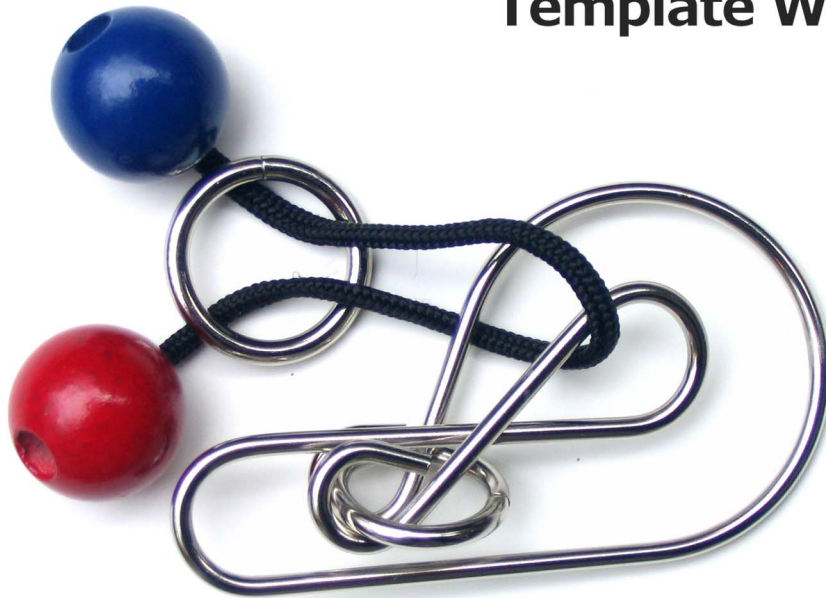


S e c o n d   E d i t i o n

# Programming Objects in Clarion<sup>TM</sup>

**A Guide to Writing  
Object-Oriented Clarion Code  
and ABC-Compatible  
Template Wrappers**



**by Russell B. Eggen**

# **PROGRAMMING OBJECTS IN CLARION**

by Russell B. Eggen

# Programming Objects in Clarion

by Russell B. Eggen

Copyright © 2004 by Russell B. Eggen

Published by:  
CoveComm Inc.  
1036 McMillan Ave  
Winnipeg, MB R3M 0V8

All rights reserved. No part of this book may be reproduced or transmitted in any form by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. For information on getting permission for reprints or excerpts, contact [books@clarionmag.com](mailto:books@clarionmag.com).

See <http://www.clarionmag.com/books> for source downloads and errata

Library and Archives Canada Cataloguing in Publication

Eggen, Russell B., 1959-  
Programming objects in Clarion / Russell B. Eggen. – 2nd ed.

Includes index.  
ISBN 0-9689553-2-0

1. Clarion for Windows (Computer program language) 2. Object-oriented programming (Computer science) I. Title.

QA76.73.C22E34 2004 005.13'3 C2004-903848-6

The information in this book, and any source code and/or information in downloads referenced in this book, is distributed on as “As Is” basis, without warranty. While every precaution has been taken in the preparation of this book, neither the editor nor the publisher assumes responsibility for errors or omissions in the book, or in the instructions and/or source code in downloads referenced in this book.

Clarion™ is a trademark of SoftVelocity, Inc. This and other trademarks which may appear in the book are used in an editorial fashion only and to the benefit of the trademark owner, with no intention of infringement of the trademark.

## Printing History

July 2004	Second Edition
November 2004	Revised Second Edition
January 2006	PDF Edition

Print version ISBN 0-9689553-2-0

## **Russell B. Eggen**

Russ Eggen is a former Topspeed/SoftVelocity employee who has conducted Clarion training sessions worldwide. Russ is a sought-after speaker and a highly effective trainer; he is well-known for his ability to communicate difficult concepts in meaningful ways.

## **Terms of use**

All rights reserved. No part of this book may be reproduced or transmitted in any form by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

As a licensed user of this e-book you may keep multiple copies for your personal use only.

Copyright © 1999-2006 by CoveComm Inc.

Published by:

CoveComm Inc.  
1036 McMillan Ave  
Winnipeg, MB R3M 0V8

T. 204.943.5165

F. 204.477-4518

[books@clarionmag.com](mailto:books@clarionmag.com)



# Table of Contents

---

<b>Foreword</b>	<b>i</b>
<b>Acknowledgements</b>	<b>iii</b>
<b>Struggling with Objects</b>	<b>1</b>
How to study this subject . . . . .	3
The sequence of study topics . . . . .	4
<i>What about ABC?</i>	
<i>Coding your own classes</i>	
<i>Template Wrappers</i>	
<b>The Theory of OOP</b>	<b>7</b>
OOP defined . . . . .	7
Your first exercise (under a minute) . . . . .	8
Encapsulation . . . . .	9
<i>The CLASS statement</i>	
Instantiation . . . . .	11
<i>How to instantiate a Class</i>	
<i>Which instantiation method to use?</i>	
Dot syntax . . . . .	15
<i>Object naming in method code</i>	
<i>SELF</i>	
<i>The Microwave object</i>	
Constructors and Destructors . . . . .	21
<i>Why use them?</i>	
<i>Why not use automatic constructors/destructors?</i>	
Extreme Encapsulation . . . . .	23
<i>When to use the Private Attribute</i>	
Inheritance and Derivation . . . . .	24
<i>Extending a class</i>	
<i>Overriding a class</i>	
<i>Multiple Inheritance</i>	
<i>Composition</i>	
Moderate Encapsulation . . . . .	27
Overriding . . . . .	28

More on Constructors and Destructors . . . . .	29
PARENT . . . . .	30
Polymorphism . . . . .	32
<i>Virtual methods</i>	
Late Binding . . . . .	36
Local Derived Methods . . . . .	36
Interfaces . . . . .	37
<i>Declaring the INTERFACE</i>	
Writing Interface Methods . . . . .	40
<i>The Dark Side of Interfaces</i>	
<i>The Good Side of Interfaces</i>	
Summary . . . . .	42

## **Applying OOP in Clarion 43**

Where to Start? . . . . .	43
What about existing applications? . . . . .	44
Back to Encapsulation . . . . .	45
<i>Function Overloading</i>	
<i>How to code large CLASS structures</i>	
Instantiation and Inheritance . . . . .	48
<i>Module and Link</i>	
<i>Instantiating multiple objects</i>	
<i>Manual instantiation</i>	
<i>Do not forget to clean up after yourself</i>	
Constructors and Destructors . . . . .	52
Derivation . . . . .	54
Overriding . . . . .	54
Virtual methods . . . . .	56
<i>The Debugger as Teacher</i>	
<i>The Virtual Apple Pie</i>	
<i>Moral of the story?</i>	
How to Make Any Kind of Pie . . . . .	61
<i>One Last Concept</i>	
Summary . . . . .	70

## **ABC and OOP 71**

Global objects . . . . .	72
Global embeds . . . . .	74
<i>How does ABC do files?</i>	
<i>DctInit</i>	



<i>DctKill</i>	
<i>FilesInit</i>	
Local embeds . . . . .	82
<i>ABC classes in relation to templates</i>	
<i>Browse procedures</i>	
<i>When to use an ABC embed or a control specific embed</i>	
Working with ABC classes . . . . .	95
<i>The ErrorClass</i>	
<i>Deriving from ErrorClass</i>	
<i>Installing your Custom Classes</i>	
<i>Using Custom Classes in an Application</i>	

## **Coding Objects 107**

Design Considerations . . . . .	107
ABC Relation Tree vs. Relational Object Tree . . . . .	110
Designs. . . . .	110
<i>Navigation</i>	
<i>Edits</i>	
A Fast Way to Code a Class . . . . .	112
<i>The Start of the Relation Tree Class</i>	
<i>Adding Properties</i>	
<i>Adding the Methods</i>	
<i>The Code for the Methods</i>	
<i>Writing the code</i>	
<i>A New Project</i>	
<i>Summary</i>	
Generic class. . . . .	126
<i>Automatic constructors and destructors</i>	
<i>About Init and Kill</i>	
<i>How much is that List in the Window?</i>	
<i>Keep Method Code Simple as Possible</i>	
The RelationTree Class Methods. . . . .	134
<i>RefreshTree()</i>	
<i>LoadLevel()</i>	
<i>UnloadLevel()</i>	
<i>NextParent()</i>	
<i>PreviousParent()</i>	
<i>NextLevel()</i>	
<i>PreviousLevel()</i>	
<i>NextSavedLevel()</i>	
<i>PreviousSavedLevel()</i>	
<i>NextRecord()</i>	

<i>PreviousRecord()</i>	
Virtual Stub Methods . . . . .	140
<i>AssignButtons()</i>	
<i>AddEntryServer()</i>	
<i>EditEntryServer()</i>	
<i>RemoveEntryServer()</i>	
Edit Methods . . . . .	141
<i>AddEntry()</i>	
<i>EditEntry()</i>	
<i>DeleteEntry()</i>	
<i>UpdateLoop()</i>	
<i>Contract and Expand</i>	

## Writing Template Wrappers 145

Templates and Objects . . . . .	146
<i>Class reader</i>	
Writing a new template. . . . .	147
<i>The #SYSTEM statement</i>	
<i>#APPLICATION vs. #EXTENSION</i>	
<i>#CONTROL Template</i>	
<i>Loading ABC classes in memory</i>	
<i>Setting up Class Items and OOP Defaults</i>	
<i>Adding Global Prompts</i>	
<i>Exporting the class</i>	
<i>Adding Local Prompts</i>	
<i>Global Symbols and Objects</i>	
<i>A List of Objects</i>	
<i>Adding Local Declarations</i>	
<i>How to Add New Methods</i>	
<i>Clarion 5.5 vs. Clarion 6</i>	
<i>Generating the embed tree</i>	
<i>How to Call the Parent</i>	
<i>Adding Embed Points</i>	
<i>Scoping Issues</i>	
The Remaining Control Template Code . . . . .	173
<i>The Control Interface Dialogs</i>	
<i>Formula Editor</i>	
<i>The Design of the Control Template</i>	
<i>What about icons?</i>	
<i>Local Data Embed</i>	
<i>ThisWindow.Init()</i>	
<i>ThisWindow.Kill()</i>	

Window Event Handling . . . . .	182
<i>EVENT:GainFocus</i>	
Control Event Handling . . . . .	182
<i>EVENT:NewSelection</i>	
<i>EVENT:Expanded and EVENT:Contracted</i>	
<i>EVENT:AlertKey</i>	
The Generation of Method Code. . . . .	187
The RelationTree Methods . . . . .	188
Updating the RelationTree . . . . .	200
Other Control Templates . . . . .	204
<i>RelObjTreeUpdateButtons</i>	
<i>Control Event Handling</i>	
<i>Toolbar Issues</i>	
<i>Special Conditions</i>	
<i>Assignment of Edit Buttons</i>	
<i>RelObjTreeExpandContractButtons</i>	

## **Template User Guide** **213**

Registration. . . . .	213
Adding the control template . . . . .	215
Setting up the Relation Tree . . . . .	217
<i>Colors tab</i>	
<i>Conditional Color Assignments</i>	
<i>Icons</i>	
<i>Conditional Icon Use</i>	
<i>Local Objects</i>	
<i>Classes</i>	
<i>Finishing the Tree List</i>	
<i>The Embeds</i>	
Dynamic Link Library Issues. . . . .	235
Congratulations! . . . . .	237

## **Appendix A: Getting Support** **239**

Getting the source code . . . . .	239
Errata . . . . .	239

## **Index** **241**



# FOREWORD

Whether you are brand new to Clarion, or an experienced developer since the "Version 1.0" days, or somewhere in between, you've probably heard of Russ Eggen.

First, try exploring the public newsgroups. No matter what the topic may be, if it is remotely related to Clarion, Russ will be there. If you ever have an opportunity to visit any of the scheduled online chat forums, Russ will be there. If there is a developer's conference scheduled in your area, chances are that you will meet Russ.

I have always thought that there is a "Team Eggen" somewhere; 10 college interns who secretly work for Russ by writing and posting and answering his many messages and responding to comments on his articles. But no, it is just one man behind all of this effort.

My point to all of this is that you will realize as you read this book that Russ has a refreshing and unstoppable passion for Clarion. You will immediately see that he really enjoys his work, which in part is writing programs and solving business problems using Clarion. More importantly, he likes to share those binary battles and software conquests with all Clarion developers.

The migration from writing procedure-based code to the object oriented approach can be a dry and tedious journey for many. What Russ has done with this book is to upgrade your journey to first-class. His passion and love for what he is doing is clearly reflected in his writing, and the time you spend reading these pages will be well worth the investment. After reading this book, I hope that you find yourself

secretly inducted into "Team Eggen." As the membership of this team continues to increase, it will certainly speak well for the future and prosperity of all Clarion developers. Thanks again, Russ!

Bob Foreman  
SoftVelocity Inc.

# ACKNOWLEDGEMENTS

During the writing of any book, an author is assisted by many people. This book is no exception. I wish to express my thanks to the following people:

Stephen Bottomley for providing the example for the pie interface.

Andrew Guidroz for the questions about the `ErrorClass`, thus bringing about the idea on how to trap specific error codes with the `ErrorClass`.

Grateful acknowledgment goes to Andrew Ireland. If it weren't for his coding disciplines, my code would be harder to read, let alone maintain. Also, his simple ideas lead to simple solutions, regardless of the complexities of the problem addressed. After all, what are friends for?

I would be remiss without expressing some thanks (in no particular order) to Mark Goldberg, Dave Harms, Tom Hebenstreit and Mike Gould for their assistance, comments and encouragement.

Special thanks to Robert Healey for the swell graphics. If not for his efforts, you would have to suffer my horrible stick figures.

To my wife, Kathleen, for feeding me, offering encouragement ("that's nice, dear" and pats on the back), proofreading (catching those typos), ordering me to get some sleep once in a while, and just putting up with me.

And most of all, my thanks goes to the Clarion community. They are constantly curious, willing to learn and not afraid to ask questions in the quest to greater understanding. I dedicate this book to them to them.

And last, but certainly not least, the fine crew at Softvelocity. Without their tireless (and often thankless) actions, Clarion might not be in existence today.

Russ Eggen  
[www.radfusion.com](http://www.radfusion.com)





# Chapter 1

---

## STRUGGLING WITH OBJECTS

When I was first asked to present the topic of Object Oriented Programming at the East Tennessee Conference in May of 2002 (also known as ETC III), I decided to ask the Clarion community what they perceived to be the problem with working with objects. I am glad I asked, as the answers were not what I was expecting. These answers did paint a picture, but not just any picture. And when someone answered one of my questions about OOP, I asked myself, “Why did they say that?”

My investigation revealed something that I had never really considered. Despite the many good articles, books, documentation and newsgroup threads available to Clarion developers, many still found dealing with objects difficult!

I discovered a few reasons why this is so. Some developers said that their applications were too far into the development cycle to make the switch to ABC (the OOP based templates). Other developers were still thinking in linear, instead of object-oriented, terms when writing code, and when trying to write ABC code.

ABC has its critics, and some of their points are valid. These criticisms often made a good enough excuse not to use ABC. And Clarion developers have put a lot of “sweat equity” into learning the non-OOP Clarion (sometimes called “Legacy”) template set. That learning curve was not exactly easy as these templates produce a lot of excess code. Clarion programs were fat with unneeded code and it took time to wade through it. The original Clarion templates (I don't know how to be diplomatic about this) are awful. I could state many arguments supporting that claim, but I am drifting off topic.

In response to concerns about the new object-oriented code, the designers of ABC told everyone, “Just never mind about this new code we’re generating!” Now, that was not a clever idea, despite the best efforts of Topspeed staff (of which I was a member). What I mean by that is that if you are to upset accepted ideas, you had better be prepared to replace them. On the other hand, the education department certainly put their best efforts in this, and so did those in documentation, but there were still a lot of worries about what that ABC code really meant.

During the Topspeed days, you may recall a group of talented developers known as Team Topspeed (TTS). These people knew Clarion very well, and each member was a specialist in some area. Moreover, they were quite willing to share their knowledge with others.

I am going to let the cat out of the bag here, but TTS also struggled with ABC in the many beta stages. They objected on many of the same issues that later developers would struggle. They were quite willing to learn, but they did raise some legitimate concerns.

For example, one TTS member made a very good point. When discussing the issue of code re-use, his rebuttal was “So what? I use routines for that. Show me why OOP is better.” Another member questioned the wisdom of naming a report object “ThisWindow”. The small People application was the result of that conversation.

The whole point of this is simply to say that others went through the pain of shifting their style of coding too. I include myself in this group. I learned the procedural, top- down style of programming before I learned OOP. Procedural code works. It can be very difficult indeed to completely change a way of programming, let alone give up what works.

During my years as a Topspeed instructor, I had the honor (or was that sheer terror?) of teaching the very first ABC class. If that was not bad enough, ABC was in beta version 3 (out of 6) at the time. For those that were around, beta 3 was a complete re-write as beta 2 was unworkable. I was nowhere near the TS offices where I could easily get some help.

There I was, standing in front of about 10 students, all of them waiting to see and hear about this new “silver bullet” called ABC. I was under no pressure at all! Looking back on that experience, it was rough, but I think I got most folks through the difficult bits.

In later classes, I kept revising the OOP lessons based on surveys students filled out at the end of class, until I got to the point where I taught one class and three of

those students were just amazed at how easy ABC really was. That was a nice thing for them to say, but I was aiming higher.

“Could you apply this information in your current projects?” I asked. “Absolutely!” they said. I grinned and informed them that within hours they were going to prove this point (I had some lab exercises waiting). And they proved they could apply the materials I presented.

I knew then that I could deliver the theory of ABC, and that the theory, when applied, worked. This book is the result of those experiences.

Now let me explain how all of this flows.

## How to study this subject

---

Exactly how the context node

Based on those teaching experiences, and what my survey revealed, I believe I have assembled a great collection of information about coding objects. However, I am changing my tactics slightly from those classroom experiences. I thought I could just repeat my education lecture, but many have heard or read this before. This material needs to be fresh. It needs to address problems many Clarion developers face head on. It must be fun. It needs real life code in as little space as possible, to assist understanding. It must have some practical use.

If you cannot apply what you have studied, no matter how well you understand it, you cannot use the theory. However, I believe exercises should be simple. None should take more than five minutes to complete. I refuse to use cruel trick questions. I'm not interested in who can be clever, I am only interested in whether you can apply what you've studied.

If I discuss LOOP statements, then I expect you to code a LOOP structure. Any LOOP structure, I don't care for fancy. You can add “fancy” by yourself later on. “Fancy” is your job, not mine.

To ensure your knowledge increases, each new topic builds on the last topic. I surely hope that you “get” OOP by reading and applying the material in this book. Objects can make your Clarion applications smaller, faster, and far easier to maintain. However, you must be willing to do a bit of work up front. That is the price you pay. That sounds bad, so let me give you some good news.

If you design and code any object well, you will do it once, but you can use that code everywhere, greatly reducing your future workload.

That price does not sound so bad now!

## The sequence of study topics

---

In live class room settings I've discovered that the proper sequence of topics starts with the basics, and moves the student through each successive topic, building knowledge and skills. Thus, the sequence of study in this book is as follows:

- Object Oriented Programming
- ABC templates
- Converting existing code to objects
- Coding template wrappers for those objects.

### ❑ What about ABC?

Object oriented programming has been part of Clarion for Windows 2.0 (yes, 2.0!) but it didn't show up on most Clarion developers' radar screens until Clarion 4, when ABC made its debut. At first, the ABC objects did such a good job of doing the "OOP bits" for you, that I used to be in the camp of those thinking they did not need to know OOP. After all, I could make a fully functioning ABC application. Many others could too, so at first the importance of understanding the underlying technology fell by the wayside.

However, this success soon created new problems. Many Clarion developers were asking about embed points. Where could they add some code to make a procedure do what they wanted?

This problem was directly symptomatic of not understanding the underlying concepts of OOP. I came to that conclusion after many years of teaching Clarion. Before one class, I decided I would hit OOP theory hard. I wanted to see if my theory and observation were indeed correct. I found that the end-of-class lab exercises were much easier for the students to not only do, but complete. I had my evidence.

This is why there is an ABC section *after* the OOP section. Chapter 2 covers basic OOP theory, Chapter 3 shows how the Clarion language implements OOP, and

Chapter 4 examines the ABC class library. Once you understand what ABC is doing from an OOP viewpoint, finding embeds is easy, and in the worst case, still easier than you think.

### ❑ **Coding your own classes**

No book of this type is complete without a discussion of converting existing code to objects. Chapter 5 covers this topic, using as its basis some code that ships with Clarion. Many ABC users would be surprised to find that ABC's relation tree is not OOP at all. There are no objects for relation trees in ABC! In this case the ABC templates borrow heavily from the old legacy templates. They do call certain objects, but that is about all the “conversion” to ABC there is for the relation tree.

In Chapter 5, you go through a guided tour of the conversion and writing your own class. When you are done, you have something useful as well.

### ❑ **Template Wrappers**

Chapter 6 covers how to write template wrappers for objects you create. This builds on the previous chapter. You will have template wrappers for the relation tree class. And finally, Chapter 7 explains in detail how to use the template you created in Chapter 6.

Now that that is out of the way, let me continue with the theory of OOP, which was the sole topic in my presentation at ETC III.



# Chapter 2

---

## THE THEORY OF OOP

In every subject one studies, there must be a basic, simple starting point. Let me get this part out of the way right now so I can get into the “magic”.

OOP is spelled O-O-P.

Did everyone get that? Excellent! Now, I will build on this concept.

OOP is not POO spelled backwards. How does that seem for everyone? Is anyone lost?

Am I going too fast? Is everyone having fun? No one is unconscious and everyone is wide-awake? Then I will proceed.

### OOP defined

---

When I study something new, I must have a list of keywords with workable definitions in order to learn that subject. Imagine my disgust when I visited many bookstores to find a good book on OOP and I did not find one - that is correct, not a single, really good book. I found rows and rows of books on the subject of OOP. Some used C++ for example code; others used Delphi or some other language. All of them I rejected. Not because there were no Clarion examples, but because all but two of these authors did not even attempt to define OOP! Moreover, the two authors that attempted definitions came up lacking. Here is one such example:

“OOP deals with objects that have certain characteristics such as <buzzword>”,

Why do I think that is lacking? That definition can't work! What do I mean by that? You cannot apply this definition, no matter how bright you happen to be. Can you even use it in a sentence that makes sense? All right, enough of my pet peeve, I will not keep you waiting any longer.

**OOP** is an acronym meaning **Object Oriented Programming**.

You are sitting there thinking, “I knew that, so what?” Defining acronyms is not helpful if one does not know the words of the acronym. However, if we get the definitions of these words, perhaps an understanding of “what is it?” will become visible. For those who have seen these definitions before, it will not hurt to revisit them.

**Object** - [n] 1) Anything that can be apprehended intellectually. *What object do you think of when I say that? Do you have an object that starts with the letter E? What object best represents success?* 2) Anything that is visible. *What is that object in your mouth? Have you ever seen an unidentified flying object?*

**Orient** (ed) [vt] To point or direct at something or in a direction. *If you would orient your gaze to the south end of the runway, you see the shuttle is about to land*

**Program** (ing) [v] The act of writing computer code to perform a certain task. *The project is now in phase two of programming.*

Combine these three words and you have another definition. Object Oriented Programming means the “writing of code directed or aimed at objects”.

## Your first exercise (under a minute)

---

I don't mean to break your train of thought, but believe it or not, you've just studied an area that many people cannot explain. So for this exercise, simply define “OOP” in your own words. Refer to the above example if you need to. And finally, use “OOP” in a few sentences of your own creation, showing the above meaning (seriously). Do this until you feel comfortable with it. Some people need to do about 10 or more sentences before they get comfortable. Some require less.

Once you have done this, please proceed.

Now you may find yourself asking, “Where do I start?”



I think a more precise question would be, “How does Clarion do OOP?” Get that question answered and I think you have your starting point. To be more precise, since the subject is really “objects”, what is the meaning of that? Let me use a real, yet over-simplified example.

In a business application, you deal with data files. There are files of all shapes, sizes and types. If you were to design a file object, what would it do? One cannot get very far without opening a file, so it must do that. In addition, it must close a file, so add that to the list. You should be getting an idea of what a file object should do. There is more to files than opening and closing them, but I will leave it at that for now.

You should notice one thing I did not mention. Which file am I talking about? In a procedural world, you must name the file you wish to open and close. In the OOP world, you do not (at least at design time). This means that you only need to think about an object that will work the same on *any* file. This is what others mean when they tell you to think of abstract data. Abstract data does not have clear borders; it is a broad brush stroke, like the file object.

What you do is gather all the attributes and behaviors that would apply to a file - any file. At this point, the definition of the object looks incomplete; by itself, it really can't do anything useful. That is correct as you do not want the object doing everything or even knowing what the file name is. You supply the file name at runtime.

I will explain a little later how an object becomes useful at runtime. . First, you need to define what actions and attributes that apply to a given file. You place these into a container or bucket.

That brings me to the first OOP keyword.

## Encapsulation

---

Yow! That is a big word! This needs a definition quick!

**Encapsulate** [vt, vi] To enclose, as in a capsule. *When the vault door closed, they were encapsulated. Your lies encapsulate your reputation as untrustworthy. Encapsulate your thoughts on this issue.*

So what does this mean in the OOP sense? Why is this important? If you have some code and data that is in its own little world, you *encapsulate* that code. It

does not depend on anything outside itself. Alternatively, you could call this collection of code and data an object. What does this really buy you? It allows you to address this “grouping” as a single thing or entity.

The following sentence is a simple example of what I mean by that: “I’ll never trust him again as he is a Judas.” You probably know what I mean by the word “Judas”. I am referring to the Biblical story of the betrayal of Jesus by a disciple named Judas. Thus, the word “Judas” encapsulates the betrayal of Jesus and how this happened. If “Judas” were not encapsulated as a concept, you would need other, and probably many more words to describe why a person is not trustworthy, what did they do to earn that reputation, etc.

If you were to code a cat, you want the cat code distinct and different from a dog. You also do not want the dog bits mixing in with the cat bits. I am getting a little ahead of myself here, but I wanted you to get an easy image in your head without getting too technical at this point. You can see that if you wanted to code a cat, you want nothing to do with anything that is “non-cat”. Thus, you enclose the cat code in its own capsule.

Encapsulation is where the concept of a *class* comes in.

### ❑ The CLASS statement

In Clarion, you define an object with the CLASS statement. The CLASS statement is encapsulation. Hold on to your shorts, but this and one other statement are the only pure OOP statements in Clarion! That should help with any nervousness about a learning curve. You learned one of the OOP statements in Clarion and there is only one more coming! How good is that?

OK, enough chin wagging, let us see some code!

```
FileClass      CLASS
Name           CSTRING(256)
Path           CSTRING(256)
Open           PROCEDURE
Close          PROCEDURE
               END
```

The above is a simple class definition. Notice I said “definition”. This means it belongs in the data section of your program. Class definitions belong in the same section with variables, WINDOW, FILE, VIEW, REPORT and other data definitions. The data section in Clarion is anything after PROGRAM (or MEMBER or PROCEDURE) and before the CODE statement.

Think of a CLASS as an enclosed MAP statement, but with data definitions as well. The above CLASS deals with the opening and closing of a file. There is more to a good file class than that, but I wish to keep these concepts simple for the moment.

There is one thing that may not be obvious to you by looking at this definition. The name of the CLASS is `FileClass` not `CustomerFileClass`. You do not wish to repeat the same code for every different file. The old Clarion templates produce code in this manner. Why code the same thing repeatedly using a different label? I do not know about you, but I would get bored. There is more to say about code re-use, which I will discuss later. First, I need to introduce some new buzzwords into your vocabulary.

### ○ Property - defined

If you notice, there are two data types in the above CLASS structure. These are *properties* to use the OOP lingo. Properties describe the attributes of an object. How big is it? What color or size is it? This is like describing yourself to someone you never met.

### ○ Method - defined

There are two procedures in this definition as well. A *method* is the term for these procedures. That is what the CLASS does.

When you describe an object with code, you declare a CLASS structure. Then list the properties and methods in it. A CLASS therefore, encapsulates methods and properties. Taken together this is the beginning of an object. What do I mean by saying “beginning”?

Declaring a CLASS is not always declaring an object. A CLASS is a definition of an object, but doesn't exist as something that can be used. To use a class, you have to create an instance of it, called an object. This is why Clarion uses the keyword CLASS and not the keyword OBJECT. A CLASS definition does not always bring an object into existence. This brings me to the next keyword in the OOP vocabulary.

## Instantiation

---

**Instantiation** 1) The creation of an object. *Instantiation of the house results from a builder working according to the blueprints.* 2) An object ready for use. *You cannot use an object without instantiating first.* [From a Latin word meaning, “Being present”]

Here is the key to instantiation: You may have more than one instance of a `CLASS` declaration. This means you define the `CLASS` once and instantiate it multiple times. The advantage of this should start coming into focus now. Simply put, define once, then instantiate the objects that you need from that definition.

For example, I declare a `FileClass`. However, I instantiate a `CustomerFileClass` based on the `FileClass` definition. I will not and do not need to declare a clone of the `FileClass` to use it on the `Customer` file.

### ❑ How to instantiate a Class

There are three ways to instantiate any class (examples follow).

- 1) A class declared without the `TYPE` attribute declares both a type of the object and an instance of that object type. A class definition with the `TYPE` attribute does not instantiate a `CLASS`, it is a definition only.
- 2) A simple data declaration statement with the data type being the label of previously declared class structure, instantiates an object of the same type as the previous class.
- 3) Declaring a reference to the label of a previously declared class structure, then using `NEW` and `DISPOSE` instantiates and destroys an object respectively.

### ○ Global objects

A class declared in the global data section is instantiated for you automatically at the `CODE` statement that marks the beginning of your program. When you `RETURN` to the operating system, the object is destroyed for you. This means that visibility and lifetime of such objects are global. The code example below demonstrates the three ways of instantiation.

```

PROGRAM                                !Program global Data and Code

MyClass      CLASS      !Declare an object and
Property     LONG      ! a type of object
Method       PROCEDURE
              END

ClassA       MyClass    !Declare MyClass object
ClassB       &MyClass    !Declare MyClass reference

CODE
!MyClass and ClassA automagically instantiated
ClassB &= NEW(MyClass) !Explicitly Instantiate object
! some code here
DISPOSE(ClassB)        !Destroy object (required)
RETURN                 !MyClass and ClassA
                       ! automatically destroyed

```

### ○ Modular objects

A class declared in the modular data section is instantiated for you automatically at the CODE statement that marks the beginning of your program. When you RETURN to the operating system, the object is destroyed for you.

This means that the lifetime of module objects is global, but their visibility (or scope) is limited to the module.

## Programming Objects In Clarion

---

```
MEMBER('MyApplication') !Module Data

MyClass      CLASS      !Declare an object and
Property     LONG       ! a type of object
Method       PROCEDURE
END

ClassA       MyClass    !Declare MyClass object
ClassB       &MyClass    !Declare MyClass reference

CODE
!MyClass and ClassA automagically instantiated
ClassB &= NEW MyClass    !Explicitly Instantiate object
! some code here
DISPOSE(ClassB)          !Destroy object (required)
RETURN                   !MyClass and ClassA automagically
                        !destroyed
```

### ○ Local objects

Objects declared in a procedure's data section are instantiated for you automatically at the CODE statement that marks the beginning of the procedure's executable code, and are automatically destroyed for you when you RETURN from the PROCEDURE. This limits their lifetime and visibility to the PROCEDURE within which they are declared.

```
SomeProc     PROCEDURE    !Local Data and Code

MyClass      CLASS      !Declare an object and
Property     LONG       ! a type of object
Method       PROCEDURE
END

ClassA       MyClass    !Declare MyClass object
ClassB       &MyClass    !Declare MyClass reference

CODE
!MyClass and ClassA automagically instantiated
ClassB &= NEW(MyClass)    !Instantiate ClassB object
! execute some code
DISPOSE(ClassB)          !Destroy ClassB object (required)
RETURN                   !MyClass and ClassA automagically
                        ! destroyed
```

These scope and visibility issues should be straightforward. The point in bringing this up is that a CLASS is treated the same as simple variable declarations

concerning when they are available and what code may access them. This also means that if you try to use an object's methods or properties when they are not in scope, you will get errors.

### ❑ Which instantiation method to use?

I often hear the question, "Which instantiation method is the best one to use?" The honest answer is "It depends." When presented with a choice, the common conclusion is that once you pick one, you cannot change your mind, or one method works better than the others. This is simply not true. You may instantiate an object any way you deem best. There is more to say as to why, but there are other topics to cover first.

## Dot syntax

---

Dot syntax, also called "field qualification syntax," is simply a way of telling which object owns which property or method. It follows the same premise as using the prefix on labels in your code. One difference is you use a period instead of the colon.

In other words, you may have two or more class structures with methods called *Init*. The way to distinguish them is by prepending the object name, followed by a period and then the method name. The same is true for properties.

Think of it like an English sentence. You first read the noun, then a verb or some other modifier to complete the sentence. Granted, they are short sentences, but you can get the idea.

```
Fred          CLASS
Pay           DECIMAL(7.2)
Hours         LONG
Work          PROCEDURE(LONG),DECIMAL
              END
```

If you use the above class in your code, you might write some code like this:

```
CODE
FRED.Pay = FRED.Work(FRED.Hours)
```

You could surmise, without seeing all the source, that the code is working on Fred's pay.

The rule is simple: when referring to any method or property from any class, use dot syntax notation.

However, you may notice there is a problem with the code. It will only calculate Fred's pay. If your name is not Fred, you would be upset getting no pay.

Here is where the procedural style coders fall down in their approach to object oriented coding. They would write the same code for each of their employees. Eventually, you would have code that works. If it works, then why fix it? But assuming that the code always correctly calculates Fred's pay, would it not make sense to re-use the above code? However, the object's name is Fred, how do you do this? I suppose you could name it Employee instead of Fred, which makes sense. However, which employee is paid? Not much progress really. I will explain this after a short detour.

### ❑ Object naming in method code

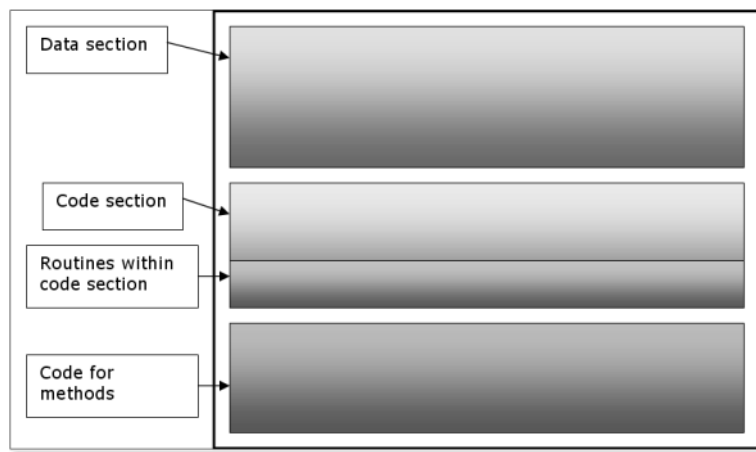
Whenever you declare a class with methods, you must code those methods. All method code must be placed last in your source file. The sequence within a method is the data declarations at the top, then the `CODE` statement. That terminates the data section. All code you write after the `CODE` statement is the main code of the method. This includes any `ROUTINEs` that follow the main code.

Method code always follows the last main code statement or routine, whichever is applicable. Just scroll down to the bottom of your source file, and that is where the code for methods belongs. It does not matter which order the methods appear in either, as long as they are at the bottom of your source file, after the main body of source.



Figure 2-1 illustrates where the different sections belong in your source file.

**Figure 2-1: Source file sections**



There is another way of coding methods using the `INCLUDE` statement, but that is covered later.

Therefore, the source where the `Fred` class is used would look like this:

```
! PROGRAM or MEMBER or PROCEDURE

Fred          CLASS          !Data section
Pay           DECIMAL (7.2)
Hours         LONG
Work          PROCEDURE (LONG),DECIMAL
END

CODE          !Main body of code
FRED.Pay = FRED.Work(FRED.Hours)
!End of main body of code

FRED.Work     PROCEDURE (LONG) !Method code
CODE
! Whatever code is needed to calculate pay
```

Granted, you need more than the above to have a working Clarion procedure that makes use of a `CLASS`. The concept is what is important at this stage. A demonstration of working code comes later.

As I explained earlier, each class method is declared using the class name, followed by a period, followed by the method name. In this case, the class is called `Fred`. In the main body of class code, you use that label. However, in the code for the method, there is no way to know which object is executing. This is because you may have many instances of this object in use. In addition, I am giving you the restriction to code this only once, so you cannot refer directly to the class itself in the class's own code. You will soon see this is not a restriction at all.

### ❑ SELF

Instead of using the label of the class in the method code (as opposed to the main body of code), you use the label `SELF`. The definition of `SELF` is “whatever the current object is.” You can see this in the following code listing:

```
PROGRAM

Employee      CLASS, TYPE !Declare object TYPE
Pay           DECIMAL(7,2)
Hours         DECIMAL(3,1)
CalcPay       PROCEDURE
Work          PROCEDURE (*DECIMAL),DECIMAL
              END
Fred          Employee    !Instantiate objects of Employee type
Barney        Employee

CODE
Fred.Pay = Fred.Work(Fred.Hours) !Code outside “Fred” object

Employee.CalcPay PROCEDURE          !Method Definition
CODE
SELF.Pay = SELF.Work(SELF.Hours) !Code inside method Listing
```

Using the above code, `SELF` is really `Fred` as `Fred` is the object name in the main body of code. Another instance of the class exists, with the label `Barney`; in that instance, `SELF` is `Barney`. The key concept to get from this is that for both `Fred` and `Barney`, you execute the same code! You are not executing a copy of the class; you are not executing a clone of the class. You are executing separate *instances* of the `Employee.CalcPay` method. You write the code once, but use it many times for different objects.

This is something you cannot do with routines. Routines act on one piece of data and that is it. If you need to do the same thing on another entity or in another

procedure, and you find yourself coding yet another routine, it's time to think about a class instead.

Let me give you a conceptual example of what I mean.

### ❑ The Microwave object

Before I watch the “big game” on TV, I like to make nachos. I have several good recipes, but I want my nachos fast. Therefore, I microwave them. The microwave does a great job of melting the cheese, warming the chili and beans. I know exactly the time and power setting I need to make those nachos perfect. I only have one microwave oven. Yet I also use it to reheat my coffee, melt or soften butter, fry bacon and do other tasks. Again, I only have one microwave oven.

Why do I keep saying that? If I were to code a microwave oven like some others write code (including the Clarion templates), I would have one oven for cooking my nachos, another for heating coffee, another to soften butter, and yet another to fry my bacon.

Using this approach, I code my microwaves with no controls on them, as the power levels and time to cook are already set. It means that if I need to do something different, no matter how small those differences may be, I need to code a brand new microwave oven!

Sticking with the Clarion analogy, I will next ask around to see if anyone has a microwave oven template that I can buy. I do not really want to code all that microwave oven behavior repeatedly. I will let a template code it repeatedly, like the legacy (Clarion) templates.

Just how much of a microwave oven is the same as any other microwave oven? Why do I need to code the box, the electron gun, and hard code the power level and hard code the time needed to cook? Are you seeing the point? Being a lazy programmer, I will define a microwave class and then re-use what I need. This means I will not define yet another microwave oven, but re-use it! Just like in real life.

## Programming Objects In Clarion

---

```
MicroWave      CLASS,TYPE
Power          LONG
Time          LONG
Cook          PROCEDURE(LONG xPower, LONG xTime),LONG,PROC
              END

Nachos         MicroWave
Reheat        MicroWave

Power:High     EQUATE(10)
Power:Low      EQUATE(1)

CODE          !Main body of code
IF Nachos.Cook(Power:High,45)
    SoundDing()
END
IF Reheat.Cook(Power:Low,60)
    SoundDing()
END

MicroWave.Cook PROCEDURE(LONG xPower, LONG xTime)
Counter      LONG,AUTO

CODE
SELF.Power = xPower
SELF.Time = xTime
LOOP Counter = 1 TO SELF.Time
    !Cooking code here.
END
RETURN True
```

You can see that both the Nachos and Reheat objects are Microwave data types and these objects are instantiated. In the main body of code, I used both objects, but if you notice, there is only one procedure labeled Cook. This is because I used the same microwave to cook my nachos and to reheat something.

Are you starting to see the picture of what is going on here? So far, I have covered encapsulation and instantiation. These are powerful, yet simple concepts. There is another benefit to these concepts before I move on.

If you notice, I have the Cook method coded in one place. Yet if my nachos burn or my coffee is cold, I have a bug in my code. Now ask yourself, how many places do you have to look to find the bug?

Coding OOP style not only allows the writing of less code, but it also reduces your maintenance. When you do need to fix something, you only have to look in one place - you don't have to hunt around for additional copies of the same code.

## Constructors and Destructors

---

These are two more keywords in the OOP lexicon, and they describe special methods in a `CLASS`. The *constructor* is a method automatically called when an object is instantiated. It does not matter how the object is instantiated. The *destructor* is a method automatically called when you destroy an object, again regardless of how the object is destroyed.

Their use is very simple. Declare a method named `Construct` with no parameters in your class. Do the same with the destructor except name it `Destruct`. You may use one, the other, neither, or both. Their use is totally up to you.

### ❑ Why use them?

If you wish code to automatically execute when an object is instantiated, then make a constructor. Such common uses are initializing variables, making reference assignments and other setup chores. If you know your design calls for the same things to happen every time at startup, this is a good use for a constructor. Your other choice is to have your other methods do these chores, but then you have to remember to call these methods yourself at the appropriate time.

If you have a constructor, the chances of coding a destructor increase. A typical use would be to clean up anything the constructor did, dereferencing, etc.

### ❑ Why not use automatic constructors/destructors?

There are limitations to constructors and destructors. They do not take parameters. They do not return any values either. If you need to pass parameters to an object or check return codes (such as when an object instantiates correctly), you need to use other means.

Returning to the `FileClass` code, one of the things required by this class is a `QUEUE`. It's a limitation of the Clarion language that `QUEUEs` cannot be declared inside a class - you have to declare a reference to a queue, and then create the queue at startup (if you're not familiar with reference variables, don't let that

worry you too much at this point). Here's what that class would look like with automatic constructor and destructor methods:

```
FileClass      CLASS,TYPE
Name           CSTRING(255)
Path           CSTRING(255)
WorkQ          &TypeQue      !TypeQue definition not shown
Open           PROCEDURE
Close          PROCEDURE
Construct      PROCEDURE
Destruct       PROCEDURE
               END
!Main body of code here

FileClass.Construct  PROCEDURE
CODE
  SELF.WorkQ &= NEW(TypeQue)
  ? ASSERT(SELF.WorkQ &= NULL, 'Cannot reference WorkQ')

FileClass.Destruct   PROCEDURE
CODE
  IF ~SELF.WorkQ &= NULL
    FREE(SELF.WorkQ)
    DISPOSE(SELF.WorkQ)
  END
```

Looking over the above code listing, it shows a simple declaration for the constructor and destructor in the class declaration. Below the comment indicating where normally you see the main code, are the two methods. In this case, you see a reference assignment of a newly created QUEUE to a QUEUE reference variable. The ASSERT is debug code that will show the assert message if the condition is false (the assertion being that the condition is always true). The point here is that this code executes every time this class is instantiated.

The destructor “undoes” what the constructor built. If the reference is not NULL, it FREES the QUEUE, then DISPOSES the QUEUE. This is a good reason to use constructors and destructors. A destructor can “clean up” what a constructor did.

I will revisit constructors and destructors a little later. For now, I need to add a bit more to encapsulation.

## Extreme Encapsulation

---

You recall that a class encapsulates data contained in it. This could imply that the class hides data from the outside world. In one sense, this is a good thing, as you certainly do not want anything from the outside to step on data and corrupt it. Encapsulation can be a guarantee that the data is in a reliable state.

Clarion supports this with the `PRIVATE` attribute. When used as an attribute in a property or method declaration in a class, only other members of that class may access that property or method. In other words, `PRIVATE` hides data and/or methods from everything outside the class.

```
MyClass      CLASS
MyProperty   LONG,PRIVATE      !private Property
Method       PROCEDURE
MyMethod     PROCEDURE,PRIVATE !private method
              END

CODE
MyClass.MyMethod      !Invalid here
MyClass.Method        !Valid here
MyClass.MyProperty = 10 !Invalid here

MyClass.Method PROCEDURE
CODE
SELF.MyMethod      !Valid here
SELF.MyProperty = 10 !Valid here
```

### ❑ When to use the Private Attribute

So when would you wish to use `PRIVATE`? In the early days of ABC, there was heavy use of the `PRIVATE` attribute. The reason was that ABC was still in a state of change, and if the code or data was `PRIVATE` it could be changed without breaking anything. That is still a valid use today, even with non-ABC objects.

Another time to use `PRIVATE` is whenever you have something that *only* that class is responsible for. Again, I'll have more to say about this later.

## Inheritance and Derivation

---

You know that inheritance means you receive a vast sum of money after a rich relative died. OK, that does not seem to apply to Clarion, let alone OOP.

**Inheritance** (n) - something received from a predecessor, as a trait, or characteristic. *He inherited his musical abilities from his father.*

Inheritance is similar to another common OOP term: *derive*.

**Derive** (v): To convey from one (treated as a source) to another, as by transmission, descent, etc.; to transmit, impart, communicate, pass on, hand on. *You derived your sense of humor from Uncle Harry. He derived his knowledge from hard study.*

This is where OOP can get a bit tricky, especially to those trying to grasp it for the first time. To derive a new class, you simply name the parent class as the parameter to the new class statement. The new class declaration inherits everything from the parent class. The difference here is that, in OOP no one has to die for the child to inherit.

This is a good opportunity to introduce some more OOP terms: *Base* class and *derived* class.

A *base class* is a class that has no parameter to its class statement, meaning it does not inherit anything. However, a *derived class* always has a parameter naming its parent class. Notice that I did not say that the parameter to the derived class statement names a base class – it does not always do this. The parameter to a derived class statement names its parent class, which could be either a base class or another derived class. This means that you can have multiple generations of inheritance.

