that the list box should use a smaller font, displaying more information without requiring the end user to scroll.

4. Choose a font (your choice), and set the size to 8 points.

In general, you want to stick with the fonts that ship with Windows; otherwise, you can't be sure your end user has the same font on their system. Our example sets the font to Arial, which is a font that ships with Windows.

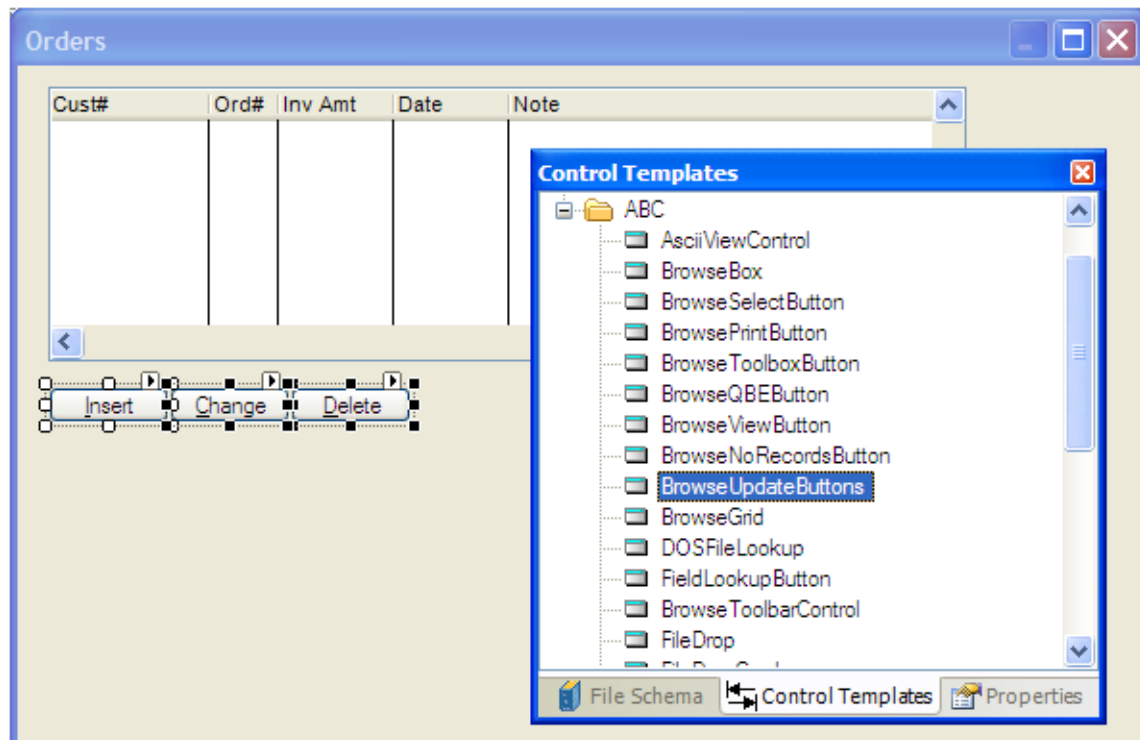
5. Press **OK** to close the *Select Font* dialog.

Adding the Browse Update Buttons Template

Next add the standard **Insert**, **Change** and **Delete** buttons for the top browse list box control. Since there are going to be two list boxes on this window, we'll leave these buttons visible for the user. Later we'll add a form procedure for adding or editing an order.

1. In the *Control Templates* pad, highlight the *BrowseUpdateButtons* control template, then DRAG below the left edge of the list box.

The **Insert**, **Change**, and **Delete** buttons all appear together.



Name the Update Procedure

1. RIGHT-CLICK on the **Delete** button only, then choose **Actions** from the popup menu.
2. Type *UpdateOrder* in the **Update Procedure** box.

This names the procedure, in the same way that you named the Update procedure for the Customer browse in its Procedure Properties dialog.

Naming the Update Procedure for one button in the Control template names it for all three.

3. Press the **OK** button.

Placing the Second Browse List Box

Next, place a list box with the contents of the Detail table. This will update automatically when the end user changes the selection in the top list box.

1. In the **Control Templates** Pad, highlight the *BrowseBox* Control template, then DRAG directly below the **Insert** button you placed before, and DROP.

The second Browse Box control is now populated.

Place the Detail table columns in the List Box Formatter

1. In the **Data / Tables Pad**, highlight the *<ToDo>* item below the second *File-Browsing List Box* and press the **Add** button.
2. Highlight the *Detail* table in the *Select* dialog, then press the **Select** button.
3. Highlight the *Detail* table, and press the **Change** button.

4. Highlight *OrderNumberKey* in the *Change Access Key* dialog, then press the **Select** button.
5. RIGHT-CLICK on the Browse box just populated, and select **List Box format...** from the popup menu.
6. Press the **Add Field** button.
7. Highlight *OrderNumber* in the *Columns* list, then press the **Select** button.
8. Highlight the "dummy" field just above the *OrderNumber* column just added, and press the **Remove** button to remove it.
9. Press the **Add Field** button.
10. Highlight *ProdNumber* in the *Columns* list, then press the **Select** button.
11. Press the **Add Field** button.
12. Highlight *Quantity* in the *Columns* list, then press the **Select** button.
13. Press the **Add Field** button.
14. Highlight *ProdAmount* in the *Columns* list, then press the **Select** button.
15. As you did in the first Browse Box, adjust the column **Width** property as needed, set the **Resizable** and **RightBorder** property to *TRUE* as before, and adjust the Header **Text** as needed.
16. Press the **OK** button to close the List Box Formatter.
17. Resize the browse list box control by dragging the handles, making it an appropriate size for display (but leave some space to its right for a button we're going to place in the bottom right corner of the window).

Set up the Range Limits

1. RIGHT-CLICK on the list box you just placed and select **Actions** from the popup menu.
2. Press the ellipsis (...) button for the **Range Limit Field**.
3. Highlight the *DTL:OrderNumber* column in the Components list, then press the **Select** button. (Note: this column may populate automatically)
4. Choose *File Relationship* from the **Range Limit Type** drop list.
5. Press the ellipsis (...) button in the **Related File** entry.
6. Highlight the *Orders* table in the *Select Table from Procedure Schema* list, then press the **Select** button.

These last four steps limit the range of rows displayed in the second list box to only those Detail rows related to the currently highlighted row in the Orders table's list box.

This tells the second control template to use the table relationship defined in the data dictionary to synchronize the bottom list to the top.

7. Press **OK** to close the Actions(*Prompts*) dialog.

Format the list box appearance

1. Make sure the second Browse Box is selected, and refer to the **Properties Pad**.
2. Set the **Vertical** and **Horizontal** properties to *TRUE*.

This adds horizontal and vertical scroll bars to the list box.

3. Press the ellipsis button to the right of the **TextFont** property.

Although there are no "long" columns in this list box, it will look better if you match the font to the same font used in the top list box.

4. Choose a font (your choice), and set the size to 8 points.
5. Press **OK** to close the *Select Font* dialog.

Adding the Close Button Control Template

Finally, you can add a Close button that closes the window.

1. Open the **Control Templates** Pad, if not already displayed.
2. Select the *CloseButton* Control template, then DRAG to the lower right corner of the window.
3. Press the **Save and Close** button on the Window Designer toolbar to close the Window Designer, and automatically save changes.

Make the window resizable

1. In the *Procedure Properties* dialog, select the **Extensions** tab.
2. In the *Extensions and Control Templates* dialog, press the **Insert** button.
3. Highlight *WindowResize* in the *Select Extension* dialog, then press the **Select** button.

We've used this Extension template several times already, but this time we'll modify its actions instead of simply taking the default behavior.

Specify the resize strategies

1. Check the **Restrict Minimum Window Size** box.

By checking this box and leaving the **Minimum Width** and **Minimum Height** set to zero (0), this template ensures that users cannot make the window any smaller than the designed size of the window.

2. Press the **Override Control Strategies** button.

The *Override Control Strategies* dialog appears. This dialog allows you to specify the resize strategy for individual controls.

3. Press the **Insert** button.
4. Select *?Insert* from the **Window Control** drop list.
5. Choose the *Fix Bottom* radio button in the **Vertical Positional Strategy** set of options.

This sets the resize strategy for the **Insert** button to keep it a fixed distance from the bottom of the window. Now we'll do the same for the other two update buttons and the Details list box.

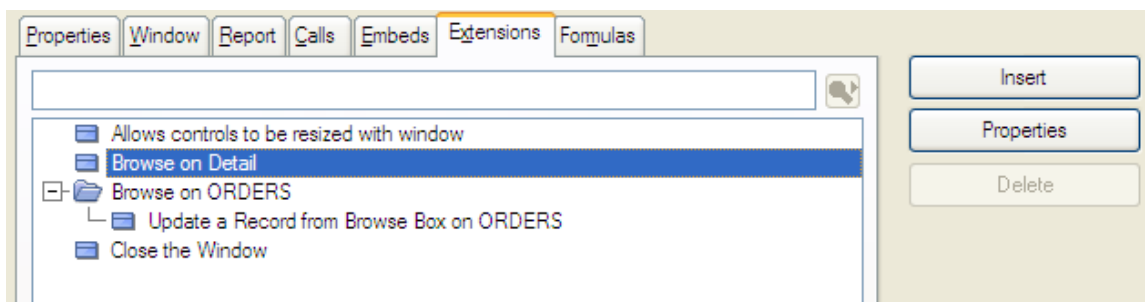
6. Press the **OK** button.
7. Press the **Insert** button, then select *?Change* from the **Window Control** drop-down list.
8. Choose the *Fix Bottom* radio button in the **Vertical Positional Strategy** set of options, then press the **OK** button.
9. Press the **Insert** button, then select *?Delete* from the **Window Control** drop-down list.
10. Choose the *Fix Bottom* radio button in the **Vertical Positional Strategy** set of options, then press the **OK** button.
11. Press the **Insert** button, then select *?List:2* from the **Window Control** drop list.
12. Choose the *Fix Bottom* radio button in the **Vertical Positional Strategy** set of options, then press the **OK** button.
13. Press the **Insert** button, then select *?List* from the **Window Control** drop list.
14. Choose the *Constant Bottom Border* radio button in the **Vertical Resize Strategy** set of options.

This sets the resize strategy for the Orders List box to keep the bottom border of the list a fixed distance from the bottom of the window. Therefore, the list will stretch as needed to fill up the space as the window becomes larger.

15. Press the **OK** button to close the *Override* dialog, and then press the next **OK** button to close the *Window Resize* dialog.

Set up a Reset Column

1. In the Extensions and Control Templates dialog, highlight *Browse on Detail* and press the **Properties** button.





2. Press the **Reset Fields** button, then press the **Insert** button.
3. Type `ORD:InvoiceAmount` in the **Reset Column** entry (or press the ellipsis and select it from the select column list), and then press the **OK** button.

This specifies that the Detail table's list box should reset itself whenever the value in the `ORD:InvoiceAmount` column changes. This ensures that any changes you make to an existing order are reflected in this dialog when you return from changing the order.

4. Press the **OK** button to close the *Reset Fields* dialog, and then press the **OK** button to return to the Procedure Properties dialog.

Close the Procedure Properties dialog and Save the Application

1. Press the **Save and Close** button  in the *Procedure Properties* dialog to close it.
2. Choose **File ► Save**, or press the Save button  on the tool bar to save your work.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- ✓ You created a new browse procedure, but did it using the *BrowseBox* and *BrowseUpdateButtons* Control Templates instead of the *Browse Procedure* Template.
- ✓ You created a second, range-limited list box to display related child rows.
- ✓ You used the *WindowResize* Extension Template and specified individual control resize strategies.
- ✓ You set a Reset Column on the Detail table's Browse list so its display is always kept current.

Next we'll create the UpdateOrder Form procedure to create and maintain the Orders and Detail table rows.

11 - Advanced Topics

Set Up the UpdateOrder Form

For the Order Update form, we'll place the columns from the Order table on an update form, perform an automatic lookup to the Customer table, add a *BrowseBox* Control template to show and edit the related detail items, calculate each line item detail, then calculate the order total.

Starting Point:

The **LCLESSON.APP** file and **Application Editor (Tree)** should be open.

Create the Orders table's data entry Form

1. Highlight *UpdateOrder* in the *Application Tree* dialog, then press the **Properties** button.
2. Select the **Defaults** tab, choose *FORM(Add/Change/Delete)*, then press the **Select** button.
3. Select the **Window** tab, and then press the **Designer** button to open the Window Designer.
4. Resize the window taller by dragging its bottom middle handle. Move the three controls to the bottom of the window after resizing.

Place the entry controls for the Orders table

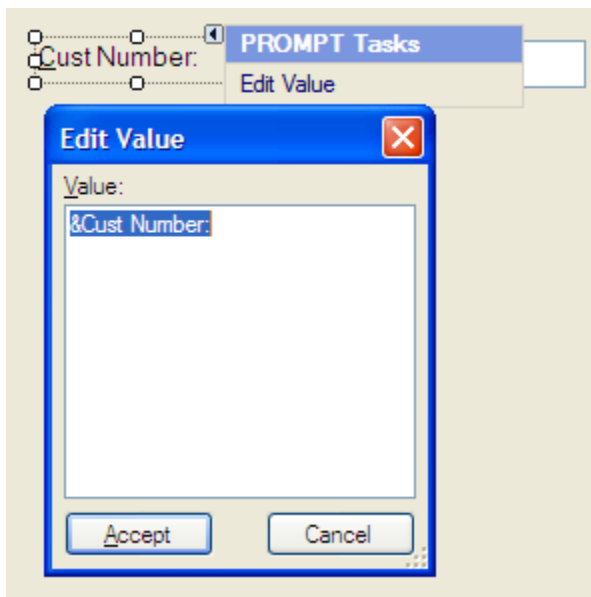
Instead of using the *Populate Column* IDE menu option, we'll use the *Data / Tables Pad* dialog to populate multiple controls.

1. In the *Data / Tables Pad* dialog, highlight the *<ToDo>* item under the *Update Record on Disk*, then press the **Add** toolbar button.
2. Highlight the *Orders* table from the *Select* list, then press the **Select** button.
3. Highlight the *Orders* table, and in the bottom half of the *Data / Tables Pad*, highlight *OrderDate*, then DRAG near the top left of the window, and DROP.
4. Back in the *Data / Tables Pad*, highlight *OrderNote*, then DRAG just to the right of the entry box placed for the date.
5. Switch to the **Toolbox** Pad, and if not visible, open it by selecting **View ► Toolbox** from the IDE Menu.
6. Choose (DRAG) the ENTRY control from the **Toolbox** Pad and drop it under the *OrderDate* entry control.
7. With the Entry control still selected, press the F4 key to switch focus to the **Properties** Pad.
8. Enter *ORD:CustNumber* in the **Use** property.

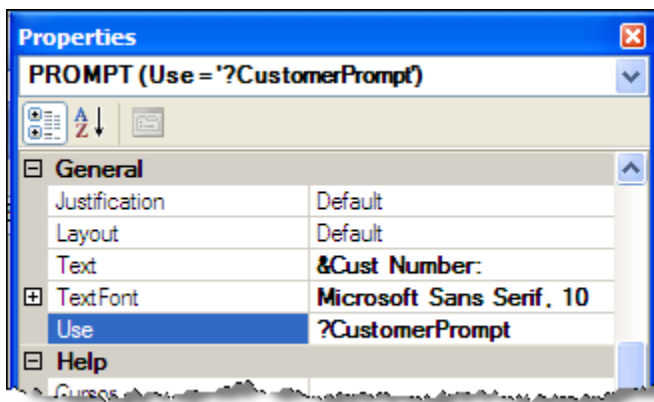
Since the *ORD:CustNumber* column is setup as a string in the dictionary, we need to add an entry control.

9. Switch back to the **Toolbox** Pad, and choose (DRAG) the PROMPT control from the **Toolbox** Pad and drop it to the left of the *CustNumber* entry control.

10. Note the **Edit Value** Smart Tag shown just after populating. Click on it, and enter *&Cust Number:* in the Text control:



11. With the PROMPT control still selected, press the F4 key to switch focus to the **Properties** Pad.
12. Change the **Use** property to *?CustomerPrompt*.



Add a lookup procedure call into the customer list

1. RIGHT-CLICK on the *ORD:CustNumber* entry control and select **Actions** from the popup menu.

The standard actions for any entry control allow you to perform data entry validation against a row in another table, either when the control is Selected (just before the user can enter data) or when the control is Accepted (right after the user has entered data).

2. In the *When the Control is Accepted* group box, press the ellipsis button (...) located to the right of the **Lookup Key** entry box.

3. Highlight the *Orders* table in the *Select Key* dialog, then press the **Add** button.
4. Highlight the *Customer* table in the *Select* list, and press the **Select** button.

These last two steps add the *Customer* table to the procedure's Table Schematic as an automatic lookup from the *Orders* table. This will automatically lookup the related *Customer* table row for you, and the lookup is based on the table relationship set up in the data dictionary.

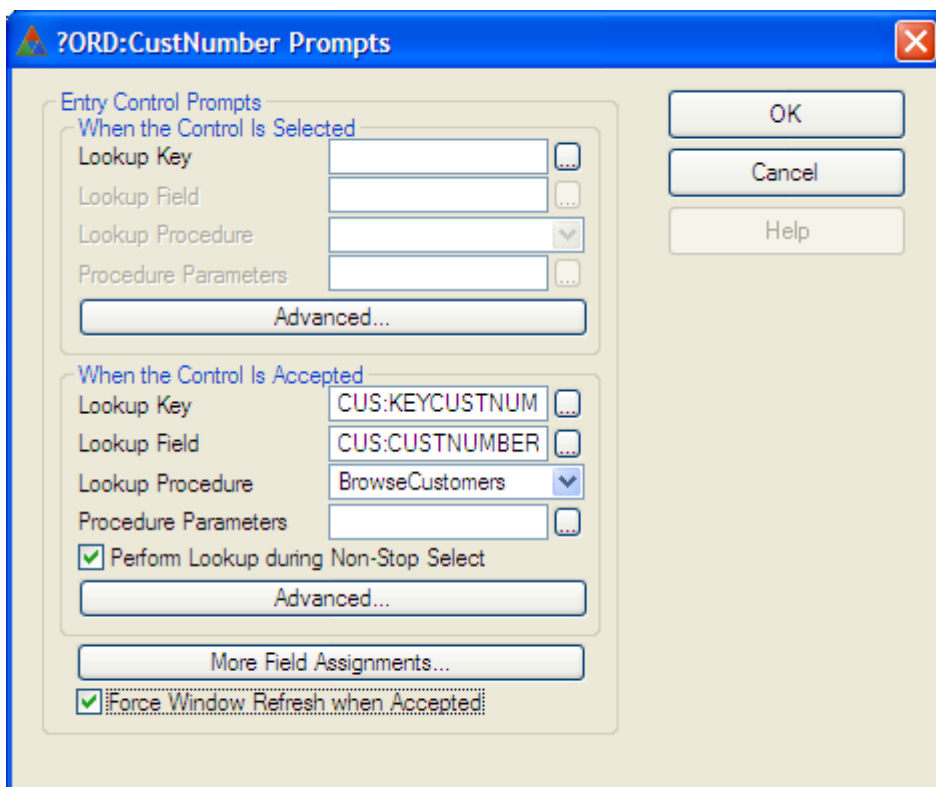
5. Highlight *CUS:KeyCustNumber* in the *Select Key* dialog, then press the **Select** button.

This makes *CUS:KeyCustNumber* the key that will be used to attempt to get a matching valid row from the *Customer* table for the value the user enters into this control.

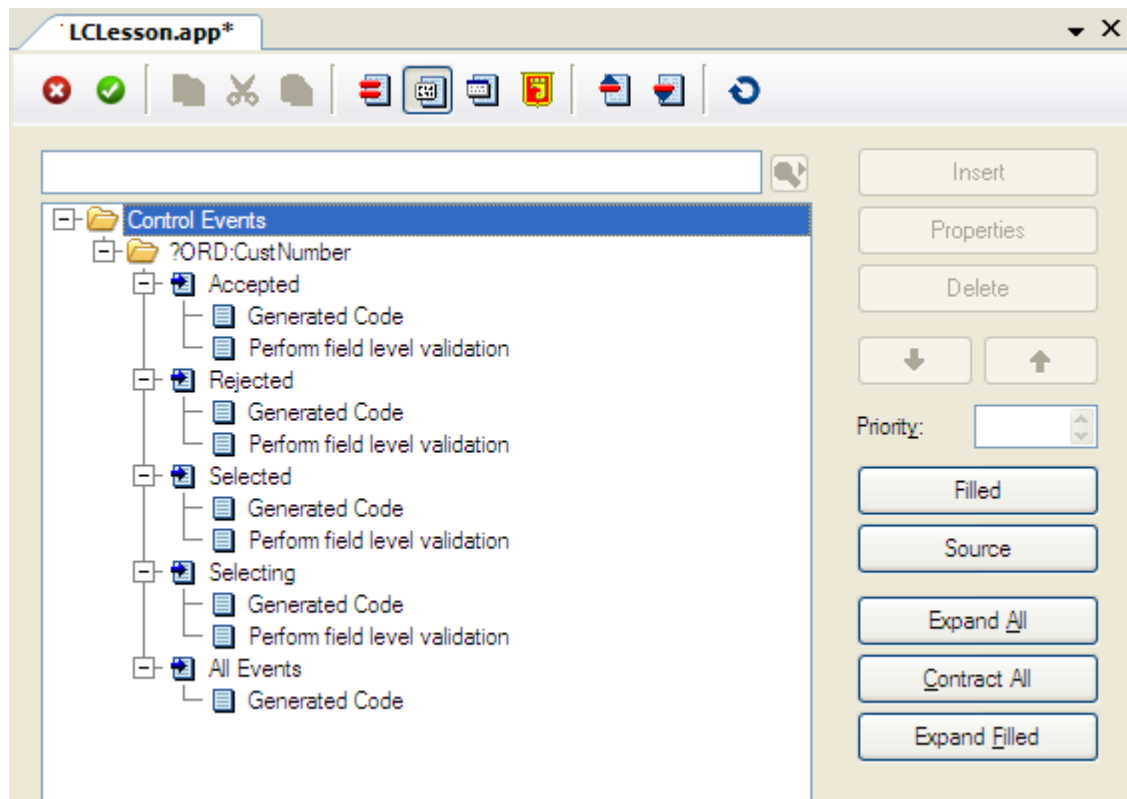
6. Press the ellipsis button (...) for the **Lookup Field** entry box. This makes *CUS:CustNumber* the column that must contain the matching value to the value the user enters into this control.
7. Choose the *BrowseCustomers* procedure from the **Lookup Procedure** drop list.

This calls the *BrowseCustomers* procedure when the user enters an invalid customer number, so the end user can select from a list of customers.

8. Check the **Perform Lookup during Non-Stop Select** and **Force Window Refresh when Accepted** boxes to ensure that the data displayed on screen is always valid and current.



9. Press the **OK** button to close the Entry Actions.
10. RIGHT-CLICK on the Entry control, and select **Embeds...** from the popup menu.



This displays a list of just the embed points associated with this one control. This is the quickest way to get to a specific control's embed points, and it's the second way you've seen so far to get to an embed point. There is a third method that's still to come.

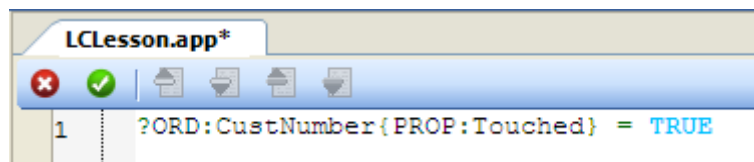
11. Highlight the *Selected* event under *Control Events* then press the **Insert** button.

The *Selected* event occurs just before the control gains input focus. This embed point allows you to do any "setup" specific to this one control.

12. Highlight *Source* then press the **Select** button.

This calls the Text Editor to allow you to write any Clarion source code you want. Notice that the floating *Populate Column* toolbox is present. Whenever you DOUBLE-CLICK on a column in this toolbox, it places the name of the column (including any prefix) in your code at the insertion point. This not only helps your productivity (less typing), but also ensures you spell them correctly (eliminating misspelled column name compiler errors).

Type the following code:



13. Press the **Save and Close** button to return to the *Embedded Source* dialog.


It is "standard Windows behavior" that, if the user does not enter data into a control and just presses tab (or CLICKs the mouse) to go on to another control, an *Accepted* event does not happen (this is very different from the way DOS programs work). This allows users to easily tab through the window's controls without triggering data-entry validation code on each control. However, sometimes you need to override this "Windows standard behavior" to ensure the integrity of your database.

The `?ORD:CustNumber{PROP:Touched} = TRUE` statement uses the Clarion language Property Assignment syntax. By setting `PROP:Touched` to `TRUE` in the *Selected* event for this control, an *Accepted* event is always generated—whether the user has entered data or not. This forces the lookup code generated for you into the *Accepted* event for this control (from the information you entered on the **Actions** tab on the previous page) to execute. This ensures that the user either enters a valid Customer number, or the Customer list pops up to allow the user to select a Customer for the Order.

14. Press the **Save and Close** button to return to the Window Designer.

Add a "display-only" control

1. Switch back to the **Toolbox** Pad, and choose (DRAG) the **STRING** control from the **Toolbox** Pad and DROP it to the right of the customer number entry box you placed before.
2. RIGHT-CLICK the string control you just placed, and select **Properties** from the popup menu.
3. In the Properties Pad, change the **IsPicture** property to `TRUE`, the **Picture** property to `@s30`, and the **Use** property to `CUS:Company`.


[-] General	
Justification	Default
Layout	Default
StringValue	@s30
IsPicture	True
Picture	@s30 
TextFont	Microsoft Sans Serif, 10
Use	CUS:Company

Placing the Detail Table's Control Templates

The next key element in this window is a browse list box control, synchronized to the Order Number of this form. This will show all the rows in the Detail table related to the currently displayed *Orders* table row.

Add a Detail list

1. Open the **Control Templates** Pad, highlight *BrowseBox*, then DRAG the control just below the customer number entry box you placed before, and DROP.
2. Highlight the *<ToDo>* item below the *File-Browsing List Box* and press the **Add** button.
3. Select the *Detail* table from the *Select a Table* dialog, then press the **Select** button.
4. Highlight the *Detail* table, and press the **Change** button.

