Building ABC Compliant OOP Classes And Wrapper Templates

Gus M. Creces
The Clarion Handy Tools Page

Presentation Objective

This Presentation Teaches You How To Build:

- ABC Compliant OOP Classes
- ABC Compliant OOP Wrapper Templates

The objective of the next 5 or 6 hours is to teach you how to build ABC Compliant OOP Classes and how to incorporate them into your applications with ABC Compliant Wrapper Templates. The outcome of this training session we hope will be that the classes and templates which you create - should you choose to do so - will work the same way as the right-from-the-box templates and classes shipped with Clarion. A corollary to this is that you will gain a greater understanding of the mechanisms underlying ABC even if you never build your own classes or templates.

The term "ABC Compliant" is bandied around quite a bit and needs to be clarified. And it will be as this talk progresses.

For the moment, let's simply agree it means that the classes and their wrapper templates are structured in a way that they look and act like ABC classes and templates particularly in the way that they **derive classes using embedding**.

I don't want to take ABC Compliancy much further than that at this point in the presentation because I suspect the explanation I've just given you, specifically, "Derive Classes Using Embedding" is probably already a source of puzzlement for individuals with relatively little Clarion experience and those individuals who may not yet have strong OOP skills. Those of you who know exactly what I mean by **"Derive Classes Using Embedding"**, please stick with me on this. My talk has useful information for everyone - experienced as well as inexperienced.

What constitutes ABC Compliancy will become obvious as I explain some of the inner workings of the Clarion IDE, Clarion OOP and Clarion Templates as we progress through the presentation.

Brief Background

- Clarion Language and Model Files
- Clarion Language and Templates
- Clarion Language + OOP and Templates

First, a brief background…

The basis for the Clarion development system has always been and still is the Clarion language. The Clarion language was created and was doing work in the real world before Clarion even had a real compiler and before Clarion templates, as you know them now, ever existed.

Clarion Professional Developer Version 2 used a Model File system that worked pretty well, but was nowhere near as flexible as the Clarion Template system that first appeared around 1990/91 with the introduction of **Clarion Data Base Developer Version 3**.

Model Files and Templates were devised to make using the language more productive by reducing repetitive programming tasks and improving consistency of code and interface design. They do not replace the Clarion language, or for that matter, improve the Clarion language.

Their primary purpose is to help the developer write applications using the Clarion language.

OOP was introduced to Clarion, officially, in **Clarion For Windows Version 4**, even though **Clarion for Windows Version 2** had at least some OOP capabilities already present. There was no Clarion For Windows Version 3, presumably, to avoid confusion with **Clarion Data Base Developer** which was at that time referred to as "Version 3" or "Clarion 3".

With the introduction of OOP to Clarion, the Clarion language became what is called a "Hybrid" language. This means that while it retains the nature of a Structured Programming Language, it acquired also the abilities of an Object Oriented Language.

Included with OOP capability is a far greater measure of code maintainability and code reuse than is possible in a purely structured programming language as found in earlier Clarion versions.

And with that, came a re-definition of what the primary purpose of Clarion Templates actually is. If any of you are still using Clarion's Legacy Templates you have yet to realize many of the personal benefits of this re-definition.

I remember once, doing a calculation, when Clarion 4 was fresh on the market that to build identical browses with pre-OOP Clarion For Windows 2 took anywhere from 5 to 10 times as much generated code as it did in Clarion 4.

That doesn't necessarily mean there is a whole lot less code in OOP browses today, since there's way more functionality provided, but that there is waaaay less generated code. Browse code is more pre-built and less generated in an ABC setting than it is in the Legacy setting.

Pre-OOP Clarion model files and templates worked on a boiler-plate principle similar to that which a lawyer might use to build a contract. All the verbiage is repeated from contract to contract with the names and dates changed by substitution. Or like a formula novel many of which read like they are based on a boiler plate novel in which the author has substituted new character names, settings and bits of the action by search and replace.

On the other hand, post-OOP Clarion Templates work on the interface principle – remember this term, since there's a test later (just kidding). While code is obviously still generated, the nature of template-generated code is to initialize and interface to the OOP code library functions already present in the ABC OOP libraries.

Most of the code that constitutes your C55 and C6.x browses already exists in complete, ready-to-compile form, on your drive before you ever click the generate/compile button. OOP library code is used and re-used. Not generated afresh with each browse procedure.

Relevant Chapters

The chapters in your training guide listed here approach the objectives of this presentation in very detailed, hands-on way, from a slightly different angle than is possible in a sit-and-listen presentation such as this.

Therefore, I will not be following these chapters slavishly. But the overall purpose remains the same.

I will cover the key things that you need to know to help you realize the objective of this presentation.

If you're serious about learning how to **Build OOP Compliant Classes and Wrapper Templates**, then you'll take this book and the CD that accompanies it and work through these chapters again in light of what I show you here today.

I hope I don't have to spend any time convincing you **why** you should know how to reach this presentation's objective. I'll assume you're here because you want to know more about the inner workings of the tool with which most of you earn your livelihoods. While I'm sure you'll all agree that Clarion allows you to "work smarter and not harder", there is something profound also to be gained by "working a bit harder at becoming smarter".

Even if you never, ever build an ABC compliant OOP class or wrapper template, and you develop applications using only out-of-the-box Clarion ABC templates and a few 3rd party ABC add-on templates, a thorough understanding of what I'll attempt to teach you today is worth the price of admission to this Pre-Conference Training.

Software For This Presentation

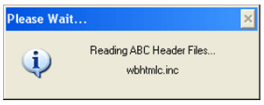
- \LIBSRC\ Devcon OOP libraries
- \TEMPLATE\ Devcon templates
- \APPS\ Devcon demo application
- \CPST\ Devcon demo projects

When you take the material we've given you today, and you begin to read the chapters covered by this presentation and work through the examples provided, you will need to copy the LIBSRC and TEMPLATE directories to your CLARION6\LIBSRC and CLARION6\TEMPLATE directories respectively.

The demonstration application and projects can be worked pretty much from any directory as long as you keep the files provided in those directories together.

While templates can also be registered from directories other than \CLARION6\TEMPLATE\ it is critical that ABC Compliant OOP libraries be located in the \LIBSRC\ directory where the IDE's OOP registration mechanism can find them.

Speaking Of Registration



Who can explain what this does?

Speaking of registration, does anyone here feel comfortable explaining in general terms what this process window does?