Let me illustrate this with the `FileClass`.



```
FileClass      CLASS,TYPE
Name           CSTRING(255)
Path           CSTRING(255)
WorkQ          &TypeQue !TypeQue definition is not shown
Open           PROCEDURE
Close          PROCEDURE
Construct      PROCEDURE
Destruct       PROCEDURE
               END
CustomerClass  CLASS(FileClass)    !Derived class
!Contents of the class structure here.
               END
CODE
!Some code to point to the customer file
CustomerClass.Open
```

### ❑ Extending a class

In the above code listing, `CustomerClass` inherits everything from `FileClass`. When you derive a new class, the purpose of doing so is usually to extend the class. In other words, you want all of the functionality of the parent, but you want to embellish existing code or add new code. With inheritance you can do all that without changing the original definition, and without rewriting the same code you already have that works.

That is a very good thing to do and makes maintenance easier. Let us say you have a `VendorFile` class derived from the `FileClass` too. If opening the `Vendor` file works, but opening the `Customer` file goes “bang”, you can safely assume the base `FileClass` is fine. That must mean there is a bug in the `CustomerClass`. Now you know where to look. So, that is a huge benefit.

This should indicate to you how to design a set of classes. Make a base class that has everything in common about the type of object you need. For example, code a file class that is as general as possible. This class has everything that you may need to handle only one file. By that, you do not yet know which file, let alone the driver type.

As I touched upon before, you need to open a file, and close it. This is a common thing to do for any file. I would strongly suggest that you keep such designs very simple. By that, I mean so simple it hurts. Have you noticed I have said nothing about checking for errors or handling relations? Those are general in nature as well.

An `ErrorClass` has broader applications than looking for file errors. Therefore, that would be another class. A `RelationClass` could be complex, but that could have a `FileClass` as its parent. In this case, the `RelationClass` derives everything it knows about files from its parent.

Derivation and inheritance are powerful, yet simple concepts in OOP. Those seeing classes derived from parents for the first time can get confused. This is because the code does not appear to be complete.

### ❑ **Overriding a class**

Another thing you can do with inheritance is override parent methods. This is very simple to do. All you need to do is declare a method with the same label and prototype as the parent method. This means that you want to toss out the parent's method and code everything in the derived class. That gives a first impression that it is double work.

Why would you do this? The parent method may not be applicable for what you need to do. In this case, simply write the replacement code. You can combine both methods if you wish, and I will explain how to do that shortly.

### ❑ **Multiple Inheritance**

Clarion does not support multiple inheritance. You may *only* derive from a single parent. Other languages such as C++ do support this concept, but this does not mean Clarion is crippled. Multiple inheritance introduces ambiguity into the code, and this is not only hard on the compiler, but the programmer.

If Clarion supported this, you would have to code disambiguate methods to explain to the compiler which objects derive from where. That is not a good use of programmer time.

### ❑ **Composition**

Where multiple inheritance would be useful, Clarion uses a simple way of giving you the advantages of multiple inheritance, but without the ambiguity. *Composition* is the term for this technique.

**Composition** - In a child class, the data type of a property is a reference to another class.

Composition gives you all the benefits of inheritance too. The main difference is that you do not have to write any code to disambiguate what you mean. One line of code is all you ever need. Let me illustrate this with an example:

```

ApplePie      CLASS,TYPE      !Declare Base Class
Apples        STRING(20)
Crust         STRING(20)
Bake          PROCEDURE
END

IceCream      CLASS,TYPE      !Declare Base Class
Flavor        STRING(20)
Scoop         PROCEDURE
END

AlaMode       CLASS(ApplePie) !Composition: Derive from a CLASS
OnTheSide     &IceCream       ! and contain a reference to
Serve         PROCEDURE      ! an object of another CLASS
END

```

You can see from the above listing the `AlaMode` class derives from the `ApplePie` class. You can tell this as `ApplePie` is a parameter in the class statement, thus it is a child class. However, look at the `OnTheSide` property. It is a reference to the `IceCream` class.

Both inherit everything from their parents. The reference to `IceCream` is the same thing as declaring the `IceCream` class inside of the `AlaMode` class. It is also the functional equivalent of listing more than one parent as a parameter of the class statement. However, the reference is the legal way to do this, and keeps your code clean.

## Moderate Encapsulation

---

Recall that a method or property with a `PRIVATE` attribute is for the exclusive use inside the class. Absent of that attribute, methods and properties are “public”, meaning you may access them from outside the class. Sometimes, you do not want either.

Use the `PROTECTED` attribute when you need to restrict methods and properties to the class they belong *and* any derived classes that may need them.

You may wish to use this attribute when data needs some protection from outside the class, but must also be available to child classes.

```
MyClass      CLASS           !Declare Base Class
Property     LONG
MyProperty   LONG,PROTECTED !Semi-Private
Method       PROCEDURE
END

ClassA       CLASS(MyClass)  !Declare Derived Class
AProperty    LONG
AMethod      PROCEDURE
END

CODE
ClassA.MyProperty = 10      !Illegal statement

ClassA.AMethod PROCEDURE
CODE
SELF.MyProperty = 1         !OK within method of class
```

## Overriding

---

This is another keyword of OOP. In the Clarion context, here is a workable definition:

**Override** - To create a method in a derived class that has the same label and prototype (any parameters and/or return type) as its parent.

That is simple enough, but why would you want to do this? The answer to that question goes back to derivation. If you recall, a derived class extends the behavior of a parent class. However, you do not change the functionality of the parent. The same is true for overridden methods. Here is a simple example.

```
ApplePie     CLASS,TYPE      !Declare Base Class
Apple        STRING(20)
Crust        STRING(20)
Bake         PROCEDURE
END
```

Dutch CrumbleTop	CLASS(ApplePie) STRING(20) END	!Declare Derived Class
American TopCrust	CLASS(ApplePie) STRING(20) END	!Declare Derived Class
Grandmas CaramelTop Bake	CLASS(ApplePie) STRING(20) PROCEDURE END	!Declare Derived Class  !Overridden method

You can see the `Bake` method in the `ApplePie` class. That makes sense, as you need to bake apple pies. Look at the `Grandmas` class. It also has a `Bake` method. Compare the two methods, and you will see they both have the same label and the same prototype.

Thus, the derived `Grandmas` class overrides the `ApplePie` method. This means that for whatever reason, the parent method may not be applicable to the child class. Thus, you write the appropriate code.

An `ApplePie.Bake` method may be fine for apple pies in general. In addition, that method is fine for `Dutch` and `American` pies. However, `Grandmas` may be a special recipe where a special bake method is required.

There is more to overriding methods than this, but I am leaving it here for now.

## More on Constructors and Destructors

---

I mentioned previously that there is more to tell you about automatic constructors and destructors. Well, now is a good time to tell you about how inheritance affects constructors and destructors.

What happens when a parent class has a `Construct` method and the derived class needs one? I already covered overriding methods in derived classes, so you might guess that the derived class `Construct` method would simply override the parent class method. That guess would be wrong, of course.

So why is it wrong? Because overriding the parent class constructor might mean that code that is required to initialize inherited properties might not execute. If the

inherited parent class properties failed to initialize, you might end up with unexpected behavior in your derived class. Therefore, *automatic* overrides do not happen with constructors. Instead, by default, they all execute in the order of object instantiation.

First, the parent class constructor executes, then the derived class constructor executes. They execute in the order of their derivation. Base class constructors always execute first followed by any derived class constructors, in derivation order. The constructor for the most derived class always executes last.

The same reasoning is true for destructors, except reverse the order of their execution. When you destroy the most derived class, its destructor automatically executes first, and on up the chain of derivation until the base class destructor executes last. In other words, LIFO: Last In First Out.

The way constructors and destructors work by default in the Clarion language is the same way they work in most other OOP languages. However, Clarion does give you some flexibility that some other OOP languages do not.

Automatic overriding of Constructors and Destructors does not happen in Clarion. The key word here is “automatic.” You can override them in Clarion if you want to –something you cannot do in some other OOP languages. If you add the `REPLACE` attribute to the prototype of your `Construct` or `Destruct` method, you are telling the compiler that you do want to override the method.

So, what does that buy you? Suppose you need to initialize a variable in your derived class before the parent class constructor executes. The only way to do that is to override the constructor. You simply add the `REPLACE` attribute on the prototype of the derived class `Construct` method, then write your code.

The concern about automatically overriding constructors and destructors is still valid, and you should consider it carefully when you manually override them. Nevertheless, the designers have thought of that. First, another small detour and then I will return.

## PARENT

---

You recall that the way to reference the current object within the code of a method is to use `SELF` instead of the object name. There is another tool in Clarion’s OOP syntax that allows you to call a method from a parent class even when you have overridden that method.

Prepending PARENT to the method name explicitly calls the parent class's method, even when overridden. This technique holds true not only for constructors and destructors, but also for any overridden methods.

```

MyClass      CLASS,TYPE      !Declare Base Class
Property     LONG
Method       PROCEDURE
Construct    PROCEDURE      !Method code not shown
END

ClassA       CLASS(MyClass)   !Declare Derived Class
Aproperty    LONG
Construct    PROCEDURE,REPLACE
END

CODE                                     !ClassA Instantiation here
<Main code here>

ClassA.Construct  PROCEDURE
CODE
  SELF.Aproperty = GLO:Flag      !Initialize then call
  PARENT.Construct              ! parent constructor

```

In the above listing, the assumption here is that MyClass writes to GLO:Flag. But when that happens, the derived ClassA class needs to grab the value of GLO:Flag before that happens. In addition, your design says you cannot change the code in MyClass. Without the REPLACE attribute, you have painted yourself in a corner.

When you need explicit control of the execution order of your constructors or destructors, you simply put the REPLACE attribute on the prototype. Then, in your constructor or destructor method's code, directly call Parent.Construct or Parent.Destruct at the exact point within your logic that is most appropriate to what you need to do. That could be before, after, or in the middle of your derived class's code – wherever you need it to be.

## Polymorphism

---

*Polymorphism* is the last major OOP buzzword you must learn. For those who are wondering, it does not mean you have a parrot with a drug problem.

**Polymorphism** - derives from two Greek words. It is the prefix *poly-* meaning “many” and the suffix *-morphos* meaning “form”. Combine them together and you have a word meaning “many forms”. Add the suffix, “-ism”, and you have a new word. For you grammar types, it is also a noun, denoting a state, condition, or characteristic.

Polymorphism is a key point of object orient programming. However, it is not exclusive to OOP. Clarion supports various forms of polymorphism. For example, you may recall the *\*?* and *?* parameter types. These mean an unknown data type for parameter passing. Even Clarion for DOS supported these.

When Clarion for Windows first appeared, another form of polymorphism arrived. It is *function overloading*. Function overloading happens when you have two procedures with the same label but different prototypes. Examples of this form of polymorphism are numerous in the Clarion language, such as the `OPEN` and `CLOSE` statements. You can `OPEN` or `CLOSE` windows, files, reports, etc. and those statements know how to respond to each type of parameter, because there are actually multiple functions declared with the same names, one for each parameter type.

### ❑ Virtual methods

Virtual methods are an important part of polymorphism. What does “virtual” mean?

**Virtual-** adj. 1) Being such in power, force, or effect, though not actually or expressly such. *They are reduced to virtual independence on charity.* 2) Apparent, but not actually in existence. (From optics). *You can see your virtual image in a mirror.*

The virtual method concept drives procedural programmers crazy. When you say “virtual method” you are using the word “virtual” as an adjective. You are describing a method that seems to be in force, or is apparent. Let me lift the fog on this a little bit by giving you another definition:

**Virtual method** - A method whose prototype is present in a parent class and a derived class, which both have the `VIRTUAL` attribute.



That is all you need to do to *create* a virtual method. You may be thinking that I did not clear any fog from this concept. Nevertheless, stay with me here as the above definition is very important. Again, this requirement means that any virtual method declared in the parent class and the derived class and both must have the VIRTUAL attribute. These methods must have the same label *and* prototype. So this not the same thing as function overloading, where the label is the same and the prototype is different.

All right, what does this give you? What is so special about virtual methods? Let me put it in context for you. Recall that a derived class has methods that can “call up” to the parent class. You do this by using PARENT as the object name in the method code. This aligns well with the procedural concept. You also know that you declare any derived class after its parent. If you do not, you get compiler errors, as compilers do not take kindly to forward references.

However, a virtual method allows a parent class to call the overridden child method. You could think of it as a forward reference that actually works.

So what good is that? What benefit could you possibly get from something like this? For one, this means that you can change a parent class behavior without touching the code in it. It also means the parent knows an overridden method is in a child class and will use that method *instead* of its own!

The best way to illustrate this is with some simple code. I am returning to the apple pie example, but with some changes.

```
ApplePie      CLASS,TYPE
PreparePie    PROCEDURE
CreateCrust   PROCEDURE,VIRTUAL      !Virtual Methods
MakeFilling   PROCEDURE,VIRTUAL
END

Dutch         CLASS(ApplePie)
CreateCrust   PROCEDURE,VIRTUAL      !Virtual Methods
MakeFilling   PROCEDURE,VIRTUAL
END
```

```
CODE
Dutch.PreparePie          !Call the Dutch object's Virtuals

ApplePie.PreparePie  PROCEDURE
CODE
SELF.CreateCrust
SELF.MakeFilling
```

First, you may make some assumptions about this code. Even a non-cook knows there are some steps involved in preparing any pie. This code is in the `ApplePie.PreparePie` method, as the assumption is that the series of steps (create crust, make filling) will not change.

The `Dutch` class derives from its parent, `ApplePie`. You can see this in the class statement for it. It also has `CreateCrust` and `MakeFilling` methods like its parent. Notice that in both declarations, the `VIRTUAL` attribute is present.

In the main body of code, there is one line, `Dutch.PreparePie`. You know this is legal as `Dutch` inherits the `PreparePie` method from its parent. Thus, there is only one version of the code for `PreparePie` and this method belongs to `ApplePie`. In addition, due to inheritance you know you do not have to code a `Dutch.PreparePie` method.

OK, that is all fine for the easy stuff, like derivation and inheritance and you know why the `Dutch.PreparePie` call is legal.

So where is the code for `PreparePie`? It is in the `ApplePie` class – *only*. You can see the code. It creates a crust and makes the filling.

When the call to `Dutch.PreparePie` executes, it is really using the code from `ApplePie.PreparePie`. Thus, the two lines of code execute *as if* this is the code:

```
ApplePie.PreparePie  PROCEDURE
CODE
Dutch.CreateCrust
Dutch.MakeFilling
```

You should now see that the parent is calling down or forward to its child class and executing its method as if it were its own. You could also surmise that if there were a `Grandmas` and `American apple pie` classes derived from `ApplePie`, they too would use the inherited `PreparePie` method (assuming the same prototype and virtual attributes).

This also means that you do not have to change the parent class because the virtual override means the parent executes the child method instead – all without extra coding.

That is the real magic to coding with objects.

If you wish, you can code a virtual method to use the parent method anyway. You would want to do this when the parent method has useful code. You do this simply by using the `PARENT` label. ABC based applications use this technique in all of their virtual methods.

## ○ DERIVED

The `DERIVED` attribute also available for virtual methods. In fact `DERIVED` means “virtual”. To be more precise, it means “virtual protection”. This attribute is for child methods only. All the rules of virtual methods apply when you use `DERIVED`.

What the `DERIVED` attribute offers is prototype protection. It enforces the definition of virtual methods, especially the prototype of overridden methods.

Let us say you have a parent virtual method that takes a `SHORT` as a parameter. As your project matures, many derived methods must use a `SHORT` as a parameter. However, you discover the parameter should be a `LONG` instead, and so you make the change *in the parent class*.

While the change may be technically correct, what you really accomplished was converting a virtual override to a function overload. It is possible to get a clean compile, depending on the child method's code. However, the derived methods no longer have a matching method in the top-level class, and so they don't get called instead of the parent method which now has the changed prototype. When you test the change, you find the application does not function as it did before, or perhaps worse.

Of course, you discover the real problem and change the derived methods so the virtual methods are back in place. But did you get them all? How can you be sure?

The rule of thumb is that virtual methods in a base class have the `VIRTUAL` attribute on all virtual methods. Any classes derived from it with virtual methods use the `DERIVED` attribute. Thus, when you do make a prototype change, the compiler knows you want virtual overrides, not function overloading. It will signal this by giving you a compiler error when it no longer finds a parent method for a `DERIVED` method. Use the error editor to fix the prototypes in all child classes used. Then re-compile your project.

## Late Binding

---

Early or static binding is the stuff taught in your programming theory class, where all procedure calls are resolved at compile time. However, since calls to virtual methods cannot be resolved at compile time, the compiler must build a Virtual Method Table or VMT at compile time and set up the method call for late or dynamic binding. At run time, when the call to the virtual method executes, late binding means that the actual method to call is determined by a lookup into the VMT. You already know a lot about lookups, since it is a standard database application technique.

Now, in many other OOP languages this late binding for virtual methods can cause a real performance hit. However, in Clarion the entire late binding process takes only one extra reference at runtime than early binding, so there's no performance penalty in Clarion.

Actually, virtual methods in Clarion are so efficient that when you look at the code generated by the ABC Templates, you will find that almost all the code generated is now in virtual methods.

That brings me to the next issue, which is the special scoping rules that apply to classes declared inside Clarion procedures.

## Local Derived Methods

---

In an ABC application, most of the work in any given procedure is usually done by an instance of one or more ABC classes. Embedded code is actually generated as virtual methods belonging to derived classes, and these methods then add new behavior to those classes. When ABC first appeared on the scene in Clarion 4 beta releases, testers quickly discovered a major problem: they could not use procedure local variables in embed points.

Local Derived Methods are in a derived class structure within a PROCEDURE. The Local Derived Method definitions (meaning the executable code) must immediately follow the end of the procedure in which they are declared. I mentioned this earlier.

The ABC designers suggested using references to local variables, but this was a cumbersome approach. The logical solution was to allow classes declared inside

procedures, to see all of the procedure's data. And that is how it now works. Local derived methods inherit the scope of the procedure within which they are declared. That means that all the procedure's local data variables and routines are visible and available inside the local derived methods. It also enables you to call the procedure's routines from within the method, just as if your code were still within the procedure itself.

This new implementation of scoping is what allowed the designers to take the OOP concept to the hilt in the ABC Templates and ABC Library. All you need to do is run the App Wizard on any dictionary, then look at the generated code to see that most generated procedures now contain very little code – everything is in local derived methods. And you are still be able to write your embed code as if it were inside the procedure itself. So in this aspect, you are not changing the way you use embeds.

## Interfaces

---

The subject of interfaces is relatively new to Clarion. It is a concept that many good OOP style coders have a little trouble coming to grips with. First, a working definition:

**Interface** - a collection of methods implemented by a class.

That is accurate enough to start with. However, it does not really convey the concept of what an `INTERFACE` is or does. I have heard another definition that goes something like “an interface is a contract with a class.” Technically accurate, but still does not tell you what it is. There are good examples in ABC with the `WindowClass`, but the concept of a window is too vague. How many different windows are there anyway? What does a window do? What makes this particular window so special? Do you have a special window in one of your apps?

The concept of an interface can be difficult to visualize, so let me paint a picture. Think of a printer. Any printer will do. We have all seen printers and they do one thing – print stuff. If you worked for a printer company, what would a printer class look like? Here's an example:

```
Printer      CLASS,TYPE
DocName      CSTRING(256)
PrintDoc     PROCEDURE
END
```

Printers today are more versatile than what the above code represents. They could print in color, number of copies, duplex, etc. If you want, you could imagine many more properties and methods, but I wish to keep this simple.

The `Printer` class prints a document. Say your marketing department has the results of a survey from your customers, and they want more features. So the design department manufactured a new printer with many new abilities. It is a multifunction printer as it can operate as a fax too. Not only that, but also this new device has scanner and copier behaviors.

Here is your dilemma: Do you recode the stable and reliable `Printer` class, thus risking introducing bugs? Or do you keep the working code? Of course, some good copy and paste should preserve the working code if you re-code it. However, what happens about future enhancements? What do you do when the design department comes up with yet another new feature? Do you really want to go through this again?

### ❑ Declaring the INTERFACE

INTERFACE to the rescue! An INTERFACE looks similar to the CLASS, but is actually just a definition of what methods a class must have. Here is what the `IFax` interface might look like:

```
IFax          INTERFACE
TransmitDoc    PROCEDURE
END
```

I prepend my interface labels with the capital letter “I” so I can tell them from classes. That is not a requirement of Clarion. I use this naming convention in my code.

Before I get carried away, a few notes about coding interfaces:

- No properties are allowed, as the INTERFACE defines a behavior.
- All methods declared in an INTERFACE are implicitly VIRTUAL. You may add the VIRTUAL attribute if you wish.
- All INTERFACES are implicitly TYPED, however you may add the TYPE attribute if you wish.

- INTERFACES are not instantiated. By themselves, they do nothing. A class must implement them.

You declare an INTERFACE before any CLASS that IMPLEMENTS them. You may even wish to INCLUDE them in your source. If so, the recommendation is to use INT as the file extension, although this is not required, only customary.

Now that the IFax INTERFACE is ready, the Printer class needs to know about it. The code now looks like this:

```
IFax          INTERFACE
TransmitDoc    PROCEDURE
GetDoc         PROCEDURE
END

Printer       CLASS, IMPLEMENTS(IFax), TYPE
DocName       CSTRING(256)
PrintDoc      PROCEDURE
END
```

What does this do anyway? It doesn't add the code for any methods to the class, but it adds the definitions for those methods to the class. You are adding new behaviors while not affecting the existing code. To put this in other words, the printer knows how to be a fax, but it did not forget how to be a printer. You could also do something like this if you really do not wish to change the Printer class:

```
IFax          INTERFACE
TransmitDoc    PROCEDURE
GetDoc         PROCEDURE
END

Printer       CLASS,TYPE
DocName       CSTRING(256)
PrintDoc      PROCEDURE
END

ExtendPrinter CLASS(Printer),IMPLEMENTS(IFax),TYPE
PrintDoc      PROCEDURE
END
```

## Writing Interface Methods

---

When you IMPLEMENT an INTERFACE, you do need to write the code for the INTERFACE methods. If you do not, the compiler will remind you. You write the code in the same area as you would any other method, using the same mechanics. However, the compiler needs to know which method belongs to a class and which method belongs to the INTERFACE.

Using the above code, you could see a method for an INTERFACE written something like this:

```
ExtendPrinter.IFax.TransmitDoc    PROCEDURE
CODE
!Code here.
```

Notice that the class label comes first, then the INTERFACE label followed by the method. Each separated by a period (see dot syntax, discussed earlier).

So what's the benefit of this, if you still have to write the method code? Simply that if a class implements a specific interface, you *know* that there are certain methods that it will have. And that lets you treat the class as if it were an instance of the interface.

### ❑ The Dark Side of Interfaces

You may see that method names of interfaces can be wordy. That is a small price to pay to define a behavior. However, there is another aspect to consider. What



happens if an interface has many methods? How do you show which ones you should implement?

The simple answer is you implement all of them. David Bayliss, the designer of the ABC classes and Clarion compiler architect, explains that the problem is that an interface should be an explicit (not implicit) contract between the interface and the class that implements it. In other words, you know what the class should do. In addition, if another programmer implements your interface in his class, then he knows what to expect.

An interface is a common boundary between two objects. If one can choose behaviors, then the interface is implicit, or ambiguous. In Clarion the interface is explicit, and unambiguous.

### ❑ The Good Side of Interfaces

When you have an interface, then any later CLASS can implement it. This is yet another way of changing a behavior of a CLASS. In the example I use here, a derived printer class implemented a fax interface. This changed the behavior of the printer yet I changed nothing in the class code itself.

If I wanted to, I could add yet another INTERFACE to a class. In my printer class, I could add IScanner and ICopier interfaces to the printer class. This means that a CLASS may IMPLEMENT more than one INTERFACE.

```
Extendprinter  CLASS(Printer),IMPLEMENTS (IFax),|
               IMPLEMENTS(IScanner),|
               IMPLEMENTS(ICopier),TYPE
PrintDoc       PROCEDURE
               END
```

You may be thinking this is a little like multiple inheritance, and in fact interfaces are how single-inheritance languages like Clarion get multiple-inheritance benefits.

If you expect to implement many interfaces in a class, then think about using derivation in your interface declarations.

```
IFax           INTERFACE(ICopier)
TransmitDoc    PROCEDURE
GetDoc         PROCEDURE
               END
```

I've defined and show the rules of what an interface is and does and how to use them. In the next chapter I will demonstrate interfaces in action.

## Summary

---

You now know the three major OOP buzzwords: Encapsulation, Inheritance, and Polymorphism, and they are implemented in the Clarion language. You have also heard the other standard OOP terms: Properties, Methods, Objects, Instantiation, Base Classes, Derived Classes, Interfaces, SELF, PARENT, Constructors and Destructors, Virtual Methods, and Late Binding.

The information in this chapter lays the foundation for what comes in the next chapters. You will discover that this information applies to ABC, and to what happens when you use embed points.

# Chapter 3

---

## APPLYING OOP IN CLARION

The last chapter covered the basics of Object Oriented Programming. Its purpose was to get the theory of OOP under your belt. This chapter covers how to apply that data. I plan to do this with actual working code that covers many of the concepts covered. Since there are many possible ways to do this, you may see a concept applied twice, but differently.

This means repeated concepts, but the difference is that working code reinforces these concepts.

### Where to Start?

---

One of the problems with using objects in your applications is trying to find a good starting point. Before you write one line of code, think about what you need and want.

Think about the `FileClass` discussed previously, which deals with things that are “file- like.” Again, what do all files have in common? This common functionality is what you usually put in a base class. Always start so simple is it painfully obvious when you are missing functionality. Do not worry about missing things at first. In the case of the `FileClass`, if you can design and code a class that opens and closes a file, you are off to a good start. If you can make this class open and close any file, no matter what, then you are further along and “bang on” the right track.

The point is that an object does not know, nor care about, the specifics of what it is dealing with. At least it does not during development time. This means that you avoid code that applies only to a specific situation, or a specific file, and so on. If you need a few hard coded values bit here and there to get you going, that is fine. Just see if you can retain the function without hard coded values later.

## What about existing applications?

---

The question I think gets asked the most is when one should convert existing applications to OOP. I have seen all types of applications in various stages of development. There is no precise answer as there are too many variables. Where applications are close to release, I can state this is the wrong time to think about converting existing code to objects. In this case, hold off until the next major version of your application.

It is never too late to use objects for code not yet written. This means that applications that started with the Clarion templates and then are maintained by hand (a common practice), can benefit from objects. Even the Clarion 6 legacy templates now get most of their new functionality via ABC objects that have been grafted into the old legacy way of doing things.

The next type of application is one that has some code written, but release is still way off in the future. This could benefit from using objects as above, but in addition to that, there is a greater chance to convert existing code to objects. Whether or not this happens depends on the design, the skill set of the developers, and if there time in the budget to squeeze a few of these changes in. An interesting phenomenon happens here. It may take some time to do a conversion and thus it will not seem worth the effort as it puts the project behind. What is missing from that viewpoint is that when you have an object you can use, that is a bit of code you do not need to write again, so there is a timesaving here. The payoff comes when you start using these objects. Therefore, you could say that coding OOP style is front heavy.

The last type of application is the new one. Decisions are easy at this stage as the real code work has yet to start. One could start a new ABC project and many Clarion developers choose this route with success.

Those are three broad scenarios to look at. But you are the one who will have to decide when and how you wish to proceed.

## Back to Encapsulation

---

Unlike the previous chapter where I described the theory of encapsulation, this section shows some actual code use. Again, starting very simply, here is the `CLASS` (which is encapsulation):

```
PlayWave      CLASS,TYPE
WaveName      CSTRING(FILE:MaxFileName)
Play          PROCEDURE
END
```

That is a simple class with one property and one method encapsulated in it. Notice the `WaveName` property, which contains the name of the wave file. `FILE:MaxFileName` is an `EQUATE` which happens to be 256. Therefore, this declares a `CSTRING` variable, 256 bytes long (minus one for the terminating character). Since it is a `CSTRING`, you do not need to `CLIP` it.

However, notice that this class makes no mention of which wave file to play. There is no mention of the file name in the method either.

You may guess, based on the name of the method, it plays the wave file in `WaveName`. This is a correct assumption. However, there is something unusual about the `Play` method. Have you spotted what it is yet?

`WaveName` does not take any parameters. How does the `Play` method know which wave file to play?

Remember, we are talking about encapsulation here. Inspect the following method code listing and see if you can see how this works.

```
PlayWave.Play  PROCEDURE
CODE
  SndPlaySound(SELF.WaveName,1) ! Call API to play .WAV file.
RETURN
```

### ❑ Function Overloading

Since this is code inside of a method, the object name is `SELF`. Thus, any instance of `PlayWave` works. `PlayWave` is not instantiated due to the `TYPE` attribute. This means that this class is a definition only. In order for this to work, the `PlayWave` class must be instantiated before you attempt to use it in code.

Before I use this class, this is a good time to bring up that a class definition does support function overloading. The declaration of the `PlayWave` class can become something like this:

```
PlayWave      CLASS,TYPE
WaveName      CSTRING(FILE:MaxFileName)
Play          PROCEDURE
Play          PROCEDURE(STRING xWaveName)
END
```

Notice the two `Play` methods. One takes a parameter the other does not. Yet, both have the same label. This is function overloading and this is legal in Clarion. The rule is that any procedure you can declare in a `MAP` statement, you may declare in a class.

The `TYPE` attribute tells you this is only a definition. Before you may use an object, it must be there first. Thus, you need to instantiate the object the above definition describes. How do you instantiate this class?

One line of code is all you need, which can appear anywhere after the class declaration and before the `CODE` statement.

```
MakeNoise      PlayWave !Instantiate the PlayWave class
```

When this program starts, the `MakeNoise` object is instantiated. Assuming there is a window declaration with two button controls on it, you could call these methods when these buttons are pressed. The code could look like this:

```
ACCEPT
CASE ACCEPTED()
OF ?PlayButton
    MakeNoise.WaveName = LONGPATH() & '\hallel.wav'
    MakeNoise.Play()
OF ?PlayTooButton
    MakeNoise.Play(LONGPATH() & '\hallel.wav')
END
END
```

Notice that the code is calling both methods. The first sets the name of the wave file to play, then calls the method. This works because the code for this method uses a property of the class. The second uses the name of the file as a parameter. This works as well.

This is a clever thing to do with a class. If you function overload two or more methods, this makes your class flexible. This does not always mean that two or

more methods do the same task; the overloaded methods may do something different. Which approach you use is up to you.

Here's the method code:

```
PlayWave.Play      PROCEDURE
CODE
  SndPlaySound(SELF.WaveName,1)
RETURN

PlayWave.Play      PROCEDURE(String xWaveName)
CODE
  SELF.WaveName = xWaveName
  SndPlaySound(SELF.WaveName,1)
RETURN
```

Notice that each method has the prototype listed. This is so the compiler knows which method belongs to which declaration. You do not have to use the attributes or return types here (if any). I should mention that some OOP coders do use attributes and return types (if any), but place them as comments at the end of the line.

Also, notice that the name of the declaring class is used. Do not use the name of instantiated objects! See OOP1.PRJ in the code folder for a working example.

## ❑ How to code large CLASS structures

Here is a little tip when you are coding classes. When you declare a method, you do not have to write *working* code for it right away. You may not know the complete design or functionality yet. Thus, the minimum method code you need to get a clean compile is this:

```
ClassLabel.MethodName      PROCEDURE(<prototype>)
CODE
```

Using this method (no pun intended), you may declare your class structure any way you want, then simply drop in the minimum method code. Then you can test a few methods at a time, so even large class structures are not a concern. You could even call these “empty” methods in your testing. Nothing happens when you do.

## Instantiation and Inheritance

---

You now have a working class labeled `PlayWave`. As this code progresses, it is getting unwieldy with many declarations. Thus, the `PlayWave` class changes as follows:

```
PlayWave  CLASS,TYPE, |      MODULE('PLAYWAVE.CLW'),|
LINK('PLAYWAVE.CLW')
WaveName  CSTRING(FILE:MaxFileName)
Play      PROCEDURE
Play      PROCEDURE(STRING xWaveName)
END
```

### ❑ Module and Link

Notice two new attributes to the class definition. These are the `MODULE` and `LINK` attributes. The `MODULE` attribute tells the compiler where the method code is. The `LINK` attribute means that the external source file is placed in the project settings, without explicitly naming it there.

This is the code for the *playwave.clw* file:

```
MEMBER()
MAP
    INCLUDE('SNDAPI.CLW'),ONCE
END
INCLUDE('PLAYWAVE.INC'),ONCE

PlayWave.Play  PROCEDURE                                ! Play a WAV file.
CODE
    SndPlaySound(SELF.WaveName, 1)
    RETURN

PlayWave.Play  PROCEDURE(STRING xWaveName)! Play a .WAV file.
CODE
    SELF.WaveName = xWaveName
    SndPlaySound(SELF.WaveName, 1)
    RETURN
```

The empty `MEMBER` statement indicates the module is a “universal member module” that you can compile in any program by adding it to the project. In other words, this code doesn’t depend on any one project.

The `MAP` statement has the sound API declarations. The `ONCE` attribute on the `INCLUDE` statement means that if the contents of this source is already present,



then do not include it again. The `ONCE` attribute makes the following type of “dirty” declaration obsolete:

```
INCLUDE('SomeInclude.inc',_SomeInclude_)
_SomeInclude_ EQUATE(1)
```

The code after the `MEMBER` statement is the same method code we have seen already. The only difference is this code is no longer in the main source module. Thus, you may keep your main program logic in one file. Simply `INCLUDE` whatever you need, to keep this module free of “clutter”.

### ❑ Instantiating multiple objects

Instantiating is simple to do in Clarion. Using the `PlayWave` class, you may instantiate new instances of `PlayWave` with a few simple lines of code:

```
Noise1      PlayWave    !Instantiate new object of PlayWave type
Noise2      PlayWave
Noise3      PlayWave
```

If you need many instances of a class, this is the easiest way to do it. In addition, each instance of a `PlayWave` class inherits everything from `PlayWave`. Assuming a window with buttons, the following code in the main body works:

```
ACCEPT
CASE ACCEPTED()
OF ?OkButton
    Noise1.WaveName = LONGPATH() & '\f22.wav'
    Noise1.Play()
OF ?OkButton2
    Noise2.WaveName = LONGPATH() & '\train.wav'
    Noise2.Play()
OF ?OkButton3
    Noise3.WaveName = LONGPATH() & '\a10flyby.wav'
    Noise3.Play()
END
END
```

For a working example, see `OOP2.PRJ` in the code folder.

### ❑ Manual instantiation

So far, all the examples use simple instantiation. Manual instantiation is just as effective, but you need to keep track of what you are doing. You may need manual instantiation if you have references in a class structure.

Assuming the `PlayWave` class is the same, what follows are a few more declarations:

```
TypeQue      QUEUE,TYPE
MyString     STRING(15)
END

WorkClass    CLASS,TYPE,MODULE('WORKCLAS.CLW'),|
              LINK('WORKCLAS.CLW')
WorkQ        &TypeQue
Noise        &PlayWave
FillQue      PROCEDURE, LONG, PROC
END
```

The `TypeQue` structure is simply a `QUEUE` definition. The `TYPE` attribute means the same here as it does on a `CLASS` structure.

The `WorkClass` declaration is a definition as well. Again, the `MODULE` and `LINK` attributes are present. However, notice that there are two properties in `WorkClass`. They are simply references to other declarations. Therefore, you can see that class structures can accommodate complex structures. By using a reference, it is the same as if the declarations are in the class structure. You are not limited to `QUEUE` and `CLASS` structures. You may use a reference in any manner a reference is legal elsewhere in Clarion. This means you may have `&WINDOW`, `&FILE` and other complex structures referenced.

The method in `WorkClass` returns a value. The `PROC` attribute prevents a compiler warning, as `PROCEDURES` did not return values in earlier versions of Clarion. This was reserved for `FUNCTIONS`. `FUNCTION` is a deprecated language statement.

You can create an instance of `WorkClass`, just like `PlayWave`:

```
Work    WorkClass !Create an instance of the WorkClass object.
```

OK, there is nothing new here. Where do the references come in? If you tried to use them in code at this point, you will get a GPF. The reason is that there is no memory allocated for them – yet.

Which means you must instantiate them. The `NEW` statement does this as follows:

```
ACCEPT
CASE EVENT()
OF EVENT:OpenWindow
    Work.WorkQ &= NEW TypeQue    !Create instance of TypeQue
?   ASSERT(~Work.WorkQ &= NULL)
    Work.Noise &= NEW PlayWave    !Create instance of PlayWave
?   ASSERT(~Work.Noise &= NULL)
    IF Work.FillQue()
        ?List1{PROP:From} = Work.WorkQ
        SELECT(?List1,1)
    END
END
```

The way to do this is to make a reference assignment with the `NEW` statement. The `ASSERT` is assuming that the reference is not `NULL`. If this is not the case, you can GPF the program, but only if in debug mode. You can spot this by the question mark in column one (which is why they are red in color in the editor). When in debug mode; GPFing a program is a fast and easy way to find bugs in your code. Placing `ASSERT` statements in your code is a good way to “bullet proof” your code.

Since the `FillQue` method returns the number of items in a `QUEUE` structure, using it with the `IF` statement acts like a Boolean function. If there were no entries in the `QUEUE`, the method returns zero, thus the `IF` statement fails. In Clarion, a zero is false, anything else is true. If `Work.FillQueue` returns true, then the `?List` control gets its data from the `Work.WorkQ`. The `PROP:From` expression is what makes this work. Finally, the `SELECT` statement selects the first record in the list box.

### ❑ Do not forget to clean up after yourself

I said earlier to ensure anything you `NEW` you must `DISPOSE`. I never stated why. Any object instantiated with `NEW` is never destroyed automatically. You must do this yourself. If you forget, you have a memory leak.

Thus, the following code destroys the objects:

```
IF ~Work.WorkQ &= NULL    !if not null
    FREE(Work.WorkQ)      !Delete entries, de-allocate memory
    DISPOSE(Work.WorkQ)   !Destroy instance of TypeQue queue
END
IF ~Work.Noise &= NULL    !if not null
    DISPOSE(Work.Noise)   !Destroy instance of PlayWave
END
```

Notice the test of “nullness”. If the queue reference is already `NULL`, the code drops through to the next statement. If not, then `FREE` deletes all entries from a `QUEUE`

and de-allocates the memory they occupied. It also de-allocates the memory used by the QUEUE's "overhead." Then the code destroys the reference to `Work.WorkQ` by using `DISPOSE`.

You can see a similar test for the `Work.Noise` object.

For further study, see `OOP3.PRJ` in the code folder.

## Constructors and Destructors

---

If you ever need to use constructors and destructors in your classes, a good rule of thumb is to place code that always needs to execute in them. An example is instantiating another object in the case of a constructor. If I use a constructor, I use a destructor to clean up what the constructor did.

Here is a good example of this. I have this class definition:

```
ConstructClass CLASS,TYPE, |
                        MODULE('CNSTRUCT.CLW'),|
                        LINK('CNSTRUCT.CLW')
WorkQ                  &TypeQue           !Reference to typed queue
Noise                  &PlayWave          !Reference to typed class
FillQue                PROCEDURE, LONG, PROC
Construct              PROCEDURE
Destruct               PROCEDURE
END
```

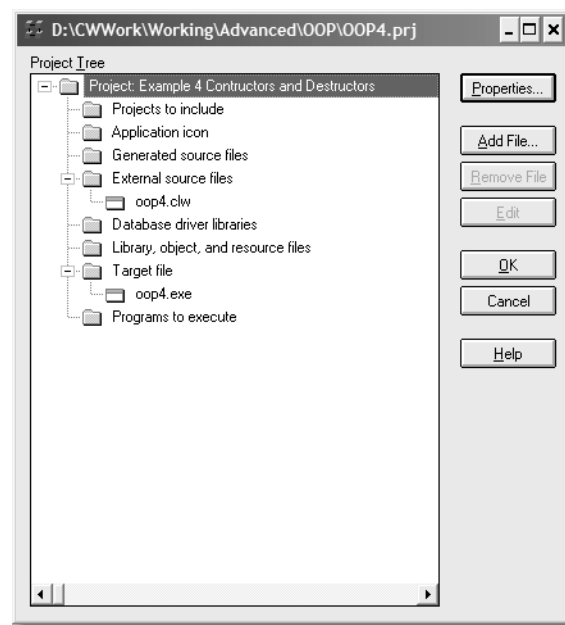
`ConstructClass` has two properties that are references, one to a queue and the other to another class. Since I need to make reference assignments to both before I can legally use them, the code to do this goes in my `Construct` method. The `Destruct` method merely undoes what `Construct` did.

```
ConstructClass.Construct    PROCEDURE
CODE
  SELF.WorkQ &= NEW(TypeQue)    !Instantiate the queue
? ASSERT(~SELF.WorkQ &= NULL)    !Assert that it is not null
  SELF.Noise &= NEW(PlayWave)    !Instantiate Playwave object
? ASSERT(~SELF.Noise &= NULL)    !Assert that it is not null

ConstructClass.Destruct    PROCEDURE
CODE
  IF ~SELF.WorkQ &= NULL        !If the queue is not null
    FREE(SELF.WorkQ)            !Free the queue
    DISPOSE(SELF.WorkQ)         !destroy the queue
  END
  IF ~SELF.Noise &= NULL        !If Noise object is not null
    DISPOSE(SELF.Noise)         !destroy the Noise object
  END
```

*Cnstruct.clw* is where you find the above code. The LINK attribute on the class tells the linker to add this to the project, even though you do not see it there.

**Figure 3-1: The project dialog.**



For further study, see OOP4.PRJ in the code folder.

## Derivation

---

The previous example has a reference to another class in its own class definition. This does give you derivation when the object is instantiated (in the `Construct` method). However, this is not the only way to do this:

```
DerivedClass  CLASS(PlayWave),TYPE
WorkQ        &TypeQue
FillQue      PROCEDURE, LONG
Construct    PROCEDURE
Destruct     PROCEDURE
END
```

As you can see in this example, the reference to the parent class is missing. However, the parameter in the class statement does the same thing.

This makes the workload on the constructor and destructor less as well. This is not much different from the previous example; more of a variation to illustrate there is more than one way to get the benefit of parent classes.

See `OOP5.PRJ` in the code folder for an example of this.

## Overriding

---

Sometimes, you need to override a parent behavior. However, do not confuse that to mean that overriding is an all-or-nothing situation. Sometimes it is, if that is what you want. However, what if you wish to extend the parent's behavior? You do have to override the method. You do this by making a method with the same label and prototype as the parent method.

However, in the code of the method there is no rule that says you cannot use the parent method. In ABC, this is a common practice. Here is the class to illustrate this:

```
OverrideClass CLASS(PlayWave),TYPE,|
                MODULE('OVERRIDE.CLW'),|
                LINK('OVERRIDE.CLW')
WinRef          &WINDOW          !Reference to a window
Control         LONG(0)
FlashSpeed      LONG(0)
Toggle         LONG, STATIC !Ensure value does not change
Flash          PROCEDURE
Play           PROCEDURE      !Override method in Parent
Init           PROCEDURE
                END
```

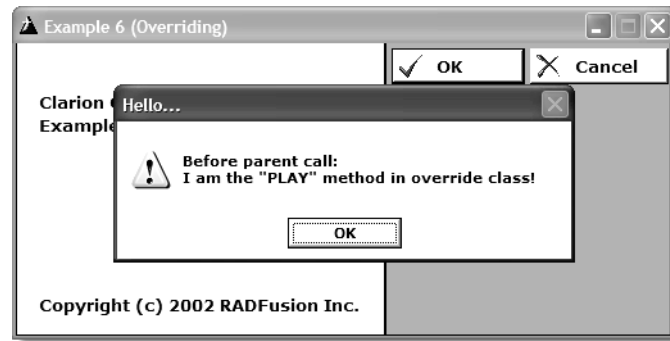
The `Play` method in `OverrideClass` overrides the `Play` method in the `PlayWave` class. You may assume that the parent `Play` method plays a wave file, like the earlier examples. The `OverrideClass.Play` method needs to do more than simply play a wave file. There is code that displays messages and does a few other things. In the source file for this method (see the `MODULE` attribute for the name of the file), there is this code:

```
OverrideClass.Play      PROCEDURE ()
CODE
SYSTEM{PROP:Font,1} = 'Verdana'
SYSTEM{PROP:Font,2} = 8
SYSTEM{PROP:Font,4} = FONT:Bold
MESSAGE('Before parent call: |I am the "PLAY" method ' |
        & 'in override class!','Hello...',ICON:Exclamation)
PARENT.Play()          !Call Play method in PlayWave (PARENT) class
SYSTEM{PROP:Font,4} = FONT:Italic
SYSTEM{PROP:Font,2} = 10
MESSAGE('After parent call: |I am the "PLAY" method ' |
        & 'in override class!','Hello...',ICON:Exclamation)
RETURN
```

Do you see the call to the parent's method? `PARENT` is like `SELF`, but defined as whatever the parent is. In this case, it is *PlayWave*. Therefore, the `OverrideClass` never plays a wave file, as there is no code to do so. Instead, it uses another method that does have that code.

In this instance, the method needs to not only play a wave file, but also display messages. In addition, there is some code to change the font of the message boxes. When run, it appears as shown in Figure 3-2.

Figure 3-2: Calling the Play method



See OOP6.PRJ in the code folder for an example.

## Virtual methods

---

Perhaps the hardest thing for those studying OOP is to understand how virtual methods work in their code. It is a strange concept for procedural, top-down coders to grasp. I have had students tell me they cannot understand how virtual methods work because the code jumps all over the place. And they are right. Of course, confusing code is nothing new. When the `ACCEPT` loop first appeared, that was perplexing to many developers too. In the old legacy (pre-C6 Clarion templates), routines with nearly identical names call each other and back.

ABC uses many virtual methods. There is more code “action” going on here, so perhaps the confusion stems from the increased activity. It really does not matter, as once you understand how virtual methods work and why you use them it’s a walk in the park.

Here are two examples. The first carries on the current theme from the example projects:



```
SECTION('PLAYWAVE')

PlayWave      CLASS,TYPE,MODULE('PLAYWAV2.CLW'),|
               LINK('PLAYWAV2.CLW')
WaveName      CSTRING(FILE:MaxFileName)
Play          PROCEDURE (),VIRTUAL
CallVirtual   PROCEDURE ()
               END

SECTION('VIRTUAL1')

Virtual1      CLASS(PlayWave),TYPE,MODULE('VIRTUAL1.CLW'),|
               LINK('VIRTUAL1.CLW')
Play          PROCEDURE (),DERIVED
               END

SECTION('VIRTUAL2')

Virtual2      CLASS(PlayWave),TYPE,MODULE('VIRTUAL2.CLW'),|
               LINK('VIRTUAL2.CLW')
Play          PROCEDURE (),DERIVED
               END

SECTION('RESTOFPROGRAM')
```

There is now a change to the `PlayWave` class. Notice the additional `CallVirtual` method. Also, notice the source where you find this method. The two almost identically named classes derive from `PlayWave`.