5. Highlight *OrderNumberKey* in the *Select Key from Detail* dialog, then press the **Select** button.
6. Highlight *ProdNumber* in the *Columns* list, then press the **Select** button.
7. RIGHT-CLICK on the Browse box just populated, and select **List Box format...** from the popup menu.
8. Select *Center* from the Data group's **DataJustification** property drop list.
9. Press the **Add Field**  button.
10. Highlight *Quantity* in the *Columns* list, then press the **Select** button.
11. Select *Center* from the Data group's **DataJustification** property drop list.
12. Press the **Add Field** button.
13. Highlight *ProdAmount* in the *Columns* list, then press the **Select** button.
14. Select *Center* from the Data group's **DataJustification** property drop list.
15. Press the **Add Field** button.
16. Highlight *LOCAL DATA UpdateOrder* in the *Select Column* list, then press the **New** button.

This **New** button allows you to add a local variable without going all the way back to the Data / Tables Pad. We will use this new variable to display the total price for each line item (the quantity multiplied by the unit price).

17. Type *ItemTotal* in the **Name** entry.
18. Select *DECIMAL* from the **Type** drop list.
19. Type 7 in the **Characters** field.
20. Type 2 in the **Places** field then press the **OK** button.
21. Press the **Cancel** button to close the Local Data entry process and return to the List Box Formatter.
22. In the *ItemTotal* column, select *Center* from the Data group's **DataJustification** property drop list.
23. Press the **Add Field** button.
24. Highlight the *Detail* table item below the *File-Browsing List Box* and press the **Add** button.
25. Select the *Products* table from the *Select* dialog, then press the **Select** button.

This adds the *Products* table to the Control template's table schematic as a lookup table. The related row from the *Products* table is automatically retrieved for you so you can display the product description in the list.

26. Highlight *ProdDesc* in the *Columns* list, then press the **Select** button.
27. Resize the columns and adjust the display formatting as you want (you've done this a couple of times already). Use the diagram below as a guide.
28. Press the **OK** button to close the List Box Formatter.
29. Resize the list box to display all the columns you populated into it.

Update Records...

ORDERDATE: <#/#/#> ORDERNOTE: \$\$\$\$\$\$\$\$\$\$\$\$\$\$

Cust Number: \$\$\$\$\$\$\$\$\$\$\$\$\$\$ \$\$\$\$\$\$\$\$\$\$\$\$\$\$

Prod#	Quantity	Amt	Item Total	Prod Desc

Synchronize the browse list box

We want this list to only display the *Details* table rows that are related to the Customer table row currently being edited. Therefore, we need to specify a Range Limit.

1. RIGHT-CLICK the list box you just placed, and select **Actions** from the popup menu.
2. Press the ellipsis (...) button for the **Range Limit Field**.
3. The *DTL:OrderNumber* column is automatically populated.
4. Choose *File Relationship* from the **Range Limit Type** drop list.
5. Type *Orders* in the **Related File** column, or use the ellipsis to select it from a selection dialog.

Add an order invoice total calculation

Now we want to calculate the order total and save it in the Orders table.

1. Select the **Totaling** tab.
2. Press the **Insert** button.
3. Press the ellipsis (...) button for the **Total Target Field**.
4. Highlight the *Orders* table in the Tables list, select *ORD:InvoiceAmount* from the Columns list, then press the **Select** button.

This is the column that will receive the result of the total calculation.

5. Choose *Sum* from the **Total Type** drop list.
6. Press the expression (E) button for the **Field to Total**.
7. In the Expression Editor dialog, DOUBLE-CLICK on *ItemTotal* from the Columns list, then press the **OK** button.

This is the column whose contents will be used in the total calculation. So far we've only declared this column and not done anything to put any value into it, but we'll get to that very soon.

8. Choose *Each Record Read* from the **Total Based On** drop list.
9. Press **OK** to close the *Browse Totaling* dialog.

Change the object name

Now we want to change the name of the browse object to make our code more readable (you'll see why a little later).

1. Select the **Classes** tab.
2. Type *BrowseDTL* in the **Object Name** column.
3. Press the **OK** button to close the Actions dialog and return to the Window Designer.

Add horizontal and vertical scroll bars

1. With the list box still selected, press F4 to switch focus to the **Properties** Pad.
2. Set the **Horizontal** and **Vertical** properties to *TRUE*.
3. Press the **Save and Close** button to save your changes, and then let's continue and re-enter the Designer by pressing the **Designer** button.

Add the standard table update buttons

1. In the *Control Templates Pad* dialog, DRAG the *BrowseUpdateButtons* control template, then DROP below the list box.

The **Insert**, **Change**, and **Delete** buttons all appear together.

2. RIGHT-CLICK on the **Delete** button only, then choose **Actions** from the popup menu.
3. Check the **Use Edit in place** box.

Checking this box for one button in the Control template checks it for all three. We will be using the Edit in Place technique to update the Details table rows instead of an update Form procedure. This will allow us to demonstrate some fairly advanced programming techniques and show just how easy they are to perform within the Application Generator.

4. Press the **OK** button.

Add a "display-only" control for the invoice total

1. Switch back to the **Toolbox** Pad, and choose (DRAG) the **STRING** control from the **Toolbox** Pad and DROP it below the bottom right corner of the list box.
2. RIGHT-CLICK the string control you just placed, and select **Properties** from the popup menu.
3. In the Properties Pad, change the **IsPicture** property to *TRUE*, the **Picture** property to *@n\$10.2*, and the **Use** property to *ORD:InvoiceAmount*.

This specifies the control will display data from a variable, not just a string constant.

Change the form window caption and exit the Window Designer

1. CLICK on the caption bar of the sample window.
2. Type *Order Form* in the **Title** field in the *Properties Pad* toolbox, then press the TAB key.
3. Choose **Save and Close** to close the Window Designer.

Making it all Work

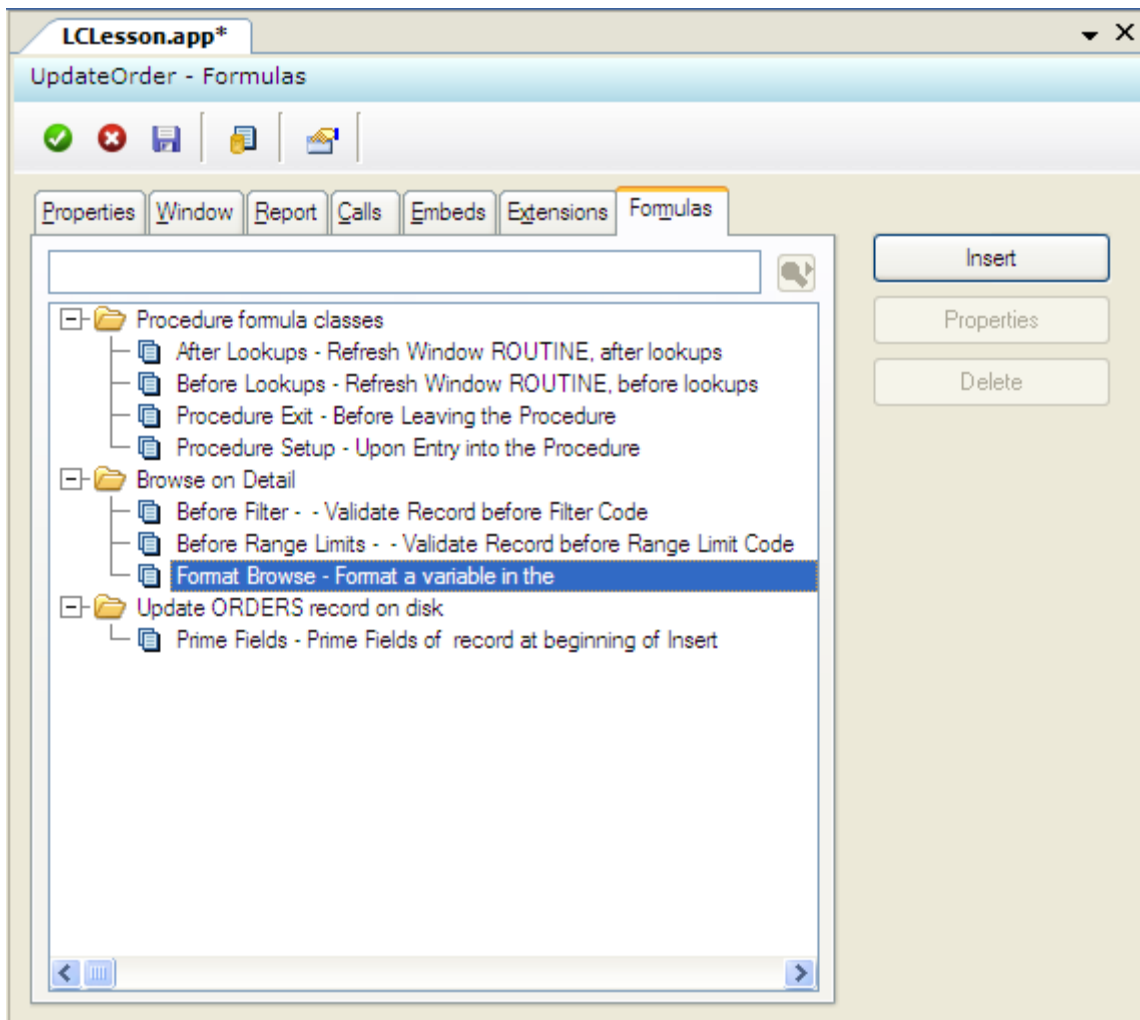
There are a couple of things we need to do to make this procedure fully functional—add a Formula, and configure the Edit in Place.

Using the Formula Editor

To make the *ItemTotal* calculate the correct amount for each Detail row in the browse list box, you need to add a Formula to the procedure. This will also allow the browse totaling to correctly place the invoice total in the *ORD:InvoiceAmount* column.

1. Press the **Formulas** tab in the *Procedure Properties* dialog.

The *Formula Editor* dialog appears. This dialog lists all the formulas in the procedure.



2. Highlight *Format Browse* in the *Template Classes* list, and press the **Insert** button to add a new formula.

The *Formula Editor* design dialog appears.

3. Type *Item Total Formula* in the **Name** field.
4. Select *Format Browse* from the Drop List selection in the **Class** Field.

The Class field simply specifies the logical position within the generated source code at which the formula is calculated. The *Format Browse* class tells the *BrowseBox* Control template to perform the calculation each time it formats a row for display in the list box.

5. Press the ellipsis (...) button to the right of the **Result** Field.
6. Highlight *LOCAL DATA UpdateOrder* in the *Tables* list, select *ItemTotal* from the *Select Columns* list, then press the **Select** button.

This names the column that will receive the result of the calculation. This is the column we defined earlier through the List Box Formatter.

7. Press the **Data** button in the **Operands** group.
8. Highlight the *Detail* table in the *Tables* list, select *Quantity* from the *Columns* list, then press the **Select** button.

This places *DTL:Quantity* into the Statement column for you. The Statement column contains the expression being built. You can type directly into the Statement field to build the expression, if you wish.

9. Press the * button in the **Operators** group.

This is the multiplication operator.

10. Press the **Data** button in the **Operands** group.
11. Highlight the *Detail* table in the *Tables* list, select *ProdAmount* from the *Columns* list, then press the **Select** button.
12. Press the **Check** button to validate the expression's syntax.

A green checkmark appears left of the statement, indicating the syntax is correct. If a red X appears, the expression's syntax is incorrect and the highlighted portion of the statement is what you must change.

13. Press the **Save and Close** button to close the Formula Editor.

The *Formula Editor* dialog with a formula list re-appears.

Configuring Edit in Place

Now we need to configure the Edit in Place characteristics. We previously used Edit in Place for the Phones table and simply took all the default behaviors, because that was a fairly simple table. However, now we're editing a line item *Detail* row for an order entry system, which means we need to do some data entry validation beyond simply ensuring the user types in a number that fits the display picture. To do this, we'll need to extend the simple Edit in Place functionality provided by the Application Builder Class (ABC) Library.

Set Column Specific Classes

1. Press the **Extensions** tab in the *Procedure Properties* dialog.
2. Highlight *Update a Record from Browse Box on Detail* then press the **Properties** button.
3. Press the **Configure Edit in place** button.

The *Configure Edit in place* dialog appears. The options on this dialog allow you to specify what behavior occurs while the user is editing data and presses enter or an arrow key, along with several save options. We'll accept all the defaults.

4. Press the **Column Specific** tab control.
5. Press the **Insert** button.
6. Press the ellipsis (...) button for the **Field** field.

7. Highlight the *Detail* table, and then *ProdNumber*, and then press the **Select** button.

The ABC Library contains an object class called `EditEntryClass` that is the default Edit in Place class. We're going to override some of the methods of the default class for this column so we can modify the default behavior. Adding this column to the **Column Specific** list of columns is what allows us to override the default Edit in Place methods for this one column. We will do this so that we can perform data validation on this column when the user enters data—we want to make sure that they can only enter a number that refers to a valid *Products* table row.

8. Press the **OK** button.
9. Press the **Insert** button.
10. Press the ellipsis (...) button for the **Field** field.
11. Highlight *DTL:Quantity*, then press the **Select** button.

The ABC Library's `EditEntryClass` defaults to using an `ENTRY` control, and for this column we want to use a `SPIN` control so the user can just spin to the quantity they want to order. Therefore, we need to override some methods for this column too, to have a `SPIN` instead of an `ENTRY` control.

12. Clear the **Use Default ABC** check box.

This allows you to specify the exact class from which to derive.


13. Choose **EditSpinClass** from the Base Class drop list.

The `EditEntryClass` does too much that's unnecessary for this, since we're going to override the methods anyway. Therefore, we're going to derive and override from the Base Class all the Edit in place classes were derived from: `EditClass`.

14. Press the **OK** button.
15. Press the **Insert** button.
16. Press the ellipsis (...) button for the **Field** field.
17. Highlight *DTL:ProdAmount*, then press the **Select** button.
18. Clear the **Allow Edit in Place** check box.

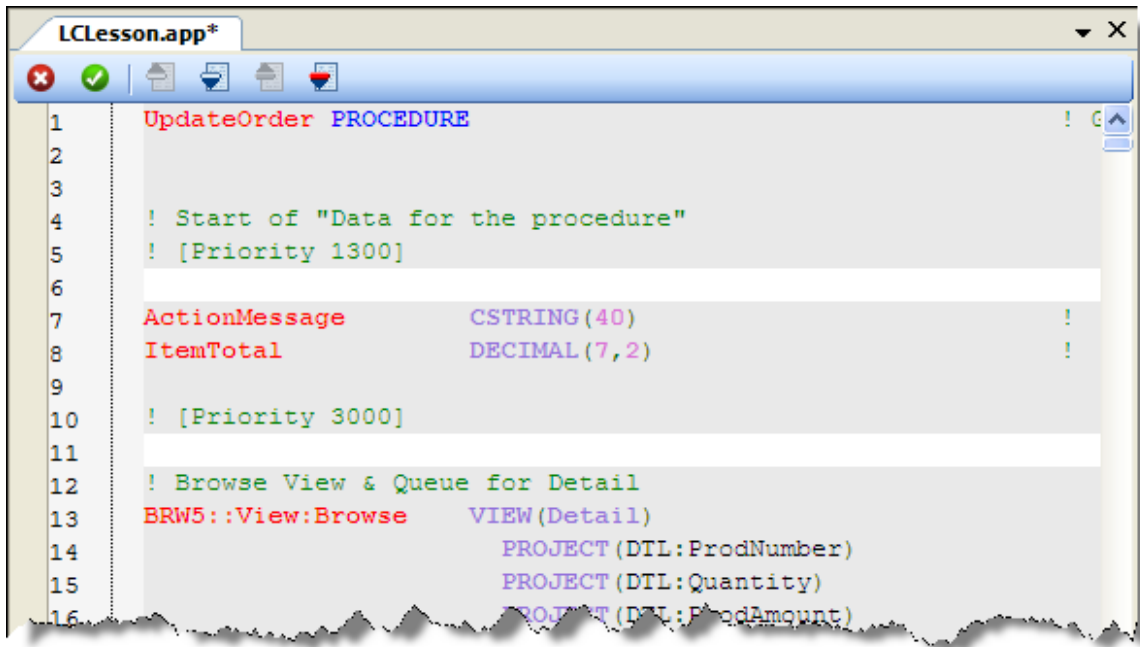
This turns off Edit in Place for this one column of the list box.

Only columns in the Primary table for the *BrowseBox* can be edited in place, and the default is that all Primary table columns are editable. In this case, this means all the Detail table columns can be edited and the Products table columns cannot be edited. However, for this procedure we do NOT want the user to be able to edit the *DTL:ProdAmount* column because we're going to get its value directly from the Products table and we don't want the user to be able to change it. That's why we turned off the **Allow Edit in Place** box.

19. Press the **OK** button twice to return back to the Extensions dialog, then press **Save and Close** to return to the Application Tree. Save your work by pressing the **Save** button  on the IDE toolbar.

Using the Embeditor

1. RIGHT-CLICK on *UpdateOrder* and select **Source** from the popup menu.



The **Source** popup menu selection opens the "Embeditor"—the third method of accessing embed points in a procedure. The Embeditor is the same Text Editor you've already used, but opened in a special mode which allows you to not only to edit all the embed points in your procedure, but to edit them within the context of template-generated code.

Notice that most of the code is on a gray background and the points where you can write your code have a white background. There are also identifying comments for each embed point. You can turn these comments on and off as you choose through the **Setup ► Application Options** dialog. Once you become familiar with them, you'll probably want to turn them off so you can see more of the actual code.


You'll notice that a message briefly appeared that said "Generating LCLesson." The Embeditor displays all possible embed points for the procedure within the context of all the possible code that may be generated for the procedure. Notice the distinction here—Embeditor does not show you the code that will be generated, but all the code which could be generated for you, if you chose every possible option and placed code into every available embed point. You are not likely to ever do that. Therefore, a lot more generated code shows up in the Embeditor than will actually be in the generated code when you compile your application. After we finish here, we'll go look at the generated code to see the difference.

At the right end of the toolbar are four buttons which are very important to know when working in the Embeditor. These are (from left to right) the Previous Embed, Next Embed, Previous Filled Embed, and Next Filled Embed buttons (hover your mouse over them and the tooltips appear naming the buttons). They allow you to quickly get from one embed point to another—particularly after you've written code into some of them.

Detecting Changed Orders

One of the things we want this procedure to do is to detect changes to existing orders and make sure the changes do not result in a data mis-match between the *Orders* and *Detail* tables. This system is storing the total dollar amount of an order in the ORD:InvoiceAmount column, so when the user changes a Detail item in an existing Order, we want to make sure the Orders table row is

updated, too. There's fairly simple way to do that which will allow us to demonstrate the ABC Library's flexible error handling.

1. CLICK on the **Next embed** button  (about 5 times) until you get to the embed point immediately preceding the line of code reading *ThisWindow* *CLASS(WindowManager)*.

```
63  
64 ! [Priority 5030]  
65  
66 ! End of "Data for the procedure"  
67 ThisWindow CLASS(WindowManager)  
68 Ask PROCEDURE(), DERIVED ! Method added to host embed code
```

Each embed point potentially has 10,000 priority levels within it. This Embed code Priority level system is designed to allow you to embed your code before or after any generated code—whether that code is generated for you by Clarion's ABC Templates or any third-party templates you choose to use. This makes the embed system completely flexible, allowing you to add your own code at any logical point needed—before or after most any "chunk" of generated code.

2. Type (or copy and paste) the following code:

```
LocalErrGroup GROUP  
    USHORT(1)  
    USHORT(99)  
    BYTE(Level:Notify)  
    PSTRING('Save the Order!')  
    PSTRING('Some Item changed -- Press the OK button.')
```

```
END  
SaveTotal LIKE(ORD:InvoiceAmount)
```

The red text indicates that the text begins in Column 1, and identifies a data label.

Clarion's ABC (Application Builder Class) Templates generate Object Oriented code for you using the ABC Library. The ABC Library contains an error handling class called *ErrorClass*. This bit of code declares a *LocalErrGroup* GROUP (in exactly the form that the *ErrorClass* requires—see the ABC Library Reference - Vol I) containing a "custom" error number and message that we are defining for use by the *ErrorClass* object in our application. The *SaveTotal* declaration is a local variable which is defined LIKE (always has the same data type) the *ORD:InvoiceAmount* column. We'll use this variable to hold the starting order total when the user is updating an existing order.

3. Choose **Search** ► **Find** to bring up the *Find* dialog.
4. Type *ThisWindow.Init* into the **Find what** entry, then press the **Find** button.

This takes you directly to the **ThisWindow.Init** method. Press **Close (the Red X)** to close the *Search and Replace* dialog.

5. In the embed point immediately following the line of code reading *SELF.Errors* *&= GlobalErrors* (this should be at Priority 5300), type the following code:

```
SELF.Errors.AddErrors(LocalErrGroup) !Add custom error
```

```

IF SELF.Request = ChangeRecord          !If Changing a row
    SaveTotal = ORD:InvoiceAmount        !Save the original order total
END

```

```

298 |
299 | ! [Priority 5050]
300 |
301 | ! Set options from global values
302 | IF ReturnValue THEN RETURN ReturnValue.
303 | SELF.FirstField = ?OK
304 | SELF.VCRRequest &= VCRRequest
305 | SELF.Errors &= GlobalErrors          ! Set this
306 |
307 | ! [Priority 5300]
308 | SELF.Errors.AddErrors(LocalErrGroup) !Add custom error
309 | IF SELF.Request = ChangeRecord      !If Changing a row
310 |     SaveTotal = ORD:InvoiceAmount    !Save the original order to
311 | END
312 | ! BIND variables
313 | BIND('ItemTotal',ItemTotal)         ! Added by:
314 |

```

This code calls the *AddErrors* method of the *GlobalErrors* object to add the *LocalErrGroup* to the list of available errors that the object handles. The *GlobalErrors* object is an instance of the *ErrorClass* which the ABC Templates declare globally to handle all error conditions in the application. Adding our *LocalErrGroup* enables the *GlobalErrors* object to handle our "custom" error condition. This demonstrates the flexibility of Clarion's ABC Library. The **IF** statement detects when the user is editing an existing order and saves the original order total.

This embed point is in the *ThisWindow.Init PROCEDURE* which performs some necessary initialization tasks. This is a virtual method of the *ThisWindow* object. *ThisWindow* is the object which handles all the window and control handling code.

You may not have noticed, but the ABC Templates generate exactly one line of executable source code within the *UpdateOrder PROCEDURE* itself (*GlobalResponse = ThisWindow.Run*) so all of the functionality of the *UpdateOrder PROCEDURE* actually occurs in object methods—either virtual methods specific to the *UpdateOrder PROCEDURE* itself or standard ABC Library methods. This is true of every ABC Template generated procedure. Generating fully Object Oriented code makes the code generated for you very tight and efficient—only the code that actually needs to be different for an individual procedure is handled differently. Everything else is standard code that exists in only one place and has been tested and debugged to ensure consistent performance.

Object Oriented Programming (OOP) in Clarion starts with the *CLASS* structure. See *CLASS* in the Language Reference Help for a discussion of OOP syntax. The Advanced Programming Resources PDF contains several articles which discuss OOP in depth, and the ABC Library Reference fully documents Clarion's Application Builder Class (ABC) Library.

6. Press **Search ► Find** (or press **CTRL + F**) to open the search dialog again. Type *ThisWindow.Kill* into the **Find what** entry, then press the **Find** button.
7. In the embed point immediately following the line of code reading *CODE* (this should be [Priority 2300]), type the following code:

```

SELF.Errors.RemoveErrors(LocalErrGroup)    !Remove custom error

```

```
436
437 ThisWindow.Kill PROCEDURE
438
439 ReturnValue          BYTE,AUTO
440
441
442 ! Start of "WindowManager Method Data Section"
443 ! [Priority 5000]
444
445 ! End of "WindowManager Method Data Section"
446 CODE
447
448 ! Start of "WindowManager Method Executable Code Section"
449 ! [Priority 2300]
450 SELF.Errors.RemoveErrors(LocalErrGroup)    !Remove custom error
451 ! UNBIND variables
452 ! [Priority 4800]
453
454 ! Parent Call
```

This calls the ABC Library method to remove our "custom" error. The ThisWindow.Kill method is a "cleanup" procedure (performs necessary exit tasks) which executes when the user is finished working in the UpdateOrder procedure, so the error is no longer needed at that point.

8. Type *EVENT:CloseWindow* into the **Find what** field, then press the **Find next** button. You will need to press the **Find** button twice to find the embed point referenced in Step 9. Close the *Search* dialog once you have found the desired embed point.
9. In the embed point immediately following the line of code reading *OF EVENT:CloseWindow* (this should be Priority 2500), type the following code:

```
IF SELF.Request = ChangeRecord      AND |    ! If Changing a row
    SELF.Response <> RequestCompleted AND|    ! and OK button not pressed
    SaveTotal <> ORD:InvoiceAmount      ! and detail recs changed
    GlobalErrors.Throw(99)             ! Display custom error
    SELECT(?OK)                        ! then select the OK button
CYCLE
END
```

```

2508
2509 ! Generated Code
2510 ! End of "Window Event Handling"
2511     OF EVENT:CloseWindow
2512
2513 ! Start of "Window Event Handling"
2514 ! [Priority 2500]
2515 IF SELF.Request = ChangeRecord      AND | ! If Changing a row
2516     SELF.Response <> RequestCompleted AND| ! and OK button not pressed
2517     SaveTotal <> ORD:InvoiceAmount      ! and detail recs changed
2518     GlobalErrors.Throw(99)             ! Display custom error
2519     SELECT(?OK)                       ! then select the OK button
2520 CYCLE
2521 END
2522

```

This is the code that will detect any attempt by the user to exit the UpdateOrder procedure without saving the *Orders* table row after they've changed an existing order. Note the vertical bar characters (|) at the end of the first two lines of code. These are absolutely necessary. Vertical bar (|) is the Clarion language line continuation character. This means that the first three lines of this code are a single logical statement which evaluates three separate conditions and will only execute the `GlobalErrors.Throw(99)` statement if all three conditions are true.