1. This window comes up when clarion is reading through the **OOP HEADER** files in your \CLARION6\LIBSRC\ directory.
2. By doing this it makes the OOP methods described here available to the Clarion template system in the form of template variables and template queues (multi dimensional template variables called MULTIs).
3. It enables the template system to act as a go-between layer – I earlier called this an interface layer – between the OOP libraries located in \CLARION6\LIBSRC\ and the developer.
4. Once the template system knows about the available OOP libraries – having read and memorized the header files – ABC Compliant templates in co-operation with the classes and the Clarion IDE, make it easy for you to incorporate application specific VIRTUAL changes to the classes without physically modifying, copying or boiler plating those classes.

That's a quick overview. Now let's look at this mechanism more closely.

First – Read OOP Headers

Reads the OOP Header Files in the
\CLARION6\LIBSRC\ directory.

The first step in class registration displays that annoying little window "Please Wait – Reading ABC Header Files". At this point, Clarion is reading through the **OOP HEADER** files in your \CLARION6\LIBSRC\ directory.

Shortly we'll explain what HEADER FILES are in case you're wondering.

Better yet, OOP HEADERS are self documenting. We'll try and explain how to read them to get most of the information you need to understand them, and finally, how to use them.

And whether they are ABC fully ABC Compliant…

Second – OOP Headers To Template Variables

OOP methods, parameters, return values and method derivability become known to the ABC template system.

The second step of class registration involves OOP METHOD NAMES, PARAMETERS, RETURN VALUES AND METHOD DERIVABILITY BECOMING KNOWN TO THE TEMPLATE SYSTEM.

Class Registration makes the OOP methods described in the HEADER FILES available to the Clarion template system in the form of template variables and template queues (multi dimensional template variables).

The ABC template system provides a wealth of template variables and template queues (called MULTIs) and template sub-routines (called GROUPS) that allow the template programmer to hook or "WRAP" the Clarion code that his/her template will generate into the finished OOP code already sitting on the disk.

By hooking to or "wrapping" that OOP code I mean calling into the procedures available in those OOP libraries. Initially, that saves having to generate those procedures again and again the way the **Clarion For Windows Version 2** "Legacy" templates once did.

But that's not the half of it!

What makes the "wrapping" process truly amazing is its ability to change the behaviour of existing OOP procedures without physically modifying the code in those procedures by using something called **derivation** – remember that term too – and make sure you understand it thoroughly when it's explained. It'll be on the test.

Third – Templates Act As Go-Between Layer

The template system is able to acts as a go-between layer, or *interface layer,* between the OOP libraries and the developer.

The **third** component of class registration is really an outcome of the previous two steps rather than a step in and of itself. The TEMPLATE SYSTEM BECOMES ENABLED TO ACT AS A GO-BETWEEN LAYER, OR INTERFACE LAYER, BETWEEN THE OOP LIBRARIES AND THE DEVELOPER.

The templates ask you questions about your application that the OOP classes need to know in order to fulfill their purpose. Questions like: What file do you want to browse? What key do you want to use? How should this window resize? And so on.

A lot of this information is used to initialize variables (known as PROPERTIES) and initialize references in the OOP classes.

Again, though, that's not where the REAL MAGIC lies! The real magic in all of this comes from the fact that ABC Compliant templates are able to place embed code on your behalf, resulting from the answers you've provided on the template interface, inside ABC Compliant OOP classes, without physically modifying the code in those ABC Compliant Classes. Let me re-state that in bold text, to make sure it sinks in:

The real magic in all of this comes from the fact that ABC Compliant templates are able to place embed code on your behalf, resulting from the answers you've provided on the template interface, inside ABC Compliant OOP classes, without physically modifying the code in those ABC Compliant Classes.

What I'm saying here, in fact, is that the same browse class OOP code can be made to browse a customer file or a product file including all of the customization and personalization required in each browse, all without physically making any changes to the ABC Compliant OOP libraries available in your \CLARION6\LIBSRC\ directory.

No physical changes to the OOP code in the libraries…

Fourth – OOP Code Changes Are Virtual, Not Physical

ABC Compliant templates in co-operation with the classes and the Clarion IDE, make it easy for you to incorporate application specific VIRTUAL changes to the classes without physically modifying, copying or boiler plating those classes.

The fourth component of template registration is again an outcome of steps 1 and 2:

ABC COMPLIANT TEMPLATES IN CO-OPERATION WITH THE CLASSES AND THE CLARION IDE, MAKE IT EASY FOR YOU TO INCORPORATE APPLICATION SPECIFIC **VIRTUAL** CHANGES TO THE CLASSES WITHOUT PHYSICALLY MODIFYING, COPYING OR BOILER PLATING THOSE CLASSES.

This specific template behaviour in ABC applications, since Clarion 4, is entirely different than the behaviour of legacy templates.

Again using browses as a basis of discussion:

- Legacy templates generated all the code required to build a browse.

- ABC Compliant templates generate:
    1) Interface code to the ABC browse and file classes and others
    2) Virtual code changes to these classes as required by the specific implementation promoted by the template.

Once you completely understand how this mechanism works you may never again need to post a message on the SV newsgroup asking, "Where is the embed point to do this or that or the other thing?"

You will know, in fact, that the embed point for any given functionality shift is inside the OOP method (procedure) that provides the functionality you are trying to modify with your embed.

This implies, of course, that you're serious about being a developer and you've taught yourself to **read and write Clarion source code**. Remember my earlier assertion that templates do not change the Clarion language, they help you to use it better, and are a means to improve the consistency of your code, and to help you with repetitive coding tasks.

Creating an application with Clarion is not a "paint-by-numbers" exercise although I'm afraid that often it comes to that.

The next portion of this presentation is intended to help you get a leg up on reading an OOP Header file.

1. What constitutes an ABC compliant class that can be template-wrapped into your application?
2. And an explanation of the mechanism that provides for VIRTUAL changes to class library code.

Demo Application DVCNTEST.APP

The demonstration application provided in /APPS/ incorporates all of the classes and templates provided with this presentation.

Much of the technical information I'm going to provide you in the next part of this presentation will take place in the context of a demonstration application that you'll find in the pre-conference day-three portion of the CD provided.

The application can be found in /APPS/ directory and is called DVCNTEST.APP

I've moved the location of this app into a directory below Clarion6. This isn't strictly necessary as you could compile the app directly from /APPS/ directory once transferred to your hard disk. Placing it below the Clarion directory simply makes it easier for me to navigate in and out of the /LIBSRC/ and /TEMPLATE/ directories, something I'll be doing a lot in the next couple of hours.

Remember, though, as I've mentioned earlier, that for this to work, you must first have copied the contents of the /TEMPLATE/ and /LIBSRC/ directories provided into your Clarion6/TEMPLATE/ and Clarion6/LIBSRC/ directories, respectively.