Notice the `Play` method in each class. The two child methods override the parent. The parent method has the `VIRTUAL` attribute, which is required. The two child methods use the `DERIVED` attribute. Remember that `DERIVED` means “virtual” plus it protects the child methods from function overloading if the parent prototype changes.

Lastly, notice the `SECTION` statements. This is not required for coding objects, but is rather a clean way of including the definitions in the method source file. Take a look at the source files.

```
INCLUDE('OOP7.CLW','PLAYWAVE'),ONCE

PlayWave.Play          PROCEDURE ()

CODE
MESSAGE('I am Play!','Base Class',ICON:Exclamation)
SndPlaySound(SELF.WaveName,1)
RETURN

PlayWave.CallVirtual    PROCEDURE
! Call whatever virtual is currently
! replacing the dummy PARENT.Play
! method shown above.

CODE
MESSAGE('I am CallVirtual!','Base Class',ICON:Exclamation)
! Self.Play calls either V1.Play or V2.Play
! depending on which one called this method.
MESSAGE('Back at CallVirtual')
```

Notice the INCLUDE statement at the top of the listing. It includes the code starting after the “Playwave” SECTION name. It includes everything until the next SECTION or end of file, whichever comes first.

There are two methods, Play and CallVirtual. CallVirtual, when run, calls Play sandwiched by two message statements. They are there only to show what is going on when the code runs, and to demonstrate more code than simply playing wave files.

Next, the *virtual1.clw* file (*Virtual2.clw* is nearly identical, so no need to look at it).

```
INCLUDE('OOP7.CLW','VIRTUAL1'),ONCE

Virtual1.Play          PROCEDURE ()
CODE
PARENT.Play ()
MESSAGE('Hi, I''m Virtual 1 ','Virtual 1...',ICON:Exclamation)
RETURN
```

Notice the two INCLUDE statements. Both are needed as the first has the parent definition and the second INCLUDE is the definition of class derived from it.

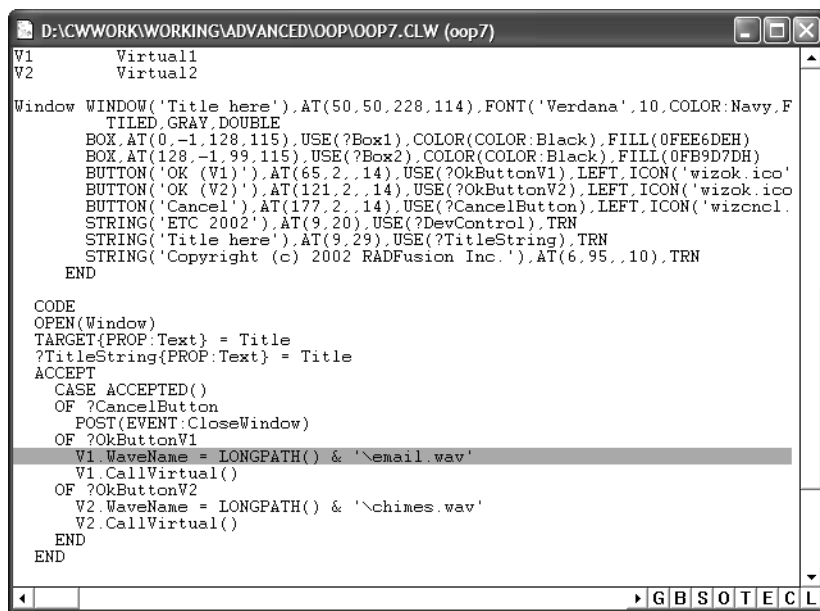
All this method does is call the parent method and show a message. The parent call does run the code to play a wave file. If you comment that line, the virtual methods still execute (just no sound).

## ❑ The Debugger as Teacher

Running the project does not really show you the calling sequence. If you wish to see this execute line-by-line, then the debugger is the best bet. Here is how I would set it up:

First, when the debugger opens the “procedures in” window, choose “\_main”. The source file for that code opens. Simply place a break point on the line of code you wish to see execute. Since the object of interest is a method call, set the break point there.

Figure 3-3: Setting a break point in the debugger



Press **Go** and the main window of the application appears. The code to run executes when you press `?OkButtonV1`. The debugger takes over and you press **Step Source** to follow the execution. Since this project is small, you can see what happens in the code. You may need a few loops through that code to get a better understanding of this. It is worth the small amount of time it takes.

### ❑ The Virtual Apple Pie

Do you remember the Apple Pie example in the previous chapter? Why not use that example with working code to show how virtual methods work? Here is all of the code:

```
PROGRAM

MAP
END

ApplePie      CLASS
PreparePie    PROCEDURE
CreateCrust   PROCEDURE,VIRTUAL    !Virtual Methods
MakeFilling   PROCEDURE,VIRTUAL
END

Dutch         CLASS(ApplePie)
CreateCrust   PROCEDURE,DERIVED    !Virtual Methods
MakeFilling   PROCEDURE,DERIVED
END

CODE          !Main body of code
MESSAGE('Lets make a Dutch Apple Pie!')
Dutch.PreparePie    !Call the Dutch object's Virtuals
MESSAGE('Now lets make an Apple Pie!')
ApplePie.PreparePie

!Code for all methods below
ApplePie.PreparePie  PROCEDURE
CODE
MESSAGE('In ApplePie Class - create the crust')
SELF.CreateCrust    !What is SELF?
MESSAGE('In ApplePie Class - make the filling')
SELF.MakeFilling    ! and why?

ApplePie.CreateCrust  PROCEDURE
CODE
MESSAGE('Creating the Apple Pie crust.')
```

```
ApplePie.MakeFilling    PROCEDURE
CODE
MESSAGE('Making the Apple Pie filling - yum!')

Dutch.CreateCrust        PROCEDURE
CODE
MESSAGE('Creating the crust for Dutch Pie.')

Dutch.MakeFilling        PROCEDURE
CODE
MESSAGE('Making the Dutch filling - yum!')
```

In the main code body, there is enough code to make two different apple pies. The main thing to keep in mind is that the `Dutch` object does not have its own `PreparePie` method. It inherited this from its parent. Thus, the `PreparePie` method is the same as the `CallVirtual` method in the previous example (functionally speaking).

When this project runs, `PreparePie` calls “down” or “forward” to the two `Dutch` methods. The only action required by the user when running this, is to press the **OK** button on each message box.

#### ❑ Moral of the story?

The advantage of virtual methods is that a parent object calls “down” or “forward” to a child class. Think about this for a minute. This also means that you can change the behavior of a parent object and *never lay a finger on the parent's code*. It also means you can alter or extend the default behavior of the parent by making a parent call in your code. You would want to do that if the parent has valid code you may use. This is the normal behavior of ABC.

---

## How to Make Any Kind of Pie

The use of virtual methods is fine and as you can see, you can get a lot done. However, if the next developer that comes along wishes to use your class to make a different type of apple pie, he must use your methods of making apple pies. However, what if they want to use a new apple pie recipe with your class? Will it work? Will they be forced to re-write your class?

The real question is, “Does the Pie Class know how to make various apple pies?”

Where I am going with this is that a pie class should know how to make any pie, apple or otherwise. This is where the interface enters the picture.

A class that implements an interface can now change its behavior without changing the class code.

```
ITEMIZE,PRE(PIE)
Crust      EQUATE      !The same as PIE:Crust EQUATE(1).
Filling    EQUATE      !The same as PIE:Filling EQUATE(2).
END

IPieType   INTERFACE    !Supply specifics for any pie type
GetType    PROCEDURE,STRING
GetIngredients PROCEDURE(BYTE bType),STRING
GetMethod  PROCEDURE,STRING
GetSuggestions PROCEDURE,STRING
END

IngredientQ QUEUE,TYPE
Ingredient  CSTRING(101)
END

PieClass   CLASS,TYPE    !The pie base class.
CrustQ     &IngredientQ
FillingQ   &IngredientQ
Type       &IPieType      !Reference to the interface
Construct  PROCEDURE
Destruct   PROCEDURE
Init       PROCEDURE(IPieType PT)
Make       PROCEDURE
ShowReceipe PROCEDURE,VIRTUAL
END
```

The first part of this code is simply another way of defining equates. The IPieType is the interface. The methods are implied virtual methods. These methods describe how one could make any type of pie. It would make sense to get what type of pie one wishes to make. What are the ingredients for a pie is a good thing to know. GetMethod describes how to make the pie. Lastly, pies do no one any good if one cannot serve the pie when ready to eat.

Next is the type definition of the IngredientQ structure. This holds the list of ingredients.

The PieClass is the base class. There are two references to the IngredientQ as the ingredients for a filling are different from a crust. The Type is a reference

property to the interface. This is another way of implementing an interface. There is a constructor and a destructor; you may assume these do housekeeping chores for the class.

The `Init` method takes a reference to the interface as a parameter. I will explain this shortly. The rest of the class makes the pie and shows the recipe.

Here are the derived classes:

```
PieView      CLASS(PieClass)
GetViewType  PROCEDURE, BYTE
ShowReceipe  PROCEDURE, DERIVED
END

PrettyPieClass CLASS(PieClass), TYPE
Make         PROCEDURE
ShowReceipe  PROCEDURE, DERIVED
END

PrettyPie    &PrettyPieClass
```

Notice the use of the `DERIVED` attribute on the virtual methods. These two classes also show two different ways of instantiation.

That is the code for the pie classes. Notice that the only thing these classes can do is make pies. Yet, nothing here hints at what types of pies one can make. One could make apple, chocolate, pumpkin or even mud pies.

This brings me to the rest of the declarations. See the following code.

```
ApplePieIngredientQ  QUEUE,TYPE      !Another TYPE queue
Type                 BYTE            !PIE:Crust or Filling
Ingredient            CSTRING(101)
                     END

ApplePie             CLASS,IMPLEMENTS(IPieType)
ListCounter          LONG
IngredientQ           &ApplePieIngredientQ
Construct             PROCEDURE
Destruct             PROCEDURE
                     END

AppleCheeseCake      CLASS,IMPLEMENTS(IPieType)
ListCounter          LONG
IngredientQ           &ApplePieIngredientQ
Construct             PROCEDURE
Destruct             PROCEDURE
                     END
```

Notice there is another queue structure, as the design requires that crust and filling ingredients use it. Thus, an ingredient type exists in the definition.

The two classes are identical except for label. Each class implements the `IPieType` interface. The rest of the declaration is just like the previous classes: a reference to a queue structure and the automatic methods for the household chores.

```
CODE
PieView.Init(ApplePie.IPieType)  !Pass the Apple Pie interface
                                !into the PieView object.
PieView.Make()
```

`PieView` is the instantiated class. The passed parameter is the `INTERFACE`. That does not look like much. Inspecting the *Init* method might reveal some information.



```
PieClass.Init    PROCEDURE(IPieType PT)
CODE
  SELF.Type &= PT      !Remember the Interface for later use.
  FREE(SELF.CrustQ)    !Empty the queues.
  FREE(SELF.FillingQ)
  LOOP                !Get crust ingredients.
    SELF.CrustQ.Ingredient = SELF.Type.GetIngredients(PIE:Crust)
    IF ~SELF.CrustQ.Ingredient
      BREAK
    END
    ADD(SELF.CrustQ)
  END
  LOOP                !Get filling ingredients.
    SELF.FillingQ.Ingredient =
SELF.Type.GetIngredients(PIE:Filling)
    IF ~SELF.FillingQ.Ingredient
      BREAK
    END
    ADD(SELF.FillingQ)
  END
END
```

The reference assignment says which INTERFACE this method uses. This is passed via the PT variable (which is an IPieType variable).

There are two loops here, one for the crust and the other for the filling. The key line of code here is the interface method `SELF.Type.GetIngredients(Type)`. This is where the “magic” happens. Here is the code for this method:

```
ApplePie.IPieType.GetIngredients    PROCEDURE(BYTE bType)
CODE
  IF ~SELF.ListCounter              !Queue is sorted
  LOOP SELF.ListCounter = 1 TO RECORDS(SELF.IngredientQ)
    GET(SELF.IngredientQ,SELF.ListCounter)  !Get the ingredient
    IF SELF.IngredientQ.Type = bType !Is what we are looking for?
      SELF.ListCounter -= 1          !Yes, decrement the counter
      BREAK                          !and break out of loop.
    END
  END
  END
  SELF.ListCounter += 1              !Increment the counter
  IF SELF.ListCounter > RECORDS(SELF.IngredientQ)
    SELF.ListCounter = 0             !re-set our counter
    RETURN ''                         !and scram.
  END
  GET(SELF.IngredientQ,SELF.ListCounter) !Get the ingredient
  IF SELF.IngredientQ.Type <> bType    !Wrong type of ingredient
    SELF.ListCounter = 0             !re-set our counter
    RETURN ''                         !and scram.
  END
  RETURN(SELF.IngredientQ.Ingredient) !Return proper ingredient.
```

The queue structure looped in the code was preloaded in the constructors. It simply returns the correct ingredient based on the ingredient type. In this example, it is either a crust or filling type.

You can now see how the `Init` method fills its own `IngredientQ` by calling this interface method. Once the `Init` method is finished, it's time to make the pie. This is the next line of code in the main body, the making of the pie.

```
PieClass.Make    PROCEDURE    !Call the VIRTUAL ShowReceipe method.
CODE
  SELF.ShowReceipe()
```

That calls the object's `ShowRecipe` method. This is virtual, thus there is nothing in the `PieClass.ShowRecipe` method. Remember, a parent class calls the derived method instead of its own when they are virtual methods.

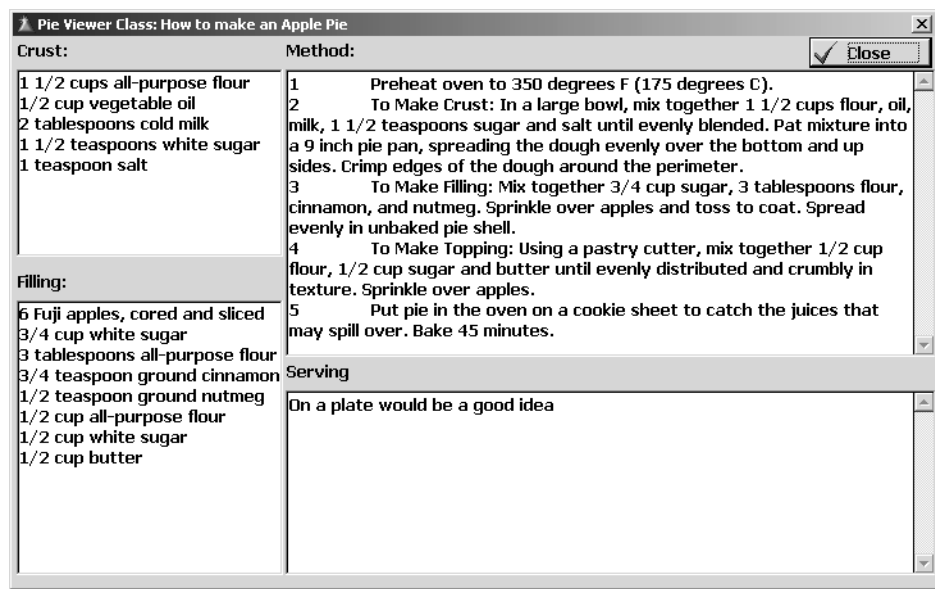
Thus, `PieView.ShowRecipe` runs instead. It is the method where the user first sees anything happening. It is the one that offers the choice of which window type they want to use and then opens that window.

```
CODE
EXECUTE(SELF.GetViewType())           !Method returns 1 or 2.
    TheWindow &= ThreeDWindow         !Reference assign if 1.
    TheWindow &= FlatWindow           !Reference assign if 2.
END
Method = SELF.Type.GetMethod()        !Get the cooking method.
Suggestions = SELF.Type.GetSuggestions() !serving suggestions.
OPEN(TheWindow)                       !Open chosen window
!Set the window caption.
TheWindow{PROP:Text} = 'Pie Viewer Class: ' &
SELF.Type.GetType()
ACCEPT                                !And away we go.
CASE FIELD()
OF ?Close
CASE EVENT()
OF EVENT:Accepted
    POST(EVENT:CloseWindow)
END
END
END
CLOSE(TheWindow)
```

The code first calls `GetViewType` to find out which type of window the user wants to use. The `EXECUTE` statement then assigns the reference to one of two windows. The `GetMethod` returns the instructions. Remember this is the interface method, so this class “knows” how to get an outside behavior, which you control. The `GetSuggestions` is the serving suggestion for an apple pie. If you made a chocolate pie, you may want to code a different `GetSuggestions` method.

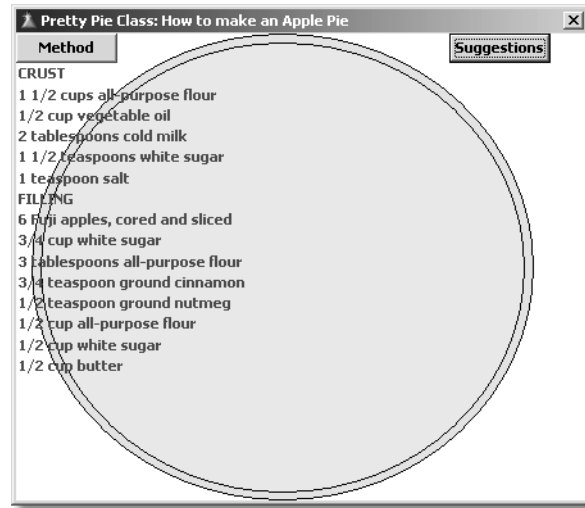
It is at this point the window opens.

Figure 3-4: Viewing the recipe



There is more to the example. Study `PIE.PRJ` in the code folder for more information. Look at the `PrettyPie` objects. This uses the same interface; however, the results are quite different, as shown in Figure 3-5.

Figure 3-5: Another recipe view



### ❑ One Last Concept

The pie code is illustrating is that the class knows how to make a pie. The interfaces deal with making different types of pies.

Let me give you another concept that may reinforce how to think with interfaces. Suppose you have a class that deals with washing dishes. You have a method to wash a plate, which may involve soaking the plate in a sink full of water and scrubbing it by hand.

What happens when you get an automatic dishwasher interface? You don't soak the plate in water and scrub it, you just put it in the dishwasher. Different actions, yet the same results – a clean plate.

Some people use the family dog to wash the plates; I'll leave that interface design to the reader.

## Summary

---

Interfaces are excellent ways to add new behaviors without touching existing code. You do have to write the code for all the methods for a given interface. That is the trade off. I feel that for the extra functionality you get with interfaces, the small price of writing code for all the methods is worth it. Remember, you may still code stub methods for behaviors you do not need.

Study and play with the provided code examples. Make changes to these projects. Please break them too. Then fix them. The only real true way to “get” OOP is to write code.

# Chapter 4

---

## ABC AND OOP

This chapter covers the ABC class library. ABC is an implementation of OOP, so you can see the importance of understanding OOP first!

ABC made its debut with the release of Clarion version 4. ABC has evolved over the years. Along the same time line, articles describing how to use various pieces of ABC were penned. Bruce Johnson, of Capesoft ([www.capesoft.com](http://www.capesoft.com)) fame, has written a text on ABC. Clarion Magazine ([www.clarionmag.com](http://www.clarionmag.com)) is another excellent resource. You will need a subscription to read most articles. In addition, another excellent resource is James Cooke's Clarion Foundry ([www.clarionpublisher.com](http://www.clarionpublisher.com)). I would be remiss without mentioning my own site, RADFusion ([www.radfusion.com](http://www.radfusion.com)).

This chapters covers certain areas to give you a better understanding of what is going on under the hood of ABC, as well as ABC problems and solutions.

The average Clarion developer can use the template interface to the ABC objects for common tasks like accessing data, developing browse lists and edit forms with little or no difficulty. Just about every business application in the world requires these tasks, and ABC handles them smoothly and without the developer having to think about what's going on in the code.

It is the uncommon areas, or exceptions to the rule, that sometimes give developers fits. The other problem area is inexplicable behaviors. If you add to this mix a lack of information (common when learning something for the first time), it magnifies the difficulty of working with ABC.

ABC does vast amounts of work for you. There is the old 90/10 rule – ABC does 90% of the work, you must do the last 10%. That last 10% can sometimes seem like the entire workload.

## Global objects

---

There are a number of derived objects that live in the global area of any application. They come straight from ABC. You can find these objects in any application by pressing **Global**, and scrolling to the right until you come to the **Classes** tab. It will look similar to Figure 4.1 (you may not have data, embeds or extensions, thus no green check):

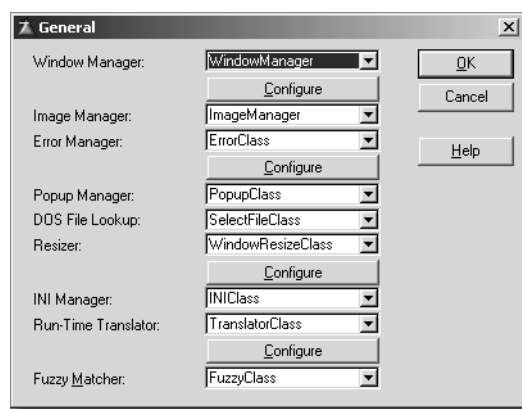
Figure 4-1:





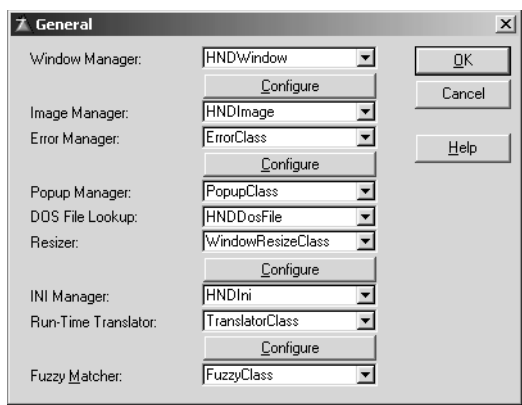
The names of the global objects as well as their settings are available here. For example, pressing General displays the dialog in Figure 4.2:

**Figure 4-2: Global General Properties**



Each class has a drop list. This is so that you may change to a different class instead of ABC's default. For example, here is what some of these setting would look like if you changed some of the defaults to use the Clarion Handy Tools ([www.cwhandy.com](http://www.cwhandy.com)) classes:

**Figure 4-3: Defaults used by the Clarion Handy Tools**



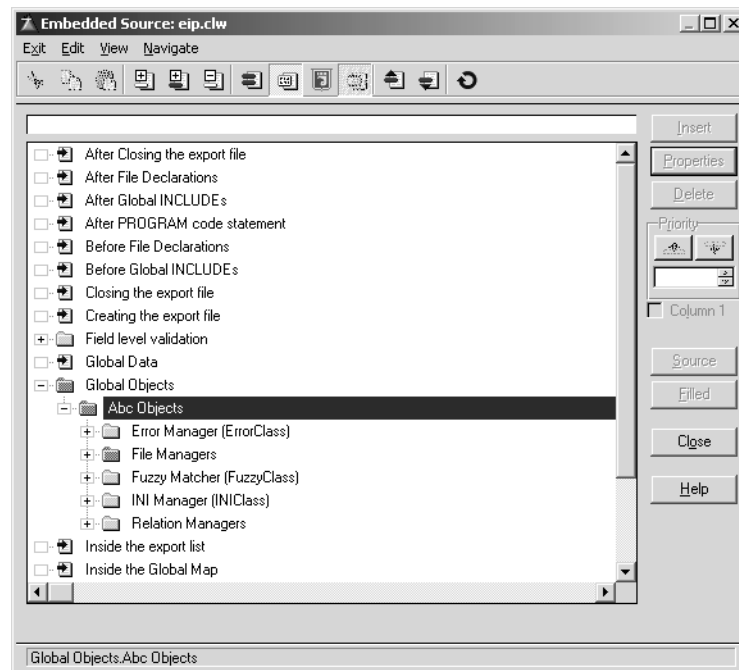
The other dialogs operate in a similar fashion. The moral of the story is that if you wish to use your own classes for your global objects, ABC allows it.

## Global embeds

---

If you press the **Embed** button, you will see a tree list of points where you may add your own code. The following figure shows the ABC global objects.

Figure 4-4: The ABC global objects



---

**NOTE** There may be more or fewer ABC objects visible depending on features needed for the application. For example, if you use the translator option, you would see a `TranslatorClass`. If **Fuzzy matching** is unchecked in the global properties, it will not appear on this list.

---

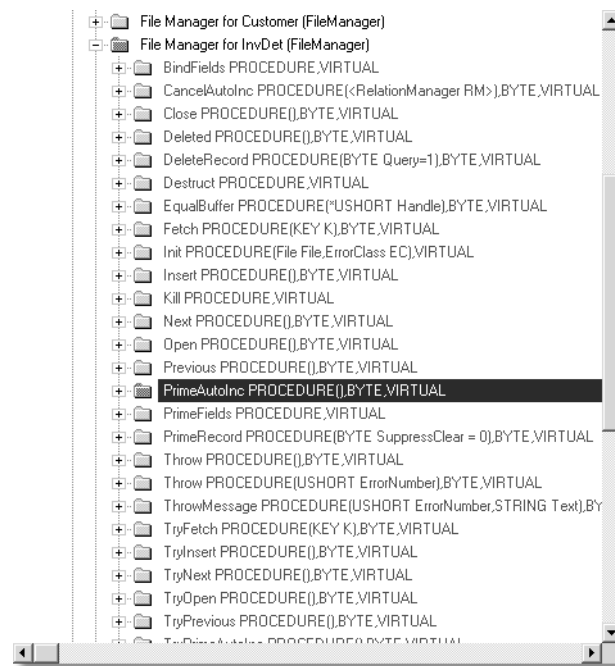
The global objects affect the entire application. Any embedded code placed here will, of course, also affect the entire application. The reason is that your code changes the way the methods of these global objects work, via virtual methods.

If you expand the **File Managers** embed, you will see a file manager object for each file used in your application. While that could make for a busy embed tree for

large dictionaries, it does give you control over certain files, while leaving others alone.

Look at the following figure.

Figure 4-5:



You can see an overridden `PrimeAutoInc` method. The documentation on this method describes it as follows:

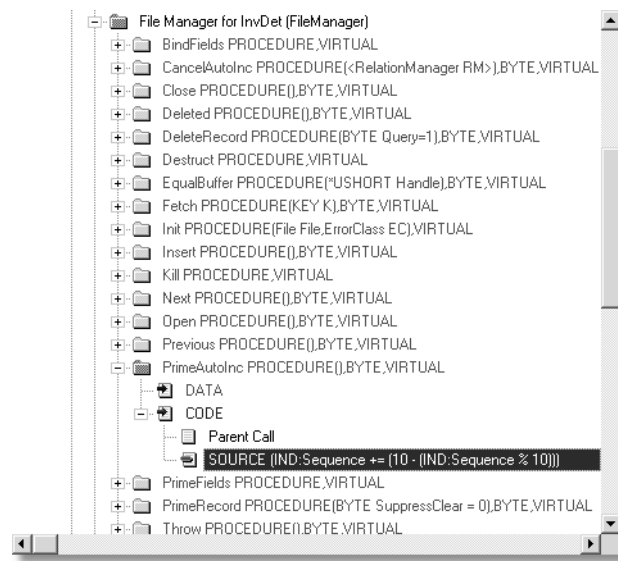
When a record is inserted, the `PrimeAutoInc` method prepares an auto-incremented record for adding to the managed file and handles any errors it encounters. If you want to provide an update form that displays the auto-incremented record ID or where RI is used to keep track of children, then you should use the `PrimeAutoInc` method to prepare the record buffer.

What that means is that this method controls how to auto increment a value. On first use, it starts with an autoincrement value of 1 and subsequently adds 1 to the last value used. If you want this behavior, then you have no business changing how this works. If you do not want this behavior, then you must add code to alter or extend how auto incrementing behaves.

You may gather that starting at 1 and then adding 1 to the last value is not the behavior wanted by this application and on this file only. So what does this embedded code do?

Look at the following figure:

Figure 4-6:



This is all the code needed to alter the default behavior. In short, instead of starting at 1 and then adding 1 to the last saved value, it starts at 10 and increments by 10. Thus, when adding details to an invoice, the line items are numbered starting with 10, then 20 and so on.

A user could then edit line item numbers by, for instance, changing 30 to 15. This would place the third line item between the first (10) and the second (20).

### ☐ How does ABC do files?

A small tour of the global file objects is in order. ABC generates two (or more) modules which are strictly under its control. They are named `AppNameBC.CLW` and `AppNameBC0.CLW`. Strictly speaking, ABC forces the eight-three convention. Eight characters for the file name, three for the extension. ABC strictly enforces the BC and BC0 part of the file name, so the actual names are the application name up to six and five characters respectively.

One final note about this naming convention; depending on the number of files in the dictionary, the BC0 could become BC1, BC2 etc. Expect to see this for every ten files.

The BC module contains code similar to this figure:

```
MEMBER('eip.clw')

MAP
  MODULE('EIPBC0.CLW')
    EIPBC0:DctInit      PROCEDURE
    EIPBC0:DctKill      PROCEDURE
    EIPBC0:FilesInit    PROCEDURE
  END
END

DctInit      PROCEDURE
CODE
  EIPBC0:DctInit
  EIPBC0:FilesInit

DctKill      PROCEDURE
CODE
  EIPBC0:DctKill
```

This code simply declares a MAP structure and the procedures in it. The code for the procedures must be there as well, and this is visible. The `DctInit` procedure calls two procedures in this case.

In Clarion 6, this code is now found in its own class, specifically a constructor. This is so that each thread has its own instance of the file and relation managers. The `DctKill` method now lives in the destructor of the `Dictionary` class. When the thread dies, so does that thread's instance of the `File` and `Relation` manager classes.

The procedure names come from the name of the application (EIP for “Edit In Place”), the BC0 designation, a colon and the name of procedure. Let's inspect the code.

#### ❑ DctInit

First up is the `DctInit` procedure.

```
EIPBC0:DctInit    PROCEDURE
CODE
Relate:Item &= Hide:Relate:Item
Relate:Config &= Hide:Relate:Config
Relate:Customer &= Hide:Relate:Customer
Relate:InvDet &= Hide:Relate:InvDet
Relate:InvHdr &= Hide:Relate:InvHdr
```

This simply makes reference assignments to a class. That is all this does. The `Hide:Relate:<File>` is derived from `RelationManager`, an ABC class. This means that `Relate:<File>` could be thought of as grandchildren of `RelationManager`.

Each of these “hidden” classes have methods in them, at the minimum an `Init` and `Kill` method. There could be more methods depending on whether there are embeds, since each embed means an automatic override of a virtual method. The class definition for `Hide:Relate:Item` is as follows:

```
Hide:Relate:Item    CLASS(RelationManager)
Init                PROCEDURE
Kill                PROCEDURE(),DERIVED
END
```

This means there must be code for both methods.

```
Hide:Relate:Item.Init PROCEDURE
CODE
Hide:Access:Item.Init
SELF.Init(Access:Item,1)

Hide:Relate:Item.Kill PROCEDURE
CODE
Hide:Access:Item.Kill
PARENT.Kill
Relate:Item &= NULL
```

The `Init` method discussion begins in the `FilesInit` section below.

### ❑ DctKill

Next is the `DctKill` procedure:

```
EIPBC0:DctKill    PROCEDURE
CODE
Hide:Relate:Item.Kill
Hide:Relate:Config.Kill
Hide:Relate:Customer.Kill
Hide:Relate:InvDet.Kill
Hide:Relate:InvHdr.Kill
```

This is simply a collection of procedure calls. For now, you may assume it undoes DctInit.

## ❑ FilesInit

Next is FilesInit.

```
EIPBC0:FilesInit    PROCEDURE
CODE
Hide:Relate:Item.Init
Hide:Relate:Config.Init
Hide:Relate:Customer.Init
Hide:Relate:InvDet.Init
Hide:Relate:InvHdr.Init
```

A little bit of explanation is in order here. Each of these procedure calls is really a method. These are really the parent methods, but recall the reference assignments from earlier. If the preceding DctInit procedure did not happen (which did the reference assignments), you would be getting GPFs at this point.

If you look back on the code for Hide:Relate:Item.Init, this is a method call in another class, which is why SELF would be improper here.

```
Hide:Access:Item    CLASS(FileManager)
Init                PROCEDURE
Kill                PROCEDURE(),DERIVED
END
```

You can see what the parent class is by the above definition. The code for the methods is as follows:

```
Hide:Access:Item.Init PROCEDURE
CODE
SELF.Init(Item,GlobalErrors)
SELF.FileNameValue = 'Item'
SELF.Buffer &= ITM:Record
SELF.Create = 1
SELF.LockRecover = 10
SELF.AddKey(ITM:ItemIdK,'Item Id Key',1)
SELF.AddKey(ITM:ItemNumberK,'Item Number Key',0)
Access:Item &= SELF

Hide:Access:Item.Kill PROCEDURE
CODE
PARENT.Kill
Access:Item &= NULL
```

While I could explain the above code in detail, it is not really necessary. You may safely assume that all the lines starting with `SELF` are methods and properties derived from `FileManager`. This means you are free to read the help for each.

The last line of the `Init` method is a bit interesting. It is making a reference to the current object (`SELF`). This is where the `Access:<File>` class comes from that you see in generated code.

I will grant you that these intermediate classes may not be entirely necessary, but since they deal with setting up objects that deal with relations of files and file management, it does make some sense to move these off to the side. Also, these methods execute only when the program starts and quits. It also serves to keep your source cleaner by having these classes in source files that are off to the side as well.

There isn't any rule in Clarion OOP that says you have to do this, it is merely a design decision of the ABC architects.

The reason why I am showing you this is to make it clear that if you wish to add code (embeds) for the file objects, the resulting code is always global. The best explanation is looking at the simple code in the method. Since I earlier used the `InvDet` file as an example, I'll focus on that file.



```
Hide:Access:InvDet CLASS(FileManager)
Init               PROCEDURE
Kill               PROCEDURE(),DERIVED
PrimeAutoInc       PROCEDURE(),BYTE,PROC,DERIVED
ValidateRecord     PROCEDURE(<*UNSIGNED Failed>),|
                   BYTE,DERIVED
                   END

Hide:Relate:InvDet CLASS(RelationManager)
Init               PROCEDURE
Kill               PROCEDURE(),DERIVED
                   END
```

The above two classes derive from the `FileManager` and `RelationManager` classes. Both are ABC classes. In addition, there is no `TYPE` attribute on either one, so these are declarations of the classes and instances (meaning they are ready for use).

Do not put too much stock in the names of these classes. You may never see these anyway, as I already discussed. But the one that is interesting is the `Hide:Access:InvDet` class. Notice the `PrimeAutoInc` method? It is there only because of embedded code in that method. Thus, it overrides the default ABC method.

Here is the generated code based on the code in the embed point:

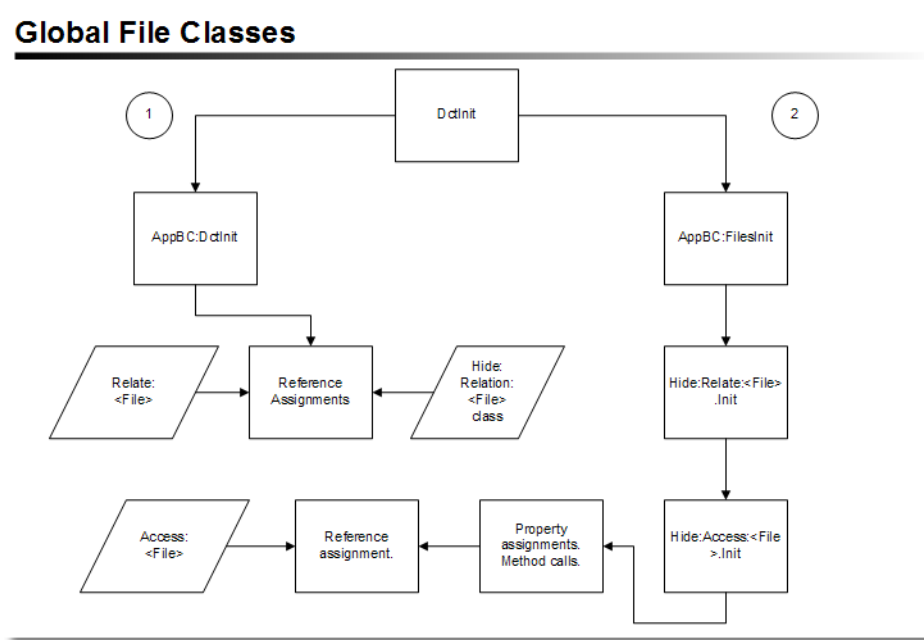
```
Hide:Access:InvDet.PrimeAutoInc    PROCEDURE
ReturnValue                       BYTE,AUTO

CODE
ReturnValue = PARENT.PrimeAutoInc()
IND:Sequence += (10 - (IND:Sequence % 10))
RETURN ReturnValue
```

Remember, this code was placed in this embed for one reason only – to change the way ABC does auto incrementing. And modifying ABC's behavior is the only reason you use any ABC embed.

Anything dealing with files declared in the dictionary (including global variables) is in the global section, so the rule of thumb is that any embed to do with files is global in nature. Figure 4.7 shows the global file classes.

Figure 4-7:



## Local embeds

---

I've talked about global embeds, and now I'll discuss local embeds. In fact there are module embeds also, but there are only two for each module, one for data declarations and the other for code. Therefore I'm going to skip any detailed discussion of module embeds.

For any given type of procedure, there are certain embed points that are popular. The reason for it is that out of the box, ABC does only so much. This is by design and these features are common enough to find in any application.

Embed points have only one purpose: to extend or replace out of the box behavior.

I think most developers understand this. The burden is that they know what they want, but sometimes get into a fog about where and how to write the code.

### ❑ ABC classes in relation to templates

This would be a good time to explain the relationship to ABC classes and ABC templates. The templates are responsible for the following tasks (out of the box):

- Write the needed OOP code to make a local class based on ABC. In other words, they ensure proper derivation.
- Instantiate the derived classes before using them.
- To generate correct code for overridden methods.

That is about all they need to do, and they do the above whether or not you add embed code.

To illustrate this, suppose you create a browse from a wizard and then run it. It works, as you can see the data in the list control. The generated code for the class derivation would be similar to the following:

```
BRW1          CLASS(BrowseClass)    !Browse using ?List
Q             &Queue:Browse         !Reference to browse queue
Init          PROCEDURE(SIGNED ListBox,*STRING Posit,VIEW V,|
              QUEUE Q,RelationManager RM,WindowManager WM)
ResetSort     PROCEDURE(BYTE Force),BYTE,PROC,DERIVED
              END
```

You can see that the BRW1 class only adds a little bit to its base ABC class. The templates can detect what is on the window and they know that to make it work, and generate the class definition accordingly.

For example, here is the complete code for the Init method:

```
BRW1.Init      PROCEDURE(SIGNED ListBox,*STRING Posit,VIEW V,|
                QUEUE Q,RelationManager RM,WindowManager WM)
CODE
PARENT.Init(ListBox,Posit,V,Q,RM,WM)
SELF.SelectControl = ?Select
SELF.HideSelect = 1
IF WM.Request <> ViewRecord
    SELF.InsertControl = ?Insert
    SELF.ChangeControl = ?Change
    SELF.DeleteControl = ?Delete
END
```

Notice the first line of code, the call to the parent class. That refers to the `BrowseClass` in this case. This means there is functional code that is suitable to ensure everything is in place to use objects for a browse.

The rest of the code deals with aspects that are unique to this particular browse. In other words, it extends the base ABC functionality. This same thing happens when you embed source code.

You really do not need to worry or concern yourself with the mechanics of declaring/coding objects. Instead, the templates free you from this chore and allow you to concentrate on what you need to do for this particular browse.

In essence, what is this browse doing different from what comes out of the box? That is all you need to concern yourself with. Let ABC do the work.

Let me go over a few procedures and point out some of the common embeds.

### ❑ Browse procedures

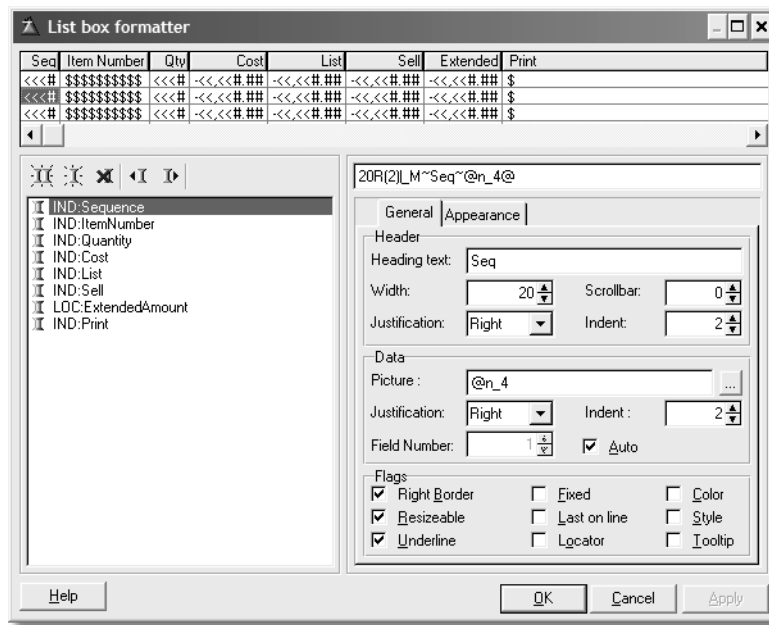
This section covers some of the common tasks in browse procedures.

### ○ Changing the appearance of a list control

One common task is adding a green bar effect. This is where each row of the list is alternating colors. Another could be calculating a local variable's contents.

What do these two tasks have in common? In addition, what is common about list controls? Examine the list formatter:

Figure 4-8:



One common task is to add local variables to a list control. In the example, `LOC:ExtendedPrice` is a variable used to calculate the extended price of an item. Since this is a local variable, you can make two assumptions:

- 1) It will not get any data by itself. You must put the data there.
- 2) You will need to do something for each row of the list control. In addition, you have no idea how the list control populates itself so the local variable can be populated at runtime.

The first thing you must determine is how does ABC populate any list control? To find out, you must look at the generated code.

```
Queue:Browse    QUEUE
CUS:Name        LIKE(CUS:Name)
CUS:Number      LIKE(CUS:Number)
CUS:Phone       LIKE(CUS:Phone)
CUS:Contact     LIKE(CUS:Contact)
CUS:CustomerId  LIKE(CUS:CustomerId)
Mark            BYTE
ViewPosition    STRING(1024)
END
```

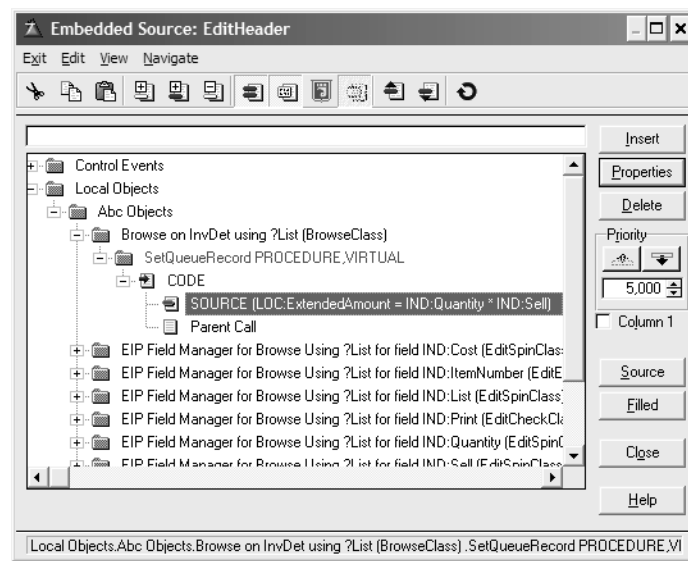
You can see that the template simply placed everything from the list in a `QUEUE` structure. One could surmise that `ABC` somehow places data in this `QUEUE` and this is what you see in the list. I should note at this point that the Clarion template chain creates queues in the same way.

Inspecting the Help confirms this:

The data items displayed in the `LIST` come from a `QUEUE` or `STRING` specified by the `FROM` attribute and are formatted by the parameters specified in the `FORMAT` attribute (which can include colors, icons, and tree control parameters).

Based on this, it is reasonable to guess that an embed point to add code to populate this local variable has something to do with queues.

Figure 4-9:



SetQueueRecord makes sense as list controls display values in QUEUE structures. In the above figure, you can see some source placed there to calculate the extended amount.

Figure 4-10:

Seq	Item Number	Qty	Cost	List	Sell	Extended	Print
10	1A2-9Z8	1	19.95	34.95	31.05	31.05	x
20	2B3-8Y7	1	11.33	29.95	25.49	25.49	x
30	3C4-7X6	1	23.95	38.77	35.00	35.00	x
40	4D5-6W5	1	23.95	38.88	36.29	36.29	x
50	5E6-5V4	1	23.95	38.88	36.29	36.29	x
60	6F7-4U3	1	23.95	38.88	33.29	33.29	x
70	1A2-9Z8	1	19.95	34.95	31.05	31.05	x
80	2B3-8Y7	1	11.33	29.95	25.49	25.49	x
90	3C4-7X6	1	23.95	38.77	35.00	35.00	x
100	4D5-6W5	1	23.95	38.88	36.29	36.29	x
110	5E6-5V4	1	23.95	38.88	36.29	36.29	x
120	6F7-4U3	1	23.95	38.88	33.29	33.29	x

### ○ What is ABC doing?

In order to answer this, one must look at the code used as a base to get the above behavior. What is the code behind SetQueueRecord?

```

BrowseClass.SetQueueRecord      PROCEDURE
CODE
  SELF.Fields.AssignLeftToRight
  SELF.ListQueue.SetViewPosition(POSITION(SELF.View))

```

That does not appear to be doing much, two lines of code. You know that SELF is whatever the current object is (see earlier for the discussion on SELF).

Looking at the dot syntax here, there are two object names. This usually indicates the class is making use of another class. Open the INC file for the BrowseClass (ABBROWSE in libsrc) and you will see that Fields is a reference to the FieldsPairClass. In other words, this is an example of composition.