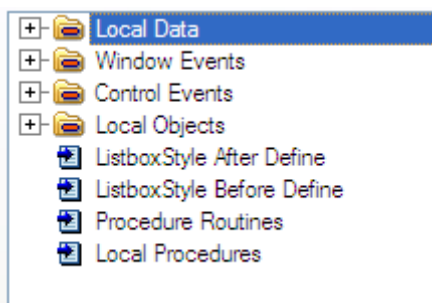
Overriding the Edit in Place Classes

OK, now you've seen an example of how you can use the ABC Library in your own embedded source code. Now we'll show you how to *override* a class to provide custom functionality that the ABC Library does not provide. The CLASS declarations for the objects that we named through the *Configure Edit in Place* dialogs are generated for you by the ABC Templates. These CLASSES are both derived from the *EditClass* ABC Library class.

1. Press **Save and Close**  to close the Embed Editor.

We will use the Embed tree to find the remaining embeds.

2. RIGHT-CLICK on *UpdateOrder* and select **Embeds** from the popup menu.
3. To make the embed tree easier to read, press the **Contract All** button.



4. By pressing the + button on the tree, expand the *Local Objects* ► *ABC Objects* ► *EIP Field Manager for Browse Using ?List for column DTL:Quantity (EditSpinClass)* ► *Init* ► *Code* ► *Parent Call*.

5. Press the **Source** button. You should now be in the embed point immediately following the line of code reading `PARENT.Init(FieldNumber,Listbox,UseVar)` (this should be [Priority 5300]), type the following code:

```
SELF.Feq{PROP:Text} = ListBox{PROPLIST:Picture,FieldNumber}
                                !Set entry picture token
SELF.Feq{PROP:RangeLow} = 1      !Set RANGE values for the SPIN
SELF.Feq{PROP:RangeHigh} = 9999
```

This code sets the data entry picture token and range of valid data for the SPIN control.

6. Scroll to the `EditInPlace::DTL:Quantity.SetAlerts` method (this should immediately follow the `EditInPlace::DTL:Quantity.ResetControl` method). Enter the following code:

```
SELF.Feq{PROP:ALRT,5} = ''
SELF.Feq{PROP:ALRT,6} = ''
```

Note:

If you wish to see the base class code that we've overridden, open the `ICLARION7LIBSRCVABEIP.CLW` file and search for `EditSpinClass`.

7. Press **Search ► Find** to open the search dialog again. Type `EditInPlace::DTL:ProdNumber.TakeEvent` into the **Find what** field, then press the **Find** button.
8. In the embed point immediately following the line of code reading `ReturnValue = PARENT.TakeEvent(Event)` (this should be [Priority 5300]), type (or copy and paste) the following code:

```
UPDATE(SELF.Feq)                                !Update Q field
IF ReturnValue AND ReturnValue <> EditAction:Cancel OR |
    EVENT() = EVENT:Accepted                    !Check for completion
    PRD:ProdNumber = BrowseDTL.Q.DTL:ProdNumber !Set for lookup
    IF Access:Products.Fetch(PRD:ProdNumberKey) !Lookup Products row
        GlobalRequest = SelectRecord            !If no row, set for select
        BrowseProducts                          ! then call Lookup proc
        IF GlobalResponse <> RequestCompleted    !Row selected?
            CLEAR(PRD:Record)                   ! if not, clear the buffer
            ReturnValue = EditAction:None        ! and set the action to
        END                                     ! stay on same entry field
    END
    BrowseDTL.Q.DTL:ProdNumber = PRD:ProdNumber !Assign Products table
    BrowseDTL.Q.DTL:ProdAmount = PRD:ProdAmount ! values to Browse QUEUE
    BrowseDTL.Q.PRD:ProdDesc = PRD:ProdDesc    ! fields
    DISPLAY                                    ! and display them
END
```

This is the really interesting code. Notice that the first executable statement (generated for you) is a call to the `PARENT.TakeEvent` method. This calls the `EditSpinClass.TakeEvent` method we're overriding so it can do what it usually does.

All the rest of the code is there to give this derived class extra functionality not present in its parent class. This is the real power of OOP—if you want everything the parent does, plus a little bit more you don't have to duplicate all the code the parent executes, you just call it. In this case, all the extra code is to perform some standard data entry validation tasks. This code will verify whether the user typed in a valid Product Number and if they didn't, it will call the `ViewProducts` procedure to allow them to choose from the list of products.

Overriding Methods in the Embeditor

There is a very important point to understand about working in the Embeditor or the Embed Tree. When you are overriding methods: *as soon as you type anything into an embed point in an overridable method, you have overridden it.* Even a simple `!` comment line makes this happen, because the Application Generator notes that you have written some of your own code, and so generates the proper method prototype into the local `CLASS` declaration for you. To prevent you from accidentally adding a comment that causes the method override which consequently "breaks" the functionality, the ABC Templates automatically generate `PARENT` method calls and `RETURN` statements for you, as appropriate.

You'll notice that all of our overridden methods contained generated calls to their `PARENT` method, and the `TakeEvent` method also had a generated `RETURN` statement. Sometimes you want these statements to execute, and sometimes you don't (usually, you do). For those cases where you do not want them to execute, simply write your code in the embed point which comes before the `PARENT` method call and write your own `RETURN` statement at the end of your code.

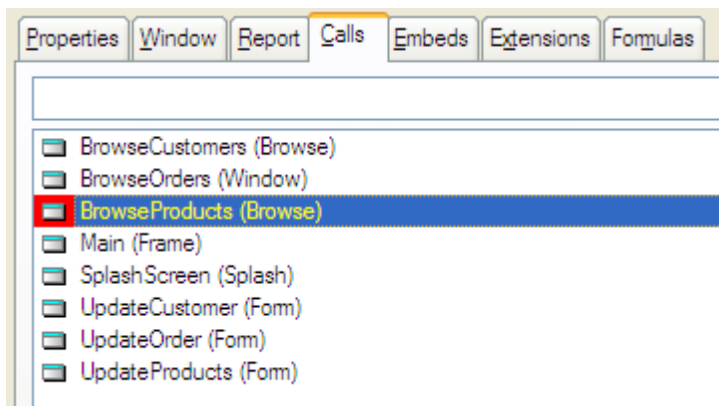
This means that the generated `PARENT` method call will never execute. Clarion's optimizing compiler is smart enough to recognize that these statements can never execute and optimizes them out of compiled object code.


1. Choose **Save and Close** to close the Embeditor, then press **Save and Close** to close the embed tree. Make sure to save your work when prompted.

Update the Procedure Call Tree


The `EditInPlace::DTL:ProdNumber.TakeEvent` method calls the `BrowseProducts` procedure from within its code. Since this is just embedded source code, the Application Generator doesn't know you've called this procedure, and needs to be told (if you don't, you'll get compiler errors), so it can generate the correct `MAP` structure for the module containing this procedure.

1. RIGHT-CLICK on `UpdateOrder` and select **Procedures** from the popup menu.
2. Highlight `BrowseProducts` then press the **Save and Close** button.



3. Choose **File** ► **Save**, or press the **Save** button  to save your work.

Generate code

1. Press the **Generate All** button  in the IDE toolbar to generate the source code for your application.
2. RIGHT-CLICK on *UpdateOrder* and select **Module** from the popup menu.

The Text Editor appears containing the generated source code for your *UpdateOrder* procedure. Notice that there is a lot less code here than there was in the Embeditor. All that generated code in the Embeditor was there to provide you with context, and to provide you with embed points with which to override methods, should you need to. However, Clarion's Application Generator and ABC Templates are smart enough to only generate the code you actually need, when you actually need it.

3. From the IDE Menu, choose **File** ► **Close** ► **File** to close the Text Editor.

Now might be a good time to try out your application. You've got all the data entry portions completed and the only things left to do now are the reports, which we'll get to in the next lesson.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

- ✓ You created a new Form procedure.
- ✓ You created a "Scrolling-Form" metaphor Edit-in-place browsebox to update the Detail table.
- ✓ You created a total column to total up the order.
- ✓ You created a Formula to total each line item in the order.
- ✓ You used the Embeditor to write your embedded source code within the context of template-generated code.
- ✓ You used the power of OOP to extend the standard error handling functionality of the ABC Library.
- ✓ You used the power of OOP to derive and override the Edit-in-place classes to extend the standard functionality of the ABC Library.
- ✓ You generated source code to compare the difference between the code shown in the Embeditor to that which is actually generated.

We're almost finished with this application. In the next lesson we'll create the application's reports.

12 - Creating Reports

Overview

The last item to cover is adding reports. First, we'll create a simple customer list to introduce you to the Report Designer. Then we'll create an Invoice Report to demonstrate how you can easily create Relational reports with multi-level group breaks, group totals, and page formatting. Then we'll copy the Invoice Report and limit the copy to print invoice for only one customer at a time.


Starting Point:

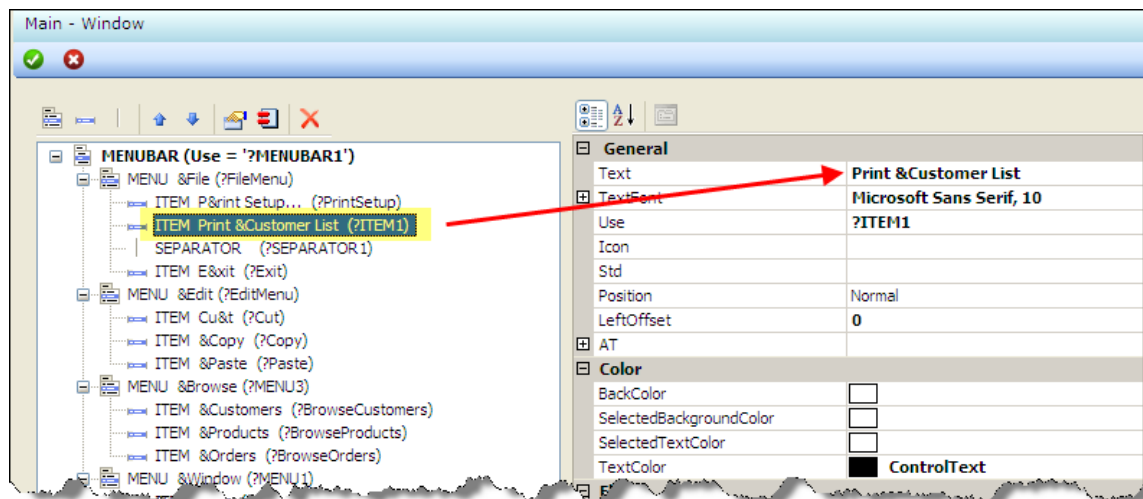
LCLESSON.APP should be open, and the **Application Tree** displayed.

Updating the Main Menu

First, we need to add menu selections so the user can call the reports, and so the Application Generator will call the appropriate "ToDo" procedures.

Add a menu item


1. RIGHT-CLICK on the *Main* procedure in the Application Tree dialog and choose **Window** from the popup menu.
2. RIGHT-CLICK on the Menubar on the window and select **Edit Menu**. The Menu Editor is now open.
3. Highlight the *P&rint Setup* item in the *Menu Editor* list.
4. Press the **Add New Item (Insert)**  button.
5. Type *Print &Customer List* in the **Text** property entry, then press the TAB key.





Specify the new item's action

1. RIGHT-CLICK on the *Print &Customer List* item in the *Menu Editor* list, and select the **Actions** item from the popup menu.
2. Choose *Call a Procedure* from the **When Pressed** drop list.
3. Type *CustReport* in the **Procedure Name** entry.
4. Check the **Initiate Thread** box.
5. Press **OK** to close the Actions (*?ITEM1 Prompts*) dialog.

Add a second menu item

1. Highlight the *Print &Customer List* item in the *Menu Editor* list, and then press the **Add New Item (Insert)**  button.
2. Type *Print &All Invoices* in the **Text** property entry, then press the TAB key.
3. RIGHT-CLICK on the *Print &All Invoices* item in the *Menu Editor* list, and select the **Actions** item from the popup menu.
4. Choose *Call a Procedure* from the **When Pressed** drop down list.
5. Type *InvoiceReport* in the **Procedure Name** entry.
6. Check the **Initiate Thread** box.


Add a third menu item

1. Highlight the *Print &All Invoices* item in the *Menu Editor* list, and then press the **Add New Item (Insert)**  button.
2. Type *Print &One Customer's Invoices* in the **Text** property entry, then press the TAB key.
3. RIGHT-CLICK on the *Print &One Customer's Invoices* item in the *Menu Editor* list, and select the **Actions** item from the popup menu.
4. Choose *Call a Procedure* from the **When Pressed** drop down list.
5. Type *CustInvoiceReport* in the **Procedure Name** entry.
6. Check the **Initiate Thread** box.
7. Press the **Save and Close** button to close the Menu Editor.
8. Press the **Save and Close** button to close the Window Designer.
9. Press the **Save and Close** button in the Window Designer Editor to return to the Application Tree.
10. Save your work by pressing the **Save** button  on the IDE toolbar.

Creating the Report

Now you can create the first report, using the Report Designer.

1. Highlight the *CustReport* procedure in the Application Tree.

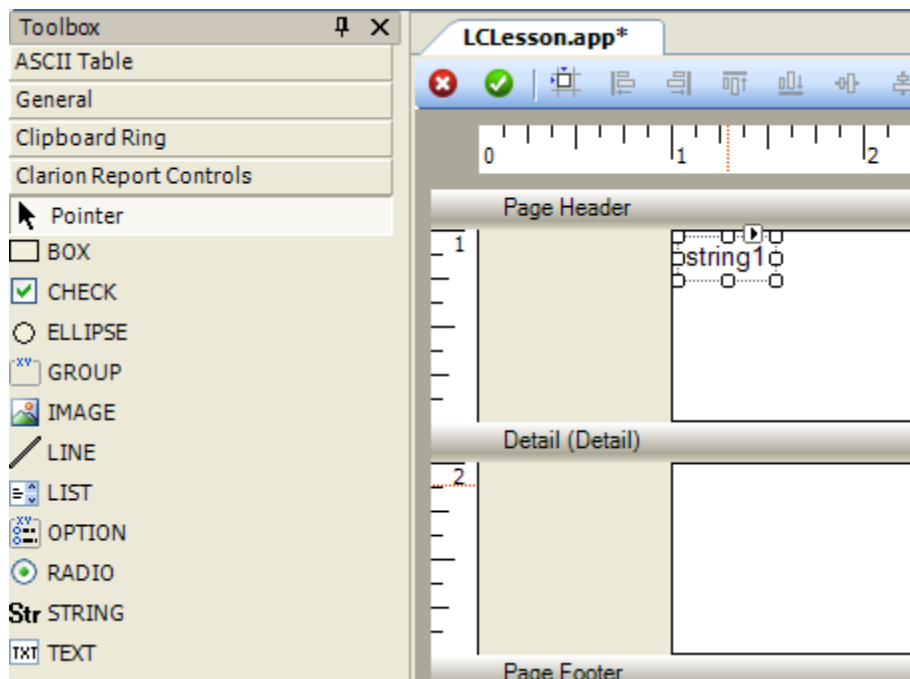
2. Press the **Properties**  button.
3. Choose the **Defaults** tab and then highlight *Report (Paper size Letter – Portrait)* in the *Select Procedure Type* dialog. Press the **Select** button.
4. Press the **Report** button in the *Procedure Properties* dialog.
5. Press the **Designer** button to open the Report Designer.

The Report Designer appears. Here you can visually edit the report and its controls. The Report Designer represents the four basic parts of the REPORT data structure by showing the Page Header, Detail, Page Footer, and Form as four "bands." Each band is a single entity that prints all together. See the User's Guide chapter on Using the Report Designer for more information on the parts of the report and how the print engine generates them.

For this report, you'll place page number controls in the header, then place the columns from the Customer table in the Detail band.

Place a string constant

1. With the Report Designer active, choose **View ► Toolbox** (or press CTRL + ALT + X).
2. Select the STRING control in the Toolbox, and DRAG the control to the top left of the **Page Header** band, and DROP.



3. RIGHT-CLICK on the control, then choose **Properties** from the popup menu.

The *String Properties* dialog appears.

4. In the Properties Pad, type *Page Number:* in the **Text** property.

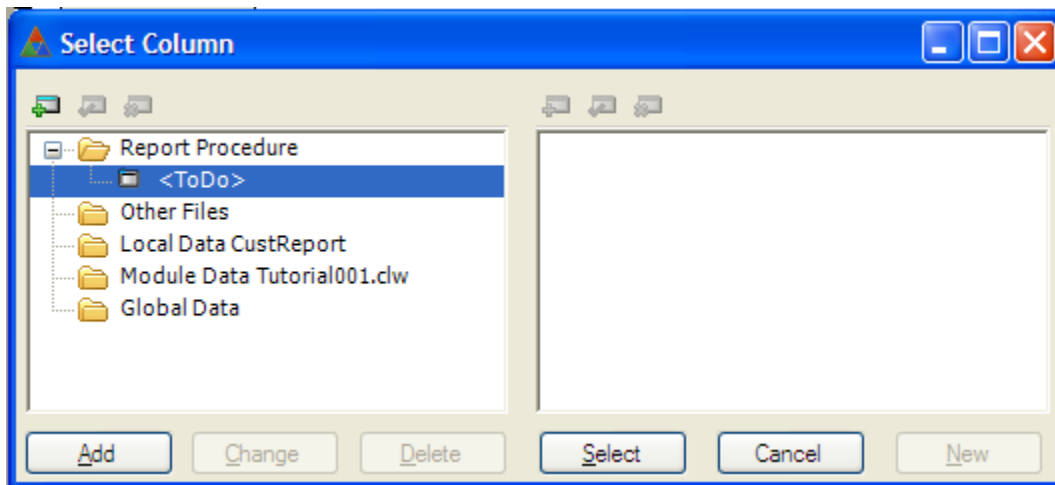
Place a control template to print the Page Number

1. Open the **Control Templates Pad**, if not already opened.
2. Highlight *ReportPageNumber* then DRAG to the right of the previously placed string.

Populating the Detail

The Detail band prints once for each row in the report. For this procedure, you'll place the columns in a block arrangement, which creates a label report at print time.

1. Select the **Report Designer ► Populate ► Multiple Columns** from the IDE toolbar.



2. In the *Select Column* dialog, highlight the *<ToDo>* folder, then press the **Add** button.
3. Select the *Customer* table from the *Select a Table* dialog, then press the **Select** button.
4. Highlight the *Customer* table, and press the **Change** button.
5. Highlight *KeyCustNumber* in the *Select Key from CUSTOMER* dialog, then press the **Select** button.
6. Highlight *Company* in the *Select Columns* list in the right pane, and press the **Select** button.
7. CLICK inside the **Detail** band, near its top left corner.
8. Highlight *FirstName* in the *Select Columns* list and press the **Select** button.
9. CLICK inside the **Detail** band, just below the first control.
10. Highlight *LastName* in the *Select Columns* list and press the **Select** button.
11. CLICK inside the **Detail** band, to the right of the control you just placed.
12. Highlight *CUS:Address* in the *Select Columns* list and press the **Select** button.
13. CLICK inside the **Detail** band, below the second control you placed.

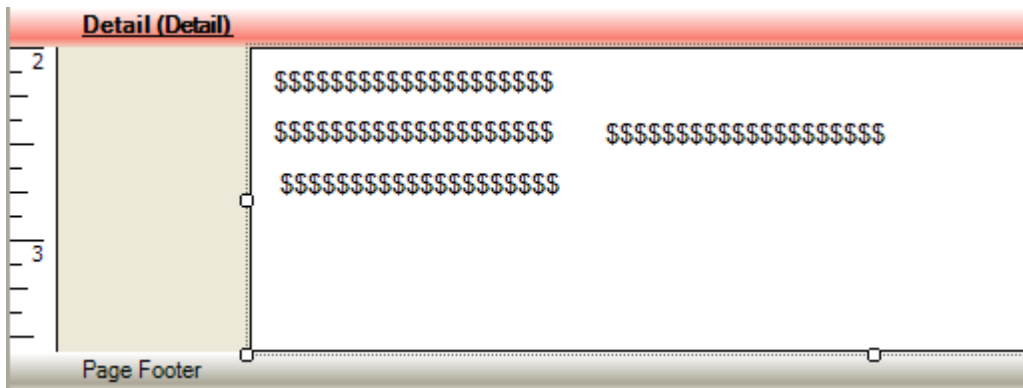
Resize the Detail band

At this point, you probably have very little room left in the Detail band, and need to make it longer.

1. Press the **Cancel** button to exit multi-populate mode.
2. CLICK inside the **Detail** band, but not on one of the string controls.

The Detail area's handles appear.

3. Resize the **Detail** band by dragging the middle handle on the bottom down—allow for enough room for about two more lines.



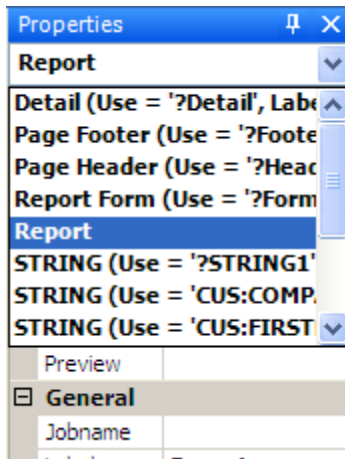
Place the rest of the columns

1. Choose **Report Designer ► Populate ► Multiple Fields** from the IDE Menu.
2. Highlight *City* in the *Columns* list, then press the **Select** button.
3. CLICK inside the **Detail** band, below the last control you placed.
4. Highlight *State* in the *Columns* list, then press the **Select** button.
5. CLICK inside the **Detail** band, to the right of the previously placed control.
6. Highlight *ZipCode* in the *Columns* list, then press the **Select** button.
7. CLICK inside the **Detail** band, to the right of the previously placed control.
8. Press the **Cancel** button to exit multi-populate mode.

Notice that you have the same set of alignment tools in the Report Designer that you have already used in the Window Designer. Feel free to align and adjust the position of your controls at this time.


Select a base font for the Report

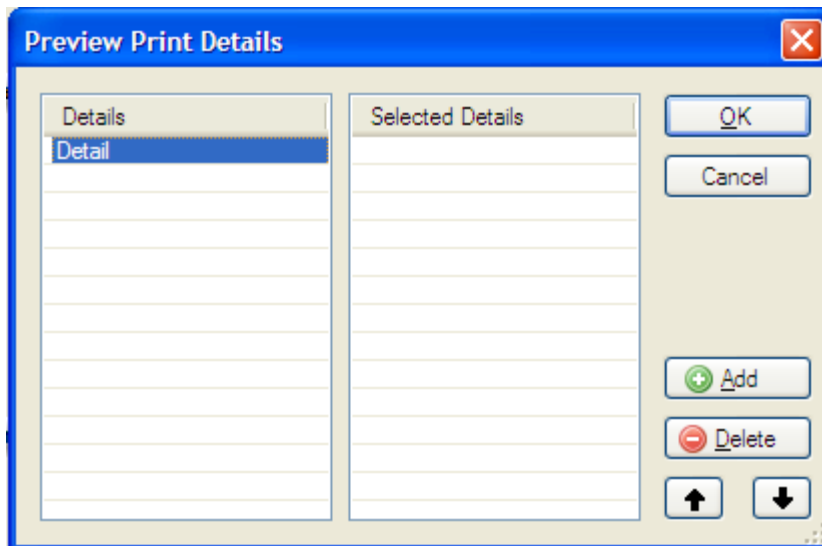
1. Open the **Properties Pad** (F4 Key), and select the **Report** control from the drop list as shown:



2. In the *Property Pad*, locate the **TextFont** property, and press the ellipsis button to the right.
3. Select a font, style, and size to use as the base font for the report.
If you don't select a font, it uses the printer's default font.
4. Press the **OK** button to close the *Select Font* dialog.


Preview the Report

1. In the Report Designer toolbar, press the **Print Preview** button  to "visualize" how the printed page will appear.



2. Highlight *Detail* in the *Details* list then press the **Add** button several times.


This populates the preview with some print bands to view. Because you can have many bands of various types within a single report, you have to select which to see before going to print preview. This way, the Report Designer knows what to compose on the screen.

3. Press the **OK** button.
4. When done "previewing," press the **Close** button.
5. Press the **Save and Close** button to return to the Procedure Properties dialog.
6. Press the **Save and Close** button to close the Report Designer Editor dialog.
7. Choose **File ► Save**, or press the Save button  on the tool bar to save your work.

An Invoice Report


Next, we will create one of the most common types of reports. An invoice will make use of most of the tables in the data dictionary, demonstrating how to create group breaks and totals. It will also show you how to control pagination based on group breaks.

Creating the Report

1. Highlight the *InvoiceReport* procedure.
2. Press the **Properties**  button.
3. Select the **Defaults** tab, then highlight *Report (Paper size Letter – Portrait)* in the *Select Procedure Type* dialog, then press the **Select** button.

The *Procedure Properties* dialog appears.

Specify the tables for the Report

1. If not already opened, open and select the **Data / Tables Pad**.
2. Highlight the *<ToDo>* folder, then press the **Add** button on the Pad toolbar.
3. Select the *Customer* table from the *Select* dialog, then press the **Select** button.
4. Highlight the *Customer* table, and press the **Change**  button.
5. Highlight *KeyCustNumber* in the *Select Key from CUSTOMER* dialog, then press the **Select** button.

The report will process all the Customer table rows in CustNumber order.

6. Highlight the *Customer* table, then press the **Add** button.
7. Select the *Orders* table from the **Related Tables** tab, then press the **Select** button.

It will process all the Orders for each Customer.

8. Highlight the *Orders* table, then press the **Add** button.
9. Select the *Detail* table from the **Related Tables** tab, then press the **Select** button.

Each Order will print all the related Detail rows.

10. Highlight the *Detail* table, then press the **Add** button.
11. Select the *Products* table from the **Related Tables** tab, then press the **Select** button.

Each Detail row will lookup the related Products table row.

Populating the Report Form Band

The Report Form band prints once for each page in the report. Its content is only composed once, when the report is opened. This makes it useful for constant information that will always be on every page of the report.

Place a string constant

1. From the *Procedure Properties* window, press the **Report** tab, and then press the **Designer** button to reenter the Report Designer.
2. With the Report Designer active, choose **View ► Toolbox** (or press CTRL + ALT + X).
3. Select the STRING control in the Toolbox, and DRAG the control to the top middle of the Report Form band.
4. RIGHT-CLICK on the control just populated, and select **Properties...** from the popup menu.

The *Property Pad* now has focus.