Demonstration: Start Clarion, Register the DevconTools.tpl template, Open, Compile, and RUN the app to show that it is working correctly.

This application is entirely built with Devcon example templates and while these are goodies in and of themselves, they are provided to give guidance with how the ABC Compliant Templates and Classes that they wrap are built. The chapters I mentioned earlier deal quite specifically with some of these classes and templates.

This application isn't mentioned in those chapters because I built it specifically for my presentation so that I could share it with my own subscribers without violating the copyright provisions of the material that Bob Foreman provided for you earlier in the week.

Demonstration: Navigate to the global area and indicate that some of the templates are global and some are procedural, and some are embed templates.

OOP Headers – What/Where

OOP HEADER files end in the .INC extension. They outline in point form the properties, functionalities and relationships of an OOP class.

Demonstration: The following are manual steps illustrated by the presenter by manipulating the Clarion IDE.

First:
Navigate to Files -> Open -> /LIBSRC/ and discuss the difference between INC and CLW files and impart the same basic information found on pages 209 – 215 (roughly).

Show some of the ABC INC files and what they do, then indicate the Devcon specific headers and what they do. FINALLY, END WITH AN EXPLANATION THAT THE BULK OF OUR INVESTIGATION AND STUDY WILL INVOLVE THE **DEVCONSTARTERSAMPLE** AND **DVCNSTARTERDERIVATION** classes.

Second:
Navigate to Application->Global Properties->Classes Tab and refresh the classes and remind them of the four steps discussed above and show them the DVCN classes in the list and point out any relationships.

Show them also the **DVCNSTARTERSAMPLE** class in the class viewer, including all properties and methods. This is useful when we examine the class header in detail to point these same things out. It corroborates the earlier assertion that the Clarion IDE had read and memorized all aspects of the class headers including relationships between files.

Third:
Now open up the procedure named **SampleStarterTemplate()** and show them that our template has indeed included the class header module for this OOP class at the procedure level and it has declared (instantiated, created) an instance of this class inside our procedure which we called STRT (determined on the classes tab).

THE MAIN POINT OF THIS IS TO INDICATE WHERE, IN THE GENERATED CODE THAT THE CLASSES ARE BEING INCORPORATED VIA
INCLUDE('CLASSNAME.INC')

OOP Headers – How

OOP HEADER files contain most of the information that you need to know about using an OOP class. This is the same information read by the Clarion IDE and made available to the template system.

Demonstration: Open DVCNSMPL.INC, the header file for a sample class called: **DVCNStarterSample**

This class does nothing much, but it is a class on which you can model your own ABC compliant classes should you decide to create your own. Use the general structure I've provided here to model your own OOP headers. Next, I will explain the key component parts of that general structure.

Demonstration: Progress through the components of this class header one by one as per the following sections:

!ABCIncludeFile (Starter)

This is the first necessary element of an ABC compliant OOP header file. The left hand element of this tag (!ABCIncludeFile) ensures that the OOP registration routine we discussed earlier and which I brought up in the context of my Speaking Of Registration slide, reads the header file into template variables for use with the templates.

I don't expect you to take my word for this so I'll demonstrate.

Demonstration: At this point go to the IDE's global classes tab and display the DVCNxxx classes. There are seven. Then remove the !ABCIncludeFile tag from two classes and refresh the classes. Then redisplay the global classes tab and display the DVCNxxx classes again.

Once one removes the !ABCIncludeFile tag from the header file of a class – or in the event it never contained one – Clarion's class registration function ignores them. In fact such classes are NOT ABC Compliant and cannot be wrapped into your Clarion application in the standard, automatic way that Clarion ABC does it.

The second part of this tag, enclosed in parenthesis serves two purposes. First of all it categorizes your classes by function when viewed in the Application Builder Class Viewer. (**Demonstration:** Show them this again in the IDE's Class Viewer). At this point in time there is no facility in this viewer to see the classes listed by category though I suggest it might be useful to have another tab in the viewer that does this.

However, that's not the main purpose of this tag anyway which probably explains why no-one thought of putting a category or "family" tab on the class viewer. The primary purpose of the category tag in the event that the entire right hand part of the tag, the parenthesis and the tag itself, are missing, or is set up as **(ABC)** is to indicate which classes are so fundamental to the operation of your application that they should be exported from your global DLL along with your file definitions and any global variables.

In other words:

- Missing! **ABCIncludeFile** means the class is not intended to be ABC compliant so the OOP registration routine ignores it. You cannot automatically embed directly into such classes at least not without significantly modifying the behaviour of Clarion's ABC mechanism itself, and that's not necessary (nor is it probably even smart).

- Having **!ABCIncludeFile** or **!ABCIncludFile(ABC)** means that the Clarion IDE will export the functions in these classes from the DLL in which you place them. Normally you do that in the Global DLL and link that into the rest of your application based on how you've set: **Application -> Global Properties -> Generate Template globals and ABCs as EXTERNAL.**

- Having **!ABCIncludeFile(Anything Else)** means that the classes so marked are not automatically exported from your DLL. (They can readily be made to do so, but demonstrating that is not the covered in this talk.) They are compiled into your DLL, if you happen to have used them in that context, but they are not automatically exported and made linkable from the DLL.

Demonstration:
Open up ERROR.INC and indicate that this class does not use the (ABC) tag.
Open up ABFILE.INC and indicate that this class does not use the (ABC) tag.
Open up ABBREAK.INC and indicate the (ABC) tag.
Open up ABPRNAME.INC and indicate the (ABC) tag.

Now set the app for DLL using, **Project -> Properties -> Target Type DLL and Run Time Library Standalone**.

Generate (no need to compile) the application and examine the EXP in the Clarion editor.

The presence of VMT$ClassName indicates that "Virtual Method Table" references to these classes are present and that the class so named is being exported from the DLL.

Search for VMT$ERRORCLASS (it comes from ERROR.INC)
Search for VMT$FILEMANAGER (it comes from ABFILE.INC)
Search for VMT$BREAKMANAGERCLASS (it comes from ABBREAK.INC)
Search for VMT$NAMEGENERATOR (it comes from ABPRNAME.INC)

Point out that our class name **VMT$DVCNStarterSample** and **VMT$DVCNStarterDerivation** are not there even though our class is ABC Compliant and is hooked into the application and compiled into the application.

The reason they have not been exported is that the category or "family" tag **(Starter)** has indicated to Clarion's OOP registration mechanism that they should not be exported from this DLL.

Demonstration: Now open DVCNSTRT.INC and DVCONDERI.INC and change **(Starter)** to **(ABC)** for each of these classes. Then refresh the classes and while they're refreshing, explain why they're being refreshed.