This method simply copies the left field to the right field as defined by the AssignLeftToRight method:

```
FieldPairsClass.AssignLeftToRight PROCEDURE
I UNSIGNED,AUTO
CODE
  LOOP I = 1 TO RECORDS(SELF.List)
    GET(SELF.List,I)
    SELF.List.Right = SELF.List.Left
    PUT(SELF.List)
  END
```

The templates take care of coding this for you as this code shows:

```
BrowseClass.SetQueueRecord          PROCEDURE
CODE
  SELF.Fields.AssignLeftToRight
  SELF.ListQueue.SetViewPosition(POSITION(SELF.View))
```

Think about this for a minute. There must be some mechanism for this particular browse to know how to populate the items in the QUEUE. It does via the `BrowseClass.AddField` method, which takes two parameters, the file field (or variable) and the displayed queue field. This way the browse maintains the link between the database and the display.

The calls to `AddField` happen in the setup for this browse procedure. Thus, you may expect to find it in the `ThisWindow.Init` method of the generated code.

Since the browse has references to the queue fields and the file fields, that first line of code in `SetQueueRecord` is the same as coding this:

```
InvDetList.Q.IND:Sequence           = IND:Sequence
InvDetList.Q.IND:ItemNumber          = IND:ItemNumber
InvDetList.Q.IND:Quantity            = IND:Quantity
InvDetList.Q.IND:Cost                = IND:Cost
InvDetList.Q.IND:List                = IND:List
InvDetList.Q.IND:Sell                = IND:Sell
InvDetList.Q.LOC:ExtendedAmount      = LOC:ExtendedAmount
InvDetList.Q.IND:Print               = IND:Print
InvDetList.Q.IND:InvDetId            = IND:InvDetId
InvDetList.Q.IND:InvHdrId            = IND:InvHdrId
ADD(InvDetList.Q); !Followed by code to check for errors
```

The second line of code in the `SetQueueRecord` method sets the position value for the QUEUE structure. `ListQueue` is a reference to `BrowseQueue`. `BrowseQueue` is not, as the name would imply, another QUEUE structure. It is an interface. The `BrowseQueue` interface is a defined set of behaviors that relate to the VIEW and QUEUE that the LIST control uses.



The `SetViewPosition` method sets the `POSITION` of the `VIEW` based on the position parameter. The `VIEW` structure gets the data from the file, so it is important that the display of the `QUEUE` be in accord with the data in a `VIEW`.

What this means is that ABC is doing a task that happens in any file-based browse. In addition, all you are doing is adding one line of code to compute a local variable. You need do nothing else, as all you want is the computation of a local variable in addition to populating the list box with data.

Let ABC do the grunt work, you merely add the few lines of code needed to fulfill the design requirements of your application.

## ○ Event processing

All event processing in Clarion is conditional. In other words, when a certain event triggers, then some code executes. ABC has many event-related embeds. You can recognize these simply by name. The first part of the name is *Take*. This means to grab hold of, or inspect or process something.

For example, consider the phrase “take a break.” This mean you are on a break or interrupt an activity. Or take a look, meaning to inspect something. While that is not the name of an ABC event embed, it applies. You could think of it as inspecting an event. Thus, common event processing such as for `EVENT:NewSelection` is handled by a method called `TakeNewSelection`.

Each different type of procedure has these *Take* embeds that apply. The `TakeNewSelection` embed appears on any window with a list control. It can appear on a form if there is a list control on it. Alternatively, it could appear on any window with a drop list, drop combo or spin box control. If there is no `NewSelection` event being generated, then you do not see such an embed.

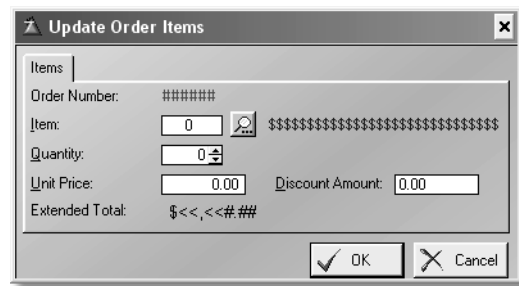
This means that the templates are smart enough to give you what you *might* need, but don't give you anything you *never* need.

You still have the usual, general control event processing and this makes it simple to determine if you need to use that embed. Again, the templates only give you the event embeds that apply to that control.

## ❑ When to use an ABC embed or a control specific embed

Look at the following window:

Figure 4-11:



You want the **Extended Total** to display any time the **Quantity** or **Unit Price** or **Discount** amounts change. The code would be as follows:

```
ITE:Total = (ITE:Quantity * ITE:Price) - ITE:DiscountAmount
```

This is a simple task. However, do you want to use one embed or three or four embeds to do this?

You could add this code to the `EVENT:Accepted` embed point for each control. That is three embeds, where is the fourth? The **Quantity** control is a spin box, so it could generate `EVENT:NewSelection` if the user presses one of the spin buttons. Therefore, you need a copy of the code in that embed too.

Okay, that's at least three embeds, and you can certainly call a `ROUTINE` to do the calculation. That eases the maintenance as you have only one spot to change the expression, if you ever need to.

How about using just one embed point? Before ABC, this was not possible. Looking at the window manager embed tree, you can scroll down the list until you find the `Take` embeds. They are, in alphabetical order, `TakeAccepted`, `TakeCloseEvent`, `TakeCompleted`, `TakeEvent`, `TakeFieldEvent`, `TakeNewSelection`, `TakeRejected`, `TakeSelected`, and `TakeWindowEvent`.

Out of these possible choices, you know one is correct. There are actually two embeds that would satisfy the requirement of one embed. Your only decision is which one is the best?

Whenever any control on a window triggers an event, any event, that is a field event. If you notice, there is an embed with that name! It does not mention which

event. In addition, since a field event is an event, there is an embed with that name too.

So, for example, when the user moves the window across the desktop, or resizes the window, those actions trigger events. Therefore, *TakeEvent* may not be the best choice. That leaves only *TakeFieldEvent*.

Nothing to it! That was simple, and that is about as complex as it gets. All ABC procedures have these event processing embeds. Which embeds appear when varies by the controls, thus the complexity of the procedure depends on the embeds presented to you.

On a final note, look at what the ABC templates coded for you (the actual source adds the comment “Method added to host embed code” to each line):

```
ThisWindow      CLASS(WindowManager)
Ask              PROCEDURE(),DERIVED
Init            PROCEDURE(),BYTE,PROC,DERIVED
Kill            PROCEDURE(),BYTE,PROC,DERIVED
Open            PROCEDURE(),DERIVED
Reset           PROCEDURE(BYTE Force=0),DERIVED
Run             PROCEDURE(),BYTE,PROC,DERIVED
Run             PROCEDURE(USHORT Number,BYTE Request),BYTE,|
                PROC,DERIVED
TakeAccepted    PROCEDURE(),BYTE,PROC,DERIVED
TakeFieldEvent  PROCEDURE(),BYTE,PROC,DERIVED
TakeSelected    PROCEDURE(),BYTE,PROC,DERIVED
                END
```

The templates generate the proper class definition for you, based on your embeds. This means that for any embed you use, it overrides the parent class' method. The code for the overridden method looks like this:

```
ThisWindow.TakeFieldEvent PROCEDURE
ReturnValue          BYTE,AUTO
Looped              BYTE

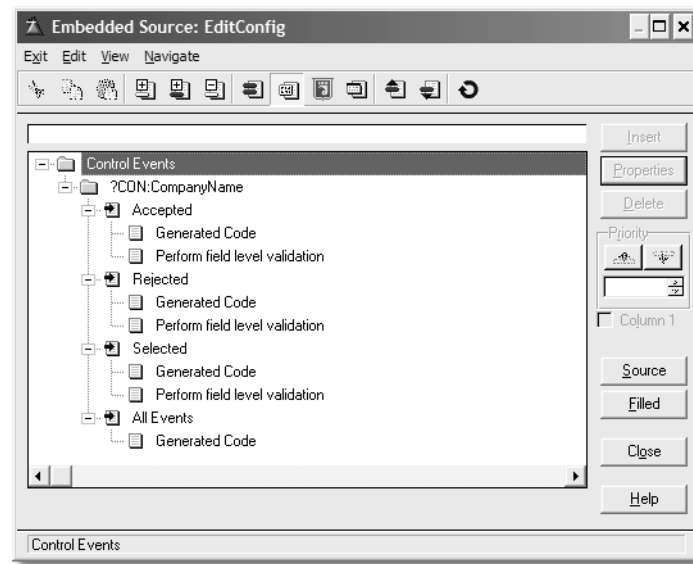
CODE
LOOP
    IF Looped
        RETURN Level:Notify
    ELSE
        Looped = 1
    END
    ITE:Total = (ITE:Quantity * ITE:Price) - ITE:DiscountAmount
    ReturnValue = PARENT.TakeFieldEvent()
    RETURN ReturnValue
END
ReturnValue = Level:Fatal
RETURN ReturnValue
```

That is really all there is to know about any of these `TakeEvent` embed points. In essence, if you wish to execute some code whenever a particular event happens, regardless of which control triggered it, then use those embeds.

If you do care which control triggers an event, then use that control's embeds. The best way is via the window formatter. Just right-click on the control, choose

**Embeds** and all embeds that are appropriate for that control show up. This is what an entry control's event embeds look like:

Figure 4-12:



To give you an idea of the dynamic nature of these embeds, remember, you only get event embed points when they are valid.

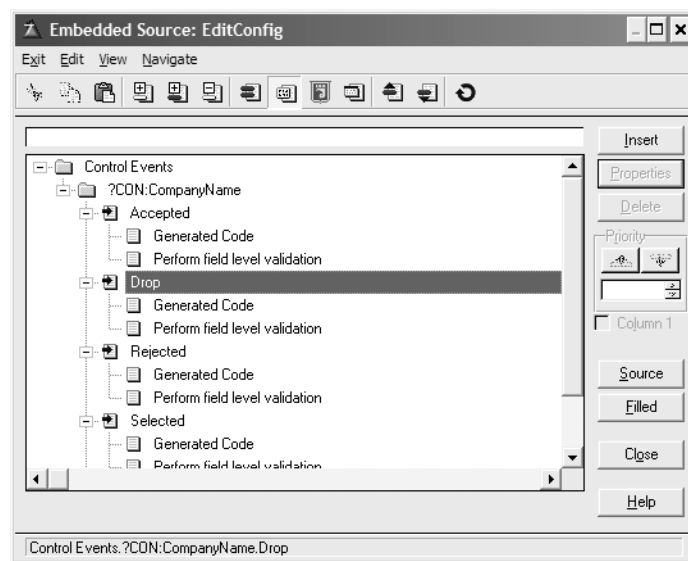
Suppose you gave this entry control a drop ID:

**Figure 4-13:**



Look what happens to the embed tree now:

**Figure 4-14:**



You can see that any events that are applicable for a control show in this list. Thus you can see that some event embeds appear only when you activate a feature. By default, drop events are not processed until a control gets a drop ID.

## Working with ABC classes

---

From time to time, you may need to work with ABC classes directly, not just by creating embed code. In my travels as an instructor, I have found the single most misunderstood class to be the *ErrorClass*. The class itself is fine except for one small sticking point. Developers insist on knowing what error was detected.

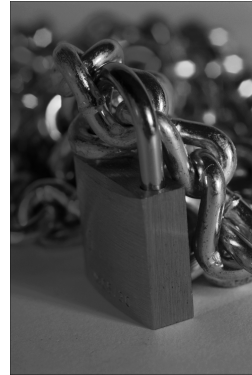
I believe this came about because many Clarion developers (including yours truly) were burned badly in the past when not checking for an error when returning from a statement. In addition, many of us are upbraided by our peers and even clients when the need to check for an error condition is missing from code. There are only so many times a developer is willing to go through this before the lesson finally sinks in. Once that happens, any developer is quite reluctant to let go of a good practice.

For years I've told Clarion developers to stop checking for errors themselves when using ABC. This statement earns me looks from others like I was from Mars. Let me repeat that. When using ABC, you do *not* need to check for errors. I did not say your application does not check for errors, it does. But if you code for it, you are checking for an error twice! When I say "checking for errors" I mean check for the usual errors, like "record mismatch", "record not found", "file not open" and the other common error conditions.

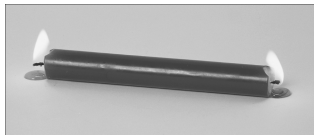
By common error conditions, I mean only the error codes Clarion knows about out of the box, such as errors dealing with files and accessing files. If Clarion knows about it, so does ABC. In addition, ABC knows which ones are fatal (like error 47) and which ones are not (error 35 or 33). And it can display a proper error message informing the user what happened. In addition, ABC displays all the usual notification messages, as when the record was changed by another station. This means that not only can ABC report errors, but it can determine the degree of severity.

That in essence is my total point about the `ErrorClass`. Do not code for error 47s or error 35s and other common data errors. ABC has your back covered on these. Therefore, it does not matter one bit that the parts of the `ErrorClass` where the actual error is known is a `PRIVATE` attribute.

I agree with the vast majority of Clarion developers that these properties should not be so restricted. I just don't agree with their reasons. A `PROTECTED` attribute is sufficient in case you wish to really change the logic of the `ErrorClass` (which seems to be a popular hobby in the Clarion community). Absent that, a method that returns the `ErrorCode` would also suffice.



I must confess I've discovered one design situation where the error condition must be known. It is in the case where your code must make a decision based on the error. Since ABC only returns the level of severity, this is not precise enough. For example, if upon opening a file, is an error 47 (record mismatch) detected? Perhaps when that happens you don't just want to exit the program, but instead you want to call a conversion procedure.



That is something the pre Clarion 6 ABC `ErrorClass` cannot do. In Clarion 6, there are two new methods that return the error code and error text.

The default behavior is to close the application as that is a fatal error. This is to protect the file's contents from corruption. I won't argue with that reasoning as that is quite valid. However, so is the design to run the conversion process, something useful that is at odds with ABC.

Is there a compromise or middle ground here? There must be, otherwise I would not bring this up. Remember, ABC is an OOP implementation. It is meant to be derived and overridden. First, an inspection of the code in the `ErrorClass` is in order.

### ❑ The `ErrorClass`

The `ErrorClass` is simple, considering what it does. There are several major functions the `ErrorClass` performs:

- Trapping errors.
- Reporting the errors if instructed to do so.



- Logging the errors to a file.
- Reporting the details of an error (which file and field triggered the error).
- How much error history to store if logging is active.
- Informing the other classes of the severity of the error.

There are several key methods one should know very well. The first is `ErrorClass.AddErrors`. This method does not add new bugs to your application! This method installs your custom error group (including translated error messages). This means that the `ErrorClass` can process your error conditions and levels of fatality. Use this method when you wish to add your own error messages for processing.

`ErrorClass.SetFatality` changes the level of fatality. It takes the error ID of the message you wish to change as the first parameter. The ID is part of the `ErrorEntry` structure. The second parameter is the new fatality level. This is usually an equate.

The `ErrorClass.Throw` method may be considered the heart of the `ErrorClass`. Look at this method's code:

```
ErrorClass.Throw  PROCEDURE(SHORT Id)
CODE
SELF.SetErrors
RETURN SELF.TakeError(Id)
```

There are just two lines of code, but what these two lines do is the point. For those who want to trap specific errors in their applications, `SetErrors` is the method of most interest:

```
ErrorClass.SetErrors  PROCEDURE
CODE
SELF.SaveErrorCode = ERRORCODE()
SELF.SaveError = CLIP(ERROR())
SELF.SaveFileErrorCode = CLIP(FILEERRORCODE())
SELF.SaveFileError = CLIP(FILEERROR())
```

`SetErrors` uses Clarion statements to store error information. It is a public method, meaning you may call it anytime you wish; but since it does not return anything, you may be thinking it is useless for your task of returning the error state. In that regard, it is. What I am pointing out are the properties used by this method to store the errors.

```
SaveError          CSTRING(255),PRIVATE  ! Clarion error message
SaveErrorCode      LONG,PRIVATE          ! Clarion error code
SaveFileError      CSTRING(255),PRIVATE  ! File error message
SaveFileErrorCode  CSTRING(255),PRIVATE  ! File error code
```

These properties store the error state. However, when some see the `PRIVATE` attribute, they get turned away. In one sense, it is a good idea to make these private to the class as you cannot really predict if or when a new error state may arise. It also implies that the class takes all responsibility for the proper use of any error states. There may be a good argument that these properties should be `PROTECTED` instead. While that has a ring of good sense to it, it also indicates that the developer may have to assume more responsibility for the proper use of these states. That is something that could defeat the design purpose of the class.

But instead of going off into hypothetical situations, lets get back to the original problem – getting the error states, even when `PRIVATE`.

Since these properties are indeed `PRIVATE`, all that means is that these properties are for the *exclusive* use of the class that defined them. Is there a method that uses these properties, and which might be used to get at those properties? Indeed there is! Take a look at the `SubString` method:

```
ErrorClass.SubsString      PROCEDURE
BuildString      CSTRING(2000)
ErrorPos      USHORT,AUTO
CODE
  BuildString = SELF.Errors.Message
  Replace('%File',SELF.FileName,BuildString)
  Replace('%ErrorCode',SELF.SaveErrorCode,BuildString)
  IF SELF.SaveErrorCode = 90
    Replace('%ErrorText',Self.SaveFileError & |
      ' (' & Self.SaveFileErrorCode & ')',BuildString)
  ELSE
    Replace('%ErrorText',Self.SaveError & |
      ' (' & Self.SaveErrorCode & ')',BuildString)
  END
  Replace('%Error',SELF.SaveError,BuildString)
  Replace('%FileErrorCode',SELF.SaveFileErrorCode,BuildString)
  Replace('%FileError',SELF.SaveFileError,BuildString)
  Replace('%Message',SELF.MessageText,BuildString)
  Replace('%Field',SELF.FieldName,BuildString)
  Replace('%Procedure',SELF.GetProcedureName(),BuildString)
  Replace('%Category',SELF.Errors.Category,BuildString)
  IF INSTRING('%Previous',BuildString,1,1)
    ErrorPos = POINTER(SELF.Errors)
    IF SELF.SetId(SELF.Errors.Id,ErrorPos-1)
      Replace('%Previous','',BuildString)
    ELSE
      Replace('%Previous',SELF.Errors.Message,BuildString)
    END
  GET(SELF.Errors,ErrorPos)
END
RETURN BuildString
```

The Replace procedure is not part of this class. It is a declared procedure in the MAP statement in *ABERROR.CLW*. ErrorClass.SubsString returns the error number and the error message. So this means that all you have to do is grab the string and parse it? Not exactly. This method is PROTECTED. This means that it is not open to anyone calling this method.

But wait, you are not turned away. All this means is that you may call the method from a class derived from the ErrorClass. And if you think about this from an encapsulation point of view, it makes perfect sense. So to make use of this method you need to create your own class derived from ErrorClass, and then tell ABC to use that class instead of ErrorClass.

### ❑ Deriving from ErrorClass

Suppose you were to code a new class, let's call it ErrorMgr. Here is the CLASS definition in a new source file, ErrorMgr.inc:

```
!ABCIncludeFile(EM)

INCLUDE('aberror.inc'),ONCE

ErrorMgr    CLASS(ErrorClass),TYPE,|
              MODULE('ErrorMgr.clw'),|
              LINK('ErrorMgr.clw','_ABCLinkMode_'), |
              DLL(_ABCDLLMode_)
GetErrStr    PROCEDURE, STRING,VIRTUAL
GetErrNum    PROCEDURE, LONG, VIRTUAL
END
```

There you have it, a simple derived error manager. Of course, the code for the methods in ErrorMgr.clw now follows.

```
MEMBER

MAP
END

INCLUDE('ErrorMgr.inc'),ONCE

ErrorMgr.GetErrStr    PROCEDURE
CODE
RETURN SELF.SubsString()    !Return entire error string
```

```

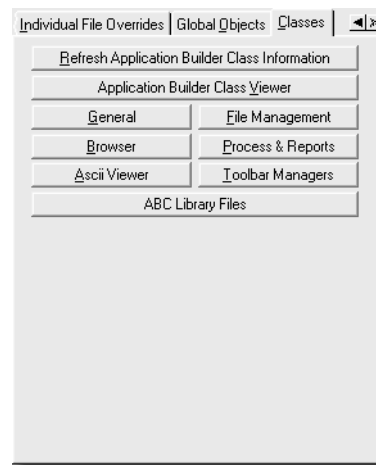
ErrorMgr.GetErrNum      PROCEDURE
CurrentString    CSTRING(2000)
CurPos          LONG
EndPos           LONG
RetVal           LONG

CODE
CLEAR(RetVal)
CurrentString = SELF.SubsString()
CurPos = INSTRING('(',CurrentString,1,1)
IF CurPos
    CurPos += 1
    EndPos = INSTRING(')',CurrentString,1,CurPos)
    IF EndPos
        EndPos -= 1
        RetVal = CurrentString [ CurPos : EndPos ]
    END
RETURN RetVal

```

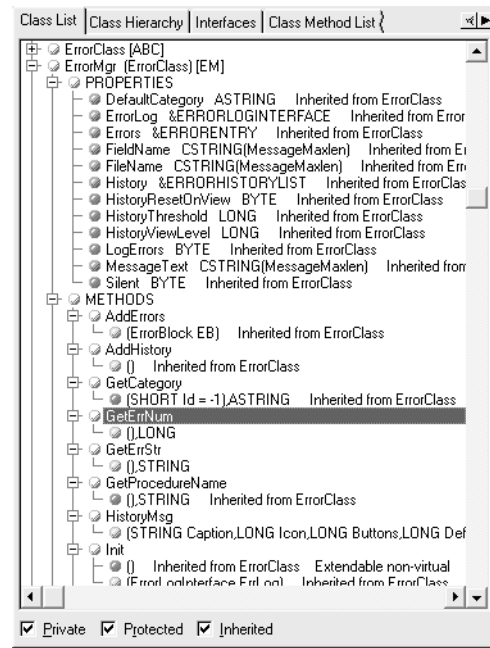
To use this class, both files must be saved in libsrc. Either start Clarion or press **Refresh Application Builder Class Information** in any class dialog to load the class in the reader.

Figure 4-15:



If you wish to see this class in the reader, press **Application Builder Class Viewer**. The ErrorMgr class should look like this:

Figure 4-16:

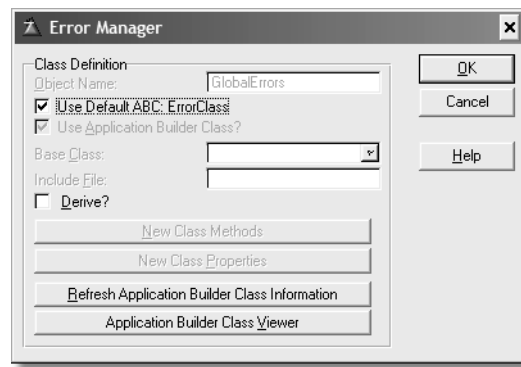


You can see the two methods in this class as well as all the properties and method inherited from the ErrorClass. Right-click on the list and choose **Show Symbol Key** for what the colors mean.

## ❑ Installing your Custom Classes

But how does one use this class in an application? The ABC template designers considered this. Choose the **Global Objects** tab. Press **Error Manager**. The following dialog displays:

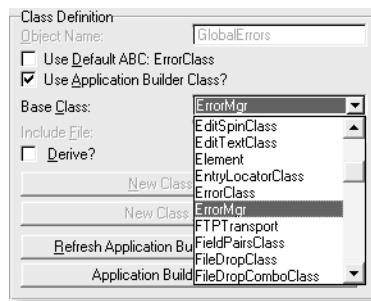
Figure 4-17:



You can see that the default error manager is `GlobalErrors`. This derives from `ErrorClass` and is the template default and is thus disabled. Uncheck the **Use Default ABC ErrorClass** box. The **Use Application Builder Class** box enables as does the **Base Class** drop list.

Select `ErrorMgr` from the drop list. Press **OK** until you are back at the application tree.

Figure 4-18:



This becomes the new class that forms the `GlobalErrors` class, which is local to your application. More importantly, any code, which includes source embeds

already present, does not have to change as the class instance (that is, the class object) name used in your application remains unchanged.

### ❑ Using Custom Classes in an Application

The next step is to actually use the custom class. As a reminder, the original problem was that the design needed to know if an error 47 (record mismatch) occurred while trying to open a file. This is normally a fatal error causing the application to shut down. However, the design needs to know if this happened rather than shutdown the application, but instead call a conversion process to handle this situation. This means some embedded code is required.

Depending on the design, the code could be placed anywhere in the application. However, since error 47s occurred when trying to open a file, the best place to add the code needed is whenever the application attempts to open a file. Where are files usually opened? Keep in mind that the `LazyOpen` flag (defer opening files) may or may not be in use, thus the placement of the embed must consider this. Also, you may not need this logic for every file in your dictionary.

A good choice executing the conversion code is when your application opens the file. A good early spot is the application frame. Lets say yo want to be able to convert the `Customer` file. In the application frame procedure's embed tree, simply walk down the tree until you find the `WindowManager`. You will see there is an `Init` method.

So the embed in the `WindowManager.Init` method becomes something like this (the actual code will vary depending on the file in use and where the code is placed):



```
!Drop severity level
GlobalErrors.SetFatality(MSG:OpenFailed, LEVEL:Notify)
GlobalErrors.Silent = True      !Make the class mute about errors
IF Access:Customer.TryOpen()    !Attempt to open a file
    IF GlobalErrors.GetErrorNum = 47 !Call method to test for 47
        RunConversionProcess      !If true, run a convertor procedure
    END
ELSE
    IF Access:Customer.UseFile() !If LazyOpen turned on
        IF GlobalErrors.GetErrorNum = 47 !Do the same test
            RunConversionProcess      !and run the converter if needed
        END
    END
END
!Restore the fatality level
GlobalErrors.SetFatality(MSG:OpenFailed, LEVEL:Fatal)
GlobalErrors.Silent = False     !Give it back its voice
Access:Customer.Close          !Close the file
```

See the example application, Error47.app in the code folder.

You can place this logic anywhere it makes sense. The first thing you must do is tell the `ErrorClass` that the open failed condition is no longer fatal, by dropping the severity level to notify. Also, mute the message handling by setting the `Silent` property to `True`. Then try to open the file. If there is an error, the `TryOpen` method will detect it and tell the `ErrorClass` which error.

Keep in mind that the object currently in use is really the child class, not the default error handler. This means your `GetErrorNum` method is a valid call as well as the normal `ErrorClass` method calls. And more importantly, this approach requires no changes to existing code generated by the template. And you did not need to edit the `ErrorClass`. That is the power of derivation and inheritance!

On a final note, if you install the next version of Clarion, your code does not break as you have not changed the ABC classes, only their behavior. A future version of the `ErrorClass` may render this requirement obsolete, but your derived class will still work!

---

**NOTE** In Clarion 6, there are two new methods that do return the error code and error message.

---

This is why I always frown when someone insists on editing the shipping classes. The next time they install a patch or newer version, they need to make the same

edits again. One may get clever to minimize any impact of editing ABC classes, but it is still extra work and you really do not have to bother.

# Chapter 5

---

## CODING OBJECTS

This chapter covers coding objects. In previous chapters I discussed OOP theory and how ABC does OOP. Now it's time to take generated legacy style (procedural) code and convert it into a class.

Generated legacy code (sometimes hand code) often results in a lot of code duplication. Often this is a great opportunity to take that code and make it into a class in its own right. The goal of this chapter is simply to show how to convert working code to a class. Initially, such a class won't have any template support. That subject is covered in the next chapter.

## Design Considerations

---

The first thing you should have in mind when writing a new class is a simple question. Ask yourself, "Could I *reuse* this code in other applications?" If you answered in the affirmative, then you have a valid reason. Another good reason would be if you find yourself using routines, or a ROUTINE that is repeated throughout your application, but with minor changes to each.

The next consideration is whether you have an example of working code. If you do, then life is easy. There really is not much to change in this aspect. If you do not have an example of working code, this just means that you should pay more attention to the design. This chapter uses a working example that any Clarion developer should already have.

This example is the Invoice example application that ships with Clarion. Load this application and open the window formatter for the procedure `BrowseAllOrders`. If you right-click on the list box, then choose **List box Format...** you see the window in Figure 5.1, and if you click on the **Appearance** tab you will see Figure 5.2:

Figure 5-1:

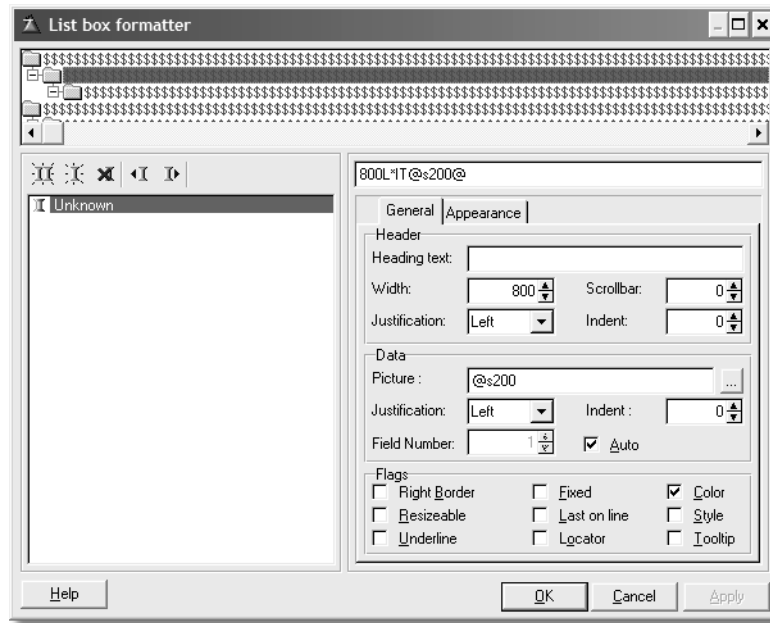
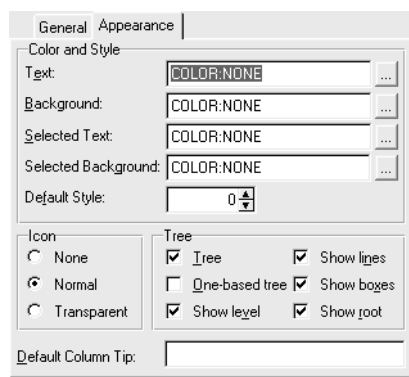
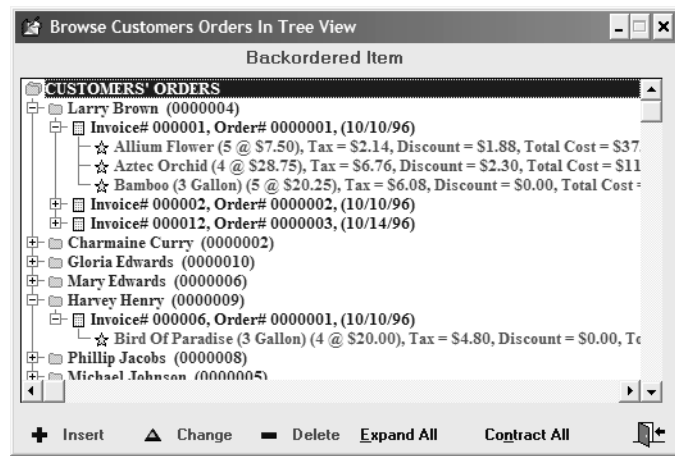


Figure 5-2:



These two views are enough to start with. The tree list displays colors and icons. At runtime, this is what you see:

Figure 5-3:



This is a functional tree list, and each item in the list has edit procedures attached. The code generated by the **Relation Tree** control template is written into routines for each file in the tree list. By that I mean they are the same routines, with just minor differences. Thus, this is a good candidate for a new class. (As an aside, Clarion as shipped does not have a relation tree *object*. This includes ABC, and it is almost identical to the one produced by the Clarion template chain.)

The goal is simple – create a class that has the same functionality as this one.

## ABC Relation Tree vs. Relational Object Tree

---

The ABC and Clarion/legacy versions of the relation tree both work, but they also both rely on the templates to manage all of the code. The only real differences are that the ABC version makes some ABC method calls to open files, get the data, etc. Both template chains use routines heavily, and depending on the files in the tree, produce similarly named routines.

If you have to make any changes to how the relation tree functions, you don't want all of that code. The reason is that such code requires you to be a template developer as well as a Clarion developer. That tells you right away that there is little or no code re-use, which one of the main goals of OOP.

When you start thinking of how to write better and tighter code, you need to think in abstract concepts.

Any class should work equally well in hand-coded projects, not just template driven apps. The only difference is that hand coding developers must handle all the declarations and instantiation themselves. Other than that, there should never be any significant differences between a hand coded project and an application; the end result should be the same.

## Designs

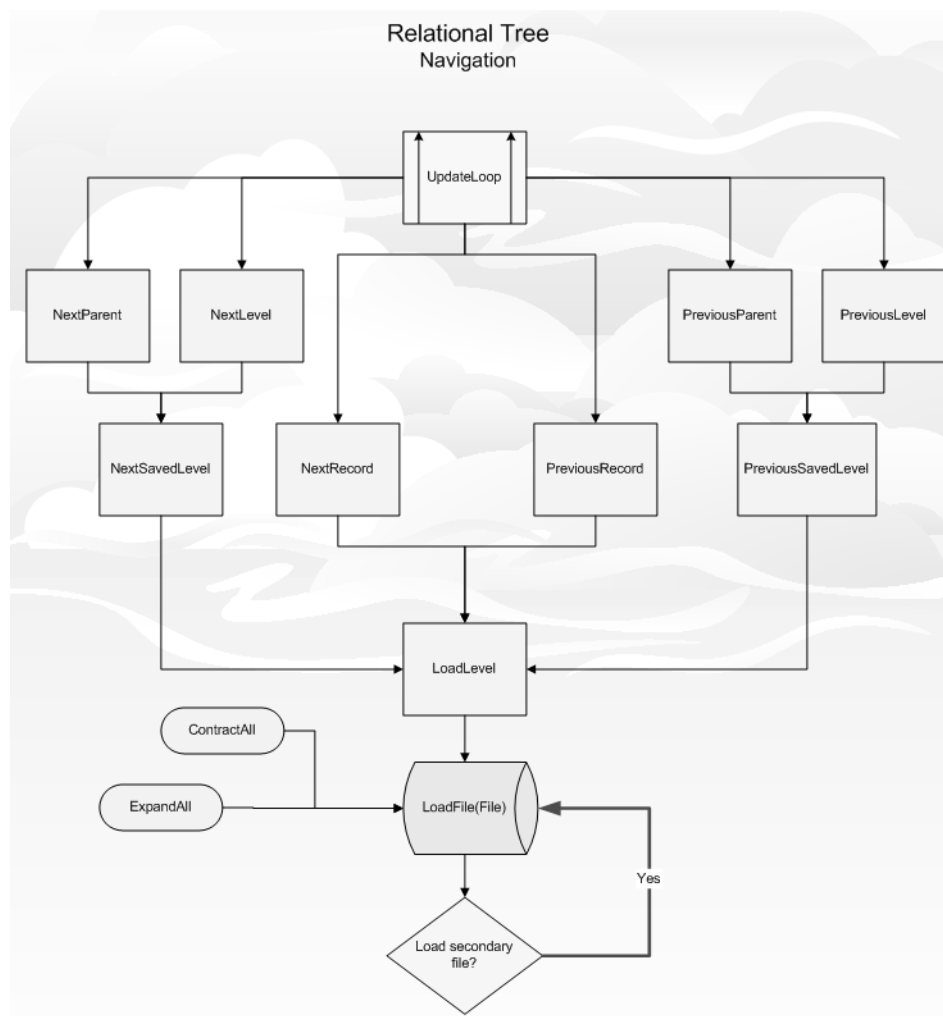
---

To convert an existing design to a new one, you must have a grasp of the old design. In the case of the relation tree, this can be broken down into two parts, editing and navigation. Therefore, I've made some simple charts of each of the major components.

## Navigation

This illustration shows how the navigation works currently.

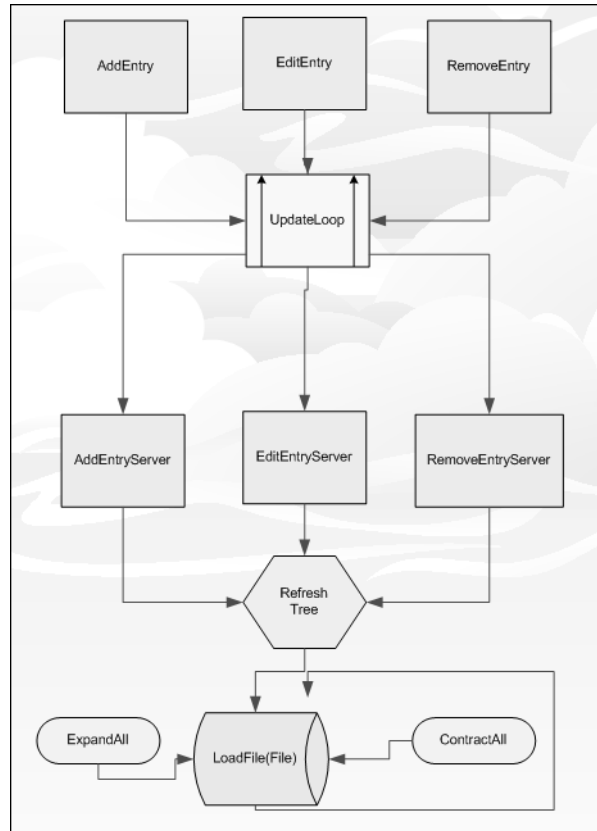
Figure 5-4:



## ■ Edits

This illustration shows the current edit functions.

Figure 5-5:



## A Fast Way to Code a Class

---

Where do you start with something like this? Let's examine the generated code. To ease this step, you should have `BrowseAllOrders` in its own module. That way you don't see anything that is "non-tree". The easiest way is to choose from the main menu, **Application | Repopulate Modules**. Choose **1 procedure per module** and then regenerate the source. Open up any editor and load the module with the



tree code in it. You can see what the module name is by either opening the procedure dialog or select the module view.

You'll need to open a new source file too. This is where the class code goes. Call the file `CBTree.inc` (CB meaning "Clarion Book") and save it in the `libsrc` directory. . You'll be switching between the open module and the `CBTree` file.

Add the following comment on line one:

```
!ABCIncludeFile(CB)
```

This comment is seen by the ABC class reader. The `(CB)` portion of the comment means that this class isn't linked in by any project that does not use this tree class. You may enter anything as a parameter. If you left the parameter off, then all projects would link in the class. I'll explain how this works later.

Let's examine the module code. Near the top you see a few data declarations and two queue structures. You'll concentrate on the `QUEUE` structures first.

Here are the generated `QUEUEs` from ABC:

```
Queue:RelTree      QUEUE,PRE()      ! Browsing Queue
REL1::Display      STRING(200)      ! Queue display string
REL1::NormalFG     LONG
REL1::NormalBG     LONG
REL1::SelectedFG   LONG
REL1::SelectedBG   LONG
REL1::Icon         SHORT
REL1::Level        LONG             ! Record level in the tree
REL1::Loaded       SHORT            ! Inferior level is loaded
REL1::Position     STRING(1024)     ! Record POSITION in VIEW
                                END
REL1::LoadedQueue  QUEUE,PRE()      ! Status Queue
REL1::LoadedLevel  LONG             ! Record level
REL1::LoadedPosition STRING(1024)   ! Record POSITION in VIEW
                                END
```

Simply copy them as-is and paste in the code in the `CBTree.inc` file.

Change the code so the source looks as follows in your `CBTree.inc` file:

```
!ABCIncludeFile(CB)

QRelTree      QUEUE,PRE(),TYPE ! Queue for Relation Tree
Display       STRING(200)      ! Queue display string
NormalFG      LONG             ! Normal foreground color
NormalBG      LONG             ! Normal background color
SelectedFG    LONG             ! Selected foreground color
SelectedBG    LONG             ! Selected background color
Icon          SHORT            ! Icon image
Level         LONG             ! Flag to indicate if this
                                ! node is expanded (positive)
                                ! or contracted (negative)
Loaded        SHORT            ! File level is loaded in
                                ! tree or not
Position      STRING(1024)     ! Record POSITION
END

LoadedQ       QUEUE,PRE(),TYPE ! Status Queue (is a file loaded?)
LoadedLevel   LONG             ! Record level
LoadedPosition STRING(1024)     ! Record POSITION
END
```

I'm not too wild about the use of the colon in data labels, except for field labels separating them from the file prefix. Thus, they are removed. I think this makes the code easier to read, as well as write. In addition, the labels of the structures could stand shortening.

I've also taken the liberty of adding more comments. In this book, the comments are wrapped for readability, but in the accompanying code, they are on one line. I know "documentation" is a four-letter word in most programmers' vocabularies, however I feel this is a necessary evil.

The `QRelTree` is a `QUEUE` that controls the display of the tree list. The presence of records with a level greater than the current level tells the list box that there is a child file loaded. This number can be positive or negative (expanded or collapsed). The `LoadedQ` keeps track of which child file is loaded, if any.

### ❑ The Start of the Relation Tree Class

It is now time to begin the `CLASS` structure. Let's label it `RelationTree`. Place it under the `QUEUE` definitions.

The code for the methods you will write later is in `CBTree.clw`. Thus, you need to add the `MODULE` attribute with `CBTree.clw` as a string literal parameter. In order to save yourself from adding this file to the project tree, add the `LINK` attribute, same text parameter and another parameter with the variable that flags how this is

linked to your application. To follow the ABC example, name this variable `_CBLinkMode_`. And lastly, add the DLL attribute with its flag parameter of `_CBDLLMode_`. To finish this, add `END` to the next line and align it with `CLASS`.

Your source should appear as follows (wrapped here to provide readability, but the entire `CLASS` portion of the declaration can be on one line if you prefer):

```
RelationTree  CLASS,MODULE('CBTree.clw'),|
               LINK('CBTree.clw',_CBLinkMode_),|
               DLL(_CBDLLMode_)
               END
```

As a general tip, when you code a structure, always code the `END` statement right away. Then move up one line and press enter to open a new line. This way, you won't forget to add the `END` statement later, which leads to compiler errors.

## ❑ Adding Properties

The next step is to add some properties. With existing code, this is not difficult. Go back to the generated code and you'll see some data definitions before and after the generated queues. Copy and paste these declarations before the `END` statement of the `CLASS`. In keeping with the naming style, remove the `REL1:` portion of the labels. I've added some comments where needed.

Your definition should now look like this:

```
RelationTree  CLASS,MODULE('CBTree.clw'),|
               LINK('CBTree.clw',_CBLinkMode_),|
               DLL(_CBDLLMode_)
SaveLevel     BYTE,AUTO      ! Current level
Action        LONG,AUTO     ! Edit action
CurrentLevel  LONG          ! Current loaded level
CurrentChoice LONG          ! Current highlighted record
NewItemLevel  LONG          ! Level for a new item
NewItemPosition STRING(1024) ! POSITION of new record
LoadAll       LONG          !Flag to load all levels
               END
```

The class is starting to take shape. However, there are at least two properties missing. What about the `QUEUE` structures? You cannot legally define a `QUEUE` structure in a `CLASS`, which is why they are declared outside of the `CLASS`.

After the last property declaration, open a new line and add the references to the two `QUEUE` declarations. There are other properties needed and they are listed here:

QRT	&QRelTree	! Reference to the QRelTree type
LDQ	&LoadedQ	! Reference to the LoadedQ type
LC	SIGNED,PROTECTED	! FEQ for list control
BaseFile	&FILE,PROTECTED	! Base file in passed VIEW
WinRef	&WINDOW,PROTECTED	! Reference to current window
VCRRequest	LONG(0)	! VCR action
InsertButton	SIGNED	! FEQ for Insert Button
ChangeButton	SIGNED	! FEQ for Change Button
DeleteButton	SIGNED	! FEQ for Delete Button

Some of these properties are PROTECTED. This is so that they can be accessed by derived classes, but not by any code outside of methods belonging to this class family. If only this base class should have access to them, then you would add the PRIVATE attribute to these properties.

If you really are not sure if a particular member should be PROTECTED, then use the PRIVATE attribute. Later as needs arise, you could change the attribute to PROTECTED or add new methods to this CLASS that return the values of these properties. The actual properties are still encapsulated regardless of what you do.

### ❑ Adding the Methods

The next step is adding the method declarations. Since the two QUEUES are declared as references, you cannot use these until a reference assignment is done. This is a good reason to use an automatic constructor. Therefore, add one, declared as follows:

```
Construct      PROCEDURE
```

There you have the first method. This method automatically executes when this class is instantiated, regardless of how it is instantiated.

Since a constructor now exists, a destructor would not be a bad idea. Make a new line and declare the destructor as follows:

```
Destruct      PROCEDURE
```

The destructor is now defined. Its purpose is to “undo” what the constructor does. This method automatically executes when this class is destroyed, regardless how it is destroyed.

The code for both of these methods follows shortly.

You may be wondering where you’ll get the code for the remaining methods. In the generated code, do a search for ROUTINE. You are interested only in the REL1::<whatever> routines.

For now, you are only interested in the routine labels. You will be interested in the code, but that step is for later. You are not interested in any method parameters, at least not yet. To get things started, simply copy the labels and paste them under the last method, which should be `Destruct`. Don't worry about the names for the moment, nor the prototypes.

You'll soon bump into the routines that are specific to a file, such as `REL1::Load:Customers` and `REL1::Format:Customers`. For simplicity's sake, just copy each of these as is. Do the same for the rest of these routines until you get to the next file, `REL1::Load:Orders`. You skip this and all other such labels with file names in them, since you only need one set of file-specific routines (which you will convert to work with any file). Continue to copy the routine names that do not refer to any file.

When you're done, the source should look like this:

```
RelationTree  CLASS,MODULE('CBTree.clw'),|
                LINK('CBTree.clw',_CBLinkMode_),|
                DLL(_CBDLLMode_)
SaveLevel     BYTE,AUTO          ! Current level
Action        LONG,AUTO          ! Edit action
CurrentLevel  LONG               ! Current loaded level
CurrentChoice LONG               ! Current highlighted record
NewItemLevel  LONG               ! Level for a new item
NewItemPosition STRING(1024)     ! POSITION of new record
LoadAll       LONG               ! Flag to load all levels
QRT           &QRelTree          ! Reference to the QRelTree type
LDQ           &LoadedQ           ! Reference to the LoadedQ type
LC            SIGNED,PROTECTED   ! FEQ for list control
BaseFile      &FILE,PROTECTED    ! Base file in passed VIEW
WinRef        &WINDOW,PROTECTED ! Reference to current window
VCRRequest    LONG(0)           ! VCR action
InsertButton  SIGNED             ! FEQ for Insert Button
ChangeButton  SIGNED             ! FEQ for Change Button
DeleteButton  SIGNED             ! FEQ for Delete Button
REL1::NextParent
REL1::PreviousParent
REL1::NextLevel
REL1::NextSavedLevel
REL1::PreviousSavedLevel
REL1::PreviousLevel
REL1::NextRecord
REL1::PreviousRecord
REL1::AssignButtons
REL1::Load:Customers
REL1::Format:Customers
REL1::LoadLevel
REL1::UnloadLevel
REL1::AddEntry
REL1::EditEntry
REL1::RemoveEntry
REL1::UpdateLoop
REL1::AddEntryServer
REL1::EditEntryServer
REL1::RemoveEntryServer
REL1::RefreshTree
REL1::ContractAll
REL1::ExpandAll
                END
```

The next step is to remove the REL1:: portion of each method label, using search and replace or whatever other method you like (some editors, though not the

Clarion editor, let you do column deletes. After that step, ensure each method has the PROCEDURE prototype.