5. Type *Invoice* in the **Text** property entry.
6. Locate the **TextFont** property, and press the ellipsis button to the right.
7. Select a font, style, and size to use for the text (something large and bold would be appropriate for this).
8. Press the **OK** button to close the *Select Font* dialog.
9. If needed, resize the control so that it's large enough to hold the text, by dragging its handles.

Place the next string constant

1. Select the STRING control in the Toolbox, and DRAG the control just below the last string you placed.
2. RIGHT-CLICK on the control, then choose **Properties** from the popup menu.
3. Type the name of your company in the **Text** entry.
4. Press the **Font** button and select a font, style, and size to use for the text (something just a little smaller than the previous column would be appropriate for this).
5. Press the **OK** button to close the *Select Font* dialog.
6. If needed, resize the control so that it's large enough to hold the text, by dragging its handles.

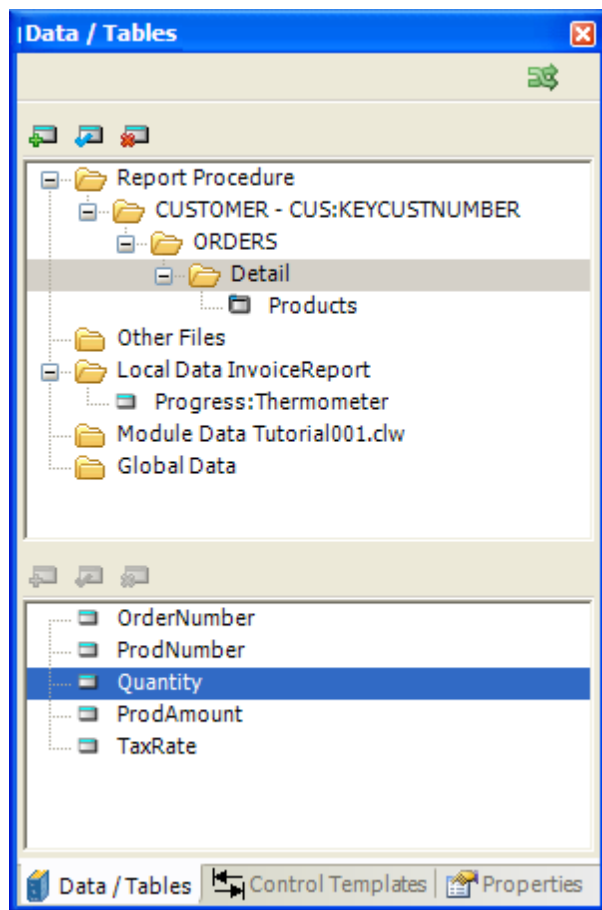


Populating the Detail Band

The Detail band will print every time new information is read from the lowest level "Child" table in the Table Schematic. For this Invoice report, the lowest level "Child" table is the *Detail* table (remember that *Products* is a Many to One "lookup" table from the *Detail* table).

In the last report, we used the **Populate > Multiple Fields** option. In this section, we will use the **Data / Tables Pad** as an alternate design technique.

1. If not already opened, open and select the *Data / Tables Pad*.
2. Highlight *Detail* in the *Tables* list then select *Quantity* in the *Columns* list and DRAG the column inside the Detail band, and DROP near its top left corner.



3. Back in the *Data / Tables Pad*, highlight *ProdNumber* in the *Columns* list and DRAG the column and DROP directly to the right of the first control.

4. Highlight *Products* in the *Tables* list then DRAG *ProdDesc* in the *Columns* list and DROP it to the right of the control just placed.
5. Highlight *Detail* in the *Tables* list then DRAG *ProdAmount* in the *Columns* list and DROP it to the right of the control just placed.
6. Highlight *LOCAL DATA InvoiceReport* in the *Tables* list, then press the **Add** button.

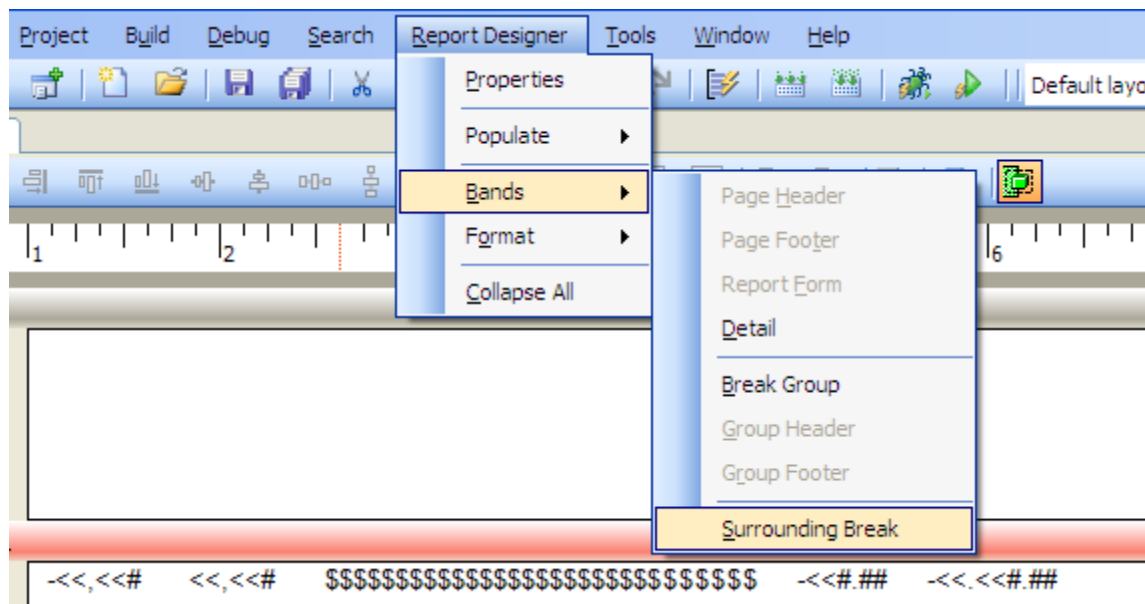
This local variable will be used to display the total price for each line item.

7. Type *LOC:ItemTotal* in the **Column Name** entry.
8. Select *DECIMAL* from the **Data Type** drop list.
9. Type 7 in the **Characters** entry.
10. Type 2 in the **Places** entry then press the **OK** button.
11. Highlight *LOCAL DATA InvoiceReport*, and DRAG *LOC:ItemTotal* and DROP it to the right of the last control placed.
12. Move all the controls to the top of the Detail band, aligned horizontally, then resize the band so it is just a little taller than the controls. Also, modify the properties of the *DTL:ProdAmount* and *LOC:ItemTotal* to change the **Justification** to *Right Justified* and the Offset to 2.

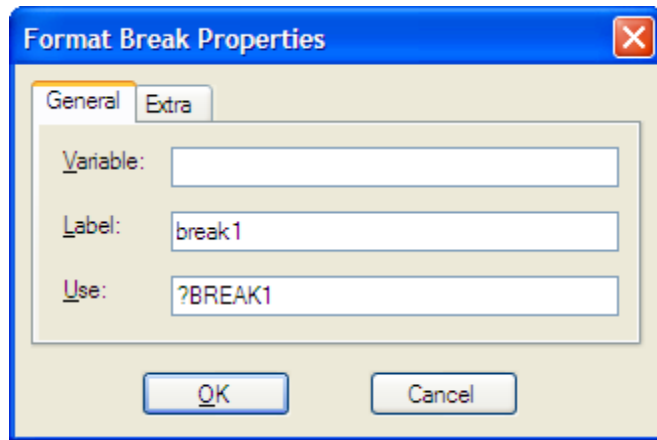
Adding Group Breaks

We need to print different information on the page for each Invoice. Therefore, we need to create BREAK structures to provide the opportunity to print something every time the *Orders* table information changes and every time the *Customer* table information changes.

1. CLICK on the *Detail* band, and then choose **Report Designer ► Bands ► Surrounding Break**.



The *Format Break Properties* dialog appears.



2. In the **Variable** entry, type *CUS:CustNumber*.
3. Type *CustNumberBreak* in the **Label** entry then press the **OK** button.

A Break (*CUS:CustNumber*) band appears above the Detail band, which appears indented, meaning it is within the Break structure.

4. CLICK on the Detail band, and then select **Report Designer ► Bands ► Surrounding Break**.

The *Format Break Properties* dialog appears.

5. In the **Variable** entry, type *ORD:OrderNumber*.
6. Type *OrderNumberBreak* in the **Label** entry then press the **OK** button.

Create the group Headers and Footers

1. CLICK on the *Break (ORD:OrderNumber)* band, and then select **Report Designer ► Bands ► Group Header**.

The Group Header (*ORD:OrderNumber*) band appears above the Detail band. This band will print every time the value in the *ORD:OrderNumber* column changes, at the beginning of each new group of rows. We will use this to print the company name, address, along with the invoice number and date.

2. CLICK on the *Break (ORD:OrderNumber)* band, and then select **Report Designer ► Bands ► Group Footer**.

The *Group Footer (ORD:OrderNumber)* band appears below the Detail band. This band will print every time the value in the *ORD:OrderNumber* column changes, at the end of each group of rows. We will use this to print the invoice total.

3. RIGHT-CLICK on the *Group Footer (ORD:OrderNumber)* band then choose **Properties** from the popup menu.

The *Properties Pad Group Footer* band now has focus.

4. Enter *1* in the **PageAfter** property.

This causes the print engine to print this band, then initiate Page Overflow. This will compose the Page Footer band, issue a form feed to the printer, then compose the Page Header band for the next page.

5. CLICK on the *Break (CUS:CustNumber)* band, and select **Report Designer ► Bands ► Group Footer**.

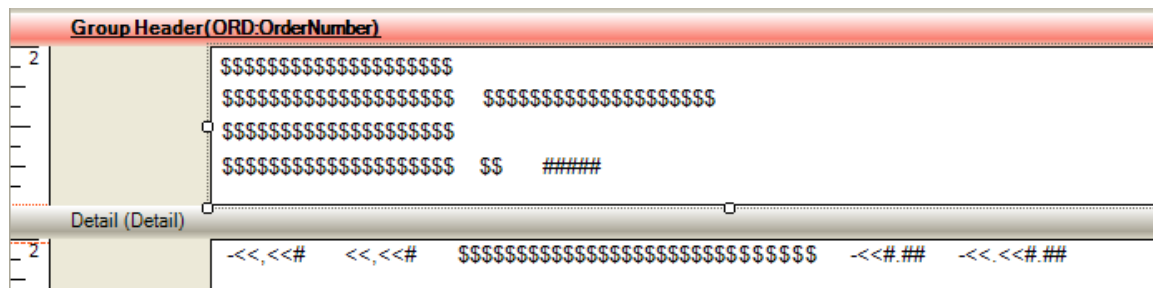
The *Group Footer (CUS:CustNumber)* band appears below the *Group Footer (ORD:OrderNumber)* band. This band will print every time the value in the CUS:CustNumber column changes, at the end of each group of rows. We will use this to print invoice summary information for each company.

6. RIGHT-CLICK on the *Group Footer (CUS:CustNumber)* band then choose **Properties** from the popup menu.
7. Enter 1 in the **PageAfter** property.

Populating the Group Header Band

Place the Customer table columns

1. Open the *Data / Tables Pad* (if not already opened).
2. Highlight *Customer* in the *Tables* list then DRAG *Company* in the *Columns* list and DROP inside the *Group Header (ORD:OrderNumber)* band, near its top left corner.
3. Back in the *Data / Tables Pad*, DRAG *FirstName* in the *Columns* list and DROP it inside the *Group Header (ORD:OrderNumber)* band, just below the first control.
4. DRAG *LastName* in the *Columns* list and DROP inside the *Group Header (ORD:OrderNumber)* band, to the right of the control you just placed.
5. DRAG *Address* in the *Columns* list and DROP inside the *Group Header (ORD:OrderNumber)* band, below the second control you placed.
6. DRAG CUS:City in the *Columns* list and DROP inside the *Group Header (ORD:OrderNumber)* band, below the last control you placed.
7. DRAG *State* in the *Columns* list, and DROP inside the *Group Header (ORD:OrderNumber)* band, to the right of the previously placed control.
8. DRAG *ZipCode* in the *Columns* list, and DROP inside the *Group Header (ORD:OrderNumber)* band, to the right of the previously placed control.

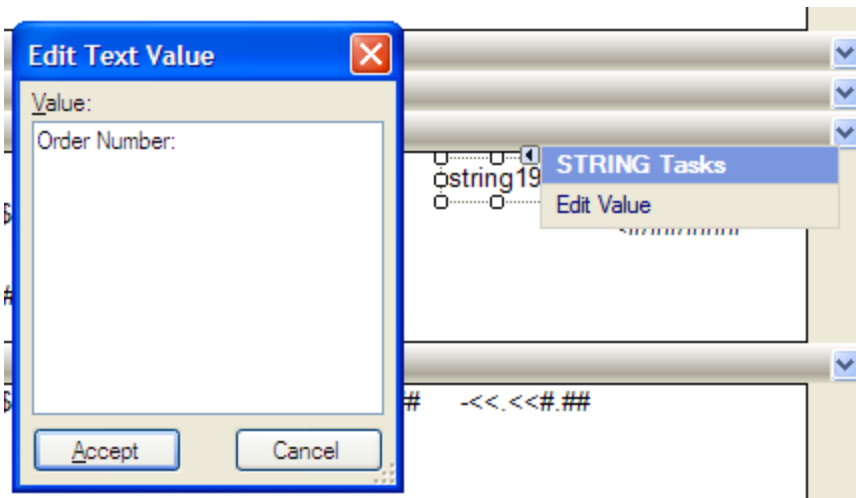


Place the Orders table columns

1. Highlight *Orders* in the *Tables* list then DRAG *OrderNumber* in the *Columns* list and DROP inside the *Group Header (ORD:OrderNumber)* band, near its top right corner.
2. DRAG *OrderDate* in the *Columns* list, then inside the *Group Header (ORD:OrderNumber)* band, below the last control you placed.

Place the constant text and column headings

1. Open the Control ToolBox if not opened already. DRAG the **STRING** from the *Controls* toolbox and DROP inside the Group Header (*ORD:OrderNumber*) band, left of the *ORD:OrderNumber* control you placed.
2. As the control is populated, select the **Edit Value** "smart link" and enter *Order Number:* in the *Edit Text Value* dialog, and press the **Accept** button..



3. Again, from the Toolbox Control Pad, DRAG the **STRING** from the *Controls* toolbox and DROP inside the Group Header (*ORD:OrderNumber*) band, left of the *ORD:OrderDate* control you placed.
4. As the control is populated, select the **Edit Value** "smart link" and enter *Order Date:* in the *Edit Text Value* dialog, and press the **Accept** button.
5. Again, from the Toolbox Control Pad, DRAG the **STRING** from the *Controls* toolbox and DROP inside the Group Header (*ORD:OrderNumber*) band, at the left end below the Customer table controls you placed.
6. As the control is populated, select the **Edit Value** "smart link" and enter *Quantity:* in the *Edit Text Value* dialog, and press the **Accept** button.
7. From the Toolbox Control Pad, DRAG the **STRING** from the *Controls* toolbox and DROP inside the Group Header (*ORD:OrderNumber*) band, inside the Group Header (*ORD:OrderNumber*) band, to the right of the last string you placed.
8. As the control is populated, select the **Edit Value** "smart link" and enter *Product:* in the *Edit Text Value* dialog, and press the **Accept** button.
9. From the Toolbox Control Pad, DRAG the **STRING** from the *Controls* toolbox and DROP inside the Group Header (*ORD:OrderNumber*) band, inside the Group Header (*ORD:OrderNumber*) band, to the right of the last string you placed, directly above the *DTL:ProdAmount* control in the Detail band.
10. As the control is populated, select the **Edit Value** "smart link" and enter *At:* in the *Edit Text Value* dialog, and press the **Accept** button.
11. From the Toolbox Control Pad, DRAG the **STRING** from the *Controls* toolbox and DROP inside the Group Header (*ORD:OrderNumber*) band, inside the Group Header (*ORD:OrderNumber*) band, to the right of the last string you placed, directly above the *LOC:ItemTotal* control in the Detail band.

12. As the control is populated, select the **Edit Value** "smart link" and enter *Item Total* in the *Edit Text Value* dialog, and press the **Accept** button.

Group Header(ORD:OrderNumber)	
<<, <<#	Order Number: <<, <<#
Order Date: <#/#/#/#/#/#	
Quantity: Product:	At: Item Total:

Detail (Detail)	
<<, <<#	<<, <<#

Place a thick line under the column headings

1. From the Toolbox Control Pad, DRAG the **LINE** from the *Controls* toolbox and DROP inside the Group Header (*ORD:OrderNumber*) band, inside the Group Header (*ORD:OrderNumber*) band, under the *Quantity* string you placed.
2. Resize the line by dragging its handles until it appears to be a line all across the report under the column headers.
3. In the *Properties Pad*, select the line control from the drop list as shown:

Properties	
LINE (Use = '?LINE1')	
<div> <div>Color</div> <div> <div>LineColor</div> <div></div> </div> </div>	
<div> <div>Design</div> <div> <div>Locked</div> <div>False</div> </div> <div> <div>TabIndex</div> <div>17</div> </div> </div>	
<div> <div>Extra</div> <div> <div>Extend</div> <div></div> </div> </div>	
<div> <div>General</div> <div> <div>Layout</div> <div>Default</div> </div> <div> <div>LineWidth</div> <div>5</div> </div> <div> <div>Use</div> <div>?LINE1</div> </div> </div>	
<div> <div>Mode</div> <div> <div>Disable</div> <div>False</div> </div> <div> <div>Hide</div> <div>False</div> </div> </div>	
<div> <div>Position</div> <div> <div>AT</div> <div>AT(52,1052,5573,-9)</div> </div> </div>	

4. Type 5 in the **LineWidth** property.
This makes the line much thicker.

Populating the Invoice Group Footer Band

Place the constant text and total column

1. From the Toolbox Control Pad, DRAG the **STRING** from the *Controls* toolbox and DROP in the middle of the Group Footer (*ORD:OrderNumber*) band.
2. As the control is populated, select the **Edit Value** "smart link" and enter *Order Total:* in the *Edit Text Value* dialog, and press the **Accept** button.
3. From the Toolbox Control Pad, DRAG the **STRING** from the *Controls* toolbox and DROP inside the Group Footer (*ORD:OrderNumber*) band, to the right of the string you just placed.
4. RIGHT-CLICK and choose **Properties** from the popup menu.
5. In the *Properties Pad*, set the **VariableString** property to *TRUE*.
6. Enter *LOC:ItemTotal* in the **Use** property (or press the ellipsis to select the column).
7. Type *@N9.2* in the **Text** property.
8. Select *Sum* in the **TotalType** property drop list.
9. Select *OrderNumberBreak* from the **Reset** property drop list.

This will add up all the *LOC:ItemTotal* contents for the Invoice and will reset to zero when the value in the *ORD:OrderNumber* column changes.

Place a line above the total

1. From the Toolbox Control Pad, DRAG the **LINE** from the *Controls* toolbox and DROP inside the Group Footer (*ORD:OrderNumber*) band, above the controls you just placed.
2. Resize the line by dragging its handles until it appears to be above both the controls you just placed.
3. In the *Properties Pad*, type 5 in the **LineWidth** entry.

This makes the line just a little bit thicker.

Populating the Customer Group Footer Band

Place the constant text

1. From the Toolbox Control Pad, DRAG the **STRING** from the *Controls* toolbox and DROP in the middle of the Group Footer (*CUS:CustNumber*) band.
2. As the control is populated, select the **Edit Value** "smart link" and enter *Invoice Summary for:* in the *Edit Text Value* dialog, and press the **Accept** button.
3. From the Toolbox Control Pad, DRAG the **STRING** from the *Controls* toolbox and DROP inside the Group Footer (*CUS:CustNumber*) band, below the string you just placed.
4. As the control is populated, select the **Edit Value** "smart link" and enter *Total Orders:* in the *Edit Text Value* dialog, and press the **Accept** button.

Enter additional local data

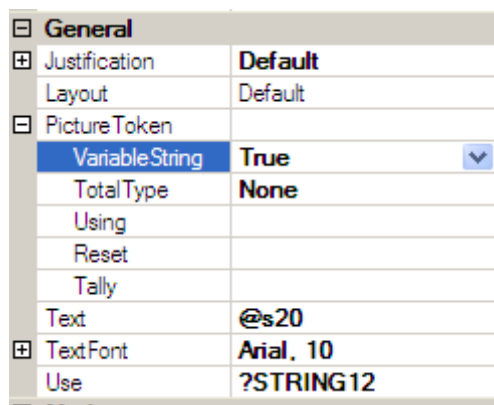
1. Open and select the **Data / Tables Pad**.
2. Highlight *LOCAL DATA InvoiceReport* in the *Tables* list, then press the **Add** button.
3. Type *LOC:InvoiceCount* in the **Column Name** entry.

This is a column that will print the number of invoices printed for an individual company.

4. Select *LONG* from the Data Type drop list.
5. Type *@N3* in the **Screen Picture** entry, then press the **OK** button.

Place the total columns

1. From the Toolbox Control Pad, DRAG the **STRING** from the *Controls* toolbox and DROP inside the Group Footer (*CUS:CustNumber*) band, just right of the string *Total Orders:*.
2. RIGHT-CLICK and choose **Properties** from the popup menu.
3. Set the **VariableString** property to TRUE.



4. Type *LOC:InvoiceCount* in the **Use** property entry.
5. Type *@N3* in the **Text** property entry.
6. Select *Count* from the **TotalType** property drop list.
7. Select *CustNumberBreak* from the **Reset** property drop list.

This is the same type of total column that we placed in the *ORD:OrderNumber* group footer, but it will only reset when *CUS:CustNumber* changes.

8. Select *OrderNumberBreak* in the **Tally** property list, and **press the Enter key to select it**.

This total column will count the number of invoices that print for each customer. The **Tally** list allows you to select the point(s) at which the total increments. By selecting *OrderNumberBreak* from the list, the count will only increment when a new invoice begins.

9. From the Toolbox Control Pad, DRAG the **STRING** from the *Controls* toolbox and DROP inside the Group Footer (*CUS:CustNumber*) band, to the right of the string you just placed.

10. RIGHT-CLICK and choose **Properties** from the popup menu to bring focus to the *Properties Pad*.
11. Set the **VariableString** property to TRUE.
12. Type *LOC:ItemTotal* in the **Use** property entry.
13. Select *Sum* from the **TotalType** drop property list.
14. Select *CustNumberBreak* from the **Reset** property drop list.

This is the same type of total column that we placed in the *ORD:OrderNumber* group footer, but it will only reset when *CUS:CustNumber* changes.

Place the display column then exit

1. Open and select the **Data / Tables Pad**.
2. Highlight *Customer* in the *Tables* list then DRAG *Company* in the *Columns* list and DROP inside the Group Footer (*CUS:CustNumber*) band, just right of the *Invoice Summary for:* string you placed.

Your report design is now complete! Resize and tighten up the bands as needed.

Here is your report so far:

The screenshot shows a report design window with the following bands and content:

- Break (ORD:OrderNumber)**: A horizontal line separating the header from the detail.
- Group Header(ORD:OrderNumber)**: Contains fields for Order Number, Order Date, and a line for Quantity and Product.
- Detail (Detail)**: A band for individual items, showing fields for Order Number, Item Number, and Item Total.
- Group Footer(ORD:OrderNumber)**: Contains the Order Total field.
- Group Footer(CUS:CustNumber)**: A red-shaded band containing the Invoice Summary for and Total Orders fields.
- Page Footer**: A band for page information.
- Report Form**: The final output area, showing the title "Invoice" and the company name "SoftVelocity Inc."

3. Press the **Save and Close** button to return to the *Procedure Properties* dialog.

Adding a Formula

To make the *ItemTotal* column contain the correct amount for each Detail row in the invoice, you need to add a Formula to the procedure.

1. Press the **Formulas** tab in the *Procedure Properties* dialog.
2. Press the **Insert** button in the *Formula Editor* dialog.

The *Formula Editor* design dialog appears.

3. Type *Item Total Formula* in the **Name** entry.
4. In the **Class** entry drop list, select *Before Print Detail* in the *Template Classes* list.

The *Before Print Detail* class tells the Report template to perform the calculation each time it gets ready to print a Detail.

5. Press the ellipsis (...) button for the **Result** entry.
6. Highlight *LOCAL DATA InvoiceReport* in the *Tables* list, select *LOC:ItemTotal* from the *Columns* list, then press the **Select** button.
7. Press the **Data** button in the **Operands** group.
8. Highlight the *Detail* table in the *Tables* list, select *DTL:Quantity* from the *Columns* list, then press the **Select** button.

This places the *DTL:Quantity* column in the **Statement** entry for you. The Statement entry contains the expression being built, and you can also type directly into it to build the expression, if you wish.

9. Press the * button in the **Operators** group.
10. Press the **Data** button in the **Operands** group.
11. Highlight the *Detail* table in the *Tables* list, select *DTL:ProdAmount* from the *Columns* list, then press the **Select** button.
12. Press the **Check** button to check the expression's syntax.
13. Press the **Save and Close** button to close the *Formula Editor* design window.

Row Filters vs Inner Joins

We want to make sure the report only prints invoices for the companies that have orders. You could add a row filter that would work by following these steps (but don't actually do them):

1. Press the **Properties** tab in the *Procedure Properties* dialog.
2. Press the **Actions** button.
3. Press the **Report Properties** button.

The *Report Properties* dialog appears.

4. Type *ORD:OrderNumber <> 0* in the **Record Filter** entry.
5. Press the **OK** button to close the *Report Properties* dialog, then press **OK** again to return to the *Procedure Properties* dialog.

This will eliminate all the customers who have not ordered anything. Internally, the Report Template generates a VIEW structure for you. This VIEW structure, by default, performs an "outer join" on the tables you placed in the Table Schematic. "Outer join" is a standard term in Relational Database theory—it just means that the VIEW will retrieve all Parent table rows, whether there are any related Child table rows or not.

If it retrieves a Parent row without a Child, the columns in the Child table are all blank or zero, while the Parent table's columns contain valid data. Therefore, this is the condition for which we test.