Now regenerate your app and open the .EXP file again and search for VMT$DVCN. Now our classes are being exported from the DLL because both **VMT$DVCNStarterSample** and **VMT$DVCNStarterDerivation** can be found in the export list.

Except for global classes like File Manager, View Manager and Error Manager, it is not absolutely critical that your classes be exported from the global DLL or they be exported from any DLL for that matter. If you look closer at the ABC's themselves you will note that a large percentage of them use category or "family" tags that cause them not to be exported automatically from your DLL.

(**Demonstration:** Point this out by pointing out the various categorizations visible in the ABC Class Viewer).

Classes are compiled into your procedures as needed where the compiler optimizes them to a great extent anyway. In a huge app consisting of many DLLs there is some space saving by exporting the classes either from the global DLL by tagging them in this way - generally with **(ABC)** – or by exporting them from a DLL of their own.

There **ARE** some other design considerations in the class header and in your wrapper templates that do have to be handled if you go that route. I'll bring those considerations to your attention as we proceed.

If your app is 100% EXE and links no DLLs of your own making, the category tag has no effect. A given class is compiled into the exe only once even though it may be instantiated on many different procedures.

If your app is divided into DLLs then the classes without category tags and those with **(ABC)** are exported from the global DLL and are linked to the rest of your application from there. The classes using other category or "family" names are compiled into (but not exported from) the program components where they're used. Exporting them from the global DLL saves some space in the other DLLs and the EXE since they can be linked from the global DLL.

As I've said, exporting them from a DLL brings with it a few design considerations that you need to be aware of so you shouldn't base your decision solely on a bit of space saved in a few DLLs.

It's your decision to make if you think your classes are important enough or used often enough in various places in the application to be included in the global DLL. One way or another your application will work just as this demo does regardless which way you decide.

Demonstration: Point out the OMIT structure at the top of the DVCNSTRT.INC file.

```
 OMIT('_EndOfOMIT_',_DVCNStarterSample_)
_DVCNStarterSample_      EQUATE(1)
 _EndOfOMIT_
```

This is an omit flag that keeps your class from being included in the compile stream more than once (from being compiled twice in the same app). The OMIT compiler directive omits the section of code between OMIT and the indicated end marker flag, in this case _EndOfOMIT_. This omit section skips the entire class

and keeps it out of the compile stream of a given procedure or section if it's already been through it at least once. OMIT compiles the code if the equate setting for **_DVCNStarterSample_** does not exist or is zero. And it does NOT compile the code if the equate exists and has any value other than zero.

Since classes are meant to use one another and call into one another and they often derive from one another, any given class header may be included dozens of times by other classes. This flagging system is absolutely critical to keeping the code from compiling twice when it doesn't need to be. It also keeps you from getting DUPLICATE SYMBOL errors which occur when an EQUATE with the same name is defined more than once.

IF TIME PERMITS
Discuss the possibility of potentially creating recursive includes that could cause the compiler to go into a loop if an omit structure like this is not used in a complex system where includes are deeply nested. At the minimum, at least the qualifier, ONCE should be used on all header includes.
END OF IF TIME PERMITS

Another good use of this flag is to detect at compile time whether it's safe to call into a given class or not, again using the OMIT and COMPILE directives. In fact, I've included an example of this in the application.

Demonstration: At his point, open up the demo application on procedure SampleStarterTemplate() and go to the section as follows:

```
 COMPILE('_EndOfCompile_',_DVCNStarterDerivation_)
  STRT.Method4()
  _EndOfCompile_
  OMIT('_EndOfOMIT_',_DVCNStarterDerivation_)
  MESSAGE('Method 4 is available only in the DVCNStarterDerivation
Class','Method not available...',ICON:Asterisk)
  _EndOfOMIT_
```

This code looks for the presence of the **DVCNStarterDerivation** class by checking for its presence flag, **_DVCNStarterDerivation_**. If present, the code calls **Method4()** as it is obviously available in that class. If the flag indicates that the class is not present, a message is thrown up to indicate that the required class is not present.

This isn't quite how you'd use that feature in an app but you might use the same construct to include menus or buttons that call into reports or processes which are available if class X is present and to not include those menus or buttons if class X is not present.

Demonstration: Next, point out the INCLUDE Directives at the top of the DVCNSTRT.INC file.

```
INCLUDE('KEYCODES.CLW'),ONCE
INCLUDE('PROPERTY.CLW'),ONCE
```

This is the place to include equate files and definitions of any sort that may be used by your classes. It's also the place to include other class header files if your classes are going to call into or derive those classes. Notice that these includes are INSIDE the OMIT compiler directive since we don't need these things recompiled in a given context any more than we want our classes recompiled in a given context. In other words, these includes are "Included" the first time our class header is compiled and not included when your class header is OMITTED. That's just how it should be.

Demonstration: At this point it may be useful to open another ABC class such as ABBROWSE.INC and note that it includes these headers from other ABC classes:

```
INCLUDE ('ABFILE.INC'),ONCE
INCLUDE ('ABPOPUP.INC'),ONCE
INCLUDE ('ABTOOLBA.INC'),ONCE
INCLUDE ('ABEIP.INC'),ONCE
```

Notice that ABBROWSE.INC includes a number of other ABC headers.

These header files are included there because the ABC BrowseClass makes use of these classes (calls into or derives) these classes. In fact you can point out that BrowseClass itself derives ViewManager which is defined in ABFILE.INC as follows:

```
BrowseClass
CLASS(ViewManager),IMPLEMENTS(WindowComponent),TYPE,MODULE('ABBROWSE.CLW'),DLL(_ABCDllMode_)
```

Demonstration: Point out the CLASS declaration at the top of the DVCNSTRT.INC file.

```
DVCNStarterSample
CLASS,TYPE,MODULE('DVCNSMPL.CLW'),LINK('DVCNSMPL.CLW',_ABCLinkMode_),DLL(_ABCDllMode_)
END
```

Between the name **DVCNStarterSample** and the **END** statement there is a complete definition (description) of this OOP class.

In a minute we'll see some of the components included in a class definition, first let's examine the first line of the definition because there's a lot of important information here.

Our class is called **DVCNStarterSample**. That does not mean we'll call this class by that name when we implement it in our application procedures and functions. A class can be thought of in the same way you think of a data **TYPE** like LONG or STRING.

Demonstration: Point out the **TYPE** keyword in the class definition.

When you use a string in your app you do not write code like STRING = "SOME VALUE" you define an INSTANCE of a STRING and give that instance a name. Then you call the string by that instance name when you use it in your procedure or application.

The same principle applies to naming a class. A class definition is a TYPE definition.

To use a class you must create an INSTANCE of the class, or INSTANTIATE a class.