The source should now appear as follows:

```

RelationTree  CLASS,MODULE('CBTree.clw'),|
                LINK('CBTree.clw',_CBLinkMode_),|
                DLL(_CBDLLMode_)
SaveLevel     BYTE,AUTO           ! Current level
Action        LONG,AUTO           ! Edit action
CurrentLevel  LONG                ! Current loaded level
CurrentChoice LONG                ! Current highlighted record
NewItemLevel  LONG                ! Level for a new item
NewItemPosition STRING(1024)      ! POSITION of new record
LoadAll       LONG                ! Flag to load all levels
QRT           &QRelTree           ! Reference to QRelTree type
LDQ           &LoadedQ            ! Reference to the LoadedQ type
LC            SIGNED,PROTECTED    ! FEQ for list control
BaseFile      &FILE,PROTECTED     ! Base file in passed VIEW
WinRef        &WINDOW,PROTECTED  ! Reference to current window
VCRRequest    LONG(0)            ! VCR action
InsertButton  SIGNED              ! FEQ for Insert Button
ChangeButton  SIGNED              ! FEQ for Change Button
DeleteButton  SIGNED              ! FEQ for Delete Button
NextParent    PROCEDURE
PreviousParent PROCEDURE
NextLevel     PROCEDURE
NextSavedLevel PROCEDURE
PreviousSavedLevel PROCEDURE
PreviousLevel  PROCEDURE
NextRecord    PROCEDURE
PreviousRecord PROCEDURE
AssignButtons PROCEDURE
Load:Customers PROCEDURE
Format:Customers PROCEDURE
LoadLevel     PROCEDURE
UnloadLevel   PROCEDURE
AddEntry      PROCEDURE
EditEntry     PROCEDURE
RemoveEntry   PROCEDURE
UpdateLoop    PROCEDURE
AddEntryServer PROCEDURE
EditEntryServer PROCEDURE

```

```
RemoveEntryServer  PROCEDURE
RefreshTree        PROCEDURE
ContractAll        PROCEDURE
ExpandAll          PROCEDURE
END
```

The class is taking on more form now. There are two method names that still need changing. These are the ones labeled `Load:Customers` and `Format:Customers`. Set these aside for now as these two methods will require some thought. More on that later, but save your work so far.

### ❑ The Code for the Methods

So far, the only thing you've done is define the class. There are methods defined, yet no code yet exists for these methods. At this stage, this class is not compilable. With your handy-dandy editor of choice, create a new file named `CBTree.clw` and ensure it is saved in `libsrc`.

At the top of the file, ensure you write your `MEMBER` statement. It must be empty (no parameters) as this code is not tied to any project. Since this is actually a source file, be sure to add the `MAP` `END` statements. If you neglect this step, you'll get compiler errors on valid Clarion code. `MAP` `END` ensures Clarion's library of commands are seen by the compiler.

```
MEMBER

MAP
END
```

Next, you need to code the `INCLUDE` statement:

```
INCLUDE('CBTree.inc'),ONCE
```

The `ONCE` attribute is really not required as this is the only time the header definitions are included. However, using the `ONCE` attribute never hurts, and it's a good habit to have.

### ❑ Writing the code

This is the fun part, actually writing the code for the methods declared in the `CLASS` structure. No matter how big a `CLASS` may be, writing the code for each method often takes only a few seconds to a few minutes. Here are the steps for the fastest way I know to code methods for a class.



- 1) Go back to the `INC` file and highlight all of the methods declared in it.  
Copy the labels (names) of the methods and the prototypes.
- 2) Copy them to the clipboard.
- 3) Switch to the `CLW` file.
- 4) Paste the declarations at the bottom of the source.
- 5) Open the search and replace feature of your editor. This works best if you have one space in column one and your editor can do column search and replace. Search for the single space and replace it with `RelationTree`. (Don't forget the ending period!)

Your source should appear like this:

```
MEMBER

    MAP
    END

    INCLUDE('CBTree.inc'),ONCE

    RelationTree.NextParent          PROCEDURE
    RelationTree.PreviousParent       PROCEDURE
    RelationTree.NextLevel            PROCEDURE
    RelationTree.NextSavedLevel       PROCEDURE
    RelationTree.PreviousSavedLevel   PROCEDURE
    RelationTree.PreviousLevel        PROCEDURE
    RelationTree.NextRecord           PROCEDURE
    RelationTree.PreviousRecord        PROCEDURE
    RelationTree.AssignButtons        PROCEDURE
    RelationTree.Load:Customers        PROCEDURE
    RelationTree.Format:Customers     PROCEDURE
    RelationTree.LoadLevel            PROCEDURE
    RelationTree.UnloadLevel          PROCEDURE
    RelationTree.AddEntry              PROCEDURE
    RelationTree.EditEntry             PROCEDURE
    RelationTree.RemoveEntry           PROCEDURE
    RelationTree.UpdateLoop            PROCEDURE
    RelationTree.AddEntryServer        PROCEDURE
    RelationTree.EditEntryServer       PROCEDURE
    RelationTree.RemoveEntryServer     PROCEDURE
    RelationTree.RefreshTree           PROCEDURE
    RelationTree.ContractAll           PROCEDURE
    RelationTree.ExpandAll             PROCEDURE
```

- 6)** Open a new line under the first method (the Construct method).
- 7)** Press SPACE twice and then enter CODE and then press enter.
- 8)** Copy the CODE line (with its CR/LF) and paste after each method listed.

When done, your source now should look similar to this:

```
MEMBER
```

```
    MAP  
    END
```

```
    INCLUDE('CBTree.inc'),ONCE
```

```
RelationTree.NextParent      PROCEDURE  
    CODE
```

```
RelationTree.PreviousParent  PROCEDURE  
    CODE
```

```
RelationTree.NextLevel       PROCEDURE  
    CODE
```

```
RelationTree.NextSavedLevel  PROCEDURE  
    CODE
```

```
RelationTree.PreviousSavedLevel PROCEDURE  
    CODE
```

```
RelationTree.PreviousLevel   PROCEDURE  
    CODE
```

```
RelationTree.NextRecord      PROCEDURE  
    CODE
```

```
RelationTree.PreviousRecord  PROCEDURE  
    CODE
```

```
RelationTree.AssignButtons   PROCEDURE  
    CODE
```

```
RelationTree.Load:Customers  PROCEDURE  
    CODE
```

```
RelationTree.Format:Customers PROCEDURE  
    CODE
```

```
RelationTree.LoadLevel       PROCEDURE  
    CODE
```

```
RelationTree.UnloadLevel     PROCEDURE  
    CODE
```

## Programming Objects In Clarion

---

RelationTree.AddEntry CODE	PROCEDURE
RelationTree.EditEntry CODE	PROCEDURE
RelationTree.RemoveEntry CODE	PROCEDURE
RelationTree.UpdateLoop CODE	PROCEDURE
RelationTree.AddEntryServer CODE	PROCEDURE
RelationTree.EditEntryServer CODE	PROCEDURE
RelationTree.RemoveEntryServer CODE	PROCEDURE
RelationTree.RefreshTree CODE	PROCEDURE
RelationTree.ContractAll CODE	PROCEDURE
RelationTree.ExpandAll CODE	PROCEDURE

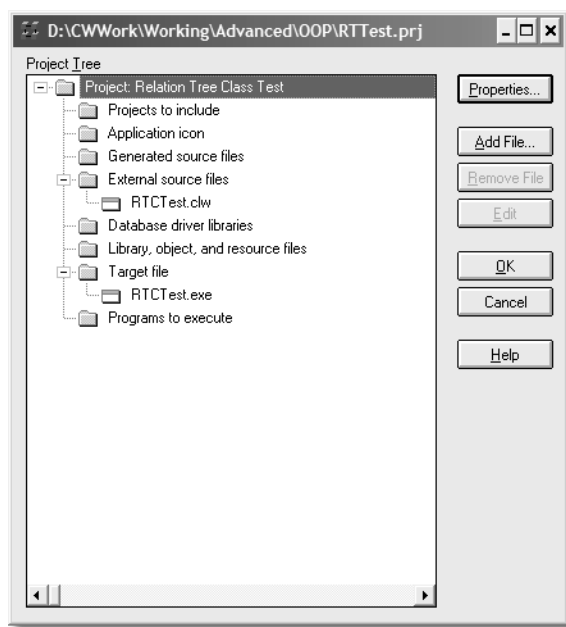
Be sure to save your work at this point. You now have a class that can compile clean. It still does not do anything (not yet). But you should let the compiler check your work at this stage.

### ❑ A New Project

Start a new project. You can do this by choosing **File | New | Project** from the menu, or use the pick list dialog and make a new project from there. It does not matter what you name the project, but I suggest you name it something meaningful to you, like `RTTest`. A description is optional, but I would suggest something like `Relation Tree Class Test`. Enter a name for the file, like `RTCTest.clw`. The EXE

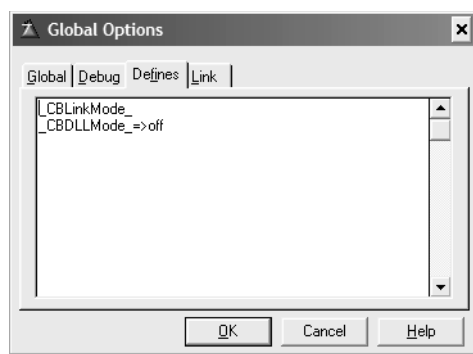
name is filled in for you. Press **OK** to save your choices and the project tree displays.

Figure 5-6:



Next you'll need to set some project variable definitions, which are used by the compiler and linker to correctly build your project. If a define is not explicitly set to => off, it is on. Press **Properties** and then choose the **Defines** tab. Be sure to add the following variables and their settings:

Figure 5-7:



Press **OK** when you are done and return to the project window. Highlight `RCTTest.clw` and press **Edit**. You should now have an empty source file. Add code to match the following figure:

```
PROGRAM

MAP
END

INCLUDE('CBTree.inc'),ONCE

RT  &RelationTree    !Reference to the RelationTree class

CODE
```

You can now press the compile button and it should make the EXE. The EXE won't do anything as there isn't any code to tell it what to do. This comes later. The test at this stage is that it compiles clean. If you did not get a clean compile, go back and review the previous steps or use the errors window to fix the problem. Usually, the problem is a simple typo.

### □ Summary

What you just did was a simple exercise that you can use to code a skeletal CLASS structure. There are two purposes to this exercise:

- 9) Getting you more familiar with a CLASS structure by easily hand coding it.
- 10) Using the labels of the code you wish to convert to a CLASS, thus increasing your familiarity with the existing code.

## Generic class

---

The main point to keep in mind when coding classes is that nothing is hard coded as far as the data it manipulates. Where the original source uses hard coded values, you will use variables and reference variables.

If you look at the generated code from the example `Invoice` application, there are many references to `?RelTree`, the field equate for the list control. What is needed is a way to tell this class which list control it should recognize.

This is what the `LC` property is for. `LC` means List Control. Since it really holds the field equate for the list, the data type is `SIGNED`. Finally, since only this class or derived classes should deal with this value, `LC` has the `PROTECTED` attribute.

---

**NOTE** It really does not matter what order properties and methods list themselves in a class structure. If you group them together, then you have an easier time maintaining the class.

---

Now that the class is taking shape, let's address the constructor and destructor.

### ❑ Automatic constructors and destructors

A *constructor* is a special method that executes when an object is instantiated, regardless of how it's instantiated. A *destructor* is a method automatically executed regardless of how an object is destroyed. You never need to call these yourself as these are called for you.

The `RelationTree` uses references to two `QUEUE` structures, and before you may use these references they must be instantiated. If you recall, both definitions have the `TYPE` attribute. It makes little sense to require a programmer to do the reference assignments for these structures before they use the object. It is also dangerous to forget to always do this. Let the *constructor* do the work, not the programmer.

The `Construct` method has these lines of code:

```
RelationTree.Construct      PROCEDURE
CODE                        !Automatic constructor
IF SELF.QRT &= NULL        !If no reference exists
  SELF.QRT &= NEW QRelTree  !Instantiate Data Queue
?  ASSERT(~SELF.QRT &= NULL, 'Cannot instantiate QRelTree') END
IF SELF.LDQ &= NULL        !If no reference exists
  SELF.LDQ &= NEW LoadedQ   !Instantiate Loaded Queue
?  ASSERT(~SELF.LDQ &= NULL, 'Cannot instantiate LoadedQ') END
```

The *destructor* simply undoes what the *constructor* did:

RelationTree.Destruct	PROCEDURE
CODE	!Automatic destructor
IF ~SELF.QRT &= NULL	!If property is not null
FREE(SELF.QRT)	!Free queue's resources
DISPOSE(SELF.QRT)	!and dispose of it
END	
IF ~SELF.LDQ &= NULL	!If property is not null
FREE(SELF.LDQ)	!Free queue's resources
DISPOSE(SELF.LDQ)	!and dispose of it
END	
IF ~SELF.WinRef &= NULL	!Reference to window still valid
SELF.WinRef &= NULL	!Remove the reference
END	

You now have two methods that automatically do the “grunt” work. If you notice in the `Destruct` method, code exists to dereference the other properties. However, the constructor does not assign these references. It can't. Constructors cannot take parameters to pass in the `Window`, `List` and `File` parameters. Thus another method that can take these parameters handles this.

Theoretically, the constructor could call a method to do the additional reference assignments, but again, how can it know about the current window or file structure? It can't. This is where the `Init` method comes in.

### ■ About Init and Kill

I've heard discussions arguing in favor of and against using methods named `Init` and `Kill`. In essence, they are considered alternate construct and destruct methods. But because they are never automatically called, you must call them explicitly.

### ○ In Favor of Init and Kill

Unlike the automatic constructor and destructor methods, `Init` and `Kill` may take parameters and return values. And they may be called anytime your design requires it. They could even call `Construct` and `Destruct`, if your design calls for it. If this is the case, then investigate the use of `REPLACE` as an attribute for the constructor and destructor (covered in Chapter 1). In this case, the logical use is to restart an object. One could argue in favor of even using this technique, but that is drifting beyond the scope of this book.

Another argument in favor of (and sometimes against) the `Init` and `Kill` methods is the argument of example – you see these methods throughout ABC. Whether the decision is for or against depends on how you feel about ABC.



`Init` and `Kill`, by themselves are nothing special, they are not reserved words nor do they have any special meaning in Clarion.

### ○ Not in Favor of `Init` and `Kill`

The argument against `Init` and `Kill` goes like this: You have the automatic constructor and destructor. Why have another set of methods that at first glance appear to mimic `Construct` and `Destruct`? If you use the automatic methods, why should you even bother with `Init` and `Kill` methods? `Init` and `Kill` are not automatically called, you must write the code to call them.

In most cases, I would say you won't need explicit constructors. But it would depend on your design and of course what code discipline you follow. That is also outside the scope of this book and I will defer such decisions to the developer.

My opinion on this matter is that I have no qualms about using `Init` or `Kill` whether or not I have automatic methods (`Construct` and `Destruct`). If there is no good reason to use them I don't, if there is a good reason to use them, I do. This varies depending on the design.

### □ How much is that List in the Window?

The next method you need to look at is one that does not yet exist. Remember, you must be able to use this class in any given application. Thus, nothing is hard coded. So what major design feature is missing at this point? The actual list control itself!

It gets worse. Where does a list control live? A window. Without a window, a list control cannot exist. So how does one write code for a window that does not yet exist? And by extension, a list control that cannot yet exist? Just how does one code for this?

The answer is quite simple and it requires a new method. I suppose I could name it `Init`, but I think `InitTree` is better. You now need to add this method. Open a new line in the class definition and give this method the label `InitTree`, *as follows*:

```
InitTree PROCEDURE (*WINDOW xWin, SIGNED xList,|
    *FILE xFile),LONG,PROC,VIRTUAL
```

I'll explain why this is a virtual method shortly.

The next step is to ensure the code for the method actually exists, or the compiler will complain. The easiest way is to copy the entire prototype line from the `INC` file and switch to the `CLW` file. You can paste the code anywhere amongst the other

method code. To make this simple, you may scroll to the bottom of the file and paste it there. Or make it the first method shown. Another standard is the same order as declared. Or alphabetical. In other words, it really does not matter where you place the code (just don't paste it inside another method!).

Once you have pasted the code where you want, remove the `LONG`, `PROC`, and `VIRTUAL` attributes. The only thing you need in the method code is the label of the method and its prototype. Attributes and return values are not legal in the method code section. Open a new line and add the word `CODE`.

Fill in the rest of the code so your method appears as follows:

```
RelationTree.InitTree    PROCEDURE(*WINDOW xWin, |
                          SIGNED xList,|
                          *VIEW xFile)
RetVal                  LONG,AUTO

CODE
RetVal = False
SELF.WinRef &= xWin      !Reference to window
IF SELF.WinRef &= NULL
    RetVal = True
    RETURN RetVal
END
SELF.LC = xList           !Store the FEQ
SELF.BaseFile &= xFile    !Set primary file
IF SELF.BaseFile &= NULL
    RetVal = True
    RETURN RetVal
END
SELF.WinRef $ SELF.LC{PROP:From} = SELF.QRT !list gets data from?
SELF.RefreshTree()        !and populate it with data
RETURN RetVal
```

Notice the use of `SELF`. This means “whatever the current object is”.

The `WinRef` property is a reference to the current window. After the reference assignment, it is tested for “nullness.” If for some reason it is `NULL`, the method returns to the caller with a return value.

The `BaseFile` property is a reference to the passed primary file. This is also tested for “nullness” and if it is `NULL`, the code quits this method and returns a value.

The list control field `equat` is assigned to the `LC` property. The `QRT` property (assigned by the `Construct` method) is where the list control gets its data. Next comes a call to `RefreshTree()` (discussed later).

The final line of code returns a value, in this case, a zero or false value, meaning no errors.

I should point out that the variable, `RetVal`, declared in this method is local to this method. That means it is for the use of this method only, despite any rules of derivation. Variables declared in this fashion are considered implicitly `PRIVATE`.

Now is the time to bring up why this is a `VIRTUAL` method. This method expects to be overridden. When you derive from this class, you may want to override or extend this method. In this case simply write your code. At the appropriate time in the execution sequence, you have the option of calling `PARENT.InitTree` in your derived method. Or don't call the `Parent` method if you want to completely replace the method's functionality.

Remember, this class calls its derived children *instead* of its own methods to do work. This is the power of virtual methods. You have the choice of extending a method by calling the parent in your overridden method, or not calling it at all if you are replacing *all* functionality of the parent method. It is up to you.

Of course, having a template wrapper to write the derived virtual method code for you, according to the options selected in the template, is the best way to proceed. This class design allows for such use, and a template wrapper is the subject of the next chapter.

You could copy more code here from the generated source, but it would limit what you could do with this class. For example, you could add this code:

```
SELF.LC{PROP:At,1} = xW{PROP:At,1} * 0.05 !horz left corner
SELF.LC{PROP:At,2} = xW{PROP:At,2} * 0.10 !vert left corner.
SELF.LC{PROP:At,3} = xW{PROP:At,3} * 0.50 !Width of list control
                                           ! as a percent of
                                           !window area.
SELF.LC{PROP:At,4} = xW{PROP:At,4} * 0.75 !Height of list control
                                           !as a percent of
                                           !window area.
SELF.LC{PROP:Format} = '1000L*IT@s200@' !Format string to tell
                                           ! it is a tree with
                                           ! color & icon support.

SELF.LC{PROP:Alrt,255} = CtrlRight      !Alert keystrokes.
SELF.LC{PROP:Alrt,255} = CtrlLeft
SELF.LC{PROP:Alrt,255} = MouseLeft2
SELF.LC{PROP:VScroll} = True           !Vertical scroll bar
SELF.LC{PROP:HSCROLL} = True           !Horizontal scroll bar
SELF.LC{PROP:VCR} = True               !VCR navigation
SELF.LC{Prop:IconList,1} = '~file.ico' !List of icons
SELF.LC{Prop:IconList,2} = '~folder.ico'
SELF.LC{Prop:IconList,3} = '~invoice.ico'
SELF.LC{Prop:IconList,4} = '~wizpgdn.ico'
SELF.LC{Prop:IconList,5} = '~wizdown.ico'
SELF.LC{Prop:IconList,6} = '~star1.ico'
SELF.LC{Prop:Selected} = 1
```

However, you can see this would limit the list control in its placement, appearance and size. There is nothing wrong with this code per se, and it may indeed be fine for many list controls. There is nothing really wrong with doing this anyway, as any derived method may override these settings.

However, this is best done *in* a derived child class, with templates writing the above, based on what is known at the time. In other words, the derived class, not the base class, should create code that is specific to that particular list control.

The point is that you can think up many useful things to do with code, but this is a base class, so minimal functionality is always best. In this example, only the code that must always be there is present. The property statements above could be there, but none are required. This does not mean you may not add functionality like this later. All without losing any existing functionality.

You must resist the temptation to do too much. When coding base classes like this, only the irreducible minimum code should be in it. Even if this means a loss in functionality! I know how strange that sounds, but keep in mind the derived

class is where the missing functionality is. The more you derive, the more specific your code gets.

### ❑ Keep Method Code Simple as Possible

Let me illustrate the importance of keeping base class code simple. If you are writing a method that opens a `FILE`, then write code that opens a `FILE`. Easy enough, any Clarion developer knows how to do that.

```
MyFileClass.Open  PROCEDURE(*FILE MyFile)
CODE
OPEN(MyFile)
```

That is a no-brainer. Nice and simple. However, no Clarion developer would use only that code as it is dangerous. What if there is an error during the open? And what type of error? Not all errors with opening files are fatal. Opening a file gets complex in a hurry, doesn't it?

Many Clarion developers can write code that can test for and handle error conditions, but now you are no longer dealing with a method that opens files, as error checking and handling doesn't necessarily have anything to do with a method that opens a file. It would make more sense to call a method that does error checking for you. To strengthen this argument, any classes that trap and report errors must be able to handle other errors, not just from the limited number of errors that could arise from attempting to open a file. Thus, a call to a class that handles errors makes sense.

You still get your error checking, but you keep distinct functionality separate.

```
MyFileClass.Open  PROCEDURE(*FILE MyFile)
CODE
OPEN(MyFile)
MyErrorClass.ErrorCheck() !Was there an error?
```

The bottom line is you should always keep code painfully simple, even at the price of omitting code that *could* be useful. This useful code may be best suited to live in another class, perhaps derived, perhaps another base class (or derived from another base class). A nice bonus for you is that maintaining such code is easier as you do not have to wade through lines and lines of code unrelated to the task at hand. If you wish to do more with file management, then you know which class to go to. The same goes if you have a nifty idea about handling errors.

## The RelationTree Class Methods

---

The rest of this chapter takes you through the other methods in the class. Each method's purpose is discussed, followed by the code.

Following each method is a brief discussion of what the code does. The source code that comes with this book is fully commented, but the comments are omitted in the following pages.

### ❑ RefreshTree()

RefreshTree is a method that is called often in the class. Its main purpose is to load the tree with fresh data. You would call this method anytime you want or need to re-load the tree.

```
RelateTree.RefreshTree      PROCEDURE()  
CODE
```

This method is a VIRTUAL. What is a determining factor on making a method virtual or not? One good rule of thumb is if the method is used within the class, then it should probably be a virtual method. In other words, do other methods call this one?

Another reason is that this is a stub method, meaning there is no code in the base class. You define methods like this knowing there is almost always a derived method later. Since refreshing the list requires doing something with the local QUEUE and re-loading a FILE, this can't be known when you code the base class. However, you do have this data when a derived local class exists. Also, this fits well with template wrappers as the template will typically know how to code this method.

Another reason you may want to code a stub method is that the method is needed, but there is no way to code this without breaking the abstract rule, but other methods must call it (like Init).

There are always exceptions – I've never been able to come up with rules set in concrete. Therefore, the above rules are guidelines.

### ❑ LoadLevel()

This is a virtual stub method.

### ❑ **UnloadLevel()**

This is a virtual stub method.

### ❑ **NextParent()**

This method determines the next parent level of the tree:

```
RelationTree.NextParent      PROCEDURE()  
CODE  
IF ~SELF.WinRef{PROP:Active}  
    SELF.WinRef{PROP:Active} = True  
END  
GET(SELF.QRT,CHOICE(SELF.LC))  
IF ABS(SELF.QRT.Level) > 1  
    SELF.SaveLevel = ABS(SELF.QRT.Level) - 1  
    SELF.NextSavedLevel()  
END
```

If the window is not the active window, then the window containing the tree control is set to the active window. The code then gets a record from the QRT structure based on the current choice. If the level is greater than 1, take 1 away and save that value in the `SavedLevel` property. Then call the `NextSavedLevel` method.

### ❑ **PreviousParent()**

This method determines the previous parent level based on the current level:

```
RelationTree.PreviousParent  PROCEDURE()  
CODE  
IF ~SELF.WinRef{PROP:Active}  
    SELF.WinRef{PROP:Active} = True  
END  
GET(SELF.QRT,CHOICE(SELF.LC))  
IF ABS(SELF.QRT.Level) > 1  
    SELF.SaveLevel = ABS(SELF.QRT.Level) - 1  
    SELF.PreviousSavedLevel()  
END
```

First, the code determines if the window is active and if not makes it active. Next, it gets the record from QRT based on the current choice (highlighted row).

If the level is greater than 1, it subtracts one from whatever the current value is and puts it into the `SaveLevel` property. The `PreviousSavedLevel` method is called.

The significance of 1 means that the current level is not a root item. Subtracting 1 from the current level yields, by definition, a parent node.

### ❑ **NextLevel()**

This method finds the next level that is the same as the current one:

```
RelateTree.NextLevel          PROCEDURE
CODE
  IF ~SELF.WinRef{PROP:Active}
    SELF.WinRef{PROP:Active} = True
  END
  GET(SELF.QRT,CHOICE(SELF.LC))
  SELF.SaveLevel = ABS(SELF.QRT.Level)
  SELF.NextSavedLevel()
```

The code first determines if the window is active and if not makes it active. Next, it gets the record from QRT based on the current choice (highlighted row). It then puts the current level value into the SaveLevel property. The NextSavedLevel method is called.

### ❑ **PreviousLevel()**

This method finds the previous level that matches the current level:

```
RelateTree.PreviousLevel      PROCEDURE
CODE
  IF ~SELF.WinRef{PROP:Active}
    SELF.WinRef{PROP:Active} = True
  END
  GET(SELF.QRT,CHOICE(SELF.LC))
  SELF.SaveLevel = ABS(SELF.QRT.Level)
  SELF.PreviousSavedLevel
```

The code first determines if the window is active and if not makes it active. Next, it gets the record from QRT based on the current choice (highlighted row).

It then puts the current level value into the SaveLevel property. The PreviousSavedLevel method is called.

### ❑ **NextSavedLevel()**

This method determines the next level that matches the current level. It reads the QUEUE structure to determine this.



```
RelateTree.NextSavedLevel          PROCEDURE
SavePointer    LONG,AUTO
CODE
LOOP
  LOOP
    GET(SELF.QRT,POINTER(SELF.QRT) + 1)
    IF ERRORCODE()
      RETURN
    END
    WHILE ABS(SELF.QRT.Level) > SELF.SaveLevel
      IF ABS(SELF.QRT.Level) = SELF.SaveLevel
        SELECT(SELF.LC,POINTER(SELF.QRT))
        RETURN
      END
      SavePointer = POINTER(SELF.QRT)
      SELF.LC{PROPLIST:MouseDownRow} = SavePointer
      SELF.LoadLevel
      GET(SELF.QRT,SavePointer)
    END
```

There are two LOOP structures. The first or outer LOOP encompasses the entire method. The next, inner LOOP first checks to see if the Level value in the QUEUE is greater than the SaveLevel property value. This is done with the WHILE statement. If the Level value is greater than SaveLevel, then another iteration of the LOOP executes. During any iteration, if there is no record found by matching key available in the QUEUE (which raises an ErrorCode value), the method quits.

If the value of the Level is not greater than the SavedLevel value, the next line determines if the levels match. Notice the use of the ABS function here. Since the Level value can be negative (indicating a collapsed node), this function forces a positive (without saving this value). The highlight bar is then placed on the row in the list control and the method quits.

If the Level value is less than the SavedLevel value, the current pointer value is saved and then passed to the list control. Next, the LoadLevel method is then called. After returning from this method, the code retrieves any changes to the current QUEUE record.

#### ❑ PreviousSavedLevel()

This method determines the previous level that matches the current level. It reads the QUEUE structure to determine this:

```
RelateTree.PreviousSavedLevel      PROCEDURE
SaveRecords      LONG,AUTO
SavePointer      LONG,AUTO
CODE
LOOP
    LOOP
        GET(SELF.QRT,POINTER(SELF.QRT) - 1)
        IF ERRORCODE()
            RETURN
        END
        WHILE ABS(SELF.QRT.Level1) > SELF.SaveLevel
            IF ABS(SELF.QRT.Level1) = SELF.SaveLevel
                SELECT(SELF.LC,POINTER(SELF.QRT))
                RETURN
            END
            SavePointer = POINTER(SELF.QRT)
            SaveRecords = RECORDS(SELF.QRT)
            SELF.LC{PROPLIST:MouseDownRow} = SavePointer
            SELF.LoadLevel()
            IF RECORDS(SELF.QRT) <> SaveRecords
                SavePointer += 1 + RECORDS(SELF.QRT) - SaveRecords
            END
            GET(SELF.QRT,SavePointer)
        END
    END
```

There are two LOOP structures. The first or outer LOOP encompasses the entire method. The next, inner LOOP first checks to see if the Level value in the QUEUE is greater than the SaveLevel property value. This is done with the WHILE statement. If Level is greater than SaveLevel, then another iteration of the LOOP executes. During any iteration, if there is no record by matching key available in the QUEUE (which raises an ErrorCode value), the method quits.

If the value of the Level is not greater than the SavedLevel value, the next line determines if the levels match. Notice the use of the ABS function here. Since the Level value can be negative, this function forces a positive (without saving this value). The highlight bar is then placed on the row in the list control and the method quits.

If the Level value is less than the SavedLevel value, the current pointer value is saved and then passed to the list control. Also saved is how many records are in the QUEUE. The LoadLevel method is then called. After returning from this method, the code determines if LoadLevel added any new records to the QUEUE. If so (as the number of records in the QUEUE won't match the value of SavedRecords), it increments the pointer. Finally the code gets the current QUEUE record based on the pointer key value.

## ❑ NextRecord()

This method finds the next record in the list box and moves the highlight to it.

```
RelateTree.NextRecord          PROCEDURE
CODE
  IF ~SELF.WinRef{PROP:Active}
    SELF.WinRef{PROP:Active} = True
  END
  SELF.LoadLevel
  IF CHOICE(SELF.LC) < RECORDS(SELF.QRT)
    SELECT(SELF.LC,CHOICE(SELF.QRT) + 1)
  END
```

The code first determines if the window is active and if not, makes it active. It then calls the `LoadLevel` method.

After returning, it performs a check to see if the highlighted row is less than the number of record in the `QUEUE`. If so, it increments the pointer by one and this value is sent to the list control, which has the effect of moving the highlight one row down.

## ❑ PreviousRecord()

This method finds the previous record in the list and moves the pointer to it:

```
RelateTree.PreviousRecord      PROCEDURE
SaveRecords    LONG,AUTO
SavePointer    LONG,AUTO
CODE
  IF ~SELF.WinRef{PROP:Active}
    SELF.WinRef{PROP:Active} = True
  END
  SavePointer = CHOICE(SELF.LC)-1
  LOOP
    SaveRecords = RECORDS(SELF.QRT)
    SELF.LC{PROPLIST:MouseDownRow} = SavePointer
    SELF.LoadLevel
    IF RECORDS(SELF.QRT) = SaveRecords
      BREAK
    END
    SavePointer += RECORDS(SELF.QRT) - SaveRecords
  END
  SELECT(SELF.LC,SavePointer)
```

The code first determines if the window is active and if not, makes it active. Next, it saves the value of the current position minus one.

A `LOOP` structure then determines the number of records in the `QUEUE` and this value is saved. The `SavePointer` value is then used to position the highlight. The `LoadLevel` method is then called.

After returning from this method, the code checks to see if the number of records still matches the `SavedRecords` value. If so, then the `LOOP` terminates. If not, the code increments the pointer by adding the difference of the number of records now present to the previous value and another iteration of the `LOOP` is performed. After the `LOOP` terminates, the new `SavePointer` value is passed to the list control, selecting the matching row based on the `SavePointer` value.

## Virtual Stub Methods

---

These are more methods that have no code in them. These are placeholders for virtual methods. At the base class level, there is no way to know precisely how to code them as they require information specific to each use. In other words, once this class is locally derived, then there is enough information to actually code the methods.

### ❑ **AssignButtons()**

This method is to assign the navigation buttons from the toolbar, if a toolbar is present in the application. The resulting code should be similar to this:

```
RelationTree.AssignButtons      PROCEDURE
                                CODE
```

The templates discussed in the next chapter write the method code in the child class.

The following three methods control the editing of the highlighted item.

### ❑ **AddEntryServer()** **EditEntryServer()** **RemoveEntryServer()**

For each method, there is an associated button control (**Insert**, **Change**, **Delete**). If the control is either hidden or disabled, then the method should `RETURN`.

Since the templates fill in the data about the back end VIEW structure, the code needed for each is determined by the templates. These methods are called from the edit methods, as follows.

## Edit Methods

---

There must be a way to edit the highlighted item in the tree as well. To keep edit tasks simple, this class uses just a few methods. The following methods set the edit action.

### ❑ AddEntry()

```
RelationTree.AddEntry          PROCEDURE
CODE
SELF.Action = InsertRecord
SELF.UpdateLoop()
```

### ❑ EditEntry()

```
RelationTree.EditEntry         PROCEDURE
CODE
SELF.Action = ChangeRecord
SELF.UpdateLoop()
```

### ❑ DeleteEntry()

```
RelationTree.DeleteEntry       PROCEDURE
CODE
SELF.Action = DeleteRecord
SELF.UpdateLoop()
```

Each of the above methods sets the Action property and then calls the UpdateLoop method.

### ❑ UpdateLoop()

This method is responsible for any edits and any toolbar navigation. If no edits are enabled and there are no toolbar navigation buttons, this method is never called.

```
RelationTree.UpdateLoop      PROCEDURE
CODE
LOOP
    SELF.VCRRequest = VCR:None
    SELF.LC{PROPLIST:MouseDownRow} = CHOICE(SELF.LC)
    CASE SELF.Action
    OF InsertRecord
        SELF.AddEntryServer()
    OF DeleteRecord
        SELF.RemoveEntryServer()
    OF ChangeRecord
        SELF.EditEntryServer()
    END
    CASE SELF.VCRRequest
    OF VCR:Forward
        SELF.NextRecord()
    OF VCR:Backward
        SELF.PreviousRecord()
    OF VCR:PageForward
        SELF.NextLevel()
    OF VCR:PageBackward
        SELF.PreviousLevel()
    OF VCR:First
        SELF.PreviousParent()
    OF VCR:Last
        SELF.NextParent()
    OF VCR:Insert
        SELF.PreviousParent()
        SELF.Action = InsertRecord
    OF VCR:None
        BREAK
    END
END
```

This code calls the actual method to do the edits. It also responds to the VCR control events, since the tree could be navigated by the toolbar VCR controls. Notice the `VCRRequest` is set to `VCR:None`. This is an equate for zero. The three possible edit methods return from their calls, with a `SELF.VCRRequest`. Thus, the CASE statement for it. `SELF.VCRRequest` is a property of this class; be sure to add it to the declaration as a `LONG(0)` – the initial value of zero is important.

Another note about this VCR business. In the INC file, be sure to add this line before any data declarations:

```
INCLUDE('ABTOOLBA.INC'),ONCE
```

This file contains all of the equates needed to make a clean compile. However, if you try it at this time, you get a link error about the ASCII driver not found. This comes from the `ErrorClass`, which is a parent class for many ABC classes, including the toolbar. This comes from the file declaration for logging errors as the `DRIVER` attribute is part of the declaration.

To remedy this, simply place the ASCII driver in your project tree. You will now get a clean compile.

## ❑ Contract and Expand

These are the methods that control the expanding and contracting of the tree:

```
RelationTree.ContractAll          PROCEDURE
CODE
FREE(SELF.QRT)
FREE(SELF.LDQ)

RelationTree.ExpandAll            PROCEDURE
CODE
FREE(SELF.QRT)
FREE(SELF.LDQ)
SELF.LoadAll = True
```

Of all the methods, these two are really incomplete as far as functionality goes. The final line of code should have a call to load a given file. The reason this code is deliberately missing is because the templates will know which file to load. Also, the plan is for the templates to place a call to the `PARENT` object, which executes the above code first. The other design plan is that the templates will extend this class by adding new methods to the derived declaration.

Those are all the methods you need for the base class. Now it's time to create the template that will provide the remaining functionality.





# Chapter 6

---

## WRITING TEMPLATE WRAPPERS

This is a book about OOP, so what's a chapter on template writing doing here? The goal in this chapter is to write a template wrapper for the `CLASS` created in the previous chapter. What this means is that the `TreeClass` is fine for what it does, but a template can certainly make implementing and using the class easier in applications. For those not familiar with templates, I recommend you read the Clarion help on the template language. You may also want to read some of the [template articles at Clarion Magazine](http://www.clarionmag.com/cmaga/topics.html) (<http://www.clarionmag.com/cmaga/topics.html>).

The target audience of this chapter is all template programmers, regardless of skill level.

The purpose is to describe how to implement ABC compliant templates that generate global, or procedure local objects (or new class types). You may wish to revisit this chapter many times as your template writing skills improve. It provides some insight into how to author ABC compliant templates, and why. It also contains some tips and tricks for getting access to symbol values which are otherwise unavailable.

There are not many template coders in the Clarion world. You could blame the structure of the templates themselves for this. Unlike the Clarion language, which provides many statements that encompass a lot of functionality (like `ACCEPT`, `OPEN` etc.), the template language requires you to be quite specific and detailed in programming many tasks. In this regard, the template language seems primitive in comparison to the Clarion language.

On the other hand, there are some template statements that do a lot in one line of code. Therefore, these statements can also be viewed as advanced, in comparison to the Clarion language.

Templates also have a few sets of commands that could be best described as synonyms. There are differences among these sets of commands, but the differences are subtle. This gives rise to confusion as one starts to wonder which command is the most appropriate. That is just one point on the learning curve.

The template language is similar to playing chess. It is easy to learn, difficult to master. The rules are simple, but one must understand the subtleties to really appreciate them. Studying the template language is not a waste of time, even if the payoff comes later.

Templates are viewed by some to be superior to the Clarion language in that they can generate more than just the Clarion language. In that respect, they are correct.

There are developers who use templates to generate ASP (as SoftVelocity has done), COBOL, C++, and Perl. PHP and JSP are others that you may see soon. They can even generate client side Java applications ([www.softmasters.com.ar/eng/jaguar.html](http://www.softmasters.com.ar/eng/jaguar.html)) and .NET languages such as C# (<http://net.radventure.nl/Fenix/>).

My biggest complaint with templates is the lack of tools to assist the template developer. There is no template debugger or dialog formatter, so for the most part templates are hand coded. The template writer tool from SoftVelocity is a good step in the right direction, but it is limited. It is, however, an excellent tool for getting started if you wish to start learning the language.

The *Clarion Handy Tools* ([www.cwhandy.com](http://www.cwhandy.com)) by Gus Creces has tools to assist the template coder. Plus there is template code in ABC that is useful for debugging class wrappers. So there are tools out there, but a template debugger is sorely lacking. Consider this my open letter to the community to author one. I'll be your first customer.

## Templates and Objects

---

Objects can be generated either globally to an application, or locally to a procedure. Currently, there is no support for generating objects into module data. Additionally, any object can now be generated as a new TYPE only.

Once you implement an object using the procedures described in this chapter, you will be able to:

- Change the object name (for objects declared inside procedures only).
- Change the base class of any object.
- Embed code in any public/protected method of any object.
- Add new methods and properties to any object.

### ❑ Class reader

For an ABC class type to be available to the user and Application Generator, the Class Reader must have read it. In other words, the header file (normally INC) containing the class declaration must have been processed. This will happen automatically, when you launch Clarion, if the header file exists in the `libsrc` directory and it contains the text `!ABCIncludeFile` as the first text in the file.

The header reader will ignore any file that does not contain this comment, so if the comment is missing, that file's classes will not be available. This usually results in generation time error messages and badly generated code. Header file reading is triggered by a call to `%ReadABCFiles`, which I will cover later.

Since automatic header reading happens only at the start of your session, you may need to force a reread or refresh, for example, if you change the header files while the environment is active. If so, you may press the **Refresh Application Builder Class Information** button available anywhere `%ClassPrompts` or `%GlobalClassPrompts` is inserted, or on the global classes tab.

**Definition:** Symbols. The template name for variables, which are identified by a leading % mark.

## Writing a new template

---

When I start a new template, I usually add it to a base or root template, which is a template file with a TPL extension. I don't put each new template in its own TPL file because TPLs have to be registered. Instead, I simply put the new template in a TPW file and use the `#INCLUDE` statement to load the TPW file.

The template registry function in Clarion's IDE looks only for TPL type files and by default, in the %ROOT%\template folder, although these files may be anywhere on your system. You just need to walk the folder tree to find them every time you register a template or add another entry in the RED file for other template folders.

The reason for a base template is simple. I can add whatever templates I wish to it. When SoftVelocity issues a new release of Clarion, then I am not forced to edit their templates.

I always state a new template with the #TEMPLATE statement like the following:

```
#TEMPLATE(CBook,'Clarion Book Template'),FAMILY('ABC')
```

---

**NOTE** There is no color syntax for templates in the Clarion editor. This makes template code difficult to read. In the *extras* folder in the source code zip, I've included some C55EDT.INI settings you may use to add color syntax support for your environment.

---

The first parameter is the name of the template, followed by a required description. The FAMILY attribute is required if you want to restrict the template to a certain template family. If left off, the template defaults to the Clarion template chain. This template is for ABC only.

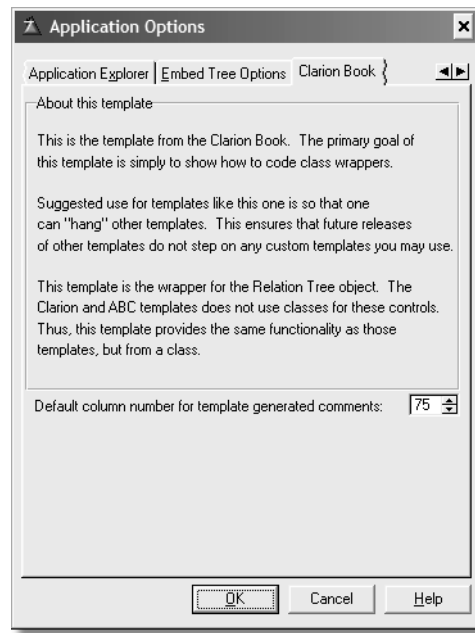
### ❑ The #SYSTEM statement

After the #TEMPLATE statement I use #SYSTEM. This is a good command that allows any dialogs to show in the **Setup | Application Options** dialog. The template code looks like this:

```
#SYSTEM
#TAB('Clarion Book')
  #BOXED('About this template'),AT(5)
    #DISPLAY('')
    #DISPLAY('This is the template from the Clarion Book.')
    #DISPLAY(' The primary goal of this template is')
    #DISPLAY(' simply to show how to code class wrappers.')
    #DISPLAY('')
    #DISPLAY('Suggested use for templates like this one')
    #DISPLAY(' is so that one can "hang" other templates.')
    #DISPLAY('This ensures that future releases ')
    #DISPLAY('of other templates do not step on any custom')
    #DISPLAY('templates you may use.')
    #DISPLAY('')
    #DISPLAY('This template is the wrapper for the Relation')
    #DISPLAY('Tree object. The Clarion and ABC templates does ')
    #DISPLAY('not use classes for these controls.')
    #DISPLAY('Thus, this template provides the same')
    #DISPLAY(' functionality as those templates, but')
    #DISPLAY('from a class.')
    #DISPLAY('')
  #ENDBOXED
  #PROMPT('Default column number for template generated ↵
comments: ',SPIN(@N2,50,99)),%ColumnPos,DEFAULT(75),↵
AT(205,,25)
#ENDTAB
```

This code produces a dialog like the following figure:

Figure 6-1:



### ❑ #APPLICATION vs. #EXTENSION

A common question is what is the difference between the `#APPLICATION` and `#EXTENSION` statements. It is quite simple. Use `#APPLICATION` when you are coding a new family of templates. If you are coding a template to “attach” or “extend” an existing template set, then `#EXTENSION` is the choice. Since this chapter is about coding object wrappers to use with ABC compliant classes, you want `#EXTENSION`.

An `#EXTENSION` could be an application level template or a procedure level template, depending on whether or not you add the appropriate `APPLICATION` or `PROCEDURE` attribute. Since the template use here is for a control, neither template is appropriate. It is applicable on a certain use, which I cover later. For now, just keep these two template in mind for future use.

## ❑ #CONTROL Template

A better choice for the RelationTree wrapper is a #CONTROL template. The code looks like this, but in the source it is on one line. I've wrapped some of the lines for better readability here:

```
#CONTROL(RelationalTree,'Relational Tree Object List↵
Box'),PRIMARY('Relational Tree Object List↵
Box',OPTKEY),DESCRIPTION('Tree structure related to ' &↵
%Primary),MULTI,WINDOW,WRAP(List)
```

The name of the template and its description is obvious. The PRIMARY attribute means a primary file for the set of controls must be placed in the procedure's Table Schematic. If you use this attribute, the description is required. The OPTKEY parameter means the key label is not required.