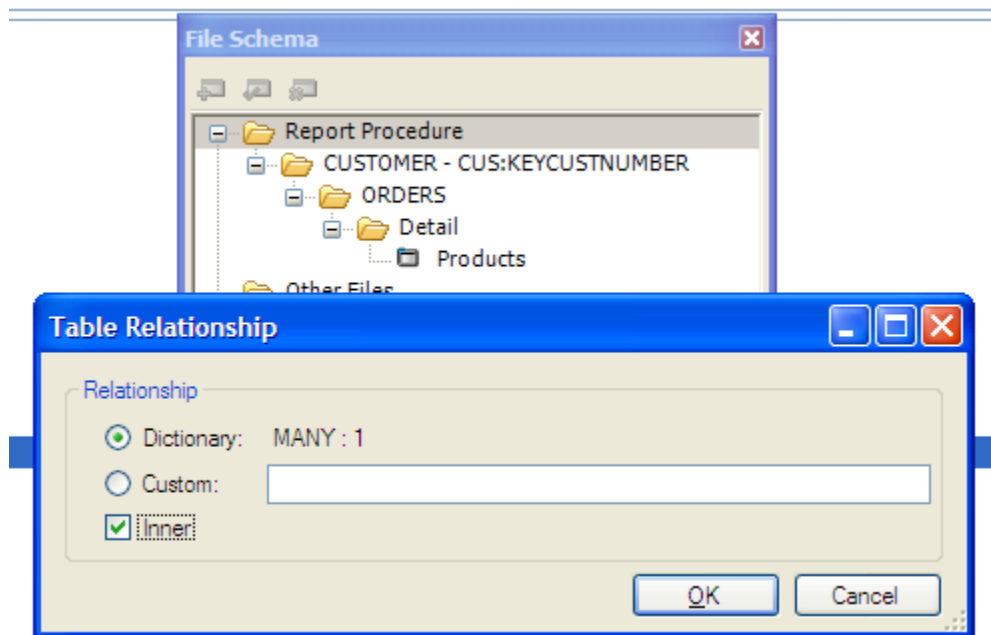
`ORD:OrderNumber <> 0` checks to see if the `ORD:OrderNumber` column has any value in it other than zero. Since `ORD:OrderNumber` is the key column in the Orders table that creates the link to the related Customers table row, it must contain a value if there are existing Orders table rows for the current Customer.

If `ORD:OrderNumber` does not contain a value other than zero, the current Customers table row is skipped ("filtered out"). This eliminates printing Parent rows without related Children (in this case, any Customers without Orders).

This means that a filter would work. However, since the VIEW structure can do the filtering required, there is a better way. This is the "inner join". What this means is that there is a Customer row only if there is a related Order row. This makes the VIEW smaller, and thus more efficient than a filter.

To use this feature, follow these steps:

1. Open the **Data / Tables Pad**
2. Select the *Orders* table and press the **Change** button.
3. Check the **Inner** box, then press the **OK** button.
4. Press the **OK** button to close this dialog.




Change the Progress Window

1. Press the **Window** tab in the *Procedure Properties* dialog, and the **Designer** button to enter the Window Designer.
2. Click on the title bar of the window, and press the **F4 key** to bring focus to the *Properties Pad*.
3. Type *Invoice Progress* in the **Title** property.

4. Press **Save and Close** to return to the *Procedure Properties* dialog.

Exit and Save

1. Press the **Save and Close** button in the *Procedure Properties* dialog to close it.
2. Choose **File ► Save**, or press the Save button  on the toolbar.

A Range Limited Report

Next, we will limit the range of rows that will print.

Creating the Report

1. From the Application Tree, highlight the *InvoiceReport* procedure.
2. Choose **Application ► Copy Procedure**.
3. The *New Procedure* dialog appears.
4. Type *CustInvoiceReport* in the entry box, then press the **OK** button.

The copied procedure appears in the application tree, replacing its "ToDo".

Modify the new report

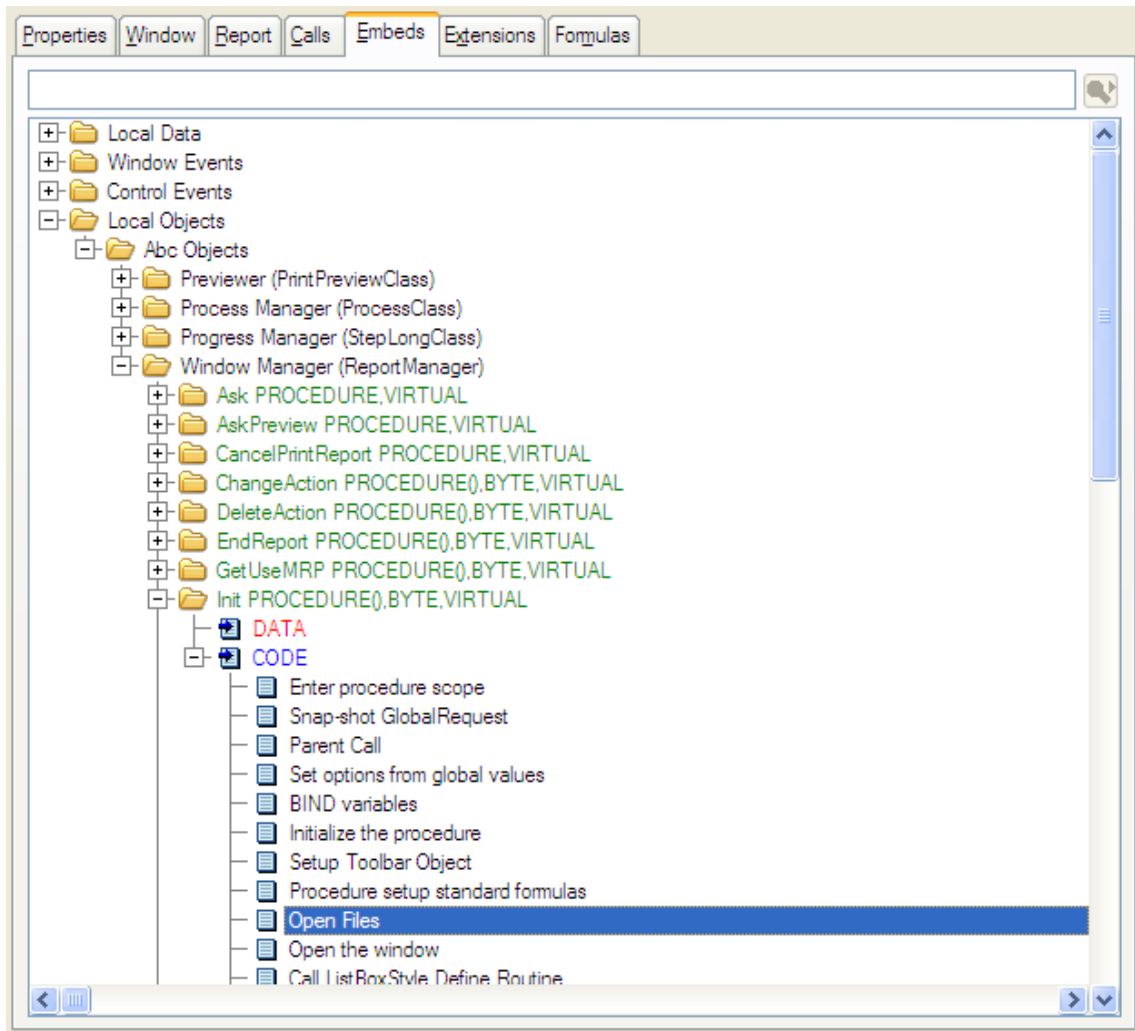
1. Highlight the *CustInvoiceReport* procedure.
2. RIGHT-CLICK and choose **Embeds** from the popup menu.
3. Press the **Contract All** button.

This will make it easier to locate the specific embed point you need.

4. Locate the *Local Objects* folder, then CLICK on its **+** sign to expand its contents. Do the same for the *ABC Objects* folder that appears under it.

The ABC templates generate object-oriented code. Each procedure instantiates a set of objects which are derived from the ABC Library. The *Local Objects* folder shows you all the object and methods that you can override for the procedure by simply embedding your own code to enhance the ABC functionality. See the *Easing Into OOP* and *Object Oriented Programming* articles in the *Clarion Language Programming* PDF for more on this powerful technique.

5. Locate the *WindowManager* folder, then CLICK on it to expand it.
6. Locate the *Init PROCEDURE(), BYTE, VIRTUAL* folder, then CLICK on it to expand it.
7. Locate the *CODE* folder, then CLICK on it to expand it.
8. Highlight *Open Files* then press the **Insert** button.




This embed point is at the beginning of the procedure, before the report has begun to process. It's important that the tables for the report already be open because we will call another procedure for the user to select a Customer row. If the tables for the report weren't already open, the procedure we call would open the Customer table for itself then close it again and we would lose the data that we want to have for the report. This has to do with multithreading and the Multiple Document Interface (MDI)—see THREAD in the Language Reference for more on this.

9. Highlight *Source* then press the **Select** button to call the Text Editor.

10. Type in the following code:

```
GlobalRequest = SelectRecord
```

This code sets up a Browse procedure to select a row (it enables the Browse procedure's Select button).

11. Press the **Save and Close** button  to return to the *Embedded Source* dialog.
12. Highlight the *SOURCE* you just added then press the **Insert** button.
13. Highlight *Call a procedure* then press the **Select** button.
14. Select *BrowseCustomers* in the list, then press the **OK** button.

This will generate a procedure call to the *BrowseCustomers* Browse procedure to allow the user to select which Customer's Invoices to print.

Notice that there are now two entries displayed under the embed point. At each embed point you can place as many items as you want, mixing Code Templates with your own SOURCE or PROCEDURE Calls. You can also move the separate items around within the embed point using the arrow buttons, changing their logical execution order (the first displayed is the first to execute). Note well that moving them will change the assigned **Priority** option setting for the moved item if you attempt to move a higher priority item in front of another with a lower priority setting.

15. Press the **Save and Close** button to return to the *Procedure Properties* dialog.

Set the Range Limit

1. Press the **Properties** button, and then press the **Actions** button.
2. In the Properties dialog, press the **Report Properties** button.

The *Report Properties* dialog appears. This dialog allows you to set either Row Filters or Range Limits (along with Hot Columns and Detail Filters).

Row Filters and Range limits are very similar. A Row Filter is a conditional expression to filter out unwanted rows from the report, while a Range Limit limits the rows printed to only those matching specific key column values. They can both be used to create reports on a subset of your tables, but a Range Limit requires a key and a Row Filter doesn't. This makes a Row Filter completely flexible while a Range Limit is very fast. You can use both capabilities if you want to limit the range then filter out unneeded rows from that range.

3. Select the **Range Limits** tab.
4. Press the ellipsis (...) button for the **Range Limit Field**.

The *CUS:CustNumber* control, the only logical choice, is automatically populated..

5. Leave *Current Value* as the **Range Limit Type** then press the **OK** button.

Current Value indicates that whatever value is in the column at the time the report begins is the value on which to limit the report. Since the user will choose a Customer row from the *BrowseCustomer* procedure, the correct value will be in the *CUS:CustNumber* column when the report begins.

Exit and Save

1. Press the **OK** button in the *Procedure Properties* dialog to close it.
2. Press the **Save and Close** button to return to the Application Tree.

A Single Invoice Report

Next, we will print a single invoice from the Browse list of orders.

Creating the Report

1. In the Application Tree, highlight the *CustInvoiceReport* procedure.


2. Choose **Application** ► **Copy Procedure**.

The *New Procedure* dialog appears.

3. Type *SingleInvoiceReport* in the entry box, then press the **OK** button.
4. Press the **Same** button in the *Procedure name clash* dialog.

The copied procedure appears unattached at the bottom of the application tree. We'll "connect the lines" after we finish with the report.

Delete the embed code

1. Highlight the *SingleInvoiceReport* procedure.
2. RIGHT-CLICK and choose **Embeds** from the popup menu.
3. Press the **Next Filled Embed** button .
4. Press the **Delete** button.
5. Answer *Yes* to the **Are you sure?** question.
6. Press the **Delete** button again and answer *Yes* to the **Are you sure?** question.
7. Press the **Save and Close** button.

Change the Table Schematic

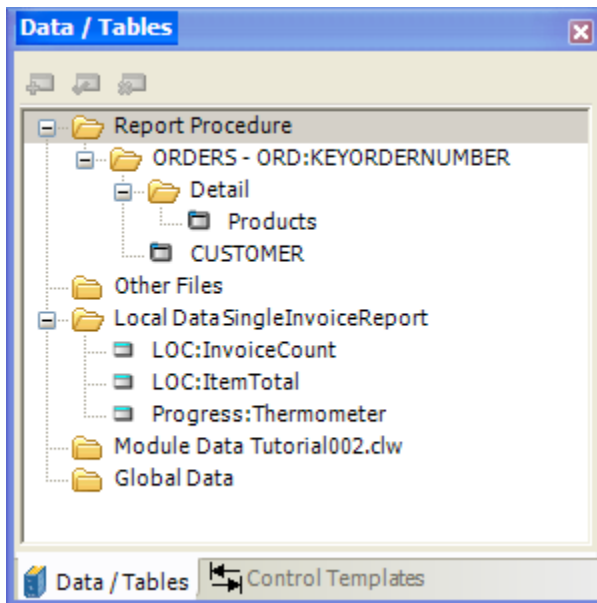
First, we need to change the order of the tables in the Table Schematic. We'll end up with all the same tables, but instead of the *Customer* table as the Primary table (first table in the Table Schematic), we need the *Orders* table to be the Primary table for the procedure so we can easily limit the range to a single invoice.

1. Highlight the *SingleInvoiceReport* procedure, and select the **Data / Tables Pad**.
2. Highlight the *Customer* table then press the **Delete** button.

This causes all the tables to disappear.

3. Highlight the *<ToDo>* folder, then press the **Add** button.
4. Select the *Orders* table from the *Select* dialog, then press the **Select** button.
5. Press the **Change** button.
6. Highlight *KeyOrderNumber* in the *Select Keyfrom Orders* dialog, then press the **Select** button.
7. Highlight the *Orders* table, then press the **Add** button.
8. Select the *Detail* table from the **Related Table** tab, then press the **Select** button.
9. Highlight the *Detail* table, then press the **Add** button.
10. Select the *Products* table from the **Related Table** dialog, then press the **Select** button.
11. Highlight the *Orders* table again, then press the *Insert* button.
12. Select the *Customer* table from the **Related Table** dialog, then press the **Select** button.

We've selected all the same tables, but now the Primary table is the *Orders* table and the related *Customer* table row will be looked up. This is important, because we need to limit this report to a single invoice and that would be much more difficult to do if the *Customer* table were the Primary.



Set the Range Limit

1. Press the **Properties** button, and then press the **Actions** button.
2. In the Properties dialog, press the **Report Properties** button.
3. Select the **Range Limits** tab.
4. Press the ellipsis (...) button for the **Range Limit Field**.

The *ORD:OrderNumber* control is automatically selected as the only logical choice.

5. Leave *Current Value* as the **Range Limit Type** then press the **OK** button.
6. Press the **OK** button to return to the *Procedure Properties* window.

Current Value indicates that whatever value is in the column at the time the report begins is the value on which to limit the report. Since the user will run this report from the *BrowseOrders* procedure, the correct value will be in the *ORD:OrderNumber* column when the report begins.

Modify the new report


Now we need to change the report itself, to only print a single invoice.

1. Press the **Report** tab, and press the **Designer** button to enter the Report Designer.
2. CLICK on the Break(*CUS:CustNumber*) band and press the **Delete** key.

This removes not only the Group Break, but also the Group Footer that was associated with it.

3. Choose **Save and Close** to return to the *Procedure Properties* dialog.


Exit and Save

1. Press the **Save and Close** button in the *Procedure Properties* dialog to close it.
2. Choose **File ► Save**, or press the Save button  on the toolbar to save your work.

Connect the Lines

1. Highlight the *BrowseOrders* procedure.
2. RIGHT-CLICK and choose **Window** from the popup menu.
3. From the **Control Templates** Pad, DRAG *BrowsePrintButton* > *Browse on ORDERS* to the right of the **Delete** button to place the new button control..
4. RIGHT-CLICK on the new button and choose **Properties** from the popup menu.
5. Type *&Print Invoice* into the **Text** property entry.
6. Type *?PrintInvoice* into the **Use** property entry.
7. RIGHT-CLICK on the new button and choose **Actions** from the popup menu.
8. Select *SingleInvoiceReport* from the **Report Procedure** drop list.

This Control Template is specifically designed to run a range-limited report based on the currently highlighted row in the list box we selected (Browse on Orders). The *Orders* table row buffer will contain the correct value to allow the *Current Value* Range Limit on the *SingleInvoiceReport* to work. It also automatically adds this button's action to the popup menu for the browse.

9. Press the **OK** button.
10. Press the **Save and Close** button to return to the Procedure Properties.
11. Press the **Save and Close** button to return to the Application Tree.
12. Choose **File ► Save**, or press the Save button  on the toolbar to save your work.

OK, What Did I Just Do?

Here's a quick recap of what you just accomplished:

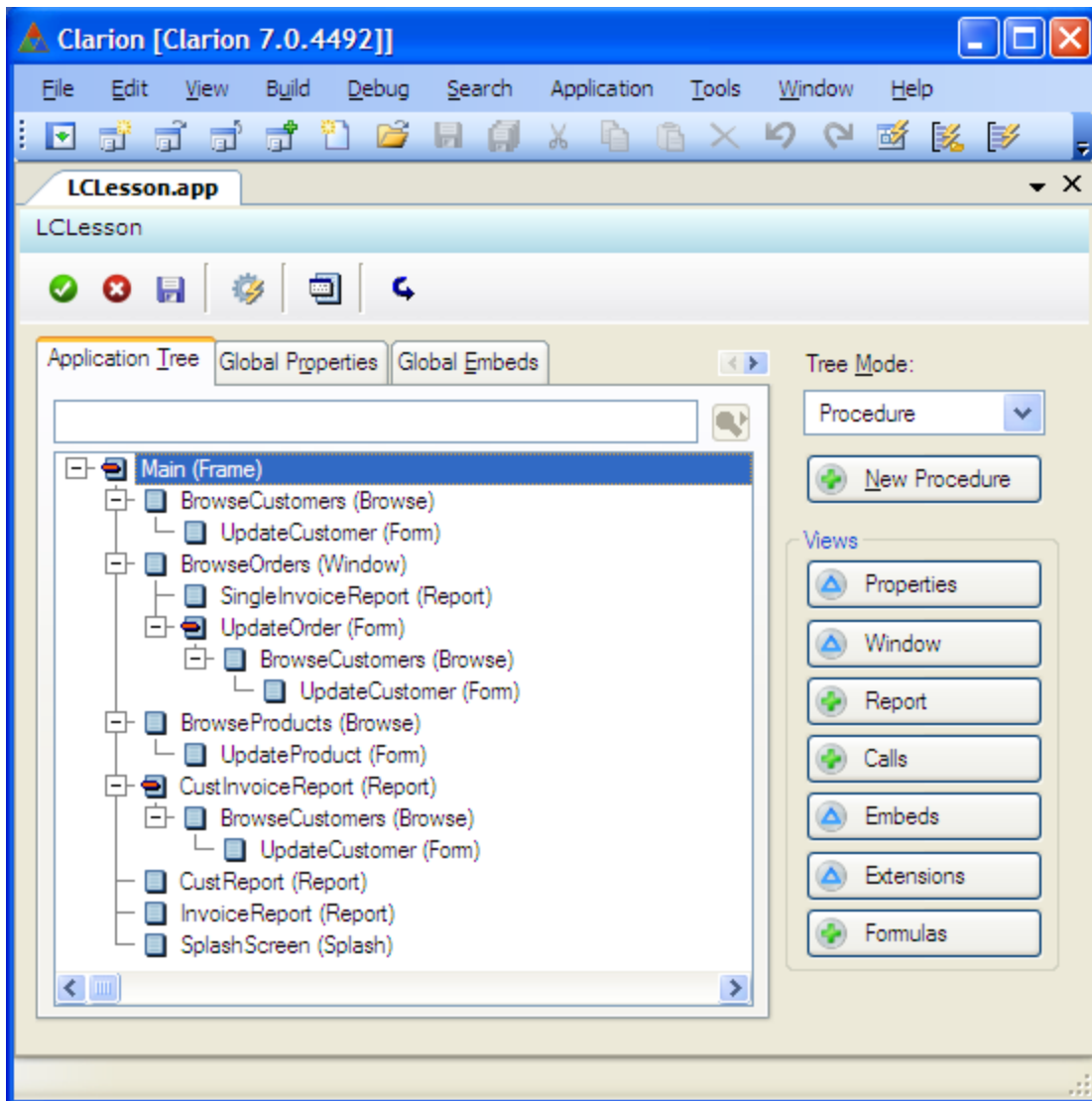
- ✓ You added several menu items to your main menu.
- ✓ You created a simple Customer List report.
- ✓ You created a relational report to print all Invoices.
- ✓ You range-limited a report to print Invoices for a single customer.
- ✓ You range-limited a report to print a single Invoice from the current row highlighted in a Browse list.

Now we'll look at where to go next.

What's Next?

Congratulations, you made it to the end of the Application Generator lesson!

Here is the completed application, created from scratch and without the help of the wizards:



Welcome to the growing community of Clarion developers!

While this lesson application is by no means a "shrink-wrap" program, it has demonstrated the normal process of using the Application Generator and all its associated tools to create an application that actually performs some reasonably sophisticated tasks. Along the way, you have used most of Clarion's high-level tool set, and seen just how much work can be done for you without writing source code. You have also seen how just a little embedded source can add extra functionality to the template-generated code, and how you can easily override the default ABC Library classes.

A Short Resource Tour

You have many resources at your disposal to help you with your Clarion programming. Here is a short tour of two of the more important ones that you have right at your fingertips:

1. Choose **Help ► Context Help**.

This is the Contents page for Clarion's extensive on-line Help system.

2. Press the **Contents** tab in the left pane. Examine the FAQ sections.

This opens Clarion's on-line Help file and takes you to a section of commonly asked questions and their answers. This list of topics is the first place you should look whenever you ask yourself any question about Clarion programming that starts with "How do I ... ?" These topics answer many of the most common questions that newcomers to Clarion have, so quite often, you'll find the answer is here.

3. Examine the **Guide to Examples** section.

This topic provides jumps to the discussions of all the example programs that come with Clarion. Here you'll find the various tips, tricks, and techniques that the examples demonstrate so you can adapt them for use in your own programs.

4. Examine the **Whats New?** section.

This topic always gives you the latest, up-to-the-minute information about the most current release of Clarion you have installed. You should always go through this section any time you get a major upgrade or interim release. There are generally a few last-minute details which you will find are only documented in this section. That makes it well worth the reading.

13 - Clarion Language Lesson

Clarion—the Programming Language

The foundation of the Clarion application development environment is the Clarion programming language. Clarion is a 4th Generation Language (4GL) that is both business-oriented and general-purpose. It is business-oriented in that it contains data structures and commands that are highly optimized for data file maintenance and business programming needs. It is also general-purpose because it is fully compiled (not interpreted) and has a command set that is functionally comparable to other 3GL languages (such as C/C++, Modula-2, Pascal, etc.).

By now, you should have completed all the Application Generator lessons in the preceding sections. The purpose of this language lesson is to introduce you to the fundamental aspects of the Clarion language—particularly as it relates to business programming using the Windows event-driven paradigm. Clarion language keywords are in ALL CAPS and this lesson concentrates on explaining the specific use of each keyword and its interaction with other language elements only in the specific context within which it is used. You should always refer to the *Language Reference* for a more complete explanation of each individual keyword and its capabilities.

When you complete this lesson, you will be familiar with:

- The basic structure of a Clarion procedural program.
- The most common event-handling code structure.
- How to compile and link hand-coded programs.

Event-driven Programming

Windows programs are event-driven. The user causes an event by CLICKing the mouse on a screen control or pressing a key. Every such user action causes Windows to send a message (an event) to the program which owns the window telling it what the user has done.

Once Windows has sent the message signaling an event to the program, the program has the opportunity to handle the event in the appropriate manner. This basically means the Windows programming paradigm is exactly opposite from the DOS programming paradigm—the operating system (Windows) tells the program what the user has done, instead of the program telling the operating system what to do.

This is the most important concept in Windows programming—that the user is in control (or should be) at all times. Therefore, Windows programs are reactive rather than proactive; they always deal with what the user has done instead of directing the user as to what to do next. The most common example of this is the data entry dialog. In most DOS programs, the user must follow one path from field to field to enter data. They must always enter data in one field before they can go on to the next (and they usually can only go on to a specific "next" entry field). This makes data validation code simple—it simply executes immediately after the user has left the field.

In Windows programs, the user may use a mouse or an accelerator key to move from control to control, at any given time, in no particular order, skipping some controls entirely. Therefore, data validation code should be called twice to ensure that it executes at least once: once when the user leaves the entry control after entering data, and again when the user presses OK to leave the dialog. If it isn't executed on the OK button, required data could be omitted. This makes Windows programs reactive rather than proactive.

Hello Windows

Traditionally, all programming language lessons begin by creating a "Hello World" type of program—and so does this one.

Starting Point:

The Clarion environment should be open. The LCLESSON.APP should now be closed.

Tip

You should not need to type any code in this lesson. Simply COPY what you need directly from this help file into the appropriate areas.

Create the Source file

1. Using the appropriate tool in your operating system (File Manager, Explorer, etc.), create a new folder called *Language* under the *CLARION7* folder.
2. Return to the Clarion IDE.
3. Choose **File ► New ► Solution, Project or Application**.

The *New Project* dialog appears. It's a special dialog, allowing you to change the directory and type in the filename.

4. Type *Hello* in the **Name** entry.
5. In the **Location** field, select the *\CLARION7\LANGUAGE* folder.
6. Uncheck the **Auto create project subdir** check box.
7. Press the **Create** button.

This creates an empty *HELLO.CLW* (.CLW is the standard extension for Clarion source code files) and places you in the Text Editor.