Demonstration: Now go back into the application and point out the CLASSES TAB for the StarterSampleTemplate () procedure and note that the name given to this instance is STRT. Then open the module code for this procedure and show how that name is used on an instance of this procedure and briefly discuss.

Near the top of our template-generated procedure module the header file of our test class is included in the compile stream.
INCLUDE('dvcnsmpl.inc'),ONCE

And further down, is the instantiation code where our sample class is declared, or in OOP parlance, is instantiated.
STRT DVCNStarterSample

Demonstration: Now go back and open the module DVCNSMPL.INC, point out this portion of the class definition line:

MODULE('DVCNSMPL.CLW'),LINK('DVCNSMPL.CLW',_ABCLinkMode_), DLL(_ABCDllMode_)

The MODULE() attribute tells us where the code for this class resides; in this case in a file with the same name ending in .CLW. We've discussed that point already.

The LINK() attribute tells the compiler/linker from where to link the code. LINK() in the first parameter is told that it should again link from the .CLW file. But this link is conditional depending on another flag that acts in the same way the OMIT directive uses its flag. If **_ABCLinkMode_** exists and is non-zero then the compiler compiles DVCNSMPL.CLW into an OBJ and that OBJ (Object code) is linked into the app. This flag is normally set to True in situations where compilation from source is intended and False if linking from elsewhere is intended. While my presentation here today only discusses the most common linking method - from DLL or Dynamic Link Library - linking from object code, in the form of an .OBJ or link .LIB is also possible.

The DLL() attribute tells the compiler/linker that the code for this class is somewhere in a DLL attached to the application. If the **_ABCDLLMode_** flag does not exist or is zero then there is no attempt made to link this class into the application from a DLL. If the flag is defined and set non zero then the compiler will attempt to link the class from one of the DLLs incorporated into your application.

The ABC templates manage these two flags for you by reversing them automatically when the time comes to link category **(ABC)** classes from a DLL.

Demonstration: Now perform the following demonstration to show the flags in action on the application DVCNTEST.APP.

Navigate to:
Project -> Properties -> Global Options for Project (first line) – Defines Tab.

Here you'll see that the settings for the two category **(ABC)** flags just discussed are as follows:
ABCDllMode=>0 – indicates don't link category **(ABC)** classes from a DLL
ABCLinkMode=>1 – indicates do compile/link category **(ABC)** classes from the OBJ created when the source is compiled

Navigate to:
Application -> Global Properties -> General Tab -> Generate template globals and ABC's as EXTERNAL

Check that setting to set it on and regenerate the application. Then go back to **Project -> Properties -> Global Options to the Project Defines Tab**.

Here you'll see that the settings on the two flags just discussed are now reversed.

ABCDllMode=>1 – indicates DO link category **(ABC)** classes from a DLL connected to the application
ABCLinkMode=>0 – indicates DON'T compile/link category **(ABC)** classes from the OBJ created when the named source is compiled (In other words, don't compile the source module.)

These flags are set by the ABC template using #PDEFINE based on the checkbox found at:

Application -> Global Properties -> General Tab -> Generate template globals and ABC's as EXTERNAL

In the event you've decided to include your classes in the **(ABC)** category to follow the convention of exporting from a DLL when you use them in a DLL compile, then your classes should use the flags **_ABCDllMode_** and **_ABCLinkMode_**.

If you decide not to include your classes in the **(ABC)** category and not to follow the convention of auto-exporting from a DLL, then your classes should use a flag of their own which is controlled by your template chain so that they are independent of the flag-setting value-shift performed when the developer sets the "**Generate template globals and ABC's as EXTERNAL**" switch to the on position.

Note that the classes **DVCNStarterSample** and **DVCNStarerDerivation** do use the **_ABCDLLMode_** and **_ABCLinkMode_** flags. Unless the category or family tag for these classes is set to **(ABC)** these flags will give you grief in a situation where the ABCs are to be linked from a global DLL if you do not change them to something else not impacted by the "Generate ABC's as External" setting already discussed. Now you know why you need to do that in your own classes.

Choosing to use ABC's default flags (_ABCDLLMode_ and _ABCLinkMode_) along with category **(ABC)** may not always be the best choice since it means your classes will always be exported from the global DATA DLL, whether you want that or not, and other parts of your app will need to link them from there.

When you use your own flags, (eg: _DVCNDLLMode_ and _DVCNLinkMode_) It's easy enough to add a PRAGMA directive to your code or to the Project Defines as ABC does to create and set these flags for you and to reverse-toggle them should you ever decide to incorporate your classes into a separate, stand-alone DLL.

If you plan to always compile your classes from source you could simply do away with the flags altogether and incorporate only True and False where the link flag

and the dll flag are now used, respectively.

Demonstration: Show this by replacing the flags temporarily in the **DVCNSampleStarter** class and compiling the application.

Demonstration: Point out CLASS PROPERTIES and which of these can be added directly to a class and which have to be created with NEW.

The following data types and a few other **simple** data types may be incorporated into a class definition directly:

```
MyString    STRING  (100)        !Can be incorporated directly.
MyCString   CSTRING (100)        !Can be incorporated directly.
MyLong      LONG                 !Can be incorporated directly.
MyByte      BYTE                 !Can be incorporated directly.
WhoAmI      CSTRING(100)         !Can be incorporated directly.
MyGroup     LIKE(PropertyGroup)!Incorporated directly with LIKE() or GROUP()
```

Creating simple data types like STRINGs, CSTRINGs, LONGs and so forth is really no different than creating ordinary variables inside your application. Creating GROUPs is also no different, though this is normally done as I've done it here using either LIKE or GROUP(PropertyGroup) based on a previously defined group TYPE definition.

Class **PROPERTIES** are in fact just **variables**. The really, really cool thing about instances of any OOP class is that while the code for a given class exists once in any context like an EXE or DLL, a separate set of properties is created for every instance of the class. So, when you create an instance of DVCNStarterSample called STRT and another called STRT2, there are two discrete sets of variables created, one set for each class. The same code is used by each set of variables. Only data is discrete. Code is shared.

There are some data types that are not included directly because they don't really exist until either referenced or created with NEW.

```
MyCStringRef    &CSTRING        !May be NEWed or Referenced.
MyQueue         &PropertyQueue  !May be NEWed or Referenced.
```

Reference variables of any type are place holders to be referenced during the course of program execution to existing data of the same type or they are pointed to data created by the class itself, using NEW.

Demonstration: Show them a reference statement in the DVCNSTRT.CLW file in METHOD 1. This reference to another entity is assigned with the assignment operator **&=.**

```
SELF.MyCStringRef &= xText
```

Data referenced this way does not need to be DEREFERENCED when your class is destroyed as when you exit the procedure in which it was instantiated, though I tend to do it out of habit.