DESCRIPTION specifies the display description of a #CONTROL that may be used multiple times in a given application or procedure. In this case, you'll see which file this tree list is for.

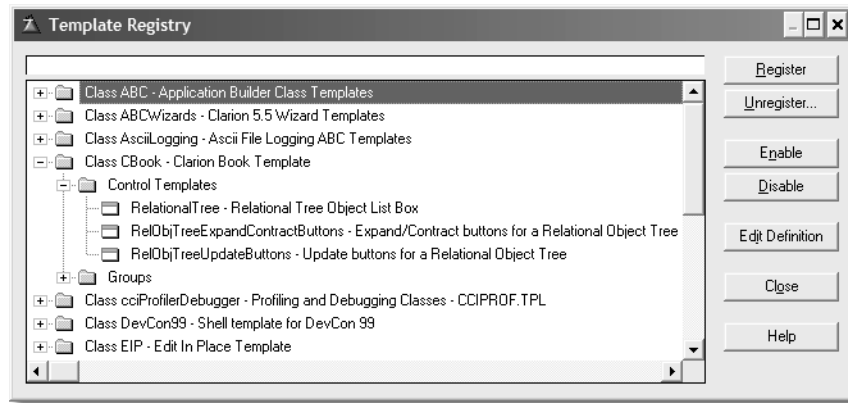
MULTI specifies the #CONTROL may be used multiple times in a given window.

WINDOW tells the Application Generator to make the #CONTROL available in the Window Formatter.

WRAP specifies the #CONTROL template is offered as an option for the control when the **Translate controls to control templates when populating** option is set in **Application Options**.

When registered, the template appears as follows (your registry will vary depending on the templates you have registered):

Figure 6-2:



### ❑ Loading ABC classes in memory

A `#GROUP` is a bit of reusable template code. You can insert it almost anywhere in your template code. The advantage is that you may re-use template code instead of coding it again.

To use a `#GROUP`, use the `#INSERT` statement (there are other statements that do the same thing, but I am sticking with `#INSERT` for now).

You must ensure that all ABC compliant classes are loaded into memory. This is done by this line of template code:

```
#INSERT(%OOPPrompts(ABC))
```

The above example shows the optional family parameter. It is optional in that this template is an ABC family member. Thus, if left off, the template system expects this group to be in this template set and will complain if it is not found there. The following is the `#GROUP` code:



```
#GROUP(%OOPPrompts)
  #BOXED(''),AT(0,0),WHERE(%False),HIDE
  #INSERT(%OOPHiddenPrompts(ABC))
#ENDBOXED
#!
#GROUP(%OOPHiddenPrompts)
  #PROMPT(' ',@S64),%ClassItem,MULTI(''),UNIQUE
  #BUTTON(' '),FROM(%ClassItem,'')
  #PROMPT(' ',@S64),%DefaultBaseClassType
  #PROMPT(' ',@S64),%ActualDefaultBaseClassType
  #PROMPT(' ',@S255),%ClassLines,MULTI('')
#ENDBUTTON
```

The `#INSERT` statement places code in a `#GROUP` where `#INSERT` executes. Using `#GROUP` serves two major purposes:

- It keeps template code clean
- It allows reuse of template code, thus eliminating retyping the same code or copying/pasting.

The above two groups are part of the ABC template chain. You call these groups by adding (ABC) after the group name. That is the family argument for a group name and is not needed if the group belongs to the current template. If you always use the group parameter, then you will never get errors about missing template code, unless you use a wrong family member name.

You really only need to use the family name with group calls when calling a `#GROUP` *not* of your template set. That way you know if a group belongs to your template set because it has no parameter. Or just always use the family name, even for your own set's groups. Either way, the choice is yours.

## ❑ Setting up Class Items and OOP Defaults

It is very important that *two* identical calls to `%SetClassDefault` are made, one from the `#PREPARE` section and one in the `#ATSTART` section.

The `#PREPARE` sets up defaults and template prompts for the user if they enter the prompts section.

`#ATSTART` catches the instance where the developer never uses the prompts section of a template, but simply generates code, using the defaults. The `#ATSTART` structure processes any code in it before code generates. You may think of it as an initialization procedure before code generation.

Thus, one allows the developer to change the class, the other allows the developer to generate working code regardless of whether they changed the class or not.

The following code sets up defaults for the `Relation Tree` object:

```
#PREPARE
  #CALL(%ReadABCFiles(ABC))
  #CALL(%SetClassDefaults(ABC),↵
'RTree' & %ActiveTemplateInstance,↵
'RTree' & %ActiveTemplateInstance,'RelationTree')
#ENDPREPARE
#ATSTART
  #CALL(%ReadABCFiles(ABC))
  #CALL(%SetClassDefaults(ABC),'RTree' & ↵
%ActiveTemplateInstance,          'RTree' &↵
%ActiveTemplateInstance,'RelationTree')
#ENDAT
```

The above is a small snippet of what is really in the `#ATSTART` statement in the template. The rest of the code is there to define symbols, set up values, etc.

One interesting aspect of the code is this section:

```
#FOR(%Control),WHERE(%ControlInstance = %ActiveTemplateInstance)
  #SET(%TreeControl,%Control)
  #SET(%TreeQueue,EXTRACT(%ControlStatement,'FROM',1))
#ENDFOR
```

The `#FOR` statement loops through the controls on the window that match the current instance. This is provided as it is possible to place more than one such control on a window.

The first `#SET` statement assigns the value of the second parameter to the first parameter. This is done so you know the name (or label) of the control. This makes it easy to use in later template code, plus you do not have to issue the `#FOR` loop again.

The second `#SET` assigns the value of the `FROM` attribute of the `LIST` control. This is done via the `EXTRACT` statement. `EXTRACT` “string slices” though a control’s attribute, looking for a specific match. The `FROM` attribute names a `QUEUE` that holds the data that is seen on a `LIST`. If you want to see how this works, you can add an `#ASSERT` and build a string as follows:

```
#ASSERT(~%TreeQueue,'TreeQueue: ' & %TreeQueue & '↵  
ControlStatement: ' & %ControlStatement)
```

This produces the following string when you generate source (emphasis added):

```
(TEST.B1$) Error:  ASSERT: TreeQueue: QTree ↵  
ControlStatement: LIST,AT(23,17,284,100),↵  
USE(?RTree),HVSCROLL,↵  
FORMAT('800L*IT@s200@'),FROM(QTree)
```

## ❑ Adding Global Prompts

Sometimes you may need global instances of a CLASS. For a control, this is not really appropriate as they are always local to a procedure. However, a procedure level class does present a problem. In the ABC templates, all classes are considered global, with local derivations. These are always set up in the global prompts, thus they are available globally.

Here's the rub; the `RelationTree` class definition is included in an app when a control is placed in a local procedure. This works fine if you plan to have only one executable. What about DLL and LIB projects? The custom with Clarion is that a root or global DLL is made, often with no procedures, and this DLL contains the exported definitions of all the classes. But so far the `RelationTree` class is not available for export, which is a requirement for DLL use. The technique for making this all work is something called a global extension. If you don't have this, and you want to use `RelationTree` with a DLL, you'll get an "unresolved external" link error.

Therefore, a global extension template is required. You will need to add this template to any application which uses a global data DLL arrangement.

The template is quite simple. Here's how it is declared.

```
#EXTENSION(GlobalTree,'Global extension for DLL/LIB use only')↵  
,APPLICATION
```

The `APPLICATION` attribute is the key to global extensions. Following the declaration is some standard code:

```
#PREPARE
  #CALL(%ReadABCFiles(ABC))
  #CALL(%SetClassDefaults(ABC),'RTree' &␣
%ActiveTemplateInstance,'RTree' &␣
%ActiveTemplateInstance,'RelationTree')
#ENDPREPARE
#ATSTART
  #CALL(%ReadABCFiles(ABC))
  #CALL(%SetClassDefaults(ABC),'RTree' &␣
%ActiveTemplateInstance,'RTree' &␣
%ActiveTemplateInstance,'RelationTree')
#ENDAT
#INSERT(%OOPPrompts(ABC))
```

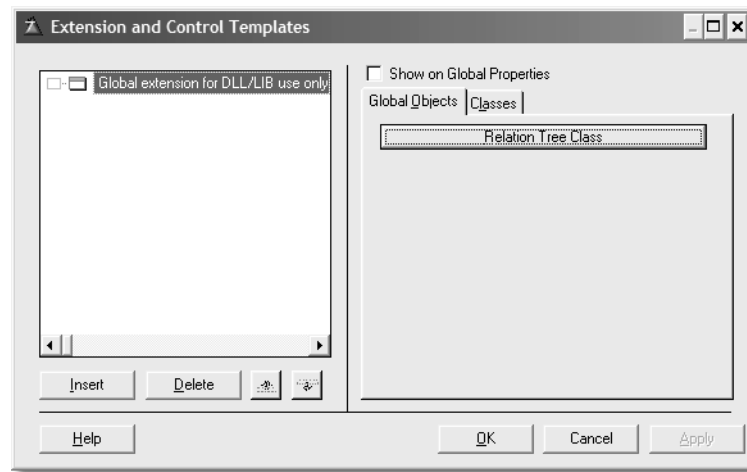
You've seen this code before, it is the same as what was discussed previously. And it's here for the same reasons. You also need to have the same code for the prompts, with one minor exception:

```
#TAB('Global &Objects')
  #BUTTON('&Relation Tree Class'),AT(.,170)
    #WITH(%ClassItem,'RTree' & %ActiveTemplateInstance)
      #INSERT(%GlobalClassPrompts(ABC))
    #ENDWITH
  #ENDBUTTON
#ENDTAB
```

The key is calling %GlobalClassPrompts. This creates a single instance of the CLASS. Since you really cannot create multiple instances of a global object (and why would you?), the intent is to either use the global instance (like ABC's GlobalError object derived from ErrorClass) or you intend to have multiple

instances locally. And this fits well if you need to ensure you have a `RelationTree` class that is exportable.

Figure 6-3:

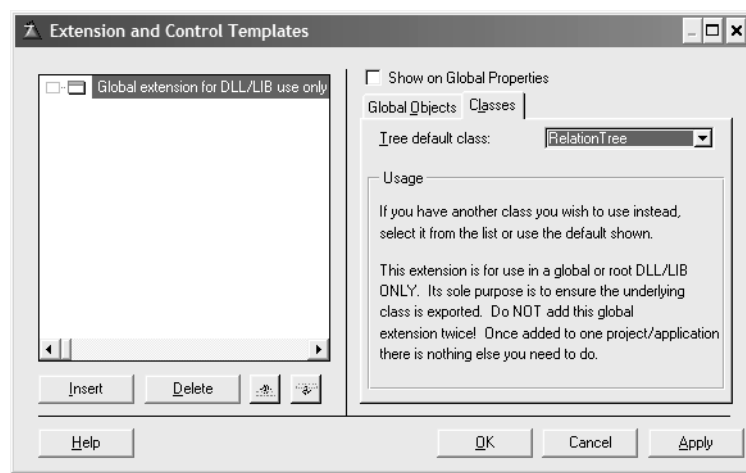


Of course, there is another tab on the global prompts. This is for the name of the base class:

```
#TAB('C&lasses')
#PROMPT('&Tree default class:',FROM(%pClassName)),
%ClassName,DEFAULT('RelationTree'),REQ
#DISPLAY()
#BOXED(' Usage ')
#DISPLAY()
#DISPLAY('If you have another class you wish to use instead, ')
#DISPLAY('select it from the list or use the default shown.')
#DISPLAY()
#DISPLAY('This extension is for use in a global DLL/LIB ')
#DISPLAY('ONLY. Its sole purpose is to ensure the underlying')
#DISPLAY('class is exported. Do NOT add this global')
#DISPLAY('extension twice! Once added to one application,')
#DISPLAY('there is nothing else you need to do.')
#DISPLAY()
#ENDBOXED
#ENDTAB
```

This produces the following dialog:

**Figure 6-4:**



For more information on using this template, see the next chapter.

### ☐ Exporting the class

You need only one more bit of template code to ensure the class is exported properly:

```
#AT(%BeforeGenerateApplication)
  #CALL(%AddCategory(ABC),'CB')
  #CALL(%SetCategoryLocation(ABC),'CB','CB')
#ENDAT
```

This not only sets things in motion for exporting, it places the correct flags and sets their values properly under the **Defines** tab in the project editor.

## ❑ Adding Local Prompts

You need to add a set of prompts for each instance of the object(s). These prompts enable the user to change the object name (unlike a global object), change the base class and add new methods and/or properties at design time.

For *local* objects this is done by inserting the %ClassPrompts #GROUP inside a uniquely defined #WITH clause. For example;

```
#TAB('Local &Objects')
  #BUTTON('&Relation Tree Class'),AT(,170)
    #WITH(%ClassItem,'RTree'& %ActiveTemplateInstance)
      #INSERT(%ClassPrompts(ABC))
    #ENDWITH
  #ENDBUTTON
#ENDTAB
```

Note that these should be populated onto a **Global Objects** tab, and a suitably named button should surround each inserted prompt set.

The second parameter of the #WITH statement wrapping each #INSERT(%ClassPrompts) is key to the whole system. It becomes the “tag” used to get the generator’s model of a specific instance of an object. For this reason, it *must* be unique within any given scope. Also, it is possible to have more than one tree list on a window.

The key to this is the symbol %ActiveTemplateInstance. This is a built-in symbol. The Clarion documentation defines this as “The instance numbers of all control templates used in the procedure.” Since this is dependent on another built-in symbol, %ActiveTemplate, it is unique for each tree control you may place on a window.

### ❑ Global Symbols and Objects

To begin, global symbols contain the session default class names for all new objects. These symbols should use the `FROM()` type to allow the user to select a class from a list of known classes.

`%pClassName` is the symbol that contains all known classes. Bear in mind that any default values supplied for class names are case sensitive and must match *exactly* the class label as defined in the header file.

You do this in the `#APPLICATION` section, or in any `#EXTENSION` with the `APPLICATION` attribute.

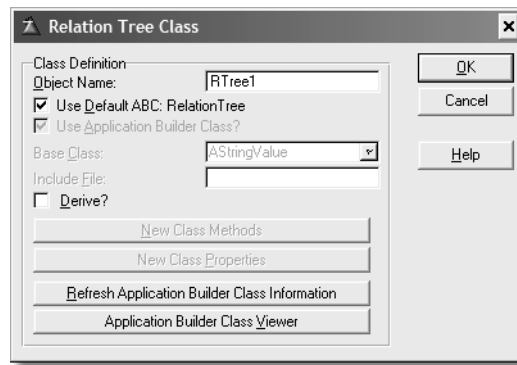
For consistency, and to ensure that the generator merges “pages” together nicely, these should appear on a tab labeled **Classes**, and buttons should be used to group logically similar/related class types together.

```
#TAB('Local &Objects')
  #BUTTON('&Relation Tree Class'),AT(.,170)
    #WITH(%ClassItem,'RTree' & %ActiveTemplateInstance)
      #INSERT(%ClassPrompts(ABC))
    #ENDWITH
  #ENDBUTTON
  #BUTTON('&Library Files'),AT(.,170)
    #BOXED('Library Files')
      #INSERT(%ABCLibraryPrompts(ABC))
    #ENDBOXED
  #ENDBUTTON
#ENDTAB
#TAB('C&lasses')
  #PROMPT('&Tree default class:',
FROM(%pClassName)),%ClassName,DEFAULT('RelationTree'),REQ
  #DISPLAY()
  #BOXED(' Usage ')
    #DISPLAY()
    #DISPLAY('If you have another class you wish to use instead, ')
    #DISPLAY('select it from the list or use the default shown.')
    #DISPLAY()
  #ENDBOXED
#ENDTAB
```



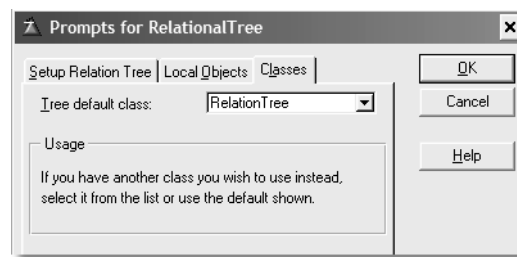
This produces the following dialog:

Figure 6-5:



The **Classes** tab takes on this appearance:

Figure 6-6:



Now that the prompts have been set up, they must be given their default values. It is important to realize that developers are *never* required to enter any of the class or global object values to generate an application that works - the defaults must always be enough.

I have snuck in a third tab, which is the same dialog as the one you find in the Clarion and ABC template chains for the tree lists.

## ❑ A List of Objects

The ABC templates maintain a list of objects that are currently in scope from the developer's perspective. This is currently used by the `CallABCMethod` and `SetABCProperty` groups. It is therefore important that objects are added to the

known object list in the %GatherObjects embed point. Each object tag should be passed to %ObjectList in turn, for example:

```
#AT(%GatherObjects)
  #CALL(%AddObjectList(ABC), 'RTree' & %ActiveTemplateInstance)
  #ADD(%ObjectList, %ThisObjectName)
  #SET(%ObjectListType, 'RelationTree')
#ENDAT
```

The #CALL statement calls a #GROUP. The #GROUP code is placed where the #CALL is placed. In this regard, it is similar to #INSERT with the NOINDENT attribute.

### ❑ Adding Local Declarations

In order to have code generated correctly in your procedures, the local instance must be instantiated. You do this with the %LocalDataClasses embed point.

```
#AT(%LocalDataClasses), ␣
WHERE(%ControlTemplate='RelationalTree(CBook)')
  #INSERT(%GenerateClass(ABC), 'RTree' &
%ActiveTemplateInstance, ␣
'Local instance and definition'), NOINDENT
#ENDAT
```

Notice the WHERE clause. %ControlTemplate is the internal multi-valued symbol that contains the names of control templates populated on a window or report. What this says is the following code generates when a specific #CONTROL template is present (which it is) and ignore other control templates.

The %ActiveTemplateInstance generates a CLASS definition for every instance of the TreeControl placed on the window.

The #INSERT statement generates compilable code for the local instance. Notice the NOINDENT attribute. I like to indent my template code. However, if I left off this attribute the data declaration generates in the column where the #INSERT appears, in column three. That is invalid for a data declaration. Alternatively, you could use #CALL instead of #INSERT(), NOINDENT.

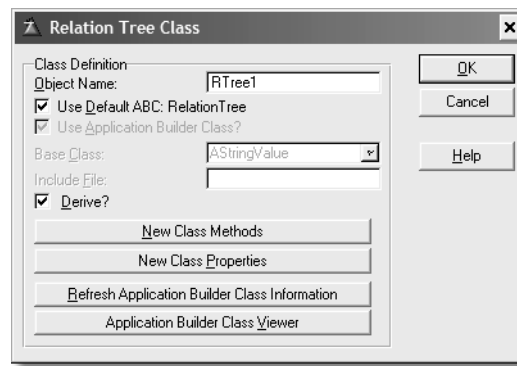
### ❑ How to Add New Methods

One of the interesting things you can do with template code is add new methods to a local class. The class definition is missing a few methods and this is deliberate. The design of the tree requires that methods load whatever files are listed in the

table schematic. This function is ideal for template use as there are internal symbols that retrieve whatever is placed there.

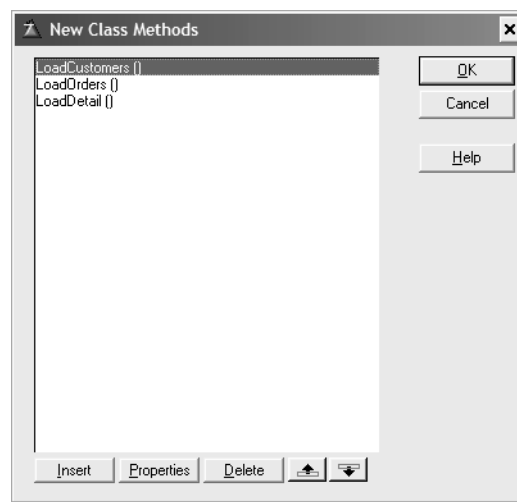
This means that a method must not only be defined, but code written for each of these methods. This is done via the Class dialog. For the local instance of the `RelationTree`, the dialog appears as follows:

Figure 6-7:



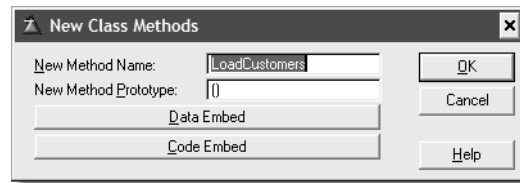
The **Derive?** check is set, which enables the **New Class Methods** and **New Class Properties** buttons. Press **New Class Methods** and the following dialog appears:

Figure 6-8:



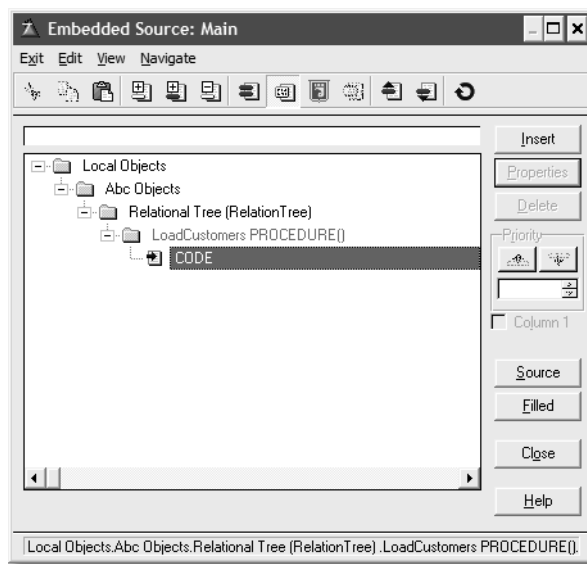
This dialog contains a list of all new methods and their prototypes. If you press **Properties** the following dialog displays:

Figure 6-9:



The buttons allow you to declare local variables for this method and/or the code for the method. Both of these buttons take you to the proper embed points.

Figure 6-10:



The point of showing you these dialogs is that without templates generating these points, you are required to fill in these manually. This requires a slight alteration of the template code shown previously to generate the class definition.

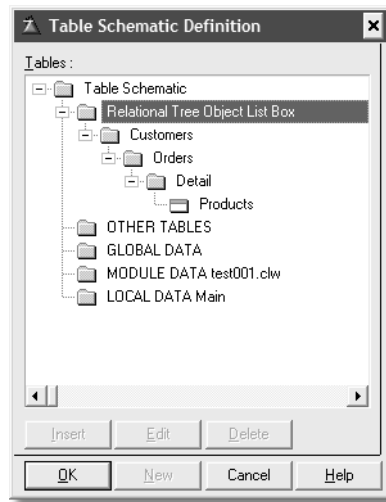
```

#AT(%LocalDataClasses),↵
WHERE(%ControlTemplate='RelationalTree(CBook)')
  #FIX(%File,%Primary)
  #CALL(%AddMethods,'Load' & %File,'()')
  #FOR(%Secondary),WHERE(%SecondaryTo = %File ↵
AND %SecondaryType = 'MANY:1')
    #FIX(%File,%Secondary)
    #CALL(%AddMethods,'Load' & %File,'()')
  #ENDFOR
  #CALL(%SetClassItem(ABC), 'RTree' & %ActiveTemplateInstance)
  #INSERT(%GenerateClass(ABC), 'RTree' &
%ActiveTemplateInstance,↵
'Local instance and definition'),NOINDENT
#ENDAT

```

The %Primary is the file listed first in the table schematic. For example, a developer may want to add the files shown in the following figure:

Figure 6-11:



The value of %Primary would be Customers. The next file is related to it. Thus #FIX sets or assigns the value of %Primary to %File. Next a call to %AddMethods, which is a local #GROUP (more on that shortly). That simply generates the proper method declaration.

The #FOR loop is filtered by the WHERE clause, and ensures that only related files are considered. It also makes the same call to %AddMethods, but only for related files.

The #GROUP code is listed below:

```
#GROUP(%AddMethods,%pMethodName,%pMethodPrototype),AUTO
#DECLARE(%MethodsPresent)
#DECLARE(%LastNewMethodsInstance)
#SET(%MethodsPresent,0)
#SET(%LastNewMethodsInstance,0)
#FOR(%NewMethods)
    #SET(%LastNewMethodsInstance,%NewMethods)
    #IF(UPPER(CLIP(%NewMethodName))=UPPER(CLIP(%pMethodName))) ↵
AND  UPPER(CLIP(%NewMethodPrototype))=↵
UPPER(CLIP(%pMethodPrototype)))
        #SET(%MethodsPresent,1)
        #BREAK
    #ENDIF
#ENDFOR
#IF(%MethodsPresent=0)
    #ADD(%NewMethods,%LastNewMethodsInstance+1)
    #SET(%NewMethodName,%pMethodName)
    #SET(%NewMethodPrototype,%pMethodPrototype)
#ENDIF
#SET(%DeriveFromBaseClass,%True)
```

The AUTO attribute opens a new scope for the group. This means that any #DECLARE statements in the #GROUP would not be available to the #PROCEDURE being generated. In other words, this is a way to define local (to the #GROUP) symbols.

The first lines define new, local instance only symbols, followed by code that sets their initial values.

The #FOR loop executes as long as there are values in %NewMethods. The #IF guarantees an accurate comparison by testing upper case letters. If there is a match, then the %MethodsPresent flag is set and since there is no longer a need to loop, a #BREAK is issued. Otherwise, the #FOR loop ends naturally once all values are read. Which means the %MethodsPresent flag is not set.

The next `#IF` tests to see if the `%MethodsPresent` is not set. If that is true, then increment the number of `%NewMethods` and set their names. The last act is to set the **Derive?** check box.

What this all means is that the class dialog box adds all the new methods to the class definition and these are visible in the dialogs as the previous illustrations show. It means the developer does not have to do any extra work.

This whole exercise is one way to replace the `%InstancePrefix:Load:%TreeLevelFile` ROUTINE in the ABC and Clarion template chains.

## ❑ Clarion 5.5 vs. Clarion 6

Due to the pre-emptive threading model added to Clarion 6, the template code for generating class definitions is different between the two versions.

CLASS structures can now have the `THREAD` attribute. To be accurate, you may add `THREAD` to the CLASS in 5.5, but it does nothing.

Here is the 5.5 version of the `%GenerateClass` group:

```
#GROUP(%GenerateClass, %Tag, %ClassComment = '', %AsType = %False)
  #ASSERT(%Tag <> '', '%GenerateClass: object Tag is blank!')
  #CALL(%SetClassItem, %Tag)
  #CALL(%GenerateClassDefinition, %ClassLines, ↵
    %ClassComment, %AsType)
```

Compare with the Clarion 6 version:

```
#GROUP(%GenerateClass, %Tag, %ClassComment = '', ↵
  %AsType = %False, %Attrs = '')
  #ASSERT(%Tag <> '', '%GenerateClass: object Tag is blank!')
  #CALL(%SetClassItem, %Tag)
  #CALL(%GenerateClassDefinition, %ClassLines, %ClassComment, ↵
    %AsType, %Attrs)
```

There is an extra parameter, `%Attrs`. Its use is for adding the `THREAD` attribute as part of a CLASS declaration as shown below:

```
#INSERT(%GenerateClass, 'ErrorManager', 'Global error manager', ↵
  %False, 'THREAD')
```

Keep in mind that `THREAD` is not appropriate for any CLASS declared with the `TYPE` attribute.

If you would like to add the `THREAD` attribute after the 5.5 declarations (where appropriate), you have to edit the shipping templates or make your own group.

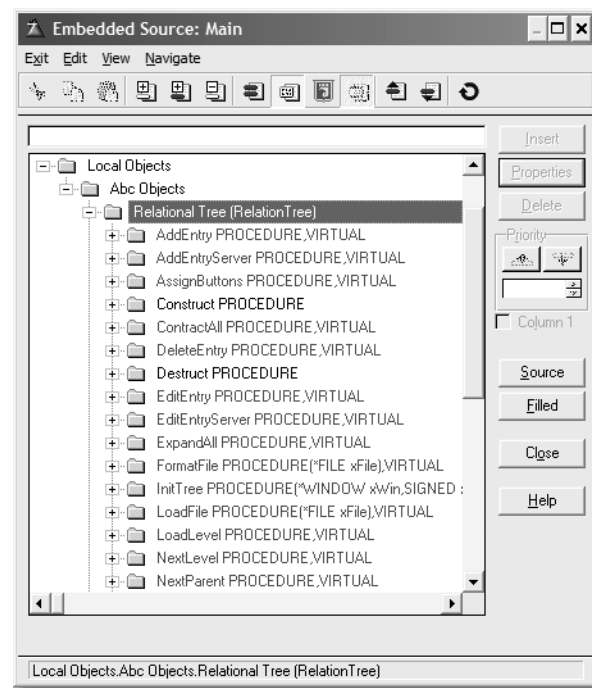
### ❑ Generating the embed tree

In order to properly place embed points for public and virtual methods in your class definition, you need some template code to ensure this happens. I've wrapped the line for readability, but the `#CALL` statement is really on one line:

```
#AT(%ProcedureRoutines),  
WHERE(%ControlTemplate='RelationalTree(CBook)')  
  #CALL(%GenerateVirtuals(ABC),          'RTree' &↵  
    %ActiveTemplateInstance,↵  
    'Local Objects|Abc Objects|Relation Tree',↵  
    '%EmbedVirtuals(CBook)')  
#ENDAT
```

Notice the same `WHERE` clause. You want the embed tree to appear for the proper instance. The `%GenerateVirtuals` from `ABC` is the key to this. Thus, you get an embed tree that looks similar to this one:

Figure 6-12:





Notice how it appears next to any other local objects. This sets up added embed code for any method for the Relation Tree.

## □ How to Call the Parent

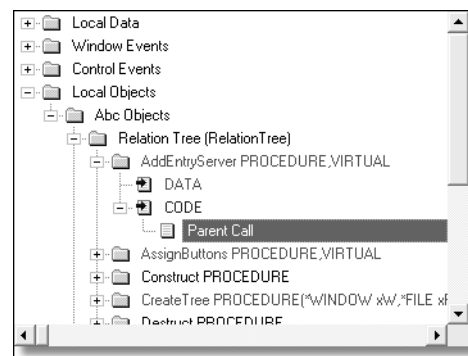
All the methods generated by the templates contain a call to their PARENT method. This code should *always* be generated at a PRIORITY of 5000; this ensures that there is plenty of “embed space” around the parent call for other code. Where the method is defined to return a value, a method local variable (of the correct type, or reference) called `ReturnValue` is automatically created, and a `RETURN ReturnValue` statement is automatically generated at the end of the procedure.

To trigger the generation of this code, a call to `%GenerateParentCall` should be made inside *every* method of *every* object – this is easily done by adding the following code inside an `#AT` for every object inside the current template. With any class, there should always be a provision to call the parent class. This is quite easy with template code. Inspect the following code (wrapped for readability):

```
#AT(%TreeClassMethodCodeSection,%ActiveTemplateInstance),↵
    PRIORITY(5000),↵
    DESCRIPTION('Parent Call'),WHERE(%ParentCallValid())
    #CALL(%GenerateParentCall(ABC))
#ENDAT
```

This produces the following embed point in the tree (in all embeds, just one is illustrated):

Figure 6-13:



What this gives you is the ability to add your own code before the parent method is called or after it is called, depending on how you want the overridden behavior to work.

### ❑ Adding Embed Points

This is the code that actually enumerates the embeds for each method and property in your class (provided it is not private). This code is essentially the same for global and local objects, and is duplicated for each object. Where an object is dependent upon some other symbol, this must be reflected in the `#EMBED` statements.

The following code generates all the embeds for the `Tree` object (wrapped for readability):

```
#IF(%BaseClassToUse())
  #CALL(%FixClassName(ABC),%BaseClassToUse())
  #FOR(%pClassMethod)
    #FOR(%pClassMethodPrototype),WHERE(%MethodEmbedPointValid())
      #CALL(%SetupMethodCheck(ABC))
      #EMBED(%TreeMethodDataSection,↵
        'Relation Tree Method Data Section '),↵
        %ActiveTemplateInstance,↵
        ,%pClassMethodPrototype,LABEL,DATA,↵
        PREPARE(,%FixClassName(%FixBaseClassToUse(↵
          'RelationTree'))),↵
        TREE(%GetEmbedTreeDesc('TREE','DATA '))
      #?CODE
      #EMBED(%TreeMethodCodeSection,↵
        'Relation Tree Method Executable Code Section'),↵
        %ActiveTemplateInstance,↵
        ,%pClassMethodPrototype,↵
        (,%FixClassName(%FixBaseClassToUse(↵
          'RelationTree'))),↵
        TREE(%GetEmbedTreeDesc('TREE','CODE'))
      #CALL(%CheckAddMethodPrototype(ABC),%ClassLines)
    #ENDFOR
  #ENDFOR
  #CALL(%GenerateNewLocalMethods(ABC),'TREE',%True)
#ENDIF
```

You must consistently use the object tag as it is passed to `%SetClassItem` and to `%FixBaseClassToUse` on both `#EMBEDs`. The `%GetEmbedTreeDesc` call takes

parameters to describe how to structure the embed tree for this embed point. Valid values for the first parameter are currently hard coded into the `#GROUP` – this may change in the future!

The `WHERE` clause on the inner `#FOR` loop enforces that the method in question is valid for embedding. For example, it ensures the method is not private. These are really `#GROUP` structures, thus they can be built like functions, i.e. they return values. This is a small subtlety of the template language, but it means that you can build some very powerful template functions.

If `%True` is passed as a parameter, the template further enforces that *only* virtual methods of the class in question are made available – this is used by the file and relation managers, for example.

The second last line calls `%GenerateNewLocalMethods`, which is responsible for generating embed points for any new methods that the user has added to the object. The first parameter describes how to structure the embed point in the embed tree and is the same as the first parameter to `%GetEmbedTreeDesc`. The second parameter is `%True` when this is a global object, or omitted otherwise.

Note that these are procedure local objects, and that there are additional `#EMBED` dependencies; namely `%ActiveTemplateInstance` (a procedure can have multiple browses).

These `#EMBED` dependencies must generally align with the dependencies of the `#WITH` statement for this object. Procedure local objects usually have an implied dependency upon `%ActiveTemplateInstance`.

For procedure local objects this code should appear inside a `#AT` that points to the `%LocalProcedures` embed point. For global objects, it should appear inside an `#AT` that points to the `%ProgramProcedures` embed point.

The Relation Tree template wrapper does not use global data, thus is not needed. However, if your templates need to generate a declaration in the global data area, the following code generates the object declaration (using the Web Guard template):

```
#AT(%GlobalData),WHERE(%GuardEnabled)
  #INSERT(%GenerateClass(ABC),'Guard','Application Security
Guard')
#ENDAT
```

The second parameter is the tag of the object to generate; the third parameter simply gets generated as a comment on the class declaration line. Optionally, you

may pass %True as a third parameter that will force the object to be generated as a TYPE (i.e., TYPE is appended to the declaration). Global objects should be generated at the %GlobalData embed point, local objects should be generated at the %LocalDataClasses embed point.

The following WebBuilder example generates a number of procedure local objects: note that the objects based upon %Control are generated as class types.

```
#AT(%LocalDataClasses),WHERE(%ProcedureHasWebWindow())
  #INSERT(%GenerateClass(ABC),'WebWindowManager')
  #IF(%IsFrame())
    #INSERT(%GenerateClass(ABC),'WebFrameManager')
  #ENDIF
  #IF(%HasQBE())
    #INSERT(%GenerateClass(ABC),'QBEWebWindowManager')
  #ENDIF
  #FOR(%Control),WHERE(%Control <> '')
    #CALL(%SetClassItem(ABC),%Control)
    #IF(ITEMS(%ClassLines))
      #INSERT(%GenerateClass(ABC),↓
        %Control,'Web Control Manager for ' & %Control,%True)
    #ENDIF
  #ENDFOR
#ENDAT
```

### ❑ Scoping Issues

As you have seen, %ClassItem is a symbol defined whenever objects are generated by the templates. All template code is executed in the context of its own local %ClassItem. This means that from any given template section (#EXTENSION, #CONTROL, etc.) you can only “see” your own %ClassItem.

Since you cannot see the %ClassItem of anything else, you cannot #FIX it using a tag that you did not create locally – this is why global objects have their names fixed. You would never be able to fix %ClassItem of the Application (or global extension) to the correct tag, and therefore you would always get generation errors reporting; **'%SetClassItem: Instance not found!'**

The most usual reason to attempt to #FIX an external %ClassItem is to get at the object name. Declaring a local variable inside the parent component and copying the object name into it can achieve this. For example:

```
#DECLARE(%ManagerName)
#FIX(%ClassItem,'Default')
#SET(%ManagerName,%ThisObjectName)
```

Providing that this is done early enough, usually in `#ATSTART`, and that the child template has the `REQ` attribute (requiring the parent template), the child can see the declared variables of its parent, and hence it can get at the parent object name.

## The Remaining Control Template Code

---

Now that the wrapper portion is done, it's time to turn your attention to the rest of the control template. Since the class is for a relation tree, this is a control on a window, thus the name of the template.

What follows next is the actual Clarion code placed on the window structure. The template does this by the `CONTROLS` statement.

```
#! Add target language code for a generic list box
#! with tree, color and icon attributes
CONTROLS
    LIST,AT(,,150,100),USE(?RTree),FORMAT('800L*IT@s200@'),|
        FROM(QTree),#REQ
END
#! end controls
```

The `#REQ` attribute in this section means that if you delete the control from the window, then all template code goes with it. If you've added embedded code in this control's embed points, they are not deleted, but moved to an orphaned section of the embed tree. You see this section only when orphaned embeds exist.

### ❑ The Control Interface Dialogs

The next section simply copies all of the `#TABS`, `#BUTTONs`, etc. from the ABC version. There is no need to code something that already exists. Besides, the point of the class and the template is to mimic or duplicate the existing ABC behavior exactly. The benefits of converting to a class and a template wrapper over the existing template should be obvious – you may extend the behavior far easier and the templates are not driving everything; that is the class's job.

I've added the developer dialogs as the first tab in the dialog section:

```
#TAB('&Setup Relation Tree')
  #!Add the developer dialogs
  #INSERT(%RelationTreeProperties)
#ENDTAB
```

You can find the %RelationTreeProperties group with the rest of the #GROUP definitions in the file CBGroups.tpw. You can tell this #GROUP belongs to this template as it is not using the family attribute that you've seen so far. This code is "stolen" from ABRELTREE.TPW with a minor modification. Each icon entry also has a lookup button. Both ABC and Clarion template chains are strictly entry controls, making the template difficult to use.

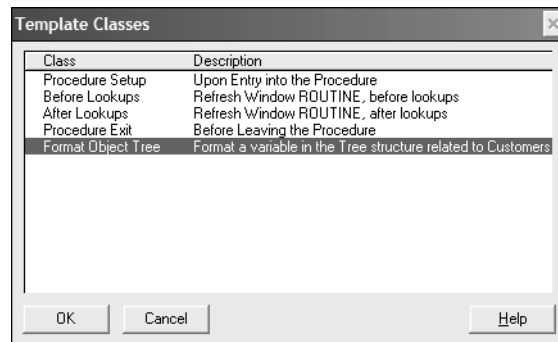
### ❑ Formula Editor

Not many programmers use the formula editor anymore, but there is template support for it, so I've included that support in this template as well, via the #CLASS statement. Thus the following line (wrapped for readability):

```
#CLASS('Format Object Tree','Format a variable in the ' ↵
%ActiveTemplateInstanceDescription)
```

The text is changed slightly so you can differentiate between the ABC tree if you place both relation trees on a window. It is theoretically possible to have both trees on the same window (but the control for both must be placed on the window first).

Figure 6-14:



### ❑ The Design of the Control Template

When considering the design of what a template might do, it is best to make some notes and itemize the major functions. What does the template produce if the

developer adds nothing from the default control? What additional features does the template support, such as list colors?

While the dialog (what colors, icons, etc.) exposed to the developer really does not need to change from the ABC version, the ABRELTREE dialog template code could easily be copied and pasted to a #GROUP. It is rather lengthy, so I tend to put such dialogs in #GROUPs. I like my template code segmented into functions, thus any maintenance I do is easier. If you run a test on a dummy application (which is smart during the coding and testing phase), you can see that the dialogs are exactly the same. This is a good thing, as anyone used to coding ABC relational tree lists won't have anything new to learn.

### ❑ What about icons?

One of the best uses of a template is to do the dull, boring code generation tasks. Adding icons to a project tree falls in this category. It is also easy to forget to do. Thus, you have this bit of template code to handle this task for you:

```
#AT(%CustomGlobalDeclarations)
  #INSERT(%FileControlSetFlags(ABC))
  #FOR(%Control),WHERE(%ControlInstance =
%ActiveTemplateInstance)
    #IF(%ControlHasIcon)
      #IF(%TreeTitleIcon)
        #INSERT(%StandardAddIconToProject(ABC),%TreeTitleIcon)
      #ENDIF
    #IF(%PrimaryDefaultIcon)

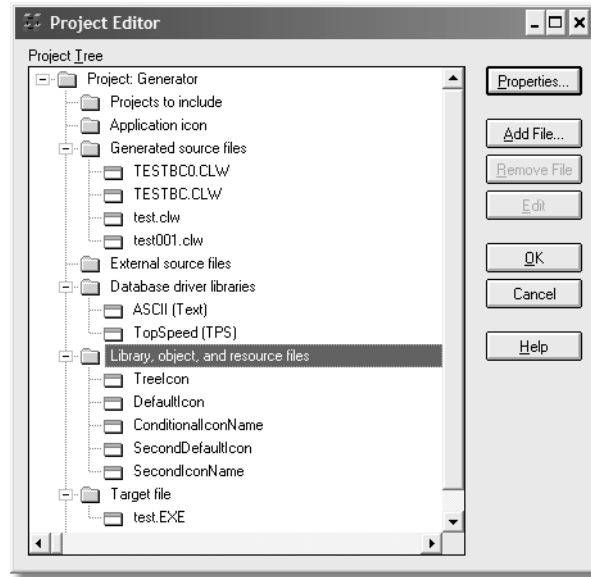
#INSERT(%StandardAddIconToProject(ABC),%PrimaryDefaultIcon)
  #ENDIF
  #FOR(%PrimaryConditionalIcons)
    #INSERT(%StandardAddIconToProject(ABC),↓
%PrimaryConditionalIcon)
  #ENDFOR
  #FOR(%Secondary),WHERE(%SecondaryType = 'MANY:1')
    #IF(%SecondaryDefaultIcon)
      #INSERT(%StandardAddIconToProject(ABC),↓
%SecondaryDefaultIcon)
    #ENDIF
    #FOR(%SecondaryConditionalIcons)
      #INSERT(%StandardAddIconToProject(ABC),↓
%SecondaryConditionalIcon)
    #ENDFOR
  #ENDFOR
#ENDIF
#ENDFOR
#ENDAT
```

In the above code each control instance matches the template instance (you could have more than one control on a window). The code then adds any icons to the project, at the %CustomGlobalDeclarations point. Notice also the testing for file types, either a primary or a secondary file that has a many-to-one relationship. This keeps the icons constant for a given level of the tree. Lastly, the template checks for any conditional icon usage. Note also that each group called has the ABC parameter, as you are calling an ABC group.



I like loading my dialogs with dummy test data, so I can see if the code generated correctly (if at all!). Thus, my project tree for my test application appears as follows:

Figure 6-15:



These “icons” were simply entered into the dialogs so I have some known values to test against. I’ve entered similar data for conditions and colors, again for testing to ensure the correct code generates at the correct time and in the correct locations.

#### ☐ Local Data Embed

There are some local data declarations needed. In this case there is not too much to keep track of. If there is navigation from the toolbar and the local

QUEUE instance the relation tree gets its data. You can take care of all of this with one embed as follows:

```
#AT(%DataSection),PRIORITY(3500)
  #IF(%AcceptToolBarControl)
    %[20]DerivedInstance      CLASS(%ToolBarRelTreeType)
    TakeEvent                 PROCEDURE(<*LONG VCR>,WindowManager WM),,
                                VIRTUAL          %[20]NULL END
  %NULL
  #ENDIF
  %[20]TreeQueue              QRelTree
  %NULL
#ENDAT
```

The #IF tests to see if toolbar navigation is in use; if so, the template generates the Clarion code to declare and instantiate a derived class with an overridden method.

The second Clarion line (after the #ENDIF), declares the local instance of the TYPED QUEUE. The [20] means that no matter how short the %TreeQueue generated value is, generate its name and ensure that the next generated code has no more than 20 spaces after it (including the characters for the %TreeQueue value). This is simply a good way to keep labels aligned and counts only for neatness.

The %NULL is an internal symbol. In this instance it is used to align closing END statements and add empty lines to the generated source. Declarations bunched up together are hard to read.

### ❑ ThisWindow.Init()

Init is a locally derived ABC method. This is where all setup actions are performed. Since ThisWindow.Init() is executed only once per procedure run, this is an ideal place to add some embeds to set up the relation tree.

First, ensure the toolbar knows about the relation tree (wrapped for easier reading):

```
#AT(%WindowManagerMethodCodeSection,'Init','()',BYTE'),↓  
PRIORITY(8500),    WHERE(%AcceptToolBarControl AND↓  
%IsFirstInstance)  
ToolBar.AddTarget(%InstancePrefix, %TreeControl)  
%InstancePrefix.AssignButtons  
#ENDAT
```

Notice the WHERE statement. If there is a toolbar, and this is the first instance, then call the toolbar's `AddTarget` method so the it “knows” about the relation tree control. Then call the `AssignButtons` method, which is one of the methods of the `Relation Tree` class. The `%InstancePrefix` is the label of the local derived class. The default is `RTree1`.

The tree control should be set in contract mode as this is the expected state and it makes the window ready for use faster. Thus, you need this template code:

```
#AT(%WindowManagerMethodCodeSection,'Init','()',BYTE'),↓  
PRIORITY(7500)↓  
#!Start tree list in contracted mode  
%InstancePrefix.ContractAll()  
#ENDAT
```

The next bit of template code takes care of what to do with icons, if any:

```
#AT(%WindowManagerMethodCodeSection,'Init','()',BYTE'),PRIORITY(8  
500)  
#FIX(%Control,%TreeControl)  
#IF(%ControlHasIcon)  
  #FOR(%IconList),WHERE(%IconListType <> 'Variable')  
    #SET(%ValueConstruct,INSTANCE(%IconList))  
    #IF(%IconListType = 'Built-In')  
%TreeControl{PROP:IconList,%ValueConstruct} = %IconList  
    #ELSIF(%IconListType = 'File')  
%TreeControl{PROP:IconList,%ValueConstruct} = '~%IconList'  
    #ENDIF  
  #ENDFOR  
#ENDIF  
%TreeControl{PROP:Selected} = True  
#ENDAT
```

`#FIX` is similar to `#SET`. The `#FIX` statement fixes the current value of the multi-valued symbol (the first parameter) to the value contained in the second parameter. This is done so that one instance of the symbol may be referenced outside a `#FOR` loop structure, or so you can reference the symbols dependent upon the multi-valued symbol.