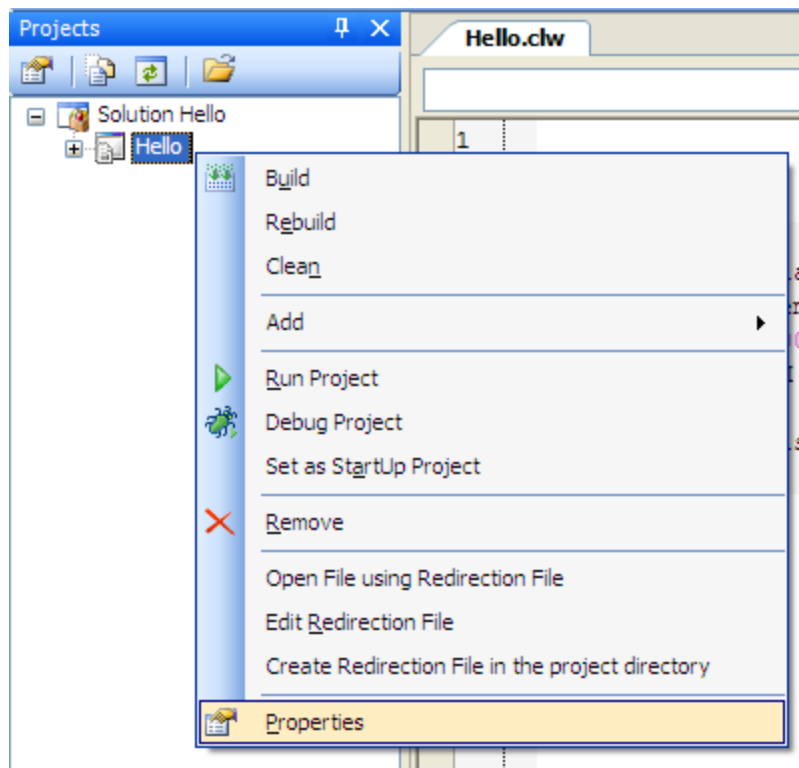
Examine the Project System

The Clarion Win32/Clarion# project system has been updated and upgraded to use the MSBuild project engine. MSBuild is the new extensible, XML-based build engine that ships with the .NET Framework 2.0 and higher. MSBuild simplifies the process of creating and maintaining build scripts, and formalizes the steps required to institute a formal build process.

All Clarion Win32 projects use a default extension of *.CWPROJ.

All projects are compiled from solution files (*.SLN). Solution files can refer to multiple projects, and allow multiple resources to be referenced and included.

These files are displayed in the Solution Explorer, shown here:



1. RIGHT-CLICK on the Hello project, and select the **Properties** item in the popup menu.

The *Project Properties* dialog appears. Project properties dialog has 4 Tabs; *Application*, *Compiling*, *Debug*, and *Build Events*.

2. The **Output Path** entry specifies where the output is created (use the ellipsis button to select a directory from the *Browse for Folder* dialog).
3. Verify that the default **Output type** is set to *Exe*. This controls the type of file that the Build action will create.
4. Verify that the default **Link Mode** is set to *Dll*.
5. Save your project changes by pressing the Save button on the IDE toolbar, and close the Project Properties by pressing the **Close (X)** button.

The HELLO.CLW file in the Text Editor now has focus.

Write the Program

1. Replace the default code in *Hello.clw* with the following:

```
PROGRAM
MAP
END
MyWin WINDOW('Hello Windows'),SYSTEM
END
CODE
OPEN(MyWin)
ACCEPT
END
```

This code begins with the **PROGRAM** statement. This must be the first non-comment statement in every Clarion program. Notice that the keyword **PROGRAM** is indented in relation to the word **MyWin**. In Clarion, the only statements that begin in column one (1) of the source code file are those with a statement label. A label must begin in column one (1), by definition. The **PROGRAM** statement begins the Global data declaration section.

Next there is an empty **MAP** structure. The **END** statement is a required terminator of the **MAP** data structure. A **MAP** structure contains prototypes which define parameter data types, return data types, and various other options that tell the compiler how to deal with your procedure calls (this is all covered later in this lesson). A **MAP** structure is required when you break up your program's code into **PROCEDURES**. We haven't done that yet, but we still need it because there is an **OPEN** statement in the executable code.

When the compiler processes a **MAP** structure, it automatically includes the prototypes in the `\CLARION7\LIBSRC\BUILTINS.CLW` file. This file contains prototypes for almost all of the Clarion language built-in procedures (including the **OPEN** statement). If the empty **MAP** structure were not in this code, the compiler would generate an error on the **OPEN** statement.

MyWin is the label of the **WINDOW** data structure (the "M" must be in column one). In Clarion, windows are declared as data structures, and not dynamically built by executable code statements as in some other languages. This is one of the aspects of Clarion that makes it a 4GL. Although Clarion can dynamically build dialogs at runtime, it is unnecessary to do so. By using a data structure, the compiler creates the Windows resource for each dialog, enabling better performance at runtime.

The ('**Hello Windows**') parameter on the **WINDOW** statement defines the title bar text for the window. The **SYSTEM** attribute adds a standard Windows system menu to the window. The **END** statement is a required terminator of the **WINDOW** data structure. In Clarion, all complex structures (both data and executable code) must terminate with an **END** or a period (.). This means the following code is functionally equivalent to the previous code:

```
PROGRAM
MAP .
MyWin WINDOW('Hello Windows'),SYSTEM.
CODE
OPEN(MyWin)
ACCEPT.
```

Although functionally equivalent, this code would become much harder to read as soon as anything is added into the **MAP**, **WINDOW**, or **ACCEPT** structures. By convention, we use the **END** statement to terminate multi-line complex statements, placing the **END** in the same column as the keyword it is terminating while indenting everything within the structure. We only use the period to terminate single-line structures, such as **IF** statements with single **THEN** clauses. This convention makes the code easier to read, and any missing structure terminators much easier to find.

The **CODE** statement is required to identify the beginning of the executable code section. Data (memory variables, data files, window structures, report structures, etc.) are declared in a data section (preceding the **CODE** statement), and executable statements may only follow a **CODE** statement.

Since this program does not contain any **PROCEDURES** (we'll get to them in the next chapter), it only has a Global Data section followed by three lines of executable code. Variables declared in the Global Data section are visible and available for use anywhere in a program.

The **OPEN(MyWin)** statement opens the window, but does not display it. The window will only appear on screen when a **DISPLAY** or **ACCEPT** statement executes. This feature allows you to dynamically change the properties of the window, or any control on the window, before it appears on screen.

ACCEPT is the event processor. Most of the messages (events) from Windows are automatically handled internally for you by **ACCEPT**. These are the common events handled by the runtime library (screen re-draws, etc.). Only those events that actually may require program action are passed on by **ACCEPT** to your Clarion code. This makes your programming job easier by allowing you to concentrate on the high-level aspects of your program.

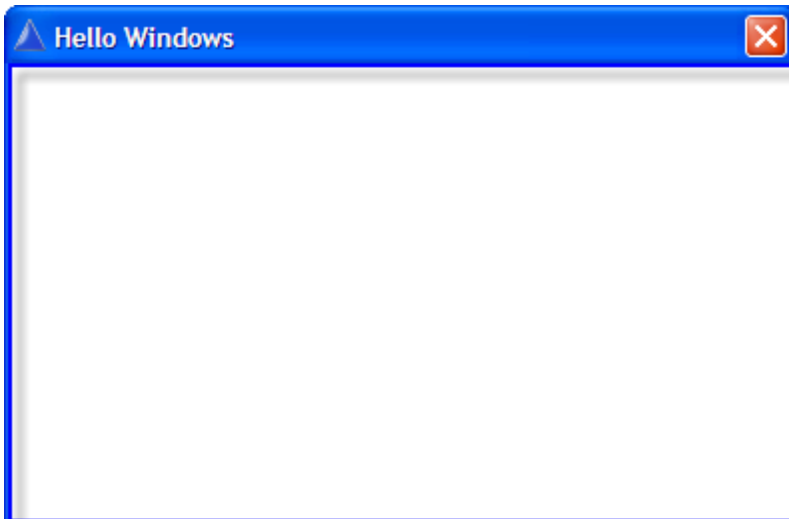
The **ACCEPT** statement has a terminating **END** statement, which means it is a complex code structure. **ACCEPT** is a looping structure, "passing through" all the events that the Clarion programmer might want to handle (none, in this program—we'll get back to this shortly), then looping back to handle the next event.

An **ACCEPT** loop is required for each window opened in a Clarion program. An open window "attaches" itself to the next **ACCEPT** loop it encounters in the code to be its event processor.

For this program, **ACCEPT** internally handles everything the system menu (placed on the window by the **SYSTEM** attribute) does. Therefore, when the user uses the system menu to close the window, **ACCEPT** automatically passes control to any statement immediately following its terminating **END** statement. Since there is no other explicit Clarion language statement to execute, the program ends. When any Clarion program reaches the end of the executable code, an implicit **RETURN** executes, which, in this case, returns the user to the operating system.

2. CLICK on the **Build and Run** button .

The program compiles and links, then executes. The window's title bar displays the "Hello Windows" message, and you must close the window with the system menu.



Hello Windows with Controls

The program you just created is close to the smallest programs possible to create in Clarion. Now we'll expand on it a bit to demonstrate adding some controls to the window and handling the events generated by those controls.

Change the Source code

1. Edit the code to read:

```
PROGRAM
MAP
END
MyWin WINDOW('Hello Windows'),AT(, ,100,100),SYSTEM      ! Changed
        STRING('Hello Windows'),AT(26,23),USE(?String1)  ! Added
        BUTTON('OK'),AT(34,60),USE(?Ok),DEFAULT          ! Added
END
CODE
OPEN(MyWin)
ACCEPT
    IF ACCEPTED() = ?Ok THEN BREAK.                      ! Added
END
```

Note:

The Window Designer is available to you in the Text Editor, just as it is in the Application Generator. To call the Window Designer, place the insertion point anywhere within the WINDOW structure then press CTRL+F. The only restrictions are that the Control Template and Dictionary Field tools are unavailable (they are specific to the Application Generator).

The change is the addition of the **STRING** and **BUTTON** controls to the WINDOW structure. The STRING places constant text in the window, and the BUTTON adds a command button.

The only other addition is the **IF ACCEPTED() = ?Ok THEN BREAK.** statement. This statement detects when the user has pressed the OK button and BREAKs out of the ACCEPT loop, ending the program. The ACCEPTED statement returns the field number of the control for which EVENT:Accepted was just generated (EVENT:Accepted is an EQUATE contained in the \CLARION7\LIBSRC\EQUATES.CLW file, which the compiler automatically includes in every program).

?Ok is the Field Equate Label of the BUTTON control, defined by the control's USE attribute (see Field Equate Labels in the Language Reference). The compiler automatically equates ?Ok to the field number it assigns the control (using Field Equate Labels helps make the code more readable).

When the ACCEPTED procedure returns a value equal to the compiler-assigned field number for the OK button, the BREAK statement executes and terminates the ACCEPT loop.

2. CLICK on the **Build and Run** button .

The program compiles and links, then executes. The window's title bar still displays the "Hello Windows" message, and now, so does the constant text in the middle of the window. You can close the window either with the system menu, or the OK button.

**Common form Source code**

There are other ways to write the code in the ACCEPT loop to accomplish the same thing. We'll go straight to the most common way, because this is more similar to the style of code that the Application Generator generates for you from the Clarion ABC Templates.

1. Edit the code to read:

```
PROGRAM
MAP
END
MyWin WINDOW('Hello Windows'),AT(, ,100,100),SYSTEM
    STRING('Hello Windows'),AT(26,23),USE(?String1)
    BUTTON('OK'),AT(34,60),USE(?Ok),DEFAULT
END
CODE
OPEN(MyWin)
ACCEPT
CASE FIELD()           !Added
OF ?Ok                 !Added
CASE EVENT()           !Added
OF EVENT:Accepted      !Added
    BREAK              !Added
END                    !Added
END                    !Added
END
```

In this code you have one **CASE** structure nested within another. A CASE structure looks for an exact match between the expression immediately following the keyword CASE and another expression immediately following an OF clause (although these only show one OF clause each, a CASE structure may have as many as necessary).

The **CASE FIELD()** structure determines to which control the current event applies. When the FIELD procedure returns a value equal to the field number of the OK button (the ?Ok Field Equate Label) it then executes the CASE EVENT() structure.

The CASE EVENT() structure determines which event was generated. When the EVENT procedure returns a value equal to EVENT:Accepted (an EQUATE contained in the \CLARION7\LIBSRC\EQUATES.CLW file) it then executes the BREAK statement.

Nesting CASE EVENT() within CASE FIELD() allows you to put all the code associated with a single control in one place. You could just as easily nest a CASE FIELD() structure within a CASE EVENT() structure, reversing the code, but this would scatter the code for a single control to multiple places.

2. CLICK on the **Build and Run** button .

Again, you can close the window either with the system menu, or the OK button, just as with the previous code, but now the code is structured in a common style.

Hello Windows with Event Handling

There are two types of events passed on to the program by ACCEPT: *Field-specific* and *Field-independent* events.

A *Field-specific* event occurs when the user does anything that may require the program to perform a specific action related to a single control. For example, when the user presses tab after entering data in a control, the field-specific EVENT:Accepted generates for that control.

A *Field-independent* event does not relate to any one control but may require some program action (for example, to close a window, quit the program, or change execution threads).

Nesting two CASE structures as we just discussed is the most common method of handling field-specific events. The most common method of handling field-independent events is a non-nested CASE EVENT() structure, usually placed immediately before the CASE FIELD() structure.

Change the Source code

1. Edit the code to read:

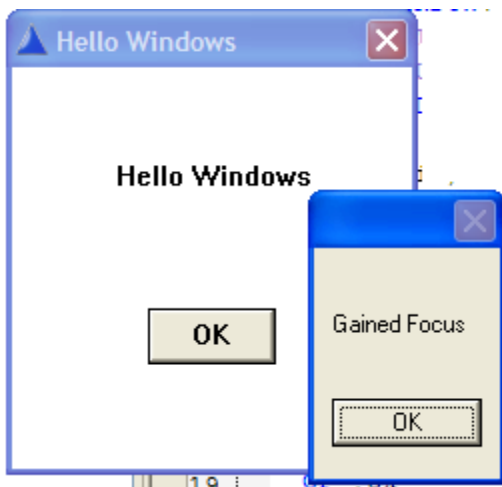
```
PROGRAM
MAP
END
MyWin WINDOW('Hello Windows'),AT(,,100,100),SYSTEM
      STRING('Hello Windows'),AT(26,23),USE(?String1)
      BUTTON('OK'),AT(34,60),USE(?Ok),DEFAULT
      END
CODE
OPEN(MyWin)
ACCEPT
CASE EVENT()
OF EVENT:OpenWindow
MESSAGE('Opened Window')
OF EVENT:GainFocus
MESSAGE('Gained Focus')
END
CASE FIELD()
OF ?Ok
CASE EVENT()
OF EVENT:Accepted
BREAK
END
END
END
```

The new **CASE EVENT()** structure handles two field-independent events: `EVENT:OpenWindow` and `EVENT:GainFocus`. The **MESSAGE** procedure used in this code is just to visually display to you at runtime that the event was triggered. Instead of the **MESSAGE** procedure, you would add here any code that your program needs to execute when the user triggers these events.

This demonstrates the basic logic flow and code structure for procedural window procedures—an **ACCEPT** loop containing a **CASE EVENT()** structure to handle all the field-independent events, followed by a **CASE FIELD()** structure with nested **CASE EVENT()** structures to handle all field-specific events.

2. CLICK on the **Build and Run** button .

Notice that `EVENT:OpenWindow` generates when the window first displays (in that order). `EVENT:GainFocus` will generate when you **ALT+TAB** to another application then **ALT+TAB** back to Hello Windows.



Adding a PROCEDURE

In Hello Windows we have an example of a very simple program. Most modern business programs are not that simple—they require the use of Structured Programming techniques. This means you break up your program into functional sections, where each performs a single logical task. In the Clarion language these functional sections are called **PROCEDURES**.

First, we'll add a **PROCEDURE** to the *Hello Windows* program.

Change the Source code

1. Edit the code to read:

```
PROGRAM
MAP
Hello PROCEDURE                                !Added
END
CODE                                             !Added
```

```

Hello                                     !Added

Hello  PROCEDURE                               !Added
MyWin  WINDOW('Hello Windows'),AT(,,100,100),SYSTEM
        STRING('Hello Windows'),AT(26,23),USE(?String1)
        BUTTON('OK'),AT(34,60),USE(?Ok),DEFAULT
        END
CODE
OPEN(MyWin)
ACCEPT
CASE EVENT()
OF EVENT:OpenWindow
    MESSAGE('Opened Window')
OF EVENT:GainFocus
    MESSAGE('Gained Focus')
END
CASE FIELD()
OF ?Ok
    CASE EVENT()
    OF EVENT:Accepted
        BREAK
    END
END
END
```

The only changes are at the beginning of the program. Inside the MAP structure we now see the Hello **PROCEDURE** statement which prototypes the Hello procedure. A *prototype* is the declaration of the procedure for the compiler, telling the compiler what to expect when your code calls the procedure. This prototype indicates that the procedure takes no parameters and does not return a value. All PROCEDURES in your program must be prototyped in a MAP structure. See PROCEDURE prototypes in the *Language Reference* for more on prototypes.

The keyword **CODE** immediately following the MAP structure terminates the Global data section and marks the beginning of the Global executable code section, which only contains the Hello statement—a call to execute the Hello procedure. A PROCEDURE which does not return a value is always called as a single executable statement in executable code.

The second Hello PROCEDURE statement terminates the Global executable code section and marks the beginning of the code definition of the Hello procedure.

A PROCEDURE contains a data declaration section just as a PROGRAM does, and so, also requires the keyword **CODE** to define the boundary between data declarations and executable code. This is why the rest of the code did not change from the previous example. This is the Local data declaration section for the PROCEDURE.

The biggest difference between Global and Local data declarations is the scope of the declared data. Any data item declared in a Local data section is visible only within the PROCEDURE which declares it, while any data item declared in the Global data section is visible everywhere in the

program. See Data Declarations and Memory Allocation in the Language Reference for a full discussion of all the differences.

2. CLICK on the **Build and Run** button .

The program executes exactly as it did before. The only difference is that the Hello PROCEDURE can now be called from anywhere within the program—even from within another PROCEDURE. This means that, even though the program may execute the procedure many times, the code for it exists just once.

Adding Another PROCEDURE

A Clarion PROCEDURE which does not directly RETURN a value can only be called as a separate executable statement—it cannot be used in an expression, parameter list, or an assignment statement. A PROCEDURE which does directly RETURN a value must always contain a RETURN statement and may be used in expressions, parameter lists, and assignment statements. You can call a PROCEDURE which does directly RETURN a value as a separate statement if you do not want the value the PROCEDURE returns, but doing this generates compiler warnings (unless the PROCEDURE's prototype has the PROC attribute).

Structurally, both types of PROCEDURE are equivalent—they both have Local data sections, followed by the executable code section that begins with the keyword CODE.

Change the Source code

1. Edit the MAP structure to read:

```
MAP
Hello    PROCEDURE
EventString PROCEDURE (LONG PassedEvent) ,STRING    !Added
END
```

This adds the prototype for the EventString PROCEDURE. EventString receives a LONG parameter called *PassedEvent* that may not be omitted, and returns a STRING. The data types of all parameters passed to a PROCEDURE are specified inside the parentheses following PROCEDURE, each separated by a comma (if there are multiple parameters being passed). The data type of the return value of a PROCEDURE is specified following the closing parenthesis of the parameter list. Both types of PROCEDURES may receive parameters, see Prototype Parameter Lists in the Language Reference for a more complete discussion of parameter passing.

2. Add the *EventString* PROCEDURE definition to the end of the file:

```
EventString    PROCEDURE (LONG PassedEvent)
ReturnString   STRING (255)
CODE
CASE PassedEvent
```

```
OF EVENT:OpenWindow
    ReturnString = 'Opened Window'
OF EVENT:GainFocus
    ReturnString = 'Gained Focus'
ELSE
    ReturnString = 'Unknown Event: ' & PassedEvent
END
RETURN(ReturnString)
```

The *EventString* label (remember, it must be in column one) on the PROCEDURE statement names this procedure, while the parameter list attached to the PROCEDURE keyword names the LONG parameter PassedEvent. There must always be an equal number of parameter names listed on the PROCEDURE statement as there are parameter data types listed in the prototype for that PROCEDURE.

ReturnString is a local variable declared as a 255 character STRING field, on the stack. The CODE statement terminates the procedure's Local data section. The CASE PassedEvent structure should look familiar, because it is the same as a CASE EVENT() structure, but its CASE condition is the PassedEvent instead of the EVENT() procedure. This CASE structure simply assigns the appropriate value to the ReturnString variable for each event that is passed to the procedure.

The interesting code here is the RETURN(ReturnString) statement. A PROCEDURE without a return value does not require an explicit RETURN, since it always executes an implicit RETURN when there is no more code to execute. However, a PROCEDURE prototyped to return a value always contains an explicit RETURN statement which specifies the value to return. In this case, the RETURN statement returns whichever value was assigned to the ReturnString in the CASE structure.

3. Edit the Hello procedure's CASE EVENT() structure to read:

```
CASE EVENT()
OF EVENT:OpenWindow
    MESSAGE(EventString(EVENT:OpenWindow))    ! Changed
OF EVENT:GainFocus
    MESSAGE(EventString(EVENT:GainFocus))    ! Changed
END
```

This changes the MESSAGE procedures to display the returned value from the EventString procedure. The event number is passed to EventString as the event EQUATE to make the code more readable.

4. CLICK on the **Build and Run** button .

The program still executes and looks exactly as it did before.

Moving Into the Real World—Adding a Menu

Hello Windows is a nice little demonstration program, but it really doesn't show you much to do with real-world business programming. Therefore, we're now going to expand it to include some real-world functionality, starting with a menu.

Change the Source code

1. Edit the beginning of the file to read:

```
PROGRAM
MAP
Main      PROCEDURE          ! Added
Hello     PROCEDURE
EventString PROCEDURE (LONG PassedEvent), STRING
END
CODE
Main      ! Changed
```

This adds the Main PROCEDURE prototype to the MAP structure and replaces the call to Hello with a call to the Main procedure.

2. Add the Main PROCEDURE definition to the end of the file:

```
Main  PROCEDURE
AppFrame APPLICATION('Hello Windows'), AT(, , 280, 200), SYSTEM, RESIZE, MAX
    MENUBAR
        MENU('&File'), USE(?File)
        ITEM('&Browse Phones'), USE(?FileBrowsePhones)
        ITEM, SEPARATOR
        ITEM('E&xit'), USE(?FileExit), STD(STD:Close)
    END
    ITEM('&About! '), USE(?About)
    END
END
CODE
OPEN(AppFrame)
ACCEPT
CASE ACCEPTED()
OF ?About
```

```
    Hello
END
END
```

The *Main* PROCEDURE accepts no parameters. It contains the AppFrame **APPLICATION** structure. An APPLICATION structure is the key to creating Windows Multiple Document Interface (MDI) programs. An MDI application can contain multiple execution threads. This is the MDI parent application frame that is required to create an MDI application.

The **MENUBAR** structure defines the menu items available to the user. The **MENU**('&File') structure creates the standard File menu that you see in most Windows programs. The ampersand (&) preceding the "F" specifies that the "F" is the menu's accelerator key, and will be underlined at runtime by the operating system.

The **ITEM**('&Browse Phones') creates a menu item that we will use to call a procedure (we'll get to that shortly). The **ITEM,SEPARATOR** statement creates a dividing line in the menu following the Browse Phones selection.

The **ITEM**('E&xit') creates a menu item to exit the procedure (and the program, since this procedure is the only procedure called from the Global executable code). The **STD(STD:Close)** attribute specifies the standard window close action to break out of the **ACCEPT** loop. This is why you don't see any executable code associated with this menu item in the **ACCEPT** loop—the Clarion runtime library takes care of it for you automatically.

The **ITEM**('&About!') statement creates a menu item on the action bar, right next to the File menu. The trailing exclamation point (!) is a programming convention to give end-users a visual clue that this item executes an action and does not drop down a menu, despite the fact that it is on the action bar.

The **ACCEPT** loop contains only a **CASE ACCEPTED()** structure. The **ACCEPTED** procedure returns the field number of the control with focus when **EVENT:Accepted** is generated. We can use the **ACCEPTED** procedure here instead of the **FIELD** and **EVENT** procedures because menu items only generate **EVENT:Accepted**. The **OF ?About** clause simply calls the **Hello** procedure.

3. CLICK on the **Build and Run** button .

The program executes and only the **About!** and **File ► Exit** items actually do anything. Notice, though, that **File ► Exit** does terminate the program, despite the fact that we wrote no code to perform that action.

Really Moving Into the Real World—Adding a Browse and Form

Having a menu is nice, but now it's time to do some real-world business programming. Now we're going to add a data file and the procedures to maintain it.

Change the Global code

1. Edit the beginning of the file to read:

```

PROGRAM
    INCLUDE ('Keycodes.CLW')                !Added
    INCLUDE ('Errors.CLW')                  !Added
    MAP
Main      PROCEDURE
BrowsePhones PROCEDURE                    !Added
UpdatePhones PROCEDURE (LONG Action), LONG !Added
Hello      PROCEDURE
EventString PROCEDURE (LONG PassedEvent), STRING
    END

Phones      FILE, DRIVER ('TopSpeed') , CREATE !Added
NameKey      KEY (Name) , DUP, NOCASE         !Added
Rec          RECORD                          !Added
Name         STRING (20)                     !Added
Number       STRING (20)                     !Added
            END                             !Added
            END                             !Added

InsertRecord EQUATE (1)                     !Added
ChangeRecord EQUATE (2)                     !Added
DeleteRecord EQUATE (3)                     !Added
ActionCode   EQUATE (1)                     !Added
ActionAborted EQUATE (2)                     !Added

CODE
Main

```

The two new **INCLUDE** statements add the standard EQUATEs for keycodes and error numbers. Using the EQUATEs instead of the numbers makes your code more readable, and therefore, more maintainable.

The MAP structure has acquired two more procedure prototypes: the *BrowsePhones* PROCEDURE and *UpdatePhones* PROCEDURE (LONG Action), LONG. The BrowsePhones procedure will display a list of the records in the file and UpdatePhones will update individual records.

In the interest of coding simplicity (you'll see why shortly), the `BrowsePhones` procedure will simply display all records in the file in a `LIST` control. This is not exactly the same as a `Browse` procedure in template generated code (which is page-loaded in order to handle very large files), but will serve a similar purpose in this program.