Demonstration: Show them a NEW in operation in the DVCNSTRT.CLW file in the CONSTRUCT method.

The class constructor sets aside memory for the QUEUE being created.

```
SELF.MyQueue &= NULL
SELF.MyQueue &= NEW PropertyQueue          !NEWed in CONSTRUCT.
                                           !Disposed in DESTRUCT.
ASSERT(~SELF.MyQueue &= NULL, 'SELF.MyQueue still null in DVCNStarterSample.Construct')
```

This is how I instantiate variables that are going to be created using NEW.

First I assign NULL to the variable. Since after each NEW sequence a NULL test is performed to assure that the variable is no longer NULL.

Next the NEW operation is performed. Then an ASSERT or conditional is used to check that the variable is no longer null.

Demonstration: Now show them how ASSERT is used to force a GPF, on the understanding that the condition created by the failure which it is asserting would cause the program to work incorrectly beyond that point in any case. Show them the **Project -> Property -> Defines** setting **"Asserts =>1"**.

This makes ASSERT () pop up its message and GPF if the flag is set to 1. If missing or set to 0, asserts are ignored in final code, in which case you might be better off using an IF THEN MESSAGE(); END structure to indicate that your important NEW operation has failed and to exit the application or procedure gracefully.

Memory created manually like this with NEW does not automatically self-destruct when your OOP class exits. You are responsible for releasing this memory when the class is destroyed. You can make this as good as automatic, so that you shouldn't need to think about it again, by disposing of all properties created with NEW in your class code inside the DESTRUCT method as we have done in our example class.

```
 !Always protect a DISPOSE() sequence with a NULL check.
 IF NOT SELF.MyQueue &= NULL THEN
    DISPOSE(SELF.MyQueue)   !Must be DISPOSED by this class.
    SELF.MyQueue &= NULL
 END
```

I always protect a dispose sequence like this with a NULL check since my own product supports both C55 and C6. If you attempt to DISPOSE memory that you have already disposed, your application can GPF - at least it does in C55. I've tested this briefly in C6.1 and DISPOSE seems to have been improved to simply ignore DISPOSE() calls to NULL data. That's a good thing and a great improvement if it turns out to be true. For my own protection, I will continue to use DISPOSE() inside an IF NOT NULL THEN; END structure.

CONSTRUCT and DESTRUCT are special OOP functions that you do not need to call. They are called automatically for you by the compiler when a class is instantiated or uninstantiated. Which is great because it lets you put all necessary initialization code like the NEW I've just illustrated, in the CONSTRUCT function and all the de-initialization code, like the DISPOSE just illustrated, in the DESTRUCT function? As long as you remember to balance NEWs with DISPOSEs the rest is handled for you automatically.

REVIEW OF PROPERTY CREATION IN A CLASS

I have not illustrated every data type being created but there are really only these two methodologies. Either the data can be incorporated into a class directly as were STRING, LONG or GROUP or it is incorporated as a reference.

References can be referenced to passed-in data, which need not – which MUST not be disposed – but which may be dereferenced or rereferenced as often as you like.

Demonstration: Show some of this in the code module.

References can be made to create their own data using NEW. That sets aside memory which belongs to your class and which your class MUST set free before it exits – or you'll create a memory leak. Set memory free with DISPOSE(). Classes usually do this in the CONSTRUCT and DESTRUCT methods which are called automatically by the compiler when a class is instantiated and uninstantiated, respectively. That way you don't have to think about it and you won't forget to do it and cause a memory leak in your application.

Demonstration: Point out the CLASS METHODS and the four basic types of method attributes.

```
!Virtual ------------------------------------------------------------------------
Method1   PROCEDURE (*CSTRING xText, <STRING xOptText>,  |
                          LONG xVal=100 ),BYTE,PROC,VIRTUAL
Method2  PROCEDURE (*PropertyGroup xGrp, BYTE xVal=0 ),BYTE,PROC,VIRTUAL

!Non-Virutual --------------------------------------------------------------------------------
Method3 PROCEDURE (STRING xText),BYTE,PROC

!Private -------------------------------------------------------------------------------
Construct                PROCEDURE (),PRIVATE
Destruct                 PROCEDURE (),PRIVATE
```

There are four basic types of methods as determined by their attributes, that your class can have and this class illustrates 3 of them.

These attributes are: **VIRTUAL, NON-VIRTUAL, PRIVATE** and **PROTECTED**

- **PRIVATE** methods can be used only by the class itself.

- **PROTECTED** methods can be used by the class itself and any other classes that DERIVE that class (DERIVE will be discussed further)

- **VIRTUAL** methods are methods that have been given the VIRTUAL attribute. Virtual methods are the KEY to application embedding as it now exists in Clarion since the introduction of OOP in Clarion For Windows Version 4. You will remember I said that the OOP version of Clarion makes it possible to incorporate code into ABC compliant classes that changes behaviours without changing the original class code provided by its designer. VIRTUAL is the single MOST IMPORTANT CLASS METHOD FEATURE that enables the templates to perform this bit of magic. Once a VIRTUAL method is DERIVED it can be used as designed or it can be added to or changed in the derived version so that the developer has the option of using the method as it was designed, or using the method as he/she thinks it should work. When a VIRTUAL method is derived, it is possible to make old code (the class code as it was written) call into the future, which is into the new code created by the class that derives this method at some future point in time. This is a little weird and sometimes difficult to understand until you see it in action and begin to internalize what it means. I'm going to illustrate this more clearly in our demo application right after we discuss this last method type.

- **NON-VIRTUAL** methods are simply methods without the VIRTUAL attribute. Non virtual methods cannot be **derived** but they can be **overridden**. Deriving or derivation provides you with options, as I've said, to use the method as-is or to enhance it or disable it any way you

see fit as long as what you do is considered legal by the compiler. With **overriding** you create in a derivative class a method of the same name with an identical prototype. This completely blows off (better term – overrides) the original method as it existed in the original class, at least as far as the derivative class is concerned. The derivative class therefore owns it's version of this overridden method while the parent class continues to own the original version of the method.

With **VIRTUAL** methods the same code exists for the PARENT method or the DERIVED method. They **share** the same code. Or the compiler makes sure that this is how it works out. That allows VIRTUAL methods to be changed without actually touching their code. It also allows virtual methods to call into the future. I'll say more about that amazing capability shortly.