The `#FOR` statement generates the icon list for the tree control only if the icons are not contained in a variable. In this case, there will be one of two lines generated as icons may be stored internally or referenced externally.

The last bit generates code to ensure the relation list control is selected, or has focus.

The next embed in the `Init` method sets up the alerted keys for the keyboard:

```
#AT(%WindowManagerMethodCodeSection,'Init','()',BYTE'),PRIORITY(8750)
  #IF(%ExpandKeyCode)
%TreeControl{PROP:Alrt,255} = %ExpandKeyCode
  #ELSE
%TreeControl{PROP:Alrt,255} = CtrlRight
  #ENDIF
  #IF(%ContractKeyCode)
%TreeControl{PROP:Alrt,254} = %ContractKeyCode
  #ELSE
%TreeControl{PROP:Alrt,254} = CtrlLeft
  #ENDIF
  #IF(%UpdatesPresent)
%TreeControl{PROP:Alrt,253} = MouseLeft2
  #ENDIF
#ENDAT
```

All that code does is allow either the default values or the developer-assigned alert keys to be used. The last bit of code detects if the relation tree is updatable. If it is, then the code alerts the left button double click as the method to go to edit mode.

The generated code looks similar to this (depending on procedure type, name and other controls you may place on a window):

```
ThisWindow.Init PROCEDURE
ReturnValu     BYTE,AUTO

CODE
GlobalErrors.SetProcedureName('Main')
SELF.Request = GlobalRequest
ReturnValu = PARENT.Init()
IF ReturnValu THEN RETURN ReturnValu.
SELF.FirstField = ?RTree
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
SELF.AddItem(Toolbar)
CLEAR(GlobalRequest)
CLEAR(GlobalResponse)
RTree1.ContractAll
OPEN(Window)
SELF.Opened=True
Toolbar.AddTarget(RTree1, ?RTree)
RTree1.AssignButtons
?RTree{PROP:Selected} = True
?RTree{PROP:Alrt,255} = CtrlRight
?RTree{PROP:Alrt,254} = CtrlLeft
SELF.SetAlerts()
RETURN ReturnValu
```

I've added the emphasis on the lines that this template generated.

#### ❑ **ThisWindow.Kill()**

There may be times you don't need code placed in the `Kill` method. In this design there is. Since some reference assignments were made, it makes sense to use the `Kill` method to clean up after them. Or you may want a local destructor, it really is your choice. The code for this is simple:

```
#AT(%WindowManagerMethodCodeSection,'Kill','()',BYTE'),PRIORITY(8
200)
%InstancePrefix.QRT &= NULL
%InstancePrefix.LDQ &= NULL
#ENDAT
```

This generates code, fairly late in the method, to de-reference the two local `QUEUE` structures.

## Window Event Handling

---

If you need to trap for window events, then ensure your template produces the code for it.

### ❑ **EVENT:GainFocus**

The design of the tree list ensures that if the user switches to another window and then comes back, the code must ensure the tree list is properly updated. This is to ensure that the other window did not update something.

```
#AT(%WindowEventHandling,'GainFocus')
%InstancePrefix.CurrentChoice = CHOICE(%TreeControl)
GET(%TreeQueue,%InstancePrefix.CurrentChoice)
%InstancePrefix.NewItemLevel = %InstancePrefix.QRT.Level
%InstancePrefix.NewItemPosition = %InstancePrefix.QRT.Position
%InstancePrefix.RefreshTree()
#ENDAT
```

The %InstancePrefix symbol is used instead of SELF as this method is outside of the tree class. When the window gains focus, set the current choice (highlighted item). Using that value, GET that item from the tree queue. Assign the level and position values to the class properties and call the RefreshTree method.

## Control Event Handling

---

List controls generate events. Thus, you need to have template code that generates proper Clarion code to address this issue.

### ❑ **EVENT:NewSelection**

This section of code handles the NewSelection event:

```
#AT(%ControlEventHandling,%TreeControl,'NewSelection')
#IF(%UpdatesPresent OR %GiveExpandContractOption)
IF KEYCODE() = MouseRight
#IF(%UpdatesPresent AND %GiveExpandContractOption)

EXECUTE(POPUP('%InsertPopupText|%'ChangePopupText|%'DeletePopup
```

```

└─
Text|-|%'ExpandPopupText|%'ContractPopupText'))
    %InstancePrefix.AddEntry
    %InstancePrefix.EditEntry
    %InstancePrefix.RemoveEntry
    %InstancePrefix.ExpandAll
    %InstancePrefix.ContractAll
END
    #ELSIF(%UpdatesPresent)
EXECUTE(POPUP(||%'ChangePopupText|%'DeletePopupText'))
    %InstancePrefix.AddEntry
    %InstancePrefix.EditEntry
    %InstancePrefix.RemoveEntry
END
    #ELSE
EXECUTE(POPUP('%'ExpandPopupText|%'ContractPopupText'))
    %InstancePrefix.ExpandAll
    %InstancePrefix.ContractAll
END
    #ENDIF
END
    #ENDIF
#ENDAT

```

This code simply tests if the %UpdatesPresent or the %GiveExpandContractOption values are true, and if so, generates the IF KeyCode() structure. Within that structure, if both of those values are true, then execute a POPUP menu that allows, insert, change, delete and expand and contract the current node.

If only %UpdatesPresent, then generate only the insert, change and delete POPUP menu. Otherwise, generate only the expand and contract POPUP menu.

In all cases, an EXECUTE structure generates for each condition and calls a method of the local instance of the relation tree class.

#### ❑ **EVENT:Expanded and EVENT:Contracted**

If the user expands or contracts the tree, the template must generate code to account for those actions. The following code accomplishes these tasks:

```
#AT(%ControlEventHandler,%TreeControl,'Expanded')
%InstancePrefix.LoadLevel
#ENDAT
#!-----
#AT(%ControlEventHandler,%TreeControl,'Contracted')
%InstancePrefix.UnloadLevel
#ENDAT
```

### ❑ **EVENT:AlertKey**

The next event is the alert key handling. This template code handles this event:



```
#AT(%ControlEventHandler,%TreeControl,'AlertKey')
CASE KEYCODE()
  #IF(%ExpandKeyCode)
OF %ExpandKeyCode
  #ELSE
OF CtrlRight
  #ENDIF
  %TreeControl{PROPLIST:MouseDownRow} = CHOICE(%TreeControl)
  POST(EVENT:Expanded,%TreeControl)
  #IF(%ContractKeyCode)
OF %ContractKeyCode
  #ELSE
OF CtrlLeft
  #ENDIF
  %TreeControl{PROPLIST:MouseDownRow} = CHOICE(%TreeControl)
  POST(EVENT:Contracted,%TreeControl)
  #IF(%UpdatesPresent)
OF MouseLeft2
  %InstancePrefix.EditEntry
  #ENDIF
END
#ENDAT
```

## ○ Other Events

The generated code for the last two sections goes in the `ABC ThisWindow.TakeFieldEvent` method. Here is a partial listing of what it may look like:

```
CASE FIELD()
OF ?RTree
  CASE EVENT()
  OF Event:AlertKey
    CASE KEYCODE()
    OF CtrlRight
      ?RTree{PropList:MouseDownRow} = CHOICE(?RTree)
      POST(Event:Expanded,?RTree)
    OF CtrlLeft
      ?RTree{PropList:MouseDownRow} = CHOICE(?RTree)
      POST(Event:Contracted,?RTree)
    END
  END
END
ReturnValue = PARENT.TakeFieldEvent()
```

```
CASE FIELD()  
OF ?Rtree  
  CASE EVENT()  
    OF EVENT:Expanded  
      Rtree1.LoadLevel  
    OF EVENT:Contracted  
      Rtree1.UnloadLevel  
  END  
END
```

The last event handler is the `OtherEvent` embed. This is an ABC embed where you may process other events not handled normally, such as user-defined events or any event in which you may embed a `POST` message. Think of it as a catch-all embed for events not otherwise handled by the templates.

```
#AT(%ControlOtherEventHandling,%TreeControl)
  #SET(%ValueConstruct,%True)
  #FOR(%ControlEvent),WHERE(%ControlEvent = 'AlertKey')
    #SET(%ValueConstruct,%False)
  #ENDFOR
  #IF(%ValueConstruct)
CASE EVENT()
OF EVENT:AlertKey
  CASE KEYCODE()
    #IF(%ExpandKeyCode)
  OF %ExpandKeyCode
    #ELSE
  OF CtrlRight
    #ENDIF
    %TreeControl{PROPLIST:MouseDownRow} = CHOICE(%TreeControl)
    POST(EVENT:Expanded,%TreeControl)
    #IF(%ContractKeyCode)
  OF %ContractKeyCode
    #ELSE
  OF CtrlLeft
    #ENDIF
    %TreeControl{PROPLIST:MouseDownRow} = CHOICE(%TreeControl)
    POST(EVENT:Contracted,%TreeControl)
    #IF(%UpdatesPresent)
  OF MouseLeft2
    %InstancePrefix.EditEntry
    #ENDIF
  END
END
  #ENDIF
#ENDAT
```

The meaning of the above code should be obvious, as it is merely a different twist on what you've read before.

## The Generation of Method Code

---

The next task is to generate the methods. There are several ways to do this depending on the overall task to be accomplished. First, let's get the toolbar out of the way:

```
#AT(%LocalProcedures),WHERE(%AcceptToolBarControl)
%DerivedInstance.TakeEvent PROCEDURE(<*LONG VCR>,WindowManager
WM)
CODE
CASE ACCEPTED()
OF ToolBar:Bottom TO ToolBar:Up
SELF.Control{PROPLIST:MouseDownRow} = CHOICE(SELF.Control)
EXECUTE(ACCEPTED()-ToolBar:Bottom+1)
%InstancePrefix.NextParent
%InstancePrefix.PreviousParent
%InstancePrefix.NextLevel
%InstancePrefix.PreviousLevel
%InstancePrefix.NextRecord
%InstancePrefix.PreviousRecord
END
#EMBED(%ReltreeToolBarDispatch,'RelObjTree ToolBar
Dispatch'),%ActiveTemplateInstance,HIDE
ELSE
PARENT.TakeEvent(VCR,%WindowManagerObject)
END
#ENDAT
```

This template code generates only when there is a need to navigate the tree list by the toolbar navigation method. Notice that depending on the toolbar button pressed, the EXECUTE structure calls one of the derived RelationTree methods. The %LocalProcedures symbol means the local objects for this procedure.

## The RelationTree Methods

---

The RelationTree methods generated by the template handle all of the instance-specific tasks.

### ○ AssignButtons

The method assigns the toolbar buttons to the tree list control. The template code to do this is as follows:

```
#AT(%TreeClassMethodCodeSection,%ActiveTemplateInstance,↵
'AssignButtons','()'),WHERE(%AcceptToolBarControl)
#EMBED(%AssignToolBarButtons,'Assign Toolbar↵
Buttons'),%ActiveTemplateInstance,HIDE
ToolBar.SetTarget(%TreeControl)
#ENDAT
RefreshTree
```

The template code below generates applicable code to refresh the tree:

```
#AT(%TreeClassMethodCodeSection,↵
%ActiveTemplateInstance,'RefreshTree','()')
FREE(%TreeQueue)
SELF.Load%Primary()
IF SELF.NewItemLevel
  SELF.CurrentChoice = 0
  LOOP
    SELF.CurrentChoice += 1
    GET(%TreeQueue,SELF.CurrentChoice)
    IF ErrorCode() THEN BREAK.
    IF ABS(SELF.QRT.Level) <> ABS(SELF.NewItemLevel) THEN CYCLE.
    IF SELF.QRT.Position <> SELF.NewItemPosition THEN CYCLE.
    SELECT(%TreeControl,SELF.CurrentChoice)
    BREAK
  END
END
#ENDAT
```

In the class, this is stub method. Almost all of this code could be in the class, but the primary file is not known to the class and neither is the tree control or the tree's data queue. You could add some properties to the class to represent these items and use those properties. In this case, the template code for the `RefreshTree` is not needed. However, you will need to change some of the class methods. If you feel a little adventurous, consider this an exercise.

## ○ ContractAll / ExpandAll

These two template embeds are rather interesting, and demonstrate the concept of what code goes in a class and what code the templates are responsible for.

```
#AT(%TreeClassMethodCodeSection,␣
    %ActiveTemplateInstance,'ContractAll','()')
SELF.Load%Primary()
#ENDAT
#AT(%TreeClassMethodCodeSection,␣
    %ActiveTemplateInstance,'ExpandAll','()')
SELF.Load%Primary()
SELF.LoadAll = False
#ENDAT
```

The point here is that this code generates after the parent call. If you inspect the method code in the class, it appears as incomplete. And this template code appears incomplete too. But after generation, the two bodies of code are really “merged”.

## ○ LoadLevel

This embed produces the code responsible for loading one level of the tree. It gets called after a user presses the plus (+) button on a tree.

```
#AT(%TreeClassMethodCodeSection,␣
%ActiveTemplateInstance,'LoadLevel','()')
SELF.CurrentChoice = %TreeControl{PROPLIST:MouseDownRow}
GET(%TreeQueue,SELF.CurrentChoice)
IF ~SELF.QRT.Loaded
  SELF.QRT.Level = ABS(SELF.QRT.Level)
  PUT(%TreeQueue)
  SELF.QRT.Loaded = True
  SELF.LDQ.LoadedLevel = ABS(SELF.QRT.Level)
  SELF.LDQ.LoadedPosition = SELF.QRT.Position
  ADD(SELF.LDQ,SELF.LDQ.LoadedLevel,SELF.LDQ.LoadedPosition)
  EXECUTE(ABS(SELF.QRT.Level))
  #FOR(%TreeLevelFile)
    BEGIN
    #IF(%TreeLevel=1)
    #IF(%PrimaryKey)
    REGET(%PrimaryKey,SELF.QRT.Position)
    #ELSE
    REGET(%Primary,SELF.QRT.Position)
    #ENDIF
    #ELSE
    REGET(%TreeLevelFile,SELF.QRT.Position)
    #ENDIF
    SELF.FormatFile(%TreeLevelFile)
  END
#ENDFOR
END
PUT(%TreeQueue)
EXECUTE(ABS(SELF.QRT.Level))
#FOR(%TreeLevelFile),WHERE(INSTANCE(%TreeLevelFile) > 1)
  SELF.Load%TreeLevelFile()
#ENDFOR
END
#ENDAT
```

The assumption is that the mouse down row is accurate enough to get the current choice. This is retrieved from the data queue structure. If the `Loaded` flag is not set, then an assignment of the current level is passed to the queue structure and written back. The `Loaded` flag is set to true. The loaded queue structure gets its data from the data queue.

Based on the level, an EXECUTE structure is entered. For every file in the tree, the template builds a BEGIN END structure. Inside these structures, there are only two lines of code generated, one to REGET and the other to format that file's data.

Inside the BEGIN structure, the template tests to see if this is the first file in the table schematic. If it is, then another test to see if a key is also in the schematic. If there is, then a REGET statement is generated based on the key. Otherwise, it is file based. The remaining files in the table schematic use the file based REGET form.

After the format call, the changed queue data is written back to the queue. Then a final EXECUTE structure generates to load the current child file.



## ○ UnloadLevel

This embed is almost the same as the `LoadLevel` embed. The purpose of this embed is to remove data from the queue structures when the user presses the minus (-) button on the tree.

```
#AT(%TreeClassMethodCodeSection,↓
%ActiveTemplateInstance,'UnloadLevel','()')
  SELF.CurrentChoice = %TreeControl{PROPLIST:MouseDownRow}
  GET(%TreeQueue,SELF.CurrentChoice)
  IF SELF.QRT.Loaded
    SELF.QRT.Level = -ABS(SELF.QRT.Level)
    PUT(%TreeQueue)
    SELF.QRT.Loaded = False
    SELF.LDQ.LoadedLevel = ABS(SELF.QRT.Level)
    SELF.LDQ.LoadedPosition = SELF.QRT.Position
    GET(SELF.LDQ,SELF.LDQ.LoadedLevel,SELF.LDQ.LoadedPosition)
    IF ~ErrorCode()
      DELETE(%LoadedQueue)
    END
    EXECUTE(ABS(SELF.QRT.Level))
  #FOR(%TreeLevelFile)
    BEGIN
    #IF(%TreeLevel=1)
    #IF(%PrimaryKey)
      REGET(%PrimaryKey,SELF.QRT.Position)
    #ELSE
      REGET(%Primary,SELF.QRT.Position)
    #ENDIF
    #ELSE
      REGET(%TreeLevelFile,SELF.QRT.Position)
    #ENDIF
    SELF.FormatFile(%TreeLevelFile)
  END
#ENDFOR
END
PUT(%TreeQueue)
SELF.CurrentLevel = ABS(SELF.QRT.Level)
SELF.CurrentChoice += 1
LOOP
  GET(%TreeQueue,SELF.CurrentChoice)
  IF ErrorCode() THEN BREAK.
  IF ABS(SELF.QRT.Level) <= SELF.CurrentLevel THEN BREAK.
  DELETE(%TreeQueue)
END
END
#ENDAT
```

This embed makes the same assumption `LoadLevel` does, in that the mouse down row property is accurate enough to detect which line of the tree to affect. The data queue item is retrieved by that value.

If the `Loaded` flag is true, the `Level` field is set to negative. The `Loaded` flag is then set to false. The data queue's fields are then primed and an attempt to get a record follows. If there is not an error, the current record is deleted.

The `EXECUTE` structure is then built exactly as `LoadLevel`. Unlike `LoadLevel`, there are two extra lines of code generated after the `PUT`. The current level is saved and the current choice is incremented by 1. This keeps the highlight bar at the same position.

The final act is to generate a `LOOP` structure. Its purpose is to remove all data items under the current node.

### ○ FormatFile

At first glance, this embed does not appear to do much work.

```
#AT(%TreeClassMethodCodeSection,␣
%ActiveTemplateInstance,'FormatFile','(*FILE xFile)')
  #FOR(%TreeLevelFile),WHERE(INSTANCE(%TreeLevelFile) = 1)
EXECUTE(ABS(SELF.QRT.Level))
BEGIN
  #INSERT(%FormatPrimary)
END
#ENDFOR
#FOR(%TreeLevelFile),WHERE(INSTANCE(%TreeLevelFile) > 1)
BEGIN
  #INSERT(%FormatSecondary)
END
#ENDFOR
END
#ENDAT
```

This template code, to put it simply, generates an `EXECUTE` structure to determine which file in the schematic needs its data formatted so that it looks nice. It uses two `#GROUP` structures to accomplish this; `%FormatPrimary` and `%FormatSecondary`.

The reason for these two groups is that the template code to do either task is quite complex and slightly different. Therefore, it makes sense for a template programmer to use this technique. It does keep certain tasks segmented so that any maintenance you may need to do is easier.

These groups are responsible for ensuring the icons, conditional icons, colors, and conditional colors are properly set. Also, icons could be external file resources or linked into the current project. Feel free to inspect this code in the file `CBGroups.tpw`, located in the template folder.

## ○ Dynamic New Methods

This embed's responsibility is to ensure that the new methods have code for them. The new methods are placed in the new methods dialog, as discussed earlier.

```
#AT(%NewMethodCodeSection,%ActiveTemplateInstance,↓
%ClassItem,%NewMethods)
#IF (~(VAREXISTS(%PrimaryFlag)))
  #DECLARE(%PrimaryFlag, LONG)
#ENDIF
#IF (~(VAREXISTS(%NewInstance)))
  #DECLARE(%NewInstance, LONG)
#ENDIF
```

The above code looks to see if a symbol is declared. If not, it declares it. The reason for this is that this template code is “re-entrant”, meaning that it gets executed many times. If this check is not done, you’ll get complaints during source generation that a symbol is already declared.

```
#SET(%NewInstance,%NewInstance + 1)
#IF(%PrimaryFlag=0)
  #SET(%PrimaryFlag,%PrimaryFlag + 1)
  #SELECT(%TreeLevelFile,1)
  #IF(%TreeTitle)
    SELF.QRT.Display = '%TreeTitle'
    SELF.QRT.Loaded = False
    SELF.QRT.Position = ''
    SELF.QRT.Level = 0
    #INSERT(%IconGroup),NOINDENT
    #CALL(%ColorGroup)
  ADD(%TreeQueue)
#ENDIF
```

This section of template code increments the local symbols. In the Clarion language, simple incrementation is easy:

```
NewInstance += 1
```

However, the templates do not support that type of syntax. The `#SET` command must be used. It assigns a value to a user-defined symbol.

Next is a check to see if the `%PrimaryFlag` is zero, which it should be the first time this template code executes. If it is, then another `#SET` command increments the current value. On the next pass, the template code after the `#ELSE` executes (a bit further in this code).

The first file in the schematic is obtained with the `#SELECT` statement. The `#SELECT` statement is really a form of `#FIX`, in that it fixes a symbol to an instance.

If there is a `%TreeTitle` (an entry on the tree dialog), then the code sets the display accordingly. Since a tree title is not based on a file, it is not loaded, nor does it have any level values.

The next two lines of template code are different, but they are not different in functionality. `#INSERT` has an optional attribute, `NOINDENT`. This means any code in this `#GROUP` generates in the same column as the line above it. `#CALL` is the same as the `#INSERT` with the `NOINDENT` attribute.

```
Access:%Primary.UseFile()
  #IF(%PrimaryKey)
SET(%PrimaryKey)
  #ELSE
SET(%Primary)
  #ENDIF
LOOP
  #EMBED(%BeforePrimaryNext,'Relational Object Tree, ↵
Before NEXT on primary file'),%ActiveTemplateInstance,↵
MAP(%ActiveTemplateInstance,↵
%ActiveTemplateInstanceDescription),LEGACY
  IF Access:%Primary.Next() <> Level:Benign
    IF Access:%Primary.GetEOF()
      BREAK
    ELSE
      POST(EVENT:CloseWindow)
      RETURN
    END
  END
  #EMBED(%AfterPrimaryNext,'Relational Object Tree, ↵
After NEXT on primary file'),↵
```

```
%ActiveTemplateInstance,MAP(%ActiveTemplateInstance,↓  
%ActiveTemplateInstanceDescription),LEGACY
```

The above code uses ABC method calls. If there is a key in the schematic, then the templates generate code to set read access by key. Otherwise, the access is set by record order.

A LOOP structure is then generated. Just inside the LOOP is an embed point. This has the LEGACY attribute on it, meaning that it won't appear in the embed tree until the developer presses the legacy button on the embed tree toolbar.

The next section of code tests an ABC call to see if reading the next record returns anything besides a LEVEL:Benign (no error). If it does, then test for end of file using an ABC method call. If it has reached the end of file, then break out of this loop. Any other error is deemed fatal and a close window event is posted to the current window, in effect closing the procedure.

Next is another legacy embed, which happens after a record is read successfully.

```
#IF(%PrimaryFilter)  
  IF ~(%PrimaryFilter) THEN CYCLE.  
#ENDIF  
  SELF.QRT.Loaded = False  
  #IF(%PrimaryKey)  
    SELF.QRT.Position = POSITION(%PrimaryKey)  
  #ELSE  
    SELF.QRT.Position = POSITION(%Primary)  
  #ENDIF  
  SELF.QRT.Level = %TreeLevel  
  #IF(%TreeChildFile)  
    #INSERT(%TreeChildGroup),NOINDENT  
    #SET(%PrimaryFlag,%True)  
  #ELSE  
    SELF.FormatFile(%TreeLevelFile)  
    ADD(%TreeQueue,POINTER(%TreeQueue) + 1)  
  #ENDIF
```

This section of code first tests to see if any filter is applied. If so, then code generates to cycle to the top of the loop if not filtered. The next #IF tests to see if a key is in effect, and if so, generate the correct code to use the key to determine the current position. Otherwise, the code uses the file to determine the current position.

If this is a child file, then the code to handle child files is inserted. Then the template assigns a true value to the %PrimaryFlag symbol to flag that primary file

processing is completed. If this is not a child file, the template calls the `FormatFile` method, with the current file passed as a parameter. After returning from this method call, it adds any queue fields values to the queue structure, in sorted order.

The following template code executes when the `%PrimaryFlag` is not zero.

```
#ELSE
  #FOR(%TreeLevelFile),WHERE↓
  (INSTANCE(%TreeLevelFile) = %NewInstance)
    #FIX(%File,%TreeLevelFile)
    #FIX(%Secondary,%TreeLevelFile)
    #FIX(%Relation,%TreeParentFile)
    #FIX(%Key,%FileKey)
    #FOR(%RelationKeyField)
      #IF(%RelationKeyField)
      %RelationKeyFieldLink = %RelationKeyField
      #ELSE
        #FIX(%KeyField,%RelationKeyFieldLink)
        #IF(%KeyFieldSequence = 'ASCENDING')
      CLEAR(%RelationKeyFieldLink)
      #ELSE
      CLEAR(%RelationKeyFieldLink,1)
      #ENDIF
    #ENDIF
  #ENDFOR
```

The `#FOR` loop is an interesting bit of code. There is a `WHERE` clause attached to it, which is a template way of filtering. In this case, the filter is if the `%TreeLevelFile` instance equals the `%NewInstance` value. Remember, the `%NewInstance` is incremented every time this template code executes. Next a series of `#FIX` commands. `#FIX` is like the Clarion assignment of:

```
File = TreeLevelFile
Secondary = TreeLevelFile
```

and so forth. Next is another `#FOR` loop for each relating key field. If the current field is part of a relating key, then the template generates as many lines of code as necessary to prime the relating or matching values. Otherwise, if not a relating field, it assigns the linking field to the key field. This is for those occasions where a custom relation is defined in the table schematic.

The next logic branch determines how to generate the CLEAR statement, the default low value, or the option second parameter to clear to high values in the case of descending field sort order.

```
Access:%File.UseFile()
SET(%FileKey,%FileKey)
LOOP
  #EMBED(%BeforeSecondaryNext,'Relational Object Tree,↓
    Before NEXT on secondary file'),↓
    %ActiveTemplateInstance,%Secondary,↓
    MAP(%ActiveTemplateInstance,↓
      %ActiveTemplateInstanceDescription),LEGACY
  IF Access:%File.Next()
    IF Access:%File.GetEOF()
      BREAK
    ELSE
      POST(EVENT:CloseWindow)
      RETURN
    END
  END
  #FOR(%RelationKeyField),WHERE(%RelationKeyField)
  IF %RelationKeyFieldLink <> %RelationKeyField THEN BREAK.
#ENDFOR
#EMBED(%AfterSecondaryNext,'Relational Object Tree,↓
  After NEXT on secondary file'),↓
  %ActiveTemplateInstance,%Secondary,↓
  MAP(%ActiveTemplateInstance,↓
    %ActiveTemplateInstanceDescription),LEGACY
```

The above template code is pretty much the same as before, but this time, the code acts on the current secondary file. The following code acts on any child files to the current child file and again, what it does follows my earlier explanation, but on the primary file.

```
#IF(%SecondaryFilter)
IF ~(%SecondaryFilter) THEN CYCLE.
#ENDIF
SELF.QRT.Loaded = 0
SELF.QRT.Position = POSITION(%TreeLevelFile)
SELF.QRT.Level = %TreeLevel
#IF(%TreeChildFile)
    #INSERT(%TreeChildGroup),NOINDENT
#ELSE
    SELF.FormatFile(%TreeLevelFile)
    ADD(%TreeQueue,POINTER(%TreeQueue) + 1)
#ENDIF
#ENDFOR
#ENDIF
END
#ENDAT
```

This embed is quite busy! If not for some calls to `#GROUPs`, this part of the template would be quite difficult to read. I've omitted the comments here, but if you open the source files you'll see that almost every line of code, template and target language has comments.

You can see that the template makes many decisions about what code to generate, based on the user settings.

## Updating the RelationTree

---

There are three embeds for the methods which handle updates to the relation tree:

- AddEntryServer
- EditEntryServer
- DeleteEntryServer

These three embeds are almost identical, so a discussion of one is enough to cover the others. The only significant differences are the edit modes.

```
#AT(%TreeClassMethodCodeSection,%ActiveTemplateInstance,↵
'AddEntryServer','()')
#IF(%UpdatesPresent)
    #EMBED(%BeginAddEntryRoutine,'Relational Object Tree,↵
Beginning of Add Record'),%ActiveTemplateInstance,↵
```



```
MAP(%ActiveTemplateInstance,↵
%ActiveTemplateInstanceDescription),NOINDENT
SELF.CurrentChoice = %TreeControl{PROPLIST:MouseDownRow}
GET(%TreeQueue,SELF.CurrentChoice)
CASE ABS(SELF.QRT.Level)
```

The above template code first checks to see if there was an update procedure entered in the developer dialog. If not, this template code quits and no target code generates. If so, then the template declares an embed point. %UpdatesPresent is a symbol that is set to %True in the #ATSTART section.

Next, the template generates target code to detect the current row and get its level. The level value is wrapped in a Clarion CASE statement.

If there is a procedure entered in %PrimaryUpdate by the developer, then the template generates target code to call the update procedure:

```
#IF(%PrimaryUpdate)
OF 0
  #EMBED(%BeforePreparingRecordOnAdd,↵
'Relational Object Tree, Before Preparing for Add'),↵
  %ActiveTemplateInstance,%Primary,↵
  MAP(%ActiveTemplateInstance,↵
%ActiveTemplateInstanceDescription),LEGACY
Access:%Primary.PrimeRecord
GlobalRequest = InsertRecord
  #EMBED(%BeforeCallingUpdateOnAdd,'Relational Object Tree,↵
Before Update on Add'),↵
  %ActiveTemplateInstance,%Primary,↵
  MAP(%ActiveTemplateInstance,↵
%ActiveTemplateInstanceDescription),LEGACY
%PrimaryUpdate
IF GlobalResponse = RequestCompleted
  SELF.NewItemLevel = 1
  SELF.NewItemPosition = POSITION(%File)
  SELF.RefreshTree()
END
  #EMBED(%AfterCallingUpdateOnAdd,↵
'Relational Object Tree, After Update Procedure on Add'),↵
  %ActiveTemplateInstance,%Primary,↵
```

```
        MAP(%ActiveTemplateInstance,↵
        %ActiveTemplateInstanceDescription),LEGACY
    #ENDIF
```

The above code handles calling the update procedure for the primary file. After a successful return, it sets the new item level and position, and calls the RefreshTree method.

The template code now must check for any records in child files:

```
    #FOR(%TreeLevelFile),WHERE(INSTANCE(%TreeLevelFile) > 1)
    #SUSPEND
```

At this point a #SUSPEND statement is encountered. All target code from here to its matching #RESUME statement, and which is prefixed with #?, is potentially not generated. The reason is to build, conditionally, any remaining OF structures – the condition being if there are child files to process. By that I mean any calls to edit procedures for each child file in the schematic.

```
        #SET(%ValueConstruct,INSTANCE(%TreeLevelFile)-1)
    #?OF %ValueConstruct
        #IF(INSTANCE(%TreeLevelFile) = ITEMS(%TreeLevelFile))
        #SET(%ValueConstruct,%ValueConstruct + 1)
```

What the above code is doing is simply passing the level number to %ValueConstruct. This makes the OF statement for each child file.

The following section is conditional, and used only at the bottom of the CASE statement as it “groups” the last two files. In other words, it REGETs the parent, then LOOPS through all the matching child records. It then calls the matching %SecondaryUpdate procedure for that child file.

```
    #?OROF %ValueConstruct
        #?LOOP WHILE ABS(SELF.QRT.Level) = %ValueConstruct
            #?SELF.CurrentChoice -= 1
            #?GET(%TreeQueue,SELF.CurrentChoice)
        #?UNTIL ERRORCODE()
        #ENDIF
        #?REGET(%TreeParentFile,SELF.QRT.Position)
            #EMBED(%BeforePreparingRecordOnAdd,↵
            'Relational Object Tree, Preparing Record for Add'),↵
        %ActiveTemplateInstance,%Primary,↵
```

```

        MAP(%ActiveTemplateInstance,↵
        %ActiveTemplateInstanceDescription),LEGACY
    #?GET(%TreeLevelFile,0)
        #FIX(%File,%TreeLevelFile)
        #FIX(%Relation,%TreeParentFile)
    #?CLEAR(%File)
        #FOR(%FileKeyField),WHERE(%FileKeyField          AND
%FileKeyFieldLink)
    #?%FileKeyField = %FileKeyFieldLink
        #ENDFOR
    #?Access:%File.PrimeRecord(1)
    #?GlobalRequest = InsertRecord
        #FIX(%File,%Primary)
        #FIX(%Secondary,%TreeLevelFile)
        #EMBED(%BeforeCallingUpdateOnAdd,↵
'Relational Object Tree, Before Update on Add'),↵
%ActiveTemplateInstance,%Secondary,↵
MAP(%ActiveTemplateInstance,↵
%ActiveTemplateInstanceDescription),LEGACY

```

As a final act, it does what the previous update calls did, which is test for successful completion of the update procedure (e.g. the user did not cancel the edit).

```

%SecondaryUpdate
    #?IF GlobalResponse = RequestCompleted
        #SET(%ValueConstruct,INSTANCE(%TreeLevelFile))
    #?SELF.NewItemLevel = %ValueConstruct
    #?SELF.NewItemPosition = POSITION(%Secondary)
    #?SELF.RefreshTree()
    #?END
        #EMBED(%AfterCallingUpdateOnAdd,↵
'Relational Object Tree, After Update ↵
Procedure on    Add'),↵
%ActiveTemplateInstance,%Secondary,↵

```

```
        MAP(%ActiveTemplateInstance,↓
        %ActiveTemplateInstanceDescription),LEGACY
        #RESUME
    #ENDFOR
END
#ENDIF
#ENDAT
```

The main difference in the other methods (Edit and Delete), is the passing of the edit signal to `GlobalRequest` and issuing of the `WATCH` statement, plus the different descriptions of the `#EMBED` statements.

## Other Control Templates

---

There are two control templates in addition to the one that places the list control on a window. These two templates are discussed here.

### ❑ RelObjTreeUpdateButtons

This template places three buttons on the window. This template also appears only if the `RelationalTree` control template exists on the window.

```
#CONTROL(RelObjTreeUpdateButtons,↓
'Update buttons for a Relational Object Tree'),↓
DESCRIPTION('Update buttons for Relational Tree for '↓
& %Primary),REQ(RelationalTree)
CONTROLS
    BUTTON('&Insert'),AT(,45,14),USE(?Insert)
    BUTTON('&Change'),AT(50,0,45,14),USE(?Change)
    BUTTON('&Delete'),AT(50,0,45,14),USE(?Delete)
END
```

The above code declares a control template and the controls to place on the window. Notice the `REQ` attribute on the `#CONTROL` statement. This means that whatever template is named there, it must be on the window (or report) before this template appears on any list.

The `CONTROLS` statement declares that what follows are the controls to place on the window. This is usually pure Clarion code. Notice the first control is missing the first two parameters of the `AT` statement. This means that once the developer picks this control template, when the mouse moves over the window, the cursor

changes to a cross-hair shape. When the left button is pressed, the controls are placed at the spot where the cross-hair is, one after the other.

If you author a control template with many controls, one useful tip is to require multiple mouse clicks to place all of the controls where you want. To do this, simply have the fourth or fourth control (for example), omit the first two parameters of its AT statement.

```
#ATSTART
  #DECLARE(%HelpControl)
  #FOR(%Control)
    #IF(UPPER(EXTRACT(%ControlStatement,'STD',1))='STD:HELP')
      #SET(%HelpControl,%Control)
      #BREAK
    #ENDIF
  #ENDFOR
```

The previous section of code starts an #ATSTART section. Remember, an #ATSTART statement specifies template code to execute before the #CONTROL generates it code. That actually applies to other template types, but the current context is control templates.

The next section following is simply code to determine if there is a control on the window with the STD:Help attribute, a Clarion equate that calls Windows standard Help when the control is accepted. If the code finds such a control, it breaks out of the loop. If it does not find one, the loop expires once all controls on the window have been inspected.

```
#DECLARE(%AddControl)
#DECLARE(%EditControl)
#DECLARE(%RemoveControl)
#FOR(%Control),WHERE(%ControlInstance =
%ActiveTemplateInstance)
#CASE(%ControlOriginal)
#OF('?Insert')
#OROF('?Add')
#SET(%AddControl,%Control)
#SET(%ValueConstruct,EXTRACT(%ControlStatement,'BUTTON',1))
#IF(%ValueConstruct)
#SET(%ValueConstruct,SUB(%ValueConstruct,2,
LEN(%ValueConstruct)-2))
#ELSE
#SET(%ValueConstruct,SUB(%ControlOriginal,2,
LEN(%ControlOriginal)-1))
#ENDIF
#SET(%InsertPopupText,%ValueConstruct)
```

The above template code declares symbols for each of the three controls for the three possible edit methods.

The `#FOR` loop looks through all controls where the control instance matches the template instance. This technique ensures that these controls get unique names when is more than one tree list control on the window.

The `#CASE` structure looks for the use variables each button might have – in this instance, possible synonyms for each of the edit actions. Once it finds one with either use variable labels, it assigns that control to the `%AddControl` (or one of the others – see below).

The `%ControlStatement` contains the control's declaration statement (and all attributes). `EXTRACT` is looking for the control type, in this case a button. Since the first attribute is a `BUTTON`, the 1 is the third parameter. In plain English, his statement means “Please give me the button’s text attribute.”

If `%ControlStatement` finds button attributes (`%ValueConstruct` has a value if it does), then the code makes `%ValueConstruct` contain only the actual attributes. If `%ValueConstruct` is empty, then the template use a slightly different method to assign the attributes.

The final line of this section of code is to assign the value of the construct to the `%InsertPopupText`. Thus, whatever the text is, the pop-up menu has the same text.

The rest of the `#ATSTART` section does the same thing for the `Edit` and `Delete` controls, and is omitted here.

## ❑ Control Event Handling

The buttons need event processing, that is, they need to know what they should do when pressed by the user. The following template code handles this:

```
#AT(%ControlEventHandler,%AddControl,'Accepted')
%TreeControl{PropList:MouseDownRow} = CHOICE(%TreeControl)
%InstancePrefix.AddEntry()
#ENDAT
```

When the user presses the button control that does adding or inserting, an `EVENT:Accepted` is generated. The `CHOICE` statement returns the row of whatever is highlighted. This is passed to the list control, then the code makes a call to the `AddEntry` method that belongs to the current object name of the `RelationTree`.

The change and delete button controls do the same thing, except for the name of the control and the method call. These method calls are in the parent class, which set the edit mode and then call `UpdateLoop` method. Before I get to that method, I must take a small detour for toolbar navigation.

## ❑ Toolbar Issues

This template code accounts for toolbar-controlled editing, but does not require a toolbar. If there is no toolbar, then the corresponding code is bypassed.

```
#AT(%ReltreeToolbarDispatch,%ActiveTemplateParentInstance),␣
WHERE(%AcceptToolbarControl)
OF Toolbar:Insert TO Toolbar>Delete
  SELF.Control{PROPLIST:MouseDownRow} = CHOICE(SELF.Control)
  EXECUTE(ACCEPTED()-Toolbar:Insert+1)
  %InstancePrefix.AddEntry()
  %InstancePrefix.EditEntry()
  %InstancePrefix.DeleteEntry()
END
#ENDAT
```

Notice the `#AT` location, this is actually a hidden (from the developer) embed. Embeds with the `HIDE` attribute are for template use only. They provide a place to embed code under template control only. You can think of this as the same functionality as `#GROUPS`, but without actually defining one.

Below is the template code where the above code gets embedded (I think you can spot where):

```
#AT(%LocalProcedures),WHERE(%AcceptToolBarControl)
%DerivedInstance.TakeEvent  PROCEDURE(<*LONG VCR>,WindowManager
WM)
CODE
CASE ACCEPTED()
OF Toolbar:Bottom TO Toolbar:Up
SELF.Control{PROPLIST:MouseDownRow} = CHOICE(SELF.Control)
EXECUTE(ACCEPTED() - Toolbar:Bottom + 1)
%InstancePrefix.NextParent()
%InstancePrefix.PreviousParent()
%InstancePrefix.NextLevel()
%InstancePrefix.PreviousLevel()
%InstancePrefix.NextRecord()
%InstancePrefix.PreviousRecord()
END
#EMBED(%ReltreeToolBarDispatch,␣
'Relational Object Tree Toolbar Dispatch'),␣
%ActiveTemplateInstance,HIDE
ELSE
PARENT.TakeEvent(VCR,%WindowManager)
END
#ENDAT
```

Notice the #AT location. %LocalProcedures is a symbol that contains the class declaration for toolbar management. The template code for this is as follows:

```
#AT(%DataSection),PRIORITY(3500)
#IF(%AcceptToolBarControl)
%[20]DerivedInstance CLASS(%ToolBarRelTreeType)
TakeEvent PROCEDURE(<*LONG VCR>,WindowManager WM),DERIVED
%[20]NULL END
%NULL
#ENDIF
%[20]TreeQueue QRelTree
%[20]LoadedQueue LoadedQ
%NULL
#ENDAT
```

This is the code that declares the local instance of the %ToolBarRelType, an ABC class. %AcceptToolBarControl is a check box on the dialog for the relation tree. If the developer checks this box, the condition is true and the code generates.



TakeEvent is an overridden virtual method. It is this method that %LocalProcedures refers to. The [20] means to generate the label normally and then add spaces, up to 20 characters (includes both label and spaces). This is a way to ensure the code aligns in the proper column. %NULL is an internal symbol and a great way to generate leading white space. This makes the closing END statement align with the PROCEDURE statement.

Therefore, if you declare a method in a class (overridden or not), you *must* follow up by writing the code for the method. That is where the #AT(%LocalProcedures) comes in.

%DerivedInstance contains the label of the local object. %InstancePrefix is the symbol for the local tree class. Since the methods listed there are not part of this class, you cannot use SELF; you must use the label of the local instance for the tree class. This also illustrates how a class can call an unrelated method in another class.

## ❑ Special Conditions

There is a special condition that should be taken into account. What if the developer adds the HIDE or DISABLE attribute to any of the edit buttons? The following template code addresses this situation:

```
#AT(%BeginAddEntryRoutine)
  #IF(%Control=%TreeControl)
    #IF(%AddControl)
      IF %AddControl{PROP:Disable}
        RETURN
      #IF (~%AcceptToolbarControl)
        ELIF %AddControl{PROP:Visible} = False
          RETURN
        #ENDIF
      END
    #ENDIF
  #ENDIF
#ENDAT
```

I am showing only the add routine as the change and delete are nearly identical. If you go back to the template code for AddEntryServer, you will find an #EMBED labeled %BeginAddEntryRoutine. This template inserts the code at that spot.

So if the current control is the current tree control, and if there is an %AddControl present, then generate code to test to see if the control is currently disabled. If it is, the generated code quits there with the RETURN statement.

Next, the template code checks to see if the toolbar control is not present, and if so, generates code to test if the control button is hidden and again, RETURN right away as there is nothing else to do.

### ❑ Assignment of Edit Buttons

The last thing for this control template is to ensure the class knows which button controls are used for editing.

```
#AT(%AssignToolBarButtons,%ActiveTemplateParentInstance),↵
WHERE(%AcceptToolBarControl)
  #IF (%RemoveControl)
    %DerivedInstance.DeleteButton = %RemoveControl
  #ENDIF
  #IF (%AddControl)
    %DerivedInstance.InsertButton = %AddControl
  #ENDIF
  #IF (%EditControl)
    %DerivedInstance.ChangeButton = %EditControl
  #ENDIF
  #IF (%HelpControl)
    %DerivedInstance.HelpButton = %HelpControl
  #ENDIF
#ENDAT
```

Again, this code generates into an #EMBED point for the AssignButtons method, but only when the %AcceptToolBarControl check box is set. The template checks each control to see if it exists, and if so, generates Clarion code to assign the control buttons to the property for each.

### ❑ RelObjTreeExpandContractButtons

This control template adds the ability to expand and collapse the entire tree. It is not very big.

```
#CONTROL(RelObjTreeExpandContractButtons,↵
'Expand/Contract buttons for a Relational Object Tree'),↵
DESCRIPTION('Expand/Contract buttons for a Relational↵
Object Tree for ' & %Primary),REQ(RelationalTree)
CONTROLS
  BUTTON('&Expand All'),AT(,45,14),USE(?Expand)
  BUTTON('Co&ntract All'),AT(50,0,45,14),USE(?Contract)
END
```

The above declares the template and its controls. It uses the same concepts discussed earlier to populate these controls on a window.

```
#ATSTART
  #DECLARE(%ExpandControl)
  #DECLARE(%ContractControl)
  #FOR(%Control),WHERE(%ControlInstance =
%ActiveTemplateInstance)
    #CASE(%ControlOriginal)
      #OF('?Expand')
        #SET(%ExpandControl,%Control)
        #SET(%ValueConstruct,EXTRACT(%ControlStatement,'BUTTON',1))
        #SET(%ExpandPopupText,SUB(%ValueConstruct,2,↓
LEN(%ValueConstruct)-2))
      #OF('?Contract')
        #SET(%ContractControl,%Control)
        #SET(%ValueConstruct,EXTRACT(%ControlStatement,'BUTTON',1))
        #SET(%ContractPopupText,SUB(%ValueConstruct,2,↓
LEN(%ValueConstruct)-2))
      #ENDCASE
    #ENDFOR
#ENDAT
```

The above code is a “pre-process” as with earlier discussions about `#ATSTART`. First, two symbols are declared. That’s followed by a `#FOR` loop, restricted to the current instance of this control. `%ControlOriginal` is the original field equate label of the control as listed in the control template from which it came. The developer can change the field equate at any time, but the `%ControlOriginal` never changes, and so is a safe variable to use.