Also in the interest of coding simplicity, `UpdatePhones` is a `PROCEDURE` that takes a parameter. The `LONG` parameter indicates what file action to take: `ADD`, `PUT`, or `DELETE`. The `LONG` return value indicates to the calling procedure whether the user completed or aborted the action. Again, this is not the same as a `Form` procedure in template generated code (which is a `PROCEDURE` using `Global` variables to signal file action and completion status), but will serve the same purpose in this simple program.

The `Phones` `FILE` declaration creates a simple data file using the `TopSpeed` file driver. There are two data fields: *Name* and *Number* which are both declared as `STRING(20)`. Declaring the `Number` field as a `STRING(20)` allows it to contain phone numbers from any country in the world (more about that to come).

The five `EQUATE` statements define constant values that make the code more readable. *InsertRecord*, *ChangeRecord*, and *DeleteRecord* all define file actions to pass as the parameter to the `UpdatePhones` procedure. The *ActionComplete* and *ActionAborted* `EQUATE`s define the two possible return values from the `UpdatePhones` procedure.

2. Edit the *Main* procedure's `CASE ACCEPTED()` code to read:

```
CASE ACCEPTED ()
OF ?FileBrowsePhones          !Added
    START (BrowsePhones, 25000) !Added
OF ?About
    Hello
END
```

The **`START(BrowsePhones,25000)`** statement executes when the user chooses the `Browse Phones` menu selection. The `START` procedure creates a new execution thread for the `BrowsePhones` procedure and the second parameter (25000) specifies the size (in bytes) of the new execution thread's stack. You must use the `START` procedure when you are calling a procedure containing an MDI child window from the `APPLICATION` frame, because each MDI child window must be on a separate execution thread from the `APPLICATION` frame. If you do not `START` the MDI child, you get the "Unable to open MDI window on `APPLICATION`'s thread" runtime error message when you try to call the `BrowsePhones` procedure and the program immediately terminates.

Once you have started a new thread, an MDI child may simply call another MDI child on its same thread without starting a new thread. This means that, although `BrowsePhones` and `UpdatePhones` both contain MDI windows, only `BrowsePhones` must `START` a new thread, because it is called from the application frame. `BrowsePhones` will simply call `UpdatePhones` without starting a new thread.

Add the `BrowsePhones` `PROCEDURE`

3. Add the data section of the *BrowsePhones* `PROCEDURE` definition to the end of the file:

```

BrowsePhones PROCEDURE
PhonesQue    QUEUE
Name         STRING(20)
Number       STRING(20)
Position     STRING(512)
            END

window WINDOW('Browse Phones'), AT(, , 185, 156), SYSTEM, GRAY, RESIZE, MDI
    LIST, AT(6, 8, 173, 100), ALRT(MouseLeft2), USE(?List) |
    , FORMAT('84L|M~Name~80L~Number~'), FROM(PhonesQue), VSCROLL
    BUTTON('&Insert'), AT(20, 117), KEY(InsertKey), USE(?Insert)
    BUTTON('&Change'), AT(76, 117, 35, 14), KEY(EnterKey), USE(?Change)
    BUTTON('&Delete'), AT(131, 117, 35, 14), KEY(DeleteKey), USE(?Delete)
    BUTTON('Close'), AT(76, 137, 35, 14), KEY(EscKey), USE(?Close)
END

```

The *PhonesQue* **QUEUE** structure defines the data structure that will contain all the records from the Phones FILE to display in the LIST control. A Clarion QUEUE is similar to a data file because it has a data buffer and allows an indeterminate number of entries, but it only exists in memory at runtime. A QUEUE could also be likened to a dynamically allocated array in other programming languages.

The PhonesQue QUEUE contains three fields. The *Name* and *Number* fields duplicate the fields in the Phones FILE structure and will display in the two columns defined in the LIST control's FORMAT attribute. The *Position* field will contain the return value from the **POSITION** procedure for each record in the Phones FILE. Saving each record's Position will allow us to immediately re-get the record from the data file before calling UpdatePhones to change or delete a record.

The *window* WINDOW structure contains one LIST control and four BUTTON controls. The LIST control is the key to this procedure. The ALRT(MouseLeft2) on the LIST alerts DOUBLE-CLICK so the ACCEPT statement will pass an EVENT:AlertKey to our Clarion code. This will let us write code to bring up the UpdatePhones procedure to change the record the user DOUBLE-CLICKs on.

The vertical bar (|) at the end of the LIST statement is the Clarion line continuation character, meaning that the LIST control continues on the next line with the **FORMAT**('84L|M~Name~80L~Number~') attribute (you can put it all on one line if you want). The parameter to the FORMAT attribute defines the appearance of the LIST control at runtime.

Tip

It is best to let the List Box Formatter tool in the Window Designer write format strings for you, since they can become very complex very quickly. This format string defines two columns. The first column is 84 dialog units wide (84), left justified(L), has a right border (|) that is resizable (M) and "Name" is its heading (~Name~). The second column is 80 dialog units wide (80), left justified(L), and "Number" is its heading (~Number~).

FROM(PhonesQue) on the LIST specifies the source QUEUE the LIST control will display, and VSCROLL adds the vertical scroll bar. The LIST will display the Name and Number fields of all entries in the PhonesQue while ignoring the Position field because the FORMAT attribute only specified two columns.

The LIST automatically handles users scrolling through the list without any coding on our part. The LIST does this because there is no IMM (immediate) attribute present. If there were an IMM attribute, we would have to write code to handle scrolling records (as the page-loaded Browse procedure template does).

The **BUTTON('&Insert')** statement defines a command button the user will press to add a new record. The KEY(InsertKey) attribute specifies that the button is automatically pressed for the user when they press insert on the keyboard. Notice that the other three BUTTON controls all have similar KEY attributes. This means you don't have to write any special code to handle keyboard access to the program versus mouse access.

4. Add the beginning of the executable code section of the *BrowsePhones* PROCEDURE definition to the end of the file:

```
CODE
DO OpenFile
DO FillQue
OPEN(window)
ACCEPT
CASE EVENT()
OF EVENT:OpenWindow
DO QueRecordsCheck
OF EVENT:AlertKey
IF KEYCODE() = MouseLeft2
POST(EVENT:Accepted,?Change)
END
END
END
```

The beginning of the CODE section starts with the *DO OpenFile* statement. DO executes a **ROUTINE**, in this case, the OpenFile ROUTINE, and when the code in the ROUTINE is done executing, control returns to the next line of code following the DO statement that called the ROUTINE.

The code defining a ROUTINE must be at the bottom of the procedure, following all the main logic, because the ROUTINE statement itself terminates the executable code section of a PROCEDURE.

There are two reasons to use a ROUTINE within a PROCEDURE: to write one set of code statements that need to execute at multiple logical points within the procedure, or to make the logic more readable by substituting a single DO statement for a set of code statements that perform a single logical task.

In this case, the DO OpenFile statement serves the second purpose by moving the code that opens or creates the data file out of the main procedure logic. Next, the DO FillQueue statement executes the code that fills the PhonesQue QUEUE structure with all the records from the data file. These two DO statements make it very easy to follow the logic flow of the procedure.

The CASE EVENT() structure's OF EVENT:OpenWindow clause executes *DO QueRecordsCheck* to call a ROUTINE that checks to see if there are any records in PhonesQue.

Next, the OF EVENT:AlertKey clause contains the IF KEYCODE() = MouseLeft2 structure to check for DOUBLE-CLICK on the LIST. Since the ALRT(MouseLeft2) attribute only appears on the LIST control, we know that the POST(EVENT:Accepted,?Change) statement will only execute when the user DOUBLE-CLICKs on the LIST.

The **POST** statement tells ACCEPT to post the event in its first parameter (EVENT:Accepted) to the control in its second parameter (?Change). The effect of the POST(EVENT:Accepted,?Change) statement is to cause the EVENT:Accepted code for the Change button to execute, just as if the user had pressed the button with the mouse or keyboard.

This illustrates a very good coding practice: write specific code once and call it from many places. This is the structured programming concept that gave us PROCEDURES and ROUTINES. Even though the EVENT:Accepted code for the change button is not sectioned off separately in a PROCEDURE or ROUTINE, using the POST statement this way means that the one section of code is all you need to maintain—if the desired logic changes, you'll only have to change it in one place.

5. Add the rest of the executable code section of the *BrowsePhones* PROCEDURE definition to the end of the file:

```
CASE FIELD()
OF ?Close
CASE EVENT()
OF EVENT:Accepted
POST(EVENT:CloseWindow)
END
OF ?Insert
CASE EVENT()
OF EVENT:Accepted
IF UpdatePhones(InsertRecord) = ActionComplete
DO AssignToQue
ADD(PhonesQue)
IF ERRORCODE() THEN STOP(ERROR()) .
SORT(PhonesQue, PhonesQue.Name)
ENABLE(?Change, ?Delete)
END
GET(PhonesQue, PhonesQue.Name)
SELECT(?List, POINTER(PhonesQue))
END
```

```
OF ?Change
CASE EVENT()
OF EVENT:Accepted
DO GetRecord
IF UpdatePhones(ChangeRecord) = ActionComplete
DO AssignToQue
PUT(PhonesQue)
IF ERRORCODE() THEN STOP(ERROR()) .
SORT(PhonesQue, PhonesQue.Name)
END
GET(PhonesQue, PhonesQue.Name)
SELECT(?List, POINTER(PhonesQue))
END
OF ?Delete
CASE EVENT()
OF EVENT:Accepted
DO GetRecord
IF UpdatePhones(DeleteRecord) = ActionComplete
DELETE(PhonesQue)
IF ERRORCODE() THEN STOP(ERROR()) .
DO QueRecordsCheck
SORT(PhonesQue, PhonesQue.Name)
END
SELECT(?List)
END
END
END
FREE(PhonesQue)
CLOSE(Phones)
```

Following the CASE EVENT() structure is the CASE FIELD() structure. Notice that each OF clause contains its own CASE EVENT() structure, and each of these only contains an OF EVENT:Accepted clause. Because of this, we could have replaced the CASE FIELD() structure with a CASE ACCEPTED() structure and eliminated the nested CASEs. This would actually have given us slightly better performance—too slight to actually notice on-screen, though. The reason we didn't is consistency; you will more often have occasion to trap more field-specific events than just EVENT:Accepted, and when you do, this nested CASE structure code is the logic to use, so it's a good habit to make now. It also demonstrates the kind of code structure that is generated for you by Clarion's templates in the Application Generator.

The OF ?Close clause executes the POST(EVENT:CloseWindow) when the user presses the Close button. Since there's no second parameter to the POST statement naming a control, the event posts to the WINDOW (and should be a field-independent event). EVENT:CloseWindow causes the ACCEPT loop to terminate and execution control drops to the first statement following

the ACCEPT's terminating END statement. In this case, control drops to the FREE(PhonesQue) statement, which frees all the memory used by the QUEUE entries (effectively closing the QUEUE). The CLOSE(Phones) statement then closes the data file. Since there are no other statements to execute following CLOSE(Phones) the procedure executes an implicit RETURN and goes back to the Main procedure (where it was called from).

The OF ?Insert clause contains the IF UpdatePhones(InsertRecord) = ActionComplete structure. This calls the UpdatePhones PROCEDURE, passing it the InsertRecord constant value that we defined in the Global data section, and then checks for the ActionComplete return value.

The DO AssignToQue statement executes only when the user actually adds a record. AssignToQue is a ROUTINE that assigns data from the Phones FILE's record buffer to the PhonesQue QUEUE's data buffer. Then the ADD(PhonesQue) statement adds a new entry to PhonesQue. The IF ERRORCODE() THEN STOP(ERROR()) statement is a standard error check that you should execute after any FILE or QUEUE action that could possibly return an error (another good habit to form).

The **Sort(PhonesQue,PhonesQue.Name)** statement sorts the PhonesQue QUEUE entries alphabetically by the Name field. Since there is no PRE attribute on the PhonesQue QUEUE structure, you must reference its fields using Clarion's Field Qualification syntax by prepending the name of the structure containing the field (PhonesQue) to the name of the field (Name), connecting them with a period (PhonesQue.Name). See Field Qualification in the Language Reference for more information.

The **ENABLE(?Change,?Delete)** statement makes sure the Change and delete buttons are active (if this was the first entry in the QUEUE, these buttons were dimmed out by the QueRecordsCheck ROUTINE). The GET(PhonesQue,PhonesQue.Name) statement re-gets the new record from the sorted QUEUE, then SELECT(?List,POINTER(PhonesQue)) puts the user back on the LIST control with the new record highlighted.

The code in the OF ?Change clause is almost identical to the code in the OF ?Insert clause. There is an added DO GetRecord statement that calls a ROUTINE to put the highlighted PhonesQue entry's related file record into the Phones file record buffer. The only other difference is the PUT(PhonesQue) statement that puts the user's changes back in the PhonesQue.

The code in the OF ?Delete clause is almost identical to the code in the OF ?Change clause. The difference is the DELETE(PhonesQue) statement that removes the entry from the PhonesQue and the call to DO QueRecordsCheck to see if the user just deleted the last record.

6. Add the ROUTINEs called in the *BrowsePhones* PROCEDURE definition to the end of the file:

AssignToQue ROUTINE

```
PhonesQue.Name = Phones.Rec.Name
PhonesQue.Number = Phones.Rec.Number
PhonesQue.Position = POSITION(Phones)
```

QueRecordsCheck ROUTINE

```
IF NOT RECORDS(PhonesQue)
    DISABLE(?Change,?Delete)
    SELECT(?Insert)
ELSE
    SELECT(?List,1)
END
```

```
GetRecord ROUTINE
  GET(PhonesQue,CHOICE(?List))
  IF ERRORCODE() THEN STOP(ERROR()).
  REGET(Phones,PhonesQue.Position)
  IF ERRORCODE() THEN STOP(ERROR()).

OpenFile ROUTINE
  OPEN(Phones,42h)
  CASE ERRORCODE()
  OF NoError
  OROF IsOpenErr
    EXIT
  OF NoFileErr
    CREATE(Phones)
    IF ERRORCODE() THEN STOP(ERROR()).
    OPEN(Phones,42h)
    IF ERRORCODE() THEN STOP(ERROR()).
  ELSE
    STOP(ERROR())
  RETURN
END

FillQue ROUTINE
  SET(Phones.NameKey)
  LOOP
    NEXT(Phones)
    IF ERRORCODE() THEN BREAK.
    DO AssignToQue
    ADD(PhonesQue)
    IF ERRORCODE() THEN STOP(ERROR()).
  END
```

As you can see, there are five ROUTINEs for this procedure. Notice that, although these ROUTINEs look similar to a PROCEDURE, they do not contain CODE statements. This is because a ROUTINE shares the procedure's Local data and does not usually have a data declaration section of its own (a ROUTINE can have its own data section—see ROUTINE in the Language Reference for a full discussion of this issue).

The *AssignToQue* ROUTINE performs three assignment statements. The `PhonesQue.Name = Phones.Rec.Name` statement copies the data in the Name field of the Phones FILE record buffer and places it in the Name field of the PhonesQue QUEUE data buffer. Since there is no PRE attribute on the Phones FILE structure, you must also reference its fields using Clarion's **Field**

Qualification Syntax by stringing together the FILE name (Phones), the RECORD name (Rec), and the name of the field (Name), connecting them all with a period (Phones.Rec.Name).

The `PhonesQue.Number = Phones.Rec.Number` statement assigns the data in the Phones file's Number field to the PhonesQue's Number field. The `PhonesQue.Position = POSITION(Phones)` statement assigns the return value of the POSITION procedure to the PhonesQue.Position field. This value lets us retrieve from disk the one specific record that is currently in the Phones file's record buffer. The POSITION procedure does this for every Clarion file driver, and is therefore the recommended method of specific record retrieval across multiple file systems.

The *QueRecordsCheck* ROUTINE checks to see if there are any records in the PhonesQue. The `IF NOT RECORDS(PhonesQue)` structure uses the logical NOT operator against the return value from the RECORDS procedure. If `RECORDS(PhonesQue)` returns zero, then the NOT makes the condition true and the code following the IF executes (zero is always false and the NOT makes it true). If `RECORDS(PhonesQue)` returns anything other than zero, the code following the ELSE will execute (any non-zero number is always true and the NOT makes it false). Therefore, if there are no records in the PhonesQue, `DISABLE(?Change,?Delete)` executes to dim out the Change and Delete buttons, then the `SELECT(?Insert)` statement places the user on the Insert button (the next logical action). If there are records in the PhonesQue, then the `SELECT(?List)` statement places the user on the LIST.

The *GetRecord* ROUTINE synchronizes the Phones file's record buffer and PhonesQue's data buffer with the currently highlighted entry in the LIST. The `GET(PhonesQue,CHOICE(?List))` statement uses the CHOICE procedure to "point to" the currently highlighted entry in the LIST and GET the related QUEUE entry into the PhonesQue's data buffer (of course, always checking for unexpected errors). Then the **`REGET(Phones,PhonesQue.Position)`** statement uses the saved record position information to retrieve the Phones FILE record into the record buffer.

The *OpenFile* ROUTINE either opens or creates the Phones FILE. The `OPEN(Phones,42h)` statement attempts to open the Phones file for shared access. The second parameter (42h) is a hexadecimal number (signaled by the trailing "h"). Clarion supports the Decimal, Hexadecimal, Binary, and Octal number systems. This number represents Read/Write, Deny None access (fully shared) to the file (see the OPEN statement in the Language Reference for more on file access modes). We're requesting shared access because this is an MDI program and the user could call multiple copies of this procedure in the same program. However, this program does not do all the concurrency checking required for a real multi-user application. See Multi-User Considerations in the Programmer's Guide for more on the concurrency checking issue.

The `CASE ERRORCODE()` structure checks for any error on the OPEN. The `OF NoError OROF IsOpenErr` clause (now you can see why we included ERRORS.CLW file) executes the EXIT statement to immediately return from the ROUTINE. It is very important not to confuse EXIT with RETURN, since RETURN immediately terminates the PROCEDURE, while EXIT only terminates the ROUTINE. RETURN is valid to use within a ROUTINE, just be sure you want to terminate the PROCEDURE and not simply terminate the ROUTINE.

The `OF NoFileErr` clause detects that there is no data file to open. The `CREATE(Phones)` statement will then create a new empty data file for us. You must be sure that, if you intend to use the CREATE statement that the FILE declaration also contains the CREATE attribute, otherwise the CREATE statement will not be able to create the file for you. The CREATE statement does not open the file for processing, so that explains the second `OPEN(Phones,42h)` statement. The code in the ELSE clause executes if any error other than these occur. The `STOP(ERROR())` statement displays the ERROR procedure's message to the user in a system modal window allowing the user the opportunity to abort the program (returning to Windows) or ignore the error. The RETURN statement then terminates the procedure if the user chooses to ignore the error.

The *FillQue* ROUTINE fills the PhonesQue QUEUE with all the records in the Phones file. The `SET(Phones.NameKey)` statement sets up the processing order and starting point for the Phones file. The `Phone.NameKey` parameter makes the processing order alphabetic based on the Name

field. The absence of a second parameter to the SET statement makes the starting point the beginning (or end) of the file. The LOOP structure has no condition, which means you must place a BREAK statement somewhere in the LOOP or else get an infinite loop. The NEXT(Phones) statement retrieves the next record from the Phones data file, then the IF ERRORCODE() THEN BREAK. statement ensures that we will BREAK out of the LOOP when all the records have been read. The DO AssignToQue statement calls the AssignToQue ROUTINE that we've already discussed, and ADD(PhonesQue) adds the new record to the QUEUE.

Add the UpdatePhones PROCEDURE

7. Add the data section of the *UpdatePhones* PROCEDURE definition to the end of the file:

```
UpdatePhones PROCEDURE(LONG Action)
ReturnValue LONG,AUTO
window WINDOW('Update Phone'),AT(, ,185,92),SYSTEM,GRAY,RESIZE,MDI
    PROMPT('N&ame:'),AT(14,14),USE(?Prompt1)
    ENTRY(@s20),AT(68,13),USE(Phones.Rec.Name),REQ
    PROMPT('N&umber:'),AT(14,43),USE(?Prompt2)
    ENTRY(@s20),AT(68,42),USE(Phones.Rec.Number)
    BUTTON('OK'),AT(45,74),USE(?Ok),REQ,DEFAULT
    BUTTON('Cancel'),AT(109,74,32,14),USE(?Cancel)
END
```

The UpdatePhones PROCEDURE(LONG Action) statement defines a PROCEDURE that receives a single LONG data typed parameter that will be called "Action" within the PROCEDURE (no matter what variable or constant is passed in).

ReturnValue LONG, AUTO declares a LONG variable that remains uninitialized by the compiler (due to the **AUTO** attribute). By default, memory variables in Clarion are initialized to all blanks or zero (depending on their data type). Specifying AUTO saves a bit of memory, but the caveat is that you must be sure you are going to assign a value to the uninitialized variable before you ever check its contents, otherwise you could create an intermittently occurring bug that would be really difficult to track down.

The WINDOW structure has the **MASK** attribute, which means that the user's data entry patterns are checked as the data is input, instead of the default Standard Windows Behavior (SWB) of "free-form" data entry.

The two PROMPT and ENTRY controls combine to provide the user's data entry controls. The two BUTTON controls will allow the user to complete or abort the current file action.

The PROMPT controls define the screen prompt text and the accelerator key to navigate to the ENTRY control following the PROMPT, The accelerator keys are formed using alt plus the letter following the ampersand. For example, ('N&ame:') indicates alt+a will give input focus to the ENTRY(@s20),AT(68,13),USE(Phones.Rec.Name) control.

The USE attributes of the ENTRY controls name the data fields that automatically receive the user's input at runtime. The runtime library ensures that the current value in the variable named in

the USE attribute displays when the control gains input focus. When the user enters data in the ENTRY control then moves on to another control, the runtime library ensures that the variable named in the USE attribute receives the current value displayed in the control.

The REQ attribute on the first ENTRY control means that the user cannot leave it blank, while the REQ attribute on the OK button checks to make sure that the user entered data into all the ENTRY controls with the REQ attribute. This required fields check is only done when the button with the REQ attribute is pressed, because the user may not have even gone to the the ENTRY with the REQ attribute.

8. Add the main logic of the *UpdatePhones* PROCEDURE definition to the end of the file:

```
CODE
OPEN(window)
DO SetupScreen
ACCEPT
CASE FIELD()
OF ?Phones:Rec:Number
CASE EVENT()
OF EVENT:Selected
DO SetupInsert
END
OF ?Ok
CASE EVENT()
OF EVENT:Accepted
EXECUTE Action
ADD(Phones)
PUT(Phones)
DELETE(Phones)
END
IF ERRORCODE() THEN STOP(ERROR()).
ReturnValue = ActionComplete
POST(EVENT:CloseWindow)
END

OF ?Cancel
CASE EVENT()
OF EVENT:Accepted
ReturnValue = ActionAborted
POST(EVENT:CloseWindow)
END
END
END
```

RETURN(ReturnValue)

The *DO SetupScreen* statement calls the SetupScreen ROUTINE to perform some window initialization code. Notice that it follows the OPEN(Window) statement. When you are going to dynamically alter the window in a procedure, it must be opened first.

The OF ?Phones:Rec:Number clause in the CASE FIELD() structure demonstrates two important points. The first is the Field Equate Label, itself. The USE(Phones.Rec.Number) attribute contains periods in the field name and periods are not valid in Clarion labels. Therefore, to construct a Field Equate Label for Phones.Rec.Number, the compiler substitutes colons for the periods (because colons are valid in a Clarion label).

The second important point is the OF EVENT:Selected clause in the CASE EVENT() structure. EVENT:Selected generates when the control gains input focus but before the user gets to input data. The DO SetupInsert statement executes to offer the user an option then setup the display and data entry format of the ENTRY control.

The OF EVENT:Accepted code in OF ?Ok is the code that actually writes the record to disk. The EXECUTE Action structure executes exactly one of the ADD(Phones), PUT(Phones), or DELETE(Phones) statements.

An EXECUTE structure is similar to the IF and CASE structures in that it conditionally executes code based on the evaluation of a condition. The EXECUTE condition must evaluate to an integer in the range of 1 to n (where n is the number of code statements within the EXECUTE structure), then it executes the single ordinal line of code within the structure that corresponds to the value of the condition.

In this code, EXECUTE looks at the Action parameter then executes ADD(Phones) if the value of Action is one (1), PUT(Phones) if Action is two (2), or DELETE(Phones) if Action is three (3).

Generally, when you evaluate which Clarion code structure to use for an instance of conditional code execution (IF/ELSIF, CASE, or EXECUTE) the IF/ELSIF structure is the most flexible and least efficient, CASE is less flexible but much more efficient than IF/ELSIF, and EXECUTE is not flexible but highly efficient. Therefore, when the condition to evaluate can resolve to an integer in the range of 1 to n, use EXECUTE, otherwise try to use CASE. If CASE it is not flexible enough, then resort to IF/ELSIF.

The IF ERRORCODE() THEN STOP(ERROR()). statement will check for an error, no matter which statement EXECUTE performed. The ReturnValue = ActionComplete statement sets up the return to the calling procedure, signalling that the user completed the file action, then the POST(EVENT:CloseWindow) terminates the ACCEPT loop, dropping control to the RETURN(ReturnValue) statement.

The OF ?Cancel code does almost the same thing, without writing anything to disk. The ReturnValue = ActionAborted assignment statement sets up the return to the calling procedure, signaling that the user aborted the file action, then the POST(EVENT:CloseWindow) terminates the ACCEPT loop, dropping control to the RETURN(ReturnValue) statement.