All right, unless you've already internalized the nature of VIRTUAL and DERIVE as implemented in OOP languages, your ability to believe what I'm saying has now probably been stretched to the limit. This **can** be illustrated in any ABC application, because the ABC templates do it all the time and if you've ever placed embeds into your applications; you've been doing it too – perhaps without being aware of the implications. If you've never fully understood this VIRTUAL/DERIVE mechanism as implemented in Clarion's ABC classes and Templates then you've probably never leveraged the Clarion Development environment to its full potential as a developer (Read: you may be working harder not smarter).

LECTURE MODE ON - IF TIME PERMITS

It's probably a truism to say that just because you know how to drive a car and you can lucidly describe from a driver's perspective how it should feel and perform, you don't really have to know how it actually works when it comes to the crunch.

In that respect Clarion is a lot like a car.

Someone with a good idea can cobble together a pretty decent application using the "paint-by-numbers" principle that I talked about earlier. Somewhere, someplace another developer has created a template or a class with a template wrapper, which more or less does what a "cobbler" wants. And he/she finds enough of these things and when they do: **Let the cobbling begin!**

That's one way to write an application but it's not my idea of a good time. What you have is a potential Rube Goldberg device not an application. Overdo it and you are not really in control of the situation!

I'm not saying developers shouldn't use 3rd party tools, far from it. What I **am** saying is a repeat of what I said earlier:

Developers must take responsibility to fully understand how **Clarion the Development System** actually works, since well behaved 3rd party tools must work within that framework. Understand the framework and you've gone most of the way to understanding the add-on tool.

Taking our car analogy a little bit further, let's now drive our car, which we know how to drive but not how to repair or maintain, and take it off the beaten path. Let's do something special and boldly go where others seldom go – splitting an infinitive or two along the way. Now you're at risk in the event something happens that lets the air out of your tires or the spark out of your coil.

You know where I'm going with this.

Using a development tool like Clarion (or a 3rd party tool for that matter) as a crutch because you need it to hold you up is different than using it as a lever because you understand how to use it and how it works or you can find out how it works by reading the code when you don't.

Understanding **The Clarion Development System** first and foremost entails understanding the Clarion language. I said this right at the outset of my talk. Ultimately, every app you write ends up as a set of Clarion language modules sitting on your hard drive, being run through the compiler to be turned into binaries for execution in a computer processor.

Once you understand and can easily read and write the Clarion language, including the fundamental OOP design principles I'm discussing here, you need to understand how the Clarion IDE and the template system leverages these language features to provide you with the really amazing development tool that you're using to earn your living.

Anyone who cajoles you into believing that "working harder not smarter" means increasing your dependence on something you never come to understand in order to spend more time at the beach is simply obfuscating.

LECTURE MODE OFF

Virtual Methods

If there is one feature in the Clarion language which you must come to internalize, that above all else defines how the development system works today, it is the principle of VIRTUAL METHODS and DERIVATION of VIRTUAL methods.

There are two features in the OOP version of the Clarion language which you must come to internalize, that above all else define how the development system works today. If you haven't yet, it's time you did.

These features are:

1. Derivation of Classes (often called Inheritance)
2. Derivation of Virtual Methods

These features, or OOP principles, are the basis for intra-application EMBEDDING, which in turn is the basis for sophisticated application-building as it exists in the Clarion development system today.

You yourself may seldom place manual embeds into your application. Some developers pride themselves in never having to do this and will go to great lengths to avoid it. In fact the ABC templates that you use daily, whether SV supplied or 3rd party supplied (if ABC compliant), are constantly embedding on your behalf. Manual embeds are just an extension of that. Where these embeds go is more often than not inside a DERIVED version (a DERIVATIVE of) a VIRTUAL method provided by an ABC compliant OOP class.

At this point let me illustrate that principle in action inside an application.

Demonstration: Open up DVCNSMPL.INC and point out the four methods provided there. Note that only two of them are virtual methods.

Now load DVCNTEST.APP and right click SampleStarterTemplate() and select MODULE to see the already-generated code for this procedure.

I've shown you this before and have brought you back here to show you once again in the already-generated code for this procedure how the class called **DVCNStarterSample** is instantiated in this procedure.

Note also (**Demonstration:** as you slide down the file with the editor.) that there are really only a few other procedures here all belonging to the window manager class.

Demonstration: Now close the module view and right click SampleStarterTemplate() and select SOURCE to enter the embeditor view for this procedure.

Take note that now as we go to the area of code where **DVCNStarterSample** is instantiated that this doesn't really look like it did in the generated code view. This class definition is not a direct instantiation (a label on the left and a class type on the right) this class definition is a DERIVATION. The target class (**STRT**) is set up to be derived. What the heck is going on here? When I look at the class via the embeditor it looks like this, when I look at the generated code it looks different.

I know, let's regenerate the module that we looked at earlier. Maybe that generated code module is out of date!

Demonstration: Now regenerate the app and re-enter the MODULE view.

Okay, shoot that theory. The embeditor (Source) view appears to be different than the generated Module view. What the heck IS going on here?

Demonstration: Now go back to the SOURCE embeditor view and scroll down to the class definition for DVCNSampleStarter.

I'm playing devil's advocate here to make a point. The target class (**STRT**) as you see it from the embeditor has been set up as if you're going to derive it from the parent class (**SVCNSampleStarter**). But why are there only two methods listed? Our parent class has four methods.

Demonstration: Now point to the DERIVED attribute.

Here's the secret. These methods in the parent class were given the attribute VIRTUAL. I showed you that earlier. The rule of thumb you need to remember:

When the methods in your ABC compliant class are VIRTUAL methods, and the class is connected to a procedure via an ABC compliant WRAPPER template, the VIRTUAL methods in that class are presented in the embeditor for DERIVATION.

We're being given a chance via these derived virtual methods to introduce some behaviour change in them that is unique to our application. So let's do that just for fun and see what the IDE and the template generator do with that.

Demonstration: Now go to the two derived methods STRT.Method1() and STRT.Method2() and add a comment inside each; close the embeditor; save and regenerate all. Then re-enter the MODULE view of the regenerated code and point out that now the generated code actually resembles what you saw in the embeditor.

The embeditor view offers us the opportunity to automatically derive a class and derive the VIRTUAL methods in that class. All we need to do is place embed code into any one or all of them. If we don't use the opportunity it simply instantiates the class under the name designated on our Wrapper Template's CLASSES tab. It does not derive the class.

On the other hand, as soon as we embed something inside a DERIVED method (VIRTUAL in the PARENT) a derivation of the class is performed and the method into which you've placed any code, is derived.

I just put hand-coded embeds here to cause a derivation to occur. The same embed could have been performed by your template and the derivation would occur just the same. In fact, most of the time that's what actually happens. A template does the embedding for you based on the information that you've provided on the template's interface.