In the case where the symbol is “Expand”, then the `%ExpandControl` gets the `%Control`’s field equate, even if this field equate was changed by the developer. The code next extracts the button’s text and places it in `%ValueConstruct`. The pop-up text is also extracted and placed in its symbol, `%ExpandPopupText`. The same process is done for the **Contract** button. The template is now ready to proceed as all of its prep work is completed.

```
#AT(%ControlEventHandler,%ExpandControl,'Accepted')
%TreeControl{PropList:MouseDownRow} = CHOICE(%TreeControl)
%InstancePrefix.ExpandAll()
#ENDAT
```

The above code does the event processing when the **Expand** control is accepted. If that event happens, it then calls the method to expand the entire tree.

```
#AT(%ControlEventHandler,%ContractControl,'Accepted')
%TreeControl{PropList:MouseDownRow} = CHOICE(%TreeControl)
%InstancePrefix.ContractAll()
#ENDAT
```

This code does the event processing when the **Contract** control is accepted. If that event happens, then call the method to collapse the entire tree.

That takes care of the details of creating the template wrapper. I've covered a lot of ground, and don't worry if you weren't able to take it all in on first reading. As I said at the beginning of this chapter, you may want to come back and reread this material as your skill level progresses. Getting a class to interact fully with the Clarion development environment is one of the most challenging template tasks you're likely to encounter. But once you've mastered the basics, you'll be able to duplicate your success easily with other classes.

In the next chapter I'll explain how to fully use this new Relation Tree template.

# Chapter 7

---

## TEMPLATE USER GUIDE

Now that the template is done, it's time to put it to use. This chapter covers each prompt, and also serves as a tutorial. In this case, the tutorial shows how to fill in the prompts so the resulting tree list looks and behaves just like the one found in the **Invoice** example.

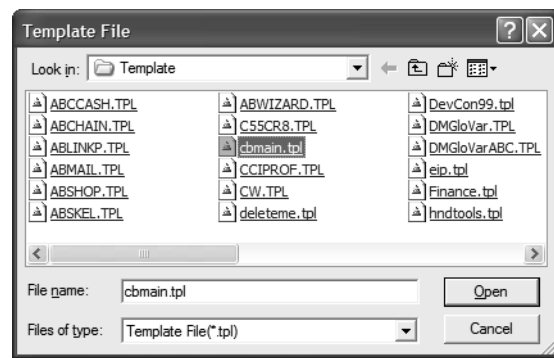
### Registration

---

You cannot use the template until you register the template. Open the registry and press **Register**. If the registry button is disabled, you are in multi-user mode. You must uncheck this box under the **Application Setup** dialog.

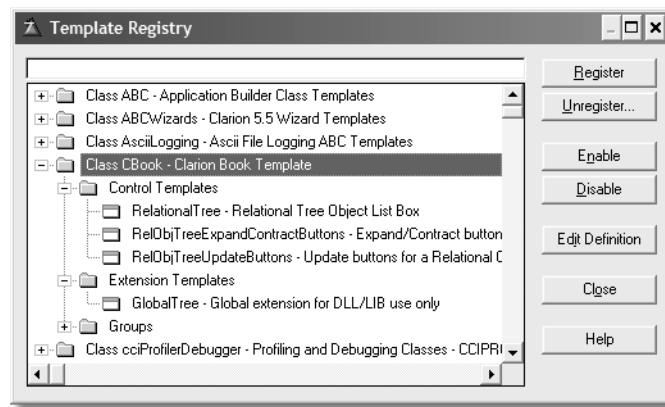
Find the `cbmain.tpl` file as shown below:

Figure 7-1:



When the template is registered, it will appear similar to the following figure (this will differ depending on which templates you've registered):

Figure 7-2:



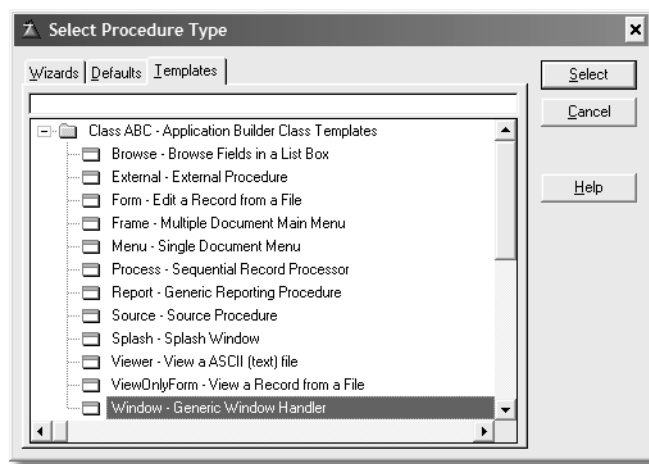
Press **OK** to close the template registry. The template is now ready for use.

## Adding the control template

You may add the relation tree control template to any procedure with a window. The best use is with either a **Browse** or **Window** procedure. For simplicity's sake, I'll use a plain window with no buttons.

Press **INS** anywhere in an existing application (even if you just created a new one). Give this new procedure any name you wish. Choose **Window** for the procedure type.

Figure 7-3:

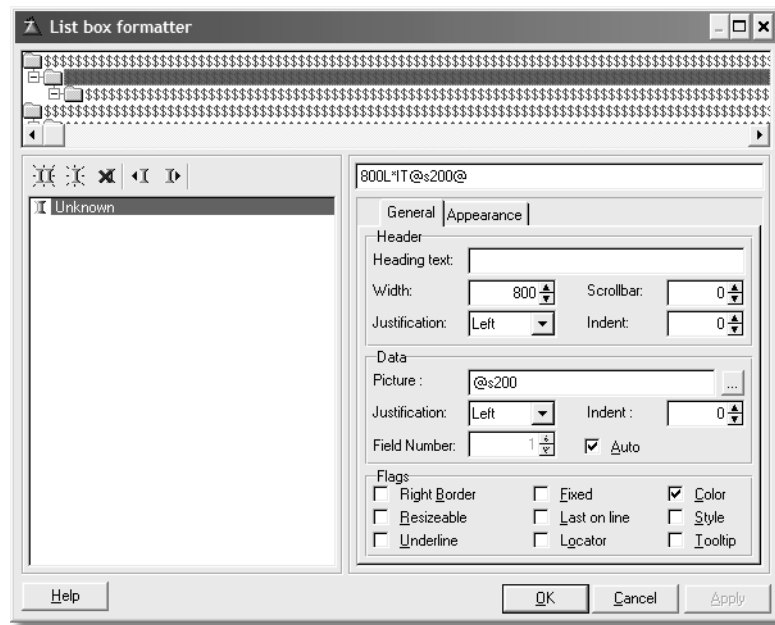


Press **Select** to accept this choice. You are now presented with a procedure dialog. Press **Window**. You are now presented with a few choices. Normally, you want an MDI Child Window, as a procedure like this is called from a **Frame** procedure. It does not really matter however as long as it is a **Window** type procedure.

Resize the window to fill the 800x600 outline so you have some room to work. From either the **Control Template** button on the **Tools** toolbar or from the populate control template option on the main menu, choose **Relational , tree object list box**. Once done the dialog closes and when your mouse cursor is over the window, it changes to a cross hair.

Once you determine where you want to drop the control, press the left mouse button. The list box formatter appears.

Figure 7-4:



Press **OK**, making no changes, as you do not use this tool. Resize the control so it fills a little more than half the window vertically with equal margins on both sides horizontally. At the moment, you have an empty list box.

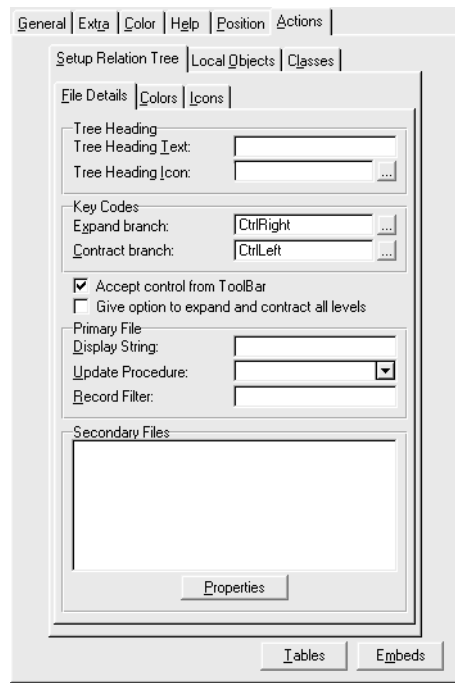
Right-click on the list box and choose **Actions**.



## Setting up the Relation Tree

The **File Details** tab is where you'll do most of your work. These are the prompts for the control template.

Figure 7-5:



### ○ Tree Heading Text

Enter the optional text for the root node of the tree.

**Example:** Enter CUSTOMERS ' ' ORDERS

### ○ Tree Heading Icon

Enter an icon file name or press the ellipsis button to lookup an icon file. Note that all icons mentioned here are in the %root%\examples\invoice folder.

**Example:** Enter File.ico

☐ **Expand Branch**

Enter the keystroke that is used to expand the current node of the tree. You may optional press the ellipsis button to open the key stroke dialog. The default is **Ctrl-Right**.

**Example:** Leave as-is.

☐ **Contract Branch**

Enter the keystroke that is used to contract or collapse the current node of the tree. You may optionally press the ellipsis button to open the key stroke dialog. The default is **Ctrl-Left**.

**Example:** Leave as-is.

☐ **Accept control from Toolbar**

If you use toolbars, check this box to add navigation and editing functions from the VCR toolbar. The default is to not use toolbar navigation.

**Example:** Check this box.

☐ **Give option to expand or contract all levels**

This enables the user to expand or collapse the entire tree. Since the tree is file loaded, a large amount of data may cause long delays. For smaller amounts of data, this is usually fine. The default is to allow this action.

**Example:** Leave the box checked.

☐ **Display String**

Enter an expression for the primary file. This uses any valid Clarion expressions to construct how this file's data appears on the list box.

**Example:** Enter `CLIP(CUS:FirstName) & ' ' & CLIP(CUS:LastName) |  
& ' ' & FORMAT(CUS:CustNumber,@P(#####)P)`

---

**NOTE** In Clarion 6 and later, press **F10** to show the zoom box. It can display more text than what is allowed on the template dialog, thus making it easier to enter long strings or expressions.

---

### ○ Update Procedure

Enter or choose from the drop list the name of the edit procedure. This procedure is usually a form procedure.

**Example:** Enter UpdateCustomers.

### ○ Record Filter

If the primary file in the tree should be filtered in some way, enter the filter expression here. Any valid Clarion expression that could be used as a filter is valid.

**Example:** Leave this blank as there is no filter in this example.

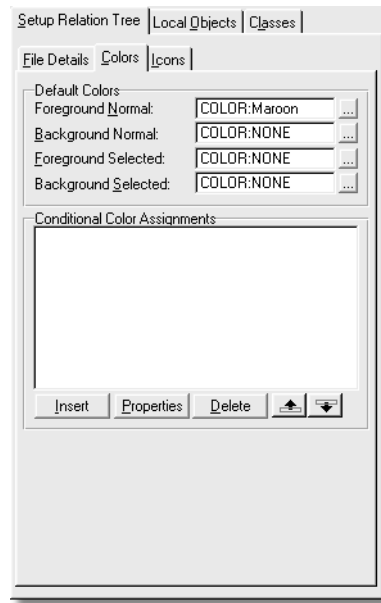
### ○ Secondary files

All files listed as related to the primary file are listed here. Since there is nothing yet in the table schematic, there is nothing to do. You'll come back to this later.

### ❑ Colors tab

This is where you will enter any colors for this level of the tree. It looks like this:

Figure 7-6:



The default colors are as follows:

#### ○ Foreground Normal

Enter a color equate or a hex value for the normal foreground color. You can also press the ellipsis button to open the color dialog box and pick the color you wish.

**Example:** Enter `COLOR:Maroon`

#### ○ Background Normal

Enter a color equate or a hex value for the normal background color. You can also press the ellipsis button to open the color dialog box and pick the color you wish.

**Example:** Leave this blank to default to no color (black).

## ○ Foreground Selected

Enter a color equate or a hex value for the foreground selected (highlight bar is on the current row) color. You can also press the ellipsis button to open the color dialog box and pick the color you wish.

**Example:** Leave this blank to default to no color.

## ○ Background Selected

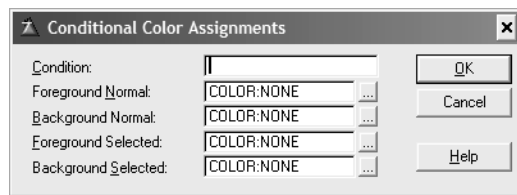
Enter a color equate or a hex value for the background selected (highlight bar is on the current row) color. You can also press the ellipsis button to open the color dialog box and pick the color you wish.

**Example:** Leave this blank to default to no color.

## □ Conditional Color Assignments

This is a list of all conditions that could apply to this node of the tree. You may press **Insert** to add a new condition, **Properties** to edit a condition, or **Delete** to remove a condition. The **Up** or **Down** buttons change the order of the conditions if you have more than one condition. Using any of the edit buttons shows this dialog:

Figure 7-7:



## ○ Condition

Enter a valid Clarion expression for the condition. The condition must be true for the colors to take effect. An example expression would be `CUS:InArrears`. Any value in this field would be true.

**Example:** Leave blank as there are no conditions for this file.

○ **Foreground Normal**

Enter a color equate or a hex value for the normal foreground color. You can also press the ellipsis button to open the color dialog box and pick the color you wish.

**Example:** Leave this blank to default to no color (black).

○ **Background Normal**

Enter a color equate or a hex value for the normal background color. You can also press the ellipsis button to open the color dialog box and pick the color you wish.

**Example:** Leave this blank to default to no color (black).

○ **Foreground Selected**

Enter a color equate or a hex value for the foreground selected (highlight bar is on the current row) color. You can also press the ellipsis button to open the color dialog box and pick the color you wish.

**Example:** Leave this blank to default to no color.

○ **Background Selected**

Enter a color equate or a hex value for the background selected (highlight bar is on the current row) color. You can also press the ellipsis button to open the color dialog box and pick the color you wish.

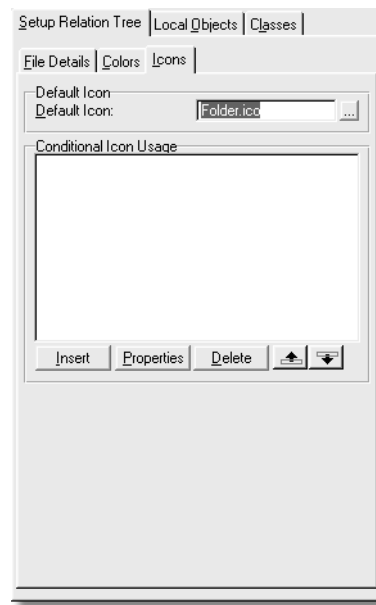
**Example:** Leave this blank to default to no color.

When you are finished, press **OK** to close this dialog and save any changes or **Cancel** to abort any edits and close this dialog. You may add another condition, or edit an existing condition, or delete a condition at this point if you wish.

## Icons

This dialog allows you specify any icons for the tree list.

Figure 7-8:



### Default icon

Enter an icon name or press the ellipsis to look up an icon file.

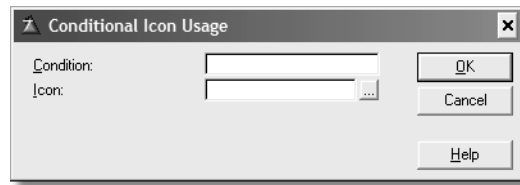
**Example:** Enter `folder.ico`

### Conditional Icon Use

This is a list of all conditions that could apply to this node of the tree. You may press **Insert** to add a new condition, **Properties** to edit a condition, or **Delete** to remove a condition. The **Up** or **Down** buttons change the order of the conditions if

you have more than one condition. Using any of the edit buttons shows this dialog:

**Figure 7-9:**



### ○ Condition

Enter a valid Clarion expression for the condition. The condition must be true for the colors to take effect. An example expression would be `CUS:InArrears`. Any value in this field other than 0 would evaluate as True.

**Example:** Leave blank as there are no conditions for this file.

### ○ Icon

Enter an icon name or press the ellipsis to look up an icon file.

**Example:** Leave blank as there are no conditional icons for this level in the tree.

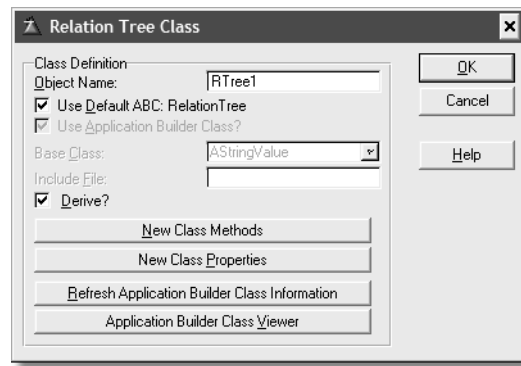
When you are finished, press **OK** to close this dialog and save any changes or **Cancel** to abort any edits and close this dialog. You may add another condition, or edit an existing condition, or delete a condition at this point if you wish.



## ❑ Local Objects

This tab shows the details for the local object. Press **Relation Tree Class** and this dialog appears:

Figure 7-10:



### ○ Object Name

This is the default name for the local instance of this class. You may enter another name for this object if you wish, or leave it as is.

**Example:** Leave the default name.

### ○ Use default ABC: RelationTree

Leave this setting checked if this is the correct parent class. If not, clear the check box to allow you to choose a different parent class.

**Example:** Leave checked.

### ○ Use Application Builder Class?

This prompt enables only if the **Use Default** checkbox is cleared. The default is checked, meaning you want to pick another ABC class as the parent.

**Example:** Not applicable.

### ☐ Base class

This drop list shows all the ABC classes to choose from. Pick a new class from this list. This is a drop list only if the previous check box is set. If it is cleared, then this control is an entry where you enter the name of the parent class.

**Example:** Not applicable.

### ☐ Include File

This entry control is enabled only if the **Use ABC** check box is cleared. Enter the name of the include file where the class definition can be found.

**Example:** Not applicable.

### ☐ Derive?

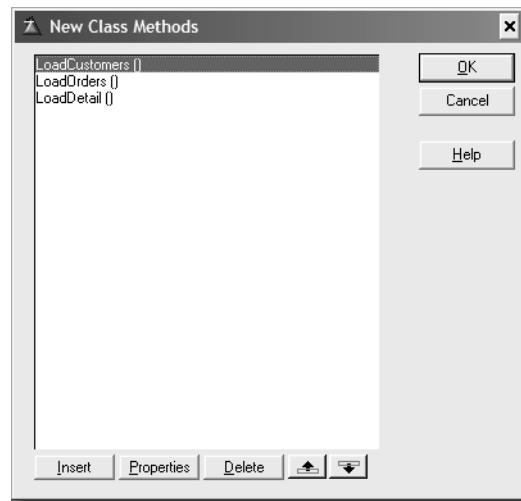
This check box is enabled automatically when one or more files are placed in the schematic. The files in the schematic must be related to the primary file. Setting this check enables the following two buttons.

**Example:** Leave this setting as you find it. It is checked if you've added a file, unchecked if not (you will add a file later).

## ○ New Class Methods

Pressing this button shows the following dialog:

Figure 7-11:



You may or may not have any methods listed here. The template automatically adds these methods based on the files in the table schematic. Since the template automatically keeps this list, you should not add any new methods of your own or change the names or parameters.

Press **Cancel** and confirm you wish to cancel (if prompted).

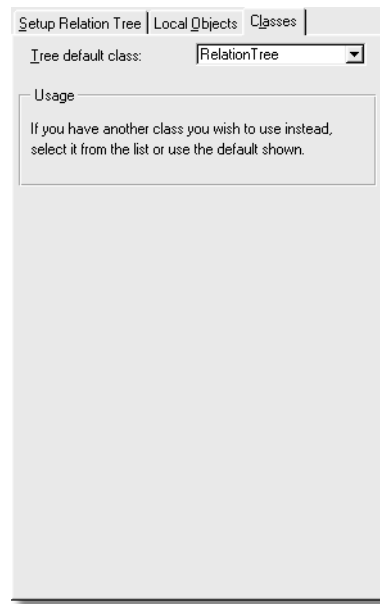
The rest of the buttons are covered in the Clarion documentation.

Press **Cancel** again and confirm you wish to cancel (if prompted).

### ❑ **Classes**

The **Classes** tab shows the name of the parent class. You may change it to another ABC compliant class that handles trees if you wish.

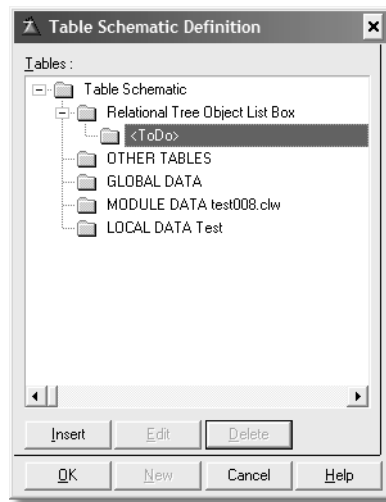
**Figure 7-12:**



Adding files to the **Table Schematic**

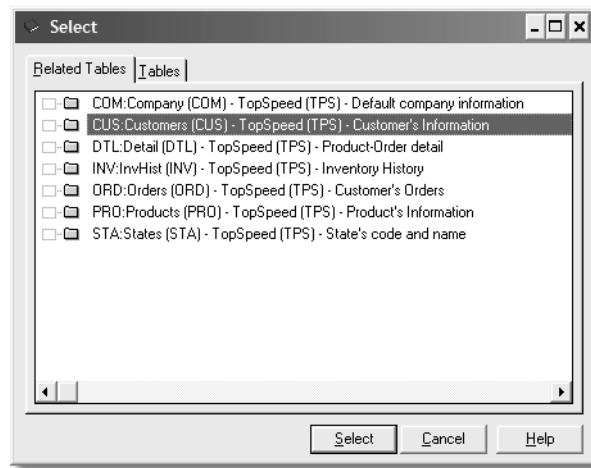
If you are at the procedure properties dialog, press the **Table** button. If you are at the application tree, Right-Click and choose **Tables**.

Figure 7-13:



Press **Insert** and the following dialog displays:

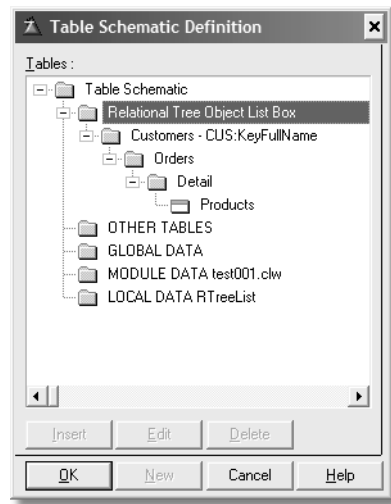
Figure 7-14:



Highlight Customers and press **Select**. The Customer file is now in the tree. Press **Edit** and choose KeyFullName. Press **Select**. With the Customer file selected,

press **Insert** and select **Orders**. Press **Select**. Repeat for **Detail** and **Products**. The table schematic should appear as follows:

Figure 7-15:



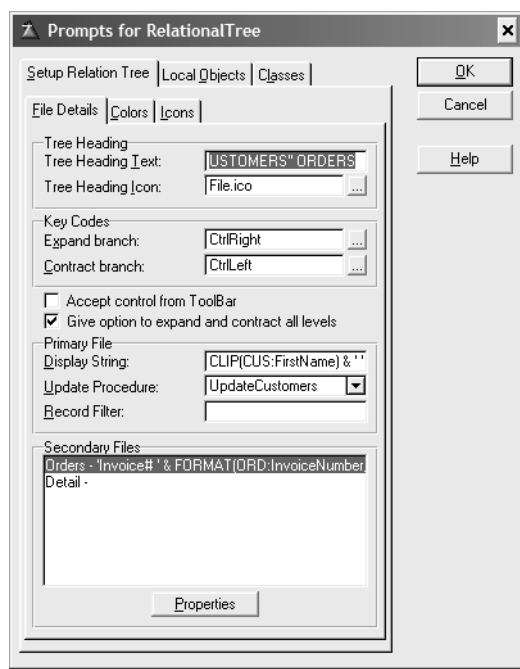
Press **OK** to close the table schematic dialog. This adds all the files needed for the tree list. Press **OK** to close the procedure dialog (if still open).

### ❑ Finishing the Tree List

You do need to fill in how other files appear in the tree list, but you do not have to always go back to the window formatter to work on the tree dialog. Right-click on the procedure and choose **Extensions**. Or you may use the right side pane of the application view (Clarion 5.5 and later). Expand the extension node of the tree. For either method, select **Tree Structure Related to Customers**, if not already selected.

The dialog will change to reflect the files added to the table schematic.

Figure 7-16:



You now see the secondary files listed. But you won't find `Products` in this list. Only files that have a 1:MANY relationship to a child file are listed. The same rule applies the **New Methods** dialog. Each dialog for the secondary files is the same as the primary file. To open those dialogs, highlight a file and press **Properties**. Fill in the prompts as follows:

Open the secondary dialog for the `Orders` file and fill in these prompts as described below.

## ○ Display String

```
Enter: 'Invoice# ' & FORMAT(ORD:InvoiceNumber,@P#####P) |
      &', Order# ' & FORMAT(ORD:OrderNumber,@P#####P) & |
      ', (' & LEFT(FORMAT(ORD:OrderDate,@D1)) & '))'
```

## ○ Update Procedure

Enter or select from the drop down `UpdateOrders`.

Select the **Color** tab.

**Foreground Normal:** Enter `COLOR:Navy`

Select the **Icon** tab.

**Default Icon:** Enter or select `invoice.ico`.

Press **OK** to close and save your changes. Open the secondary dialog for `Detail`.

**Display String:** Leave this blank as the string required is too big to fit in the maximum entry control size of 255 characters. You'll use an embed instead.

**Update Procedure:** Enter or select from the drop down `UpdateDetail`.

Select the **Color** tab.

**Foreground Normal:** Enter `COLOR:Green`

You need to enter a conditional color assignment. Press **Insert**.

**Condition:** Enter `DTL:BackOrdered = TRUE`

**Foreground Normal:** Enter `COLOR:Red`

Press **OK** to save these edits. Select the **Icon** tab.

**Default Icon:** Enter or select `star.ico`.

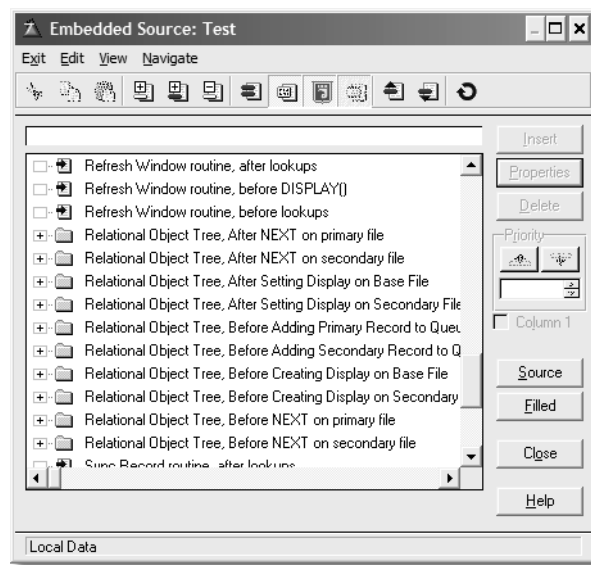
Press **OK** to close this dialog and save your changes. Press **OK** to close any other remaining dialogs until you are back to the application tree.



## ❑ The Embeds

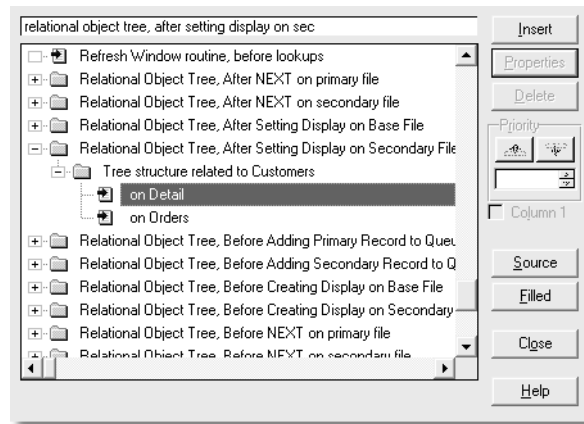
Open the embed tree by your favorite method. There are many embeds with the LEGACY attribute in the templates. This means that they are visible only when the legacy button is pressed on the embed tree toolbar.

Figure 7-17:



Using the locator, find **Relational Object Tree, After Setting Display on Secondary File**. Expand this tree.

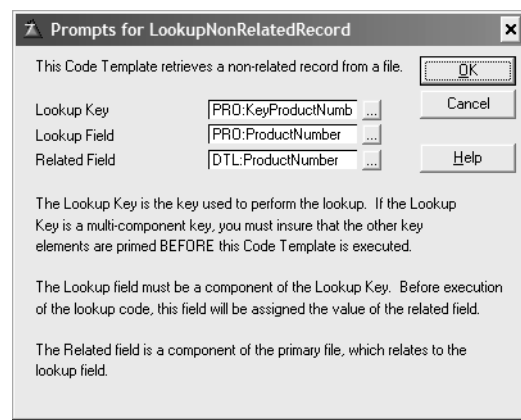
Figure 7-18:



Find the `Detail` file. You need to insert two embeds:

Press **Insert** and choose **Lookup non-related record**. Fill in the prompts as shown in the next figure.

Figure 7-19:



Press **OK** to close the dialog when done. This does a lookup into the `Product` file.

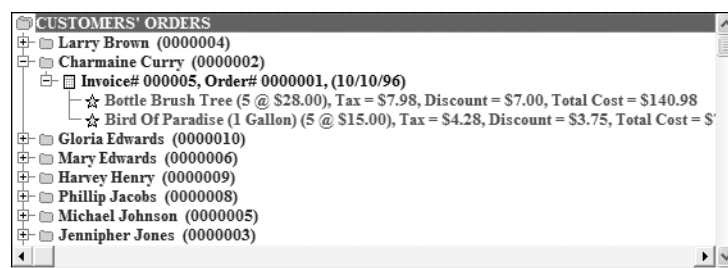
You need to add another embed. Press **Insert** and choose **Source**. Enter the source as follows:

```
!Format DisplayString
SELF.QRT.Display = CLIP(PRO:Description) & ' ('|
& CLIP(LEFT(FORMAT(DTL:QuantityOrdered,@N5))) & ' @ '|
& CLIP(LEFT(FORMAT(DTL:Price,@N$10.2))) & '), Tax = '|
& CLIP(LEFT(FORMAT(DTL:TaxPaid,@N$10.2))) & ', Discount = '|
& CLIP(LEFT(FORMAT(DTL:Discount,@N$10.2))) & ', ' & |
'Total Cost = ' & LEFT(FORMAT(DTL:TotalCost,@N$14.2))
```

Select **Exit** when done and save your edits.

Save all of your work and test. You should see something like the following figure when the test app runs:

Figure 7-20:



## Dynamic Link Library Issues

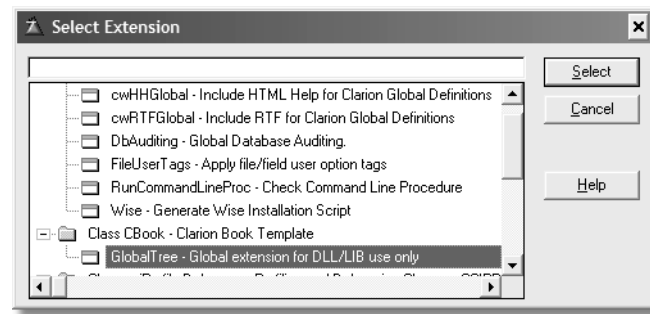
Since this template is a pure control template, it will not, by itself, work in a DLL setting. This is because this class is local and used only when a specific control is placed on a window.

If you tried to link in an ABC DLL that contains all the ABC class definitions, you will get a link error for each method call. This is because there is nothing to set up the exports for this class.

This is why a global extension exists. This extension belongs where the ABC classes are defined to your application such as a root or global DLL. It is used only once as that is all you need.

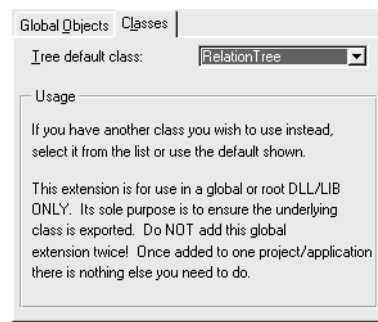
Open the Global properties dialog and press **Extensions**. Choose the only global extension in the **Class CBook** node:

Figure 7-21:



Press **Select** to choose the extension. This adds the extension template to the DLL application. Look at the following figure:

Figure 7-22:



This template ensures that the `RelationTree` class and its methods are available for export, meaning they are available for later applications to call once this DLL is inserted into the application.

You'll need to ensure that your DLL projects are correctly set up. If you follow the instructions for doing this for an ABC application, that is all you need. This template is designed to follow ABC's lead, thus nothing extra is required by the developer.

## **Congratulations!**

---

By working through the material in this book you've gained some incredibly valuable skills for Clarion development. You've learned the basics of object-oriented programming, and you've discovered the magic of virtual methods, which is at the heart of Clarion's ABC class library. And you've seen how easy it can be to convert procedural code into maintainable, usable classes, using the example of Clarion's Relation Tree.

You've also learned how to write templates that automate the process of using your own custom ABC-compatible classes with the Application Generator. And finally you've seen how you can use all this technology within the AppGen, again with the example of the Relation Tree conversion.



# APPENDIX A: GETTING SUPPORT

## Getting the source code

---

Source code is available for download via:

<http://www.clarionmag.com/books/poic/>

If you do not have access to the Internet, please contact the publisher at the following address:

CoveComm Inc.  
1036 McMillan Ave  
Winnipeg, MB  
R3M 0V8  
Tel: 204-943-5165

## Errata

---

Corrections to the book are listed at:

<http://www.clarionmag.com/books/poic/>

If you find an error in the text, please report it via the above web page.





# INDEX

## Symbols

%ROOT% 148

\_CBDLLMode\_ 115

\_CBLinkMode\_ 115

## A

ABC 71–106

and OOP 4

beta releases 2

BrowseClass

AddField 88

derived 83

Init

derived 84

SetQueueRecord 87, 88

BrowseQueue

SetViewPosition 89

class reader 147

compatible classes 113

custom global classes 101–106

ErrorClass 95–104

AddErrors 97

deriving 100

major functions 96

SetErrors 97

SetFatality 97

SubString 99

Throw 97

FieldPairsClass

AssignLeftToRight 88

FileManager

code generation 76

PrimeAutoInc 75, 81

global embeds 74

global objects 72

hidden classes 78

RelationManager

code generation 76

relationship of classes to templates 83

using custom classes 73

WindowManager

Init 88, 104, 178

Kill 181

TakeAccepted 90

TakeCloseEvent 90

TakeCompleted 90

TakeEvent 90, 209

TakeFieldEvent 90, 92, 185

TakeNewSelection 89, 90

TakeRejected 90

TakeSelected 90

TakeWindowEvent 90

abstraction 9

Application Builder Classes

*see* ABC

automatic constructor

*see* constructor, automatic

automatic destructor

*see* destructor, automatic

## B

binding

early vs late 36

late 36

## C

CLASS 10

DERIVED 63

example 10

LINK 48, 53

MODULE 48

THREAD 167

TYPE 12

class

- see also* object
- base 24
- code simplicity 133
- creating 120
- derived 24
- designing 107–112
- exporting 158
- generic 126
- method 11
  - adding with template 163
- multiple instances 49
- planning for template wrapper 131
- property 11
- single global instance 157
- template 146
- vs routine 19, 107
- colon
  - in data labels 114
- composition
  - definition 26
- construct method name 21
- constructor 21
  - and derived class 29
  - automatic 21, 127
    - creating structures 22
    - limitations 21
    - vs Init 128
    - vs Kill 128
- converting to ABC 107–143

## D

- DctInit 77
- DctKill 77, 78
- debugger
  - and virtual methods 59
- derivation 24, 54
  - and constructors, destructors 29
  - definition 24
  - example 25
- DERIVED attribute 35, 63

- destruct
  - method name 21
- destructor 21
  - and derived class 29
  - automatic 21, 127
    - limitations 21
- DISPOSE 51
- DLL
  - and global extension 235
- dot syntax 15

## E

- embed point
  - adding with template 170
  - choosing 90
  - hidden 207
  - local data 178
- encapsulation 9, 45
  - extreme 23
- error code
  - checking for in ABC 95
- EVENT
  - Accepted 67, 90, 207
  - AlertKey 184, 185
  - CloseWindow 67, 196, 199
  - Contracted 183, 185
  - Expanded 183, 185
  - GainFocus 182
  - NewSelection 89, 90, 182
  - OpenWindow 51
- examples
  - AlaMode class 27
  - American class 29
  - AppleCheeseCake class 64
  - ApplePie class 27, 28, 33, 60, 64
  - ClassA 31
  - ConstructClass 52
  - DerivedClass 54
  - Dutch class 29, 33, 60
  - Employee class 18

- ErrorMgr 100
  - ExtendPrinter class 41
  - FileClass 10, 22, 25
  - Fred class 17
  - Grandmas class 29
  - IceCream class 27
  - IFax interface 38, 40
  - IPieType interface 62
  - MicroWave class 20
  - MyClass 23, 28, 31
  - OverrideClass 55
  - PieClass 62
  - PieView class 63
  - PlayWave class 45–49, 57
  - PrettyPieClass 63
  - Printer class 38, 39
  - RelationTree
    - AddEntry 141
    - AddEntryServer 140
    - AssignButtons 140
    - ContractAll 143
    - DeleteEntry 141
    - EditEntry 141
    - EditEntryServer 140
    - ExpandAll 143
    - LoadLevel 134
    - NextLevel 136
    - NextParent 135
    - NextRecord 139
    - NextSavedLevel 136
    - PreviousLevel 136
    - PreviousParent 135
    - PreviousRecord 139
    - PreviousSavedLevel 137
    - RefreshTree 134
    - RemoveEntryServer 140
    - UnloadLevel 135
    - UpdateLoop 142
  - RelationTree class 114–143
  - Virtual1 class 57
  - Virtual2 class 57
  - WorkClass 50–52
  - EXECUTE 67, 188, 191, 194
    - BEGIN/END structure 192
- F**
- file
    - converting 104
  - FileClass 43
  - FilesInit 77, 79
  - formula editor 174
  - FUNCTION
    - deprecated 50
  - function overloading 45
    - and polymorphism 32
- G**
- green bar 84
- I**
- INCLUDE
    - ONCE 49
  - inheritance 24
    - definition 24
    - multiple 26
  - instantiation 11
    - and TYPE 12
    - options 15
  - INTERFACE 37–??, 37, ??–41
    - and properties 38
    - and virtual method 38, 62
    - declaration order 39
    - definition 37
    - implementing 40
    - method naming 41
    - multiple 41
- L**
- late binding 36
  - legacy code

converting to ABC 107–143

LEVEL

Benign 197

LINK attribute 53

list formatter 85

listbox

formatting 87

local derived method 36

LONGPATH 46

## M

MEMBER

empty 48

method

PRIVATE

*see* PRIVATE

virtual

*see* virtual method

modules

repopulating 113

multiple inheritance

and INTERFACE 41

in Clarion 41

## N

NEW 51

and ASSERT 51

## O

object

*see also* class

global 155

example 12

local

example 14

module

example

SELF

*see* SELF

Object Oriented Programming

*see* OOP

OOP

*see also* class

*see also* object

and ABC 4

base class 24

benefits 44

code maintenance 21

coding conventions 16

converting to 44

definition 7

source files 16

theory of 7–42

vs procedural code 16

override

constructor

automatic 30

definition 28

destructor

automatic 30

method 26

overriding 54

## P

PARENT

and constructor 31

and destructor 31

definition 30

polymorphism

and Clarion for DOS 32

and function overloading 32

definition 32

POPUP 183

POST 67

PRIVATE 23, 96, 98

and ABC 23

vs PROTECTED 116

PROC attribute 50

procedure

data

visible to class 37

project

creating 124

PROP

Active 135, 136, 139

Alrt 132, 180, 181

At 132

Disable 209

Font 55

Format 132

From 51, 130

HScroll 132

IconList 132, 179

Selected 132, 179, 181

Text 67

VCR 132

Visible 209

VScroll 132

PROPLIST

MouseDownRow 137, 138, 139, 142, 185, 187,  
188, 191, 193, 201, 207, 208, 211,  
212

PROTECTED 96, 98, 99, 127

vs PRIVATE 116

when to use 27

## Q

QUEUE

disposing 52

instantiating in constructor 22

## R

reference variable 50

REGET 192

relation tree

ABC vs Legacy 110

edit functions 112

navigation 111

REPLACE 99

## S

scope 37

local 15

SECTION 57

SELF

definition 18

example 18

## T

Team Topspeed 2

template

#CONTROL

with many components 205

class wrapper 146

writing 145–212

embed point

*see* template, language, #EMBED

family 152

function

*see* #GROUP

initialization 153

language

#ADD 166

#APPLICATION 150

vs #EXTENSION 150

#ASSERT 155, 167

#AT 162, 168, 169

#ATSTART 153, 156, 173, 205, 207

#BOXED 149

#BUTTON 156

#CALL 156, 162, 167, 168, 169, 170, 195

vs #INSERT 162, 196

#CASE 206

#CLASS 174

#CONTROL 151, 172, 204, 210

#DECLARE 166, 173, 195, 205

#DISPLAY 149

#EMBED 171, 188, 189, 196, 210

dependencies 171

HIDE 208

#ENDTAB 149

- #EXTENSION 150, 155, 172
  - vs #APPLICATION 150
- #FIX 173, 179, 196, 198
- #FOR 154, 166, 179
- #GROUP 152, 162, 167, 171, 175, 194, 200
  - and dialogs 175
  - AUTO 166
  - example 166
- #IF 166
- #INSERT 152, 156, 162, 167, 171, 176, 195
  - NOIDENT 162
  - NOINDENT 196
  - vs #CALL 162
- #PREPARE 153, 156
- #PROMPT 149
- #RESUME 202
- #SELECT 195, 196
- #SET 154, 166, 173, 179, 196, 202
- #SUSPEND 202
- #SYSTEM 148
- #TAB 149
- #TEMPLATE 148
- #WITH 156, 159, 171
- changes in Clarion 6 167
- EXTRACT 154, 205
- INSTANCE 202
- WHERE 162
- language, #EMBED 170
- root 147
- scope 172
- symbols
  - %ABCLibraryPrompts 160
  - %AcceptToolBarControl 188
  - %ActiveTemplateInstance 154, 156, 159, 160, 162, 165, 168, 169, 170, 171, 176, 188, 189, 190, 191, 193, 194, 195, 196, 199, 200, 201, 203, 204, 206, 208, 211
    - defined 159
  - %ActiveTemplateInstanceDescription 174, 199, 201, 203
  - %AddControl 206, 209
  - %AddMethods 165, 166
  - %AssignToolBarButtons 189
  - %Attrs 167
  - %BaseClassToUse 170
  - %BeforeCallingUpdateOnAdd 203
  - %BeforeGenerateApplication 159
  - %BeginAddEntryRoutine 209
  - %ClassItem 153, 156, 159, 160, 172, 173, 195
  - %ClassPrompts 147, 159, 160
  - %Control 154, 172, 176, 179, 205, 206, 209, 211
  - %ControlEvent 187, 207, 211, 212
  - %ControlEventHandling 182, 184, 185
  - %ControlHasIcon 176
  - %ControlInstance 154, 176, 206, 211
  - %ControlOriginal 211
  - %ControlOtherEventHandling 187
  - %ControlStatement 154, 155, 205, 206, 211
  - %ControlTemplate 162, 165, 168
  - %DerivedInstance 188
  - %DeriveFromBaseClass 166
  - %EditControl 206
  - %File 165, 198, 199, 201, 203
  - %FileControlSetFlags 176
  - %FileKey 198, 199
  - %FileKeyField 203
  - %FileKeyFieldLink 203
  - %FixBaseClassToUse 170
  - %FixClassName 170
  - %FormatPrimary 194
  - %FormatSecondary 194
  - %GatherObjects 162
  - %GenerateClass 162, 165, 167, 171, 172
    - defined 167
  - %GenerateNewLocalMethods 170, 171
  - %GenerateParentCall 169
  - %GenerateVirtuals 168
  - %GetEmbedTreeDesc 171

- %GlobalClassPrompts 147, 156
  - %GuardEnabled 171
  - %HelpControl 205
  - %InstancePrefix 179, 181, 182, 184, 185, 187, 207
  - %KeyFieldSequence 198
  - %LastNewMethodsInstance 166
  - %LocalDataClasses 162, 165, 172
  - %LocalProcedures 171, 188, 208, 209
  - %MethodsPresent 166
  - %NewInstance 195
  - %NewMethodCodeSection 195
  - %NewMethods 166, 195
  - %OOPHiddenPrompts 153
  - %OOPPROMPTS 152, 153, 156
  - %pClassMethod 170
  - %pClassMethodPrototype 170
  - %pClassName 158, 160
  - %pMethodName 166
  - %pMethodPrototype 166
  - %pMethodsPresent 166
  - %Primary 151, 165, 189, 190, 191, 193, 196, 197, 201, 202, 203, 204, 210
  - %PrimaryDefaultIcon 176
  - %PrimaryFilter 197
  - %PrimaryKey 191, 197
  - %ProgramProcedures 171
  - %ReadABCFiles 147, 154, 156
  - %RelationKeyField 198, 199
  - %RelationKeyFieldLink 198, 199
  - %RemoveControl 206
  - %Secondary 165, 176, 198, 199, 203
  - %SecondaryConditionalIcons 176
  - %SecondaryDefaultIcon 176
  - %SecondaryFilter 200
  - %SecondaryType 165, 176
  - %SecondaryUpdate 202, 203
  - %SetClassDefault 153, 154, 156
  - %SetClassItem 172
  - %SetupMethodCheck 170
  - %StandardAddIconToProject 176
  - %TreeClassMethodCodeSection 189
  - %TreeControl 154, 179, 180, 182, 184, 185, 187, 189, 191, 193, 201, 207, 209, 211, 212
  - %TreeTitleIcon 176
  - %ValueConstruct 179, 187, 202, 203, 206, 211
  - %WindowEventHandling 182
  - %WindowManagerMethodCodeSection 179, 180, 181
  - writing 145–212
  - template registry 148
  - ThisWindow 2
    - see also* ABC
    - WindowManager
  - toolbar
    - and template 207
  - TTS
    - see* Team Topspeed
- ## V
- virtual method 32, 56–69
    - advantages of 61
    - and INTERFACE 62
    - calling parent 35
    - definition 32
    - DERIVED 35
    - enforcing prototype 35
    - example 33–35
    - prototype change 35
  - virtual method table
    - see* VMT
  - VMT 36