9. Add the ROUTINEs called in the UpdatePhones PROCEDURE definition to the end of the file:

```
SetupScreen  ROUTINE
CASE Action
OF InsertRecord
  CLEAR(Phones.Rec)
```



```

    TARGET{PROP:Text} = 'Adding New Number'
OF ChangeRecord
    TARGET{PROP:Text} = 'Updating ' & CLIP(Phones.Rec.Name) |
        & ''s Phone Number'
    IF Phones.Rec:Number[1] <> '+'
        ?Phones.Rec:Number{PROP:Text} = '@P###-###-####P'
    END
OF DeleteRecord
    TARGET{PROP:Text} = 'Deleting ' & CLIP(Phones.Rec.Name) |
        & ''s Phone Number'
    DISABLE(FIRSTFIELD(),LASTFIELD())
    ENABLE(?Ok,?Cancel)
END

SetupInsert ROUTINE
IF Action = InsertRecord
    CASE MESSAGE('International?', 'Format', ICON:Question, |
        BUTTON:Yes+BUTTON:No, BUTTON:No, 1)
    OF BUTTON:Yes
        TARGET{PROP:Text} = 'Adding New International Number'
        ?Phones.Rec:Number{PROP:Text} = '@S20'
        Phones.Rec:Number[1] = '+'
        DISPLAY
        SELECT(?, 2)
    OF BUTTON:No
        TARGET{PROP:Text} = 'Adding New Domestic Number'
        ?Phones.Rec:Number{PROP:Text} = '@P###-###-####P'
    END
END

```

The SetupScreen ROUTINE starts by evaluating CASE Action. When the user is adding a record (OF InsertRecord) the CLEAR(Phones.Rec) statement clears out the record buffer by setting all the fields to blank or zero. The TARGET{PROP:Text} = 'Adding New Number' statement uses Clarion's runtime property syntax to dynamically change the title bar text for the window to "Adding New Number." Clarion's property syntax allows you to dynamically change any property (attribute) of an APPLICATION, WINDOW, or REPORT structure in executable code. See Appendix C - Property Assignments in the Language Reference for more on properties.

TARGET is a built-in variable that always "points to" the currently open WINDOW structure. The curly braces ({}) delimit the property itself, and PROP:Text is an EQUATE (contained in PROPERTY.CLW, automatically included by the compiler like EQUATES.CLW) that identifies the parameter to the data element (in this case, the WINDOW statement).

The *OF ChangeRecord* code `TARGET{PROP:Text} = 'Updating ' & CLIP(Phones.Rec.Name) & 's Phone Number'` does the same thing, but changes the title bar text to read "Updating Someone's Phone Number." The ampersand (&) is the Clarion string concatenation operator and the `CLIP(Phones.Rec.Name)` procedure removes trailing spaces from the name. The `IF Phones.Rec.Number[1] <> '+'` structure checks for a plus sign in the first character of the Number string field. The plus sign is used here as a signal that the number is an international number.

Notice that `Phones.Rec.Number[1]` is addressing the first byte of the field as if it were an array. But you'll recall that there was no DIM attribute on the declaration (the DIM attribute declares an array). All STRING, CSTRING, and PSTRING data types in Clarion are also implicitly an array of `STRING(1),DIM(SIZE(StringField))`. This means you can directly refer to any single character in any string, whether it was declared as an array or not.

If the number is not international, the `?Phones:Rec:Number{PROP:Text} = '@P###-###-####P'` statement uses the same type of property syntax to change the control's entry picture token. Notice that `PROP:Text` is used to do this, just as it was used previously to change the window's title bar text. The reason is that `PROP:Text` refers to whatever is the parameter of the control. Therefore, on a `WINDOW('title text')` it refers to the title text, and on an `ENTRY(@S20)` it refers to the picture token (@S20).

The *OF DeleteRecord* code is similar to the *ChangeRecord* code. The `DISABLE(FIRSTFIELD(),LASTFIELD())` statement uses the `FIRSTFIELD()` and `LASTFIELD()` procedures to dim out all the controls on the window, then `ENABLE(?Ok,?Cancel)` un-dims just the OK and Cancel buttons.

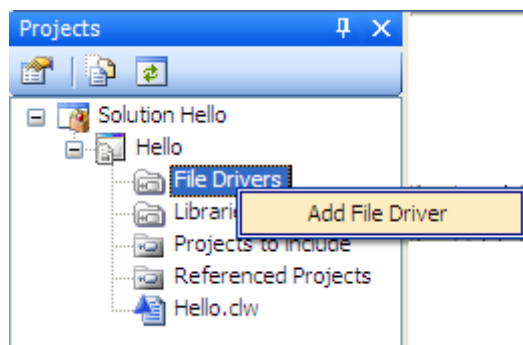
The *SetupInsert* ROUTINE executes just before the user gets to the Number ENTRY control. The *IF Action = InsertRecord* checks Action and only executes the CASE MESSAGE structure when the user is adding a new record. The MESSAGE procedure can be used to create simple Yes/No choices for users. In this case, the user is asked whether the new number is International.


The *OF BUTTON:Yes* code executes when the user has pressed the Yes button on the MESSAGE dialog. The `TARGET{PROP:Text} = 'Adding New International Number'` statement changes the window's title bar text, then `?Phones:Rec:Number{PROP:Text} = '@S20'` changes the ENTRY control's picture token. The `Phones:Rec:Number[1] = '+'` statement places a plus sign in the first character position, then `DISPLAY` displays it and `SELECT(?,2)` places the user's insertion point at the second position in the current control.

The *OF BUTTON:No* code is similar, changing the window's title bar text and the control's entry picture token.

Update the Project file

Since we added a FILE structure to the program, we have to add its file driver to the Project so it can be linked into the program. If you don't, you'll get an error message something like "link error: TOPSPEED is unresolved in file hello.obj."



10. The *Solution Explorer* should still be opened. If not, open it again by pressing **CTRL + ALT + L**.
11. RIGHT-CLICK on the *File Drivers* project node, and select *Add File Driver* from the popup menu.
12. Highlight *TOPSPEED(TPS)* in the Select File Drivers window, then press the **Select** button.
13. CLICK on the **Build and Run** button .

After exiting the program, close the Text Editor, and return to the Clarion IDE.

ABC Template Generated OOP Code

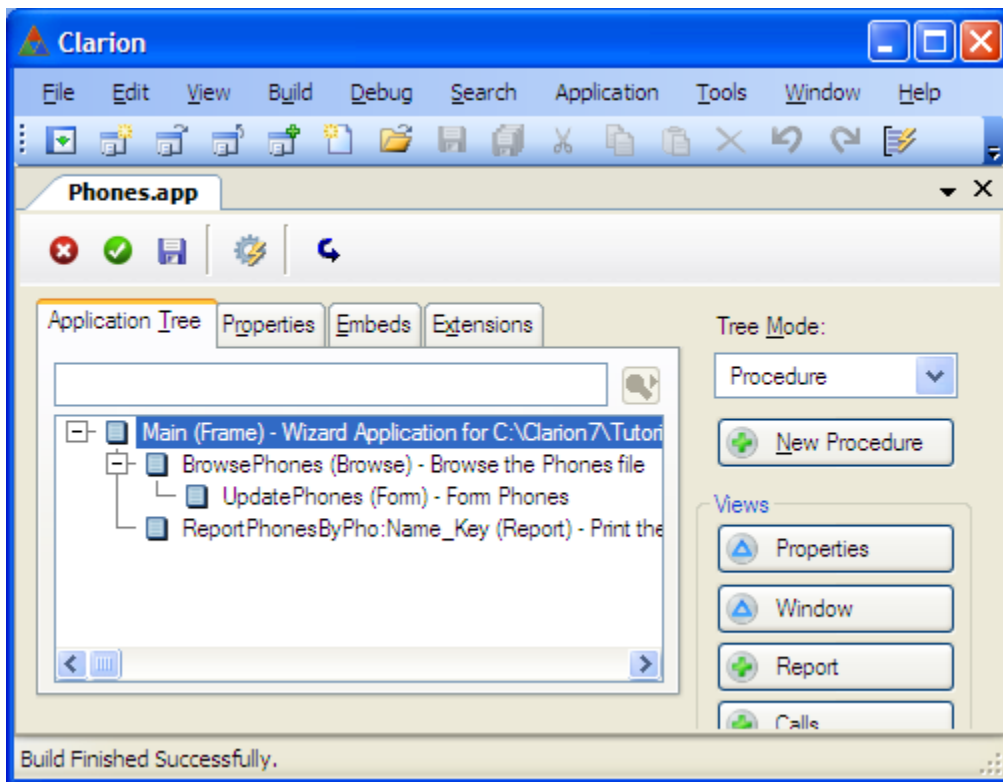
When you examine the source code generated for you by the Application Generator, you'll see some fundamental differences from the code we just wrote. The reason for that is the Application Builder Class (ABC) templates generate code that extensively uses the ABC Library—a set of Object Oriented Programming (OOP) classes.

The code we just finished writing is Procedural code, not OOP code. Now we want to take a quick look at the generated OOP code to show you how what you just learned relates to the code you'll see generated for you. This will be just a quick look to highlight the major differences.

Load a simple application

In the ...\\Lessons\\LearningClarion\\Lesson13App folder, we have created a very simple ABC template based application, based on a single *Phones* table.

1. Load the *Phones* application into the Clarion Win32 IDE, and open the Application Tree.



Look at the Program Source

Now let's examine the source code that was generated for you.

2. Switch to the **Module Tree Mode**.
3. RIGHT-CLICK on *Phones.clw*, and choose **Module** from the popup menu.

The PROGRAM Module

This is the PROGRAM module (the code should look similar to this, but may not be exactly the same). The basic structure of a Clarion OOP program is the same as the procedural. The first two statements are EQUATE statements which define constant values that the ABC Library requires. Following those are several INCLUDE statements. The INCLUDE statement tells the compiler to place the text in the named file into the program at the exact spot the INCLUDE statement.

```
PROGRAM
```

```
INCLUDE ('ABERROR.INC') ,ONCE
INCLUDE ('ABFILE.INC') ,ONCE
INCLUDE ('ABFUZZY.INC') ,ONCE
INCLUDE ('ABUTIL.INC') ,ONCE
INCLUDE ('ERRORS.CLW') ,ONCE
INCLUDE ('KEYCODES.CLW') ,ONCE
```

The first four INCLUDE files (all starting with "AB" and ending with ".INC") contain CLASS definitions for some of the ABC Library classes. The next two INCLUDE files (all ending with ".CLW") contain a number of standard EQUATE statements used by the ABC Template generated code and ABC Library classes. The ONCE attribute on each means that if they have already been included, they won't be included again.

```
MAP
    MODULE ('PHONEBC.CLW')
DctInit    PROCEDURE
DctKill    PROCEDURE
    END
!--- Application Global and Exported Procedure Definitions -----
    MODULE ('PHONE001.CLW')
Main       PROCEDURE    !Clarion 5.6 Quick Application
    END
END
```

The MAP structure contains two MODULE structures. The first declares two procedures *DctInit* and *DctKill* that are defined in the PHONEBC.CLW file. These two procedures are generated for you to properly initialize (and un-initialize) your data files for use by the ABC Library. The second MODULE structure simply names the application's first procedure to call (in this case, *Main*).

```
Phones      FILE, DRIVER ( ' TOPSPEED ' ) , PRE ( PHO ) , CREATE , BINDABLE , THREAD
Name_Key    KEY ( PHO : Name ) , DUP , NOCASE
Record      RECORD , PRE ( )
Name        STRING ( 20 )
Number      STRING ( 20 )
            END
            END
```

The above is your file declaration.

These next two lines of code are your first OOP statements:

```
Access:Phones      &FileManager, THREAD
Relate:Phones      &RelationManager, THREAD
```

The Access:Phones statement declares a reference to a FileManager object, while the Relate:Phones statement declares a reference to a RelationManager object. These two references are initialized for you by the DctInit procedure, and un-initialized for you by the DctKill procedure. Both of these procedures are embedded in the Dictionary Class, which is discussed below. These are very important statements, because they define the manner in which you will address the data file in your OOP code. The THREAD attribute declares that the CLASS is allocated memory separately for each execution thread in the program.

The next three lines of code declare a GlobalErrorStatus object, GlobalErrors object, a search object, and an INIMgr object.

```
FuzzyMatcher      FuzzyClass
GlobalErrorStatus  ErrorStatusClass, THREAD
GlobalErrors       ErrorClass, THREAD
INIMgr             INIClass
```

These objects handle all search criteria, errors, and your program's .INI file (if any), respectively. These objects are used extensively by the other ABC Library classes, so must be present (as you will shortly see).

```
GlobalRequest  BYTE ( 0 ) , THREAD
GlobalResponse BYTE ( 0 ) , THREAD
VCRRequest     LONG ( 0 ) , THREAD
```

Following that are three Global variable declarations that the ABC Templates use to communicate between procedures. Notice that the global variables all have the THREAD attribute. THREAD is required since the ABC Templates generate an MDI application by default, which makes it necessary to have separate copies of global variables for each active thread (which is what the THREAD attribute does).

```

Dictionary      CLASS, THREAD
Construct       PROCEDURE
Destruct        PROCEDURE
                END

```

The *Dictionary Class* initializes both File and Relation Managers (contained in the *DctInit* generated procedure, called from *Construct*) whenever a new thread is started. Likewise, the Managers' kill code (contained in *DctKill*, called from *Destruct*) must be called whenever a thread is terminated. The Construct is implicitly called when the global Dictionary object is instantiated.

The global CODE section only has eight lines of code:

```

CODE
GlobalErrors.Init(GlobalErrorStatus)

FuzzyMatcher.Init                                ! Initilaize the browse 'f
uzzy matcher'

FuzzyMatcher.SetOption(MatchOption:NoCase, 1)    ! Configure case matching
FuzzyMatcher.SetOption(MatchOption:WordOnly, 0)  ! Configure 'word only' ma
tching

INIMgr.Init('phones.INI', NVD_INI)               ! Configure INIManager to
use INI file

Main
INIMgr.Update
INIMgr.Kill                                       ! Destroy INI manager
FuzzyMatcher.Kill                               ! Destroy fuzzy matcher

```

The first, second, and the fourth statements call *Init* methods (in OOP parlance, a procedure in a class is called a "method"). These are the constructor methods for the FuzzyMatcher and INIMgr objects. You'll notice that the INIMgr.Init method takes two parameters. In the ABC Library, all object constructor methods are explicitly called and are named Init. There are several reasons for this. The Clarion language does support automatic object constructors (and destructors) and you are perfectly welcome to use them in any classes you write. However, automatic constructors cannot receive parameters, and many of the ABC Library Init methods must receive parameters. Therefore, for consistency, all ABC object constructor methods are explicitly called and named Init. This has the added benefit of enhanced code readability, since you can explicitly see that a constructor is executing, whereas with automatic constructors you'd have to look at the CLASS declaration to see if there is one to execute or not.

The call to the *Main* procedure begins execution of the rest of your program for your user. Once the user returns from the Main procedure, the INIMgr and FuzzyMatcher perform some necessary update and cleanup operations before the return to the operating system.

```
Dictionary.Construct PROCEDURE
```

```
CODE
IF THREAD() <> 1
    DctInit()
END
```

Dictionary.Destruct PROCEDURE

```
CODE
DctKill()
```

The *DctInit* procedure call initializes the *Access:Phones* and *Relate:Phones* reference variables so the template generated code (and any embed code that you write) can refer to the data file methods using *Access:Phones.Methodname* or *Relate:Phones.Methodname* syntax. This gives you a consistent way to reference any file in an ABC Template generated program—each FILE will have corresponding *Access:* and *Relate:* objects.

The *DctKill* procedure call performs some necessary update and cleanup operations before the return to the operating system.

The UpdatePhones Module

Now let's examine the source code that was generated for you for one of your procedures. We'll look at the *UpdatePhones* procedure as a representative, since all ABC Template generated procedures will basically follow the same form (again, your code should look similar to this, but may not be exactly the same).

1. Choose **File ► Close ► File**, or press **CTRL + F4**.
2. In the Application Tree, highlight *UpdatePhones*, then RIGHT-CLICK and choose **Module** from the popup menu.

The first thing to notice is the **MEMBER** statement on the first line. This is a required statement telling the compiler which PROGRAM module this source file "belongs" to. It also marks the beginning of a Module Data Section—an area of source code where you can make data declarations which are visible to any procedure in the same source module, but not outside that module (see *Data Declaration and Memory Allocation* in the Language Reference).

```
MEMBER('Phones.clw')          ! This is a MEMBER module
INCLUDE('ABRESIZE.INC'), ONCE
INCLUDE('ABTOOLBA.INC'), ONCE
INCLUDE('ABWINDOW.INC'), ONCE
MAP
    INCLUDE('PHONE004.INC'), ONCE !Local module prodecure declarations
END
```

The three INCLUDE files contain CLASS definitions for some of the ABC Library classes. Notice that the list of INCLUDE files here is different than the list at the global level. You only need to

INCLUDE the class definitions that the compiler needs to know about to compile this single source code module. That's why the list of INCLUDE files will likely be a bit different from module to module.

Notice the MAP structure. By default, the ABC Templates generate "local MAPs" for you containing INCLUDE statements to bring in the prototypes of the procedures defined in the module and any procedures called from the module. This allows for more efficient compilation, because you'll only get a global re-compile of your code when you actually change some global data item, and not just by adding a new procedure to your application. In this case, there are no other procedures called from this module.

The PROCEDURE statement begins the UpdatePhones procedure (which also terminates the Module Data Section).

```
UpdatePhones PROCEDURE      !Generated from procedure template - Window
```

```
CurrentTab      STRING(80)
ActionMessage   CSTRING(40)
History::PHO:Record  LIKE(PHO:RECORD),THREAD
```

Following the PROCEDURE statement are three declaration statements. The first two are common to most ABC Template generated procedures. They provide local flags used internally by the template-generated code. The ActionMessage and History::PHO:Record declarations are specific to a Form procedure. They declare a user message and a "save area" for use by the Field History Key ("ditto" key) functionality provided on the toolbar.

After the WINDOW structure comes the following object declarations:

```
ThisWindow  CLASS(WindowManager)
Ask          PROCEDURE(),DERIVED
Init         PROCEDURE(),BYTE,PROC,DERIVED
Kill         PROCEDURE(),BYTE,PROC,DERIVED
Run          PROCEDURE(),BYTE,PROC,DERIVED
TakeAccepted PROCEDURE(),BYTE,PROC,DERIVED
            END

Toolbar      ToolbarClass
ToolBarForm  ToolbarUpdateClass
Resizer      CLASS(WindowResizeClass)
Init         PROCEDURE(BYTE AppStrategy=AppStrategy:Resize,|
                     BYTE SetWindowMinSize=False,|
                     BYTE SetWindowMaxSize=False)
            END
```

The two after the `ThisWindow` class are simple object declarations which create the local objects which will enable the user to use the toolbar. The next class enables resizing the window at runtime. The interesting code here is the `ThisWindow` CLASS declaration. This CLASS structure declares an object derived from the `WindowManager` class in which the `Ask`, `Init`, and `Kill`, `Run`, and `TakeAccepted` methods of the parent class (`WindowManager`) are overridden locally. These are all VIRTUAL methods, which means that all the methods inherited from the `WindowManager` class will be able to call the overridden methods. This is a very important concept in OOP.

Skipping the rest of the data declarations is all of the executable code in your procedure:

CODE

```
GlobalResponse = ThisWindow.Run()
```

That's right—one single, solitary statement! The call to `ThisWindow.Run` is the only executable code in your entire procedure! So, you ask, "Where's all the code that provides all the functionality I can obviously see happening when I run the program?" The answer is, "In the ABC Library!" or, at least most of it is! The good news is that all the standard code to operate any procedure is built in to the ABC Library, which makes your application's "footprint" very small, since all your procedures share the same set of common code which has been extensively debugged (and so, is not likely to introduce any bugs into your programs).

All the functionality that must be explicit to this one single procedure is generated for you in the overridden methods. In this procedure's case, there are only three methods that needed to be overridden. Depending on the functionality you request in the procedure, the ABC Templates will override different methods, as needed. You also have embed points available in every method it is possible to override, so you can easily "force" the templates to override any method for which you need slightly different functionality by simply adding your own code into those embed points (using the Embeditor in the Application Generator).

OK, so let's look at the overridden methods for this procedure.

`ThisWindow.Ask` PROCEDURE

CODE

```
CASE SELF.Request
  OF ViewRecord
    ActionMessage = 'View Row'
  OF InsertRecord
    ActionMessage = 'Adding a phones Row'
  OF ChangeRecord
    ActionMessage = 'Changing a phones Row'
END
QuickWindow{Prop:Text} = ActionMessage
PARENT.Ask
```

The really interesting line of code in the `ThisWindow.Ask` PROCEDURE is last. The last statement, `PARENT.Ask`, calls the parent method that this method has overridden to execute its standard functionality. The `PARENT` keyword is very powerful, because it allows an overridden method in a derived class to call upon the method it replaces to "do its thing" allowing the overridden method to incrementally extend the parent method's functionality.

```

ThisWindow.Init PROCEDURE
ReturnValue          BYTE,AUTO

CODE
GlobalErrors.SetProcedureName('Updatephones')
SELF.Request = GlobalRequest
ReturnValue = PARENT.Init()
IF ReturnValue THEN RETURN ReturnValue.
SELF.FirstField = ?PHO:Name:Prompt
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
CLEAR(GlobalRequest)
CLEAR(GlobalResponse)
SELF.AddItem(Toolbar)
SELF.HistoryKey = CtrlH
SELF.AddHistoryFile(PHO:Record,History::PHO:Record)
SELF.AddHistoryField(?PHO:Name,1)
SELF.AddHistoryField(?PHO:Number,2)
SELF.AddUpdateFile(Access:phones)
SELF.AddItem(?Cancel,RequestCancelled)
Relate:phones.Open
SELF.FilesOpened = True
SELF.Primary &= Relate:phones
IF SELF.Request = ViewRecord AND NOT SELF.BatchProcessing
    SELF.InsertAction = Insert:None
    SELF.DeleteAction = Delete:None
    SELF.ChangeAction = Change:None
    SELF.CancelAction = Cancel:Cancel
    SELF.OkControl = 0
ELSE
    SELF.ChangeAction = Change:Caller                ! Changes allowed
    SELF.CancelAction = Cancel:Cancel+Cancel:Query    ! Confirm cancel
    SELF.OkControl = ?OK
    IF SELF.PrimeUpdate() THEN RETURN Level:Notify.
END
OPEN(QuickWindow)
SELF.Opened=True
Do DefineListboxStyle
IF SELF.Request = ViewRecord          ! Configure controls for View Only mode
    ?PHO:Name{PROP:ReadOnly} = True
    ?PHO:Number{PROP:ReadOnly} = True

```

```
END

Resizer.Init(AppStrategy:Surface,Resize:SetMinSize)

INIMgr.Fetch('UpdatePhones',QuickWindow)    ! Restore window settings from non-
-volatile store

Resizer.Resize                               !Reset required after window size altere
d by INI manager

Resizer.Reset

SELF.SetAlerts()

RETURN ReturnValue
```

There are several interesting lines of code in the ThisWindow.Init PROCEDURE. This is the ThisWindow object's constructor method, so all the code in it performs the initialization tasks specifically required by the UpdatePhones procedure.

The second statement, SELF.Request = GlobalRequest, retrieves the global variable's value and places it in the SELF.Request property. SELF is another powerful Clarion OOP keyword, which always means "the current object" or "me." SELF is the object prefix which allows class methods to be written generically to refer to whichever object instance of a class is currently executing.

The next statement calls the PARENT.Init() method (the parent method's code to perform all its standard functions) before the rest of the procedure-specific initialization code executes. Following that are a number of statements which initialize various necessary properties.

The Relate:Phones.Open statement opens the Phones data file for processing, and if there were any related child files needed for Referential Integrity processing in this procedure, it would also open them (there aren't, in this case).

```
ThisWindow.Kill PROCEDURE

ReturnValue          BYTE,AUTO

CODE

ReturnValue = PARENT.Kill()

IF ReturnValue THEN RETURN ReturnValue.

IF SELF.FilesOpened

    Relate:phones.Close

END

IF SELF.Opened

    INIMgr.Update('UpdatePhones',QuickWindow) ! Save window data to non-
-volatile store

END

GlobalErrors.SetProcedureName

RETURN ReturnValue
```

In addition to calling the PARENT.Kill() method to perform all the standard closedown functionality (like closing the window), ThisWindow.Kill closes all the files opened in the procedure, then sets the GlobalResponse variable.

3. From the IDE Menu, select **File ▶ Close ▶ File**.

Where to Go From Here?

This lesson has been just a brief introduction to the Clarion programming language and the ABC Template generated code. There is much more to the Clarion language and the ABC Library than has been covered here, so there's a lot more to learn. So where do you go from here?

- The articles in the Programmer's Guide. These essays cover various aspects of programming in the Clarion language. Although they do not take a tutorial approach, they do provide in depth information on the specific areas they cover, and there are several articles which deal specifically with OOP in Clarion.
- The ABC Library Reference (Vol I and II) fully documents the ABC Libraries. The Template Guide documents the ABC Templates. This is your prime resource for how to get the most out of Clarion's Application Generator technology.
- The Language Reference is the "Bible" of the Clarion language. Reading the entire manual is always a good idea.
- Examine and dissect source code generated for you by the Application Generator. After doing this lesson, the basic structure of the code should look familiar, even if the specific logic does not.
- The User's Guide contains lessons on using the Debuggers. This will allow you to step through your Clarion code as it executes to see exactly what effect each statement has on the operation of your program.
- SoftVelocity offers educational seminars at various locations. Call Customer Service at (800) 354-5444 or (954) 785-4555 to enroll.
- Join (or form) a local Clarion User's Group and participate in joint study projects.
- Participate in SoftVelocity's Internet Newsgroup (comp.lang.clarion and more) to network with other Clarion programmers all around the world (Strongly recommended!).

The Getting Started and Learning Clarion lesson applications

The completed application and dictionary files are located in the
... \Lessons\GettingStarted\Solution and \CLARION ROOT\Lessons\LearningClarion\Solution
directory. These are the files created by following the steps in the Getting Started and Learning
Clarion documents.

There are two other lessons contained in this folder. The files used for the Online User's Guide's
Debugger lesson can be found in the
... \Lessons\Debugger folder.

A lesson targeting the use of the Dictionary Synchronizer The files used for the Online User's
Guide's Synchronizer lesson can be found in the
... \Lessons\SQL folder.

Good luck and keep going—the programming power that Clarion puts at your fingertips just keeps
on growing as you learn more!

Index

1 - Planning the Application	7	4 - Adding Keys and Relations	35
10 - Control and Extension Templates	117	5 - Importing Existing Data	45
11 - Advanced Topics	127	6 - Starting the Application.....	51
12 - Creating Reports	149	7 - Creating a Browse	75
13 - Clarion Language Lesson	177	8 - Creating an Update Form	91
2 - Creating a Data Dictionary	13	9 - Copying Procedures	107
3 - Adding Tables and Columns	21	Intro to Learning Clarion	1