Before we dazzle you any further, let's try and understand what just happened and what the point of it is. I'll repeat some of the assumptions and see if they ring true with you.

- A major purpose for providing OOP classes is to pre-write the code so that the templates don't have to generate the same code over and over again.

- A corollary to that is, fix it in one place and it's fixed everywhere. Or perhaps break it in one place and it is broken everywhere. But even that risk is assuaged by the corollary knowledge that we only have one place to look in order to make a fix.

- We already know from experience that it is not possible to pre-write code so good, so prescient, that it takes into account and anticipates every little nuance and customization that you're going to want to add to your procedures and applications.

- Since our pre-written classes cannot anticipate everything a developer is going to want to tweak, they provide a special set of methods called VIRTUALS. These methods are purposely made VIRTUAL so that the developer or the ABC template system, on behalf of the developer, can DERIVE them and modify the behaviour of those methods as needed on a procedure-by-procedure basis.

- Class VIRTUALS can include empty methods which I call "Place Holders" (sometimes named "Call Backs") which contain no code at all, but provide the opportunity to do some virtual embedding inside your classes. We'll add one of these to the **DVCNStarterSample** class later when I'm demonstrating embedding via template.

Demonstration: If it seems appropriate, because they're getting the concept of Derivation:

Show them how class derivation (inheritance) syntax is a lot like adding extra fields into a GROUP structure being derived from another group structure.

```
MyGroup   GROUP(SomeGroupType)
Extra1    STRING(11)
          END
```

This syntax derives a new group from a pre-defined group type, expanding it's functionality by adding an extra field while still incorporating the fields from the PARENT group. If you can understand that, you can understand class derivation!

Since groups do not contain executable code the derivation principle stops there where groups are concerned. Because classes do contain executable code this derivation or inheritance principle is expanded in OOP classes to incorporate the ability to include extra methods and more important, to incorporate the derivation of virtual methods for the purpose of behaviour modifying these methods where our implementation of them requires it.

Definition Of ABC Compliant

ABC compliant templates provide auto-derivation functionality inside your applications, the purpose of which is to make virtual changes to the behaviours of the ABC compliant classes that they attach to your procedures.

You now have a more or less complete definition of an ABC compliant class and ABC compliant wrapper template:

ABC compliant templates provide auto-derivation functionality inside your applications, the purpose of which is to make virtual changes to the behaviours of the ABC compliant classes that they attach to your procedures.

VIRTUAL methods are an attribute of OOP classes. Class and Method DERIVATION is a technique that can be used with OOP classes to make an existing class incorporate new behaviours without changing the original code.

OPTIONAL SECTION

Demonstration: If there is plenty of time, manually embed some changes into the STRT.Method1() and STRT.Method2() to show that this is in fact the case. Illustrate also that there are two key places that embeds can be placed, either before the PARENT call (with or without a RETURN) or after the PARENT call. Where embeds are placed and what you do inside a derived method via embedding is dependent on your objective. Point out also that the syntax for class derivation involves naming the parent class inside parenthesis following the CLASS() declaration.

END OF OPTIONAL SECTION

Derivative Classes

Derivative classes are classes that incorporate the abilities of a parent class in order to extend or modify that class for some new, usually more specific purpose. They do not physically modify the original parent class code which continues to work in other contexts as originally designed.

I'm next going to illustrate virtual methods in a more manual way so that it becomes perfectly clear that what you've just seen is not solely some magic provided by the template system.

The real magic is based firmly in the OOP attributes of the language itself. That reemphasizes the point I made earlier about the **Clarion Language** being the basis of the **Clarion Development System**. With screen designers, embeditors and templates leveraging – for our benefit – what the language already offers.

With that point driven home yet again, I hope you'll believe me when I repeat how important it is for you to gain as great an understanding of the Clarion language and the mechanisms in the Clarion IDE that tie into this, as your time and career path permit. It's not that hard and it's the smart thing to do.

OPTIONAL
The often-repeated aphorism "work smarter not harder" could be understood to imply that you should also become lazy, spend more time at the beach, and become a "cobbler" or "paint-by-number" developer, becoming co-dependent on someone else to provide you with the solutions to your development challenges. On the other hand, it could also be understood to imply – and this is my take on it – that you gain a thorough understanding of the development tool you are using and the underlying principles that make it behave as it does, in which case you'll really be smarter since you'll have way more options, more control over your application and still not have to work any harder.
END OF OPTIONAL

Demonstration: Now open the Devcon example OOP module called DVCNDERI.INC and begin to discuss it.

This class is called **DVCNStarterDerivation**. I've purposely kept this one simple since it's only here to illustrate derivation of classes or as our current slide suggests, DERIVATIVE CLASSES.

We immediately know that this class derives from DVCNStarterSample by looking at its CLASS definition.

```
DVCNStarterDerivation CLASS(DVCNStarterSample)
```

Enclosed in parenthesis right after the CLASS () declaration is the name of its PARENT class.

If you look above this line at the include statement preceding it you'll see that the HEADER FILE for the PARENT class has been included here so that it is incorporated into the compile steam before the class declaration. Since this class is going to derive that other class, we're making sure here that the other class has a chance to be scanned by the compiler before we make a new declaration incorporating it.

```
   INCLUDE ('DVCNSMPL.INC'), ONCE
```

Derivative Classes inherit, and can make direct use of, all of the functionalities of the PARENT class that are not marked PRIVATE. Perhaps you'll remember that the CONSTRUCT and DESTRUCT methods in the DVCNStarterSample class were declared as PRIVATE methods. Methods 1 and 2 were declared as VIRTUAL and Method 3 was declared as (Public) NON-VIRTUAL.

Demonstration: You may need to go back to DVCNSMPL.INC to point this out if you think that this has been forgotten or has gone over their heads.

This means that in this derivative class we have the opportunity to derive methods 1 and 2 in such a way that these methods (even when called from inside the parent class) behave according to the new definition of them here in our derivative class. That's because these methods are VIRTUAL. Virtual methods are able to call into the future. VIRTUAL methods behave even inside the parent class according to the plan laid down in the derivative class. The whole point is to make old code behave in a new way (if required).

Make sure you understand this. We haven't physically changed the code of the parent class. If that parent class is used directly in another setting, it behaves

exactly as it was originally coded to do. If it's used in the context of a derivation, then it behaves along the lines dictated by the derivation.

Method 3 in DVCNStarterSample defaults to becoming a public NON-VIRTUAL method, since it does not use the VIRTUAL attribute, nor does it use the PRIVATE or PROTECTED attribute. That means this class can **override** its behaviour and replace it with new behaviour which is exhibited when that method is called from the derivative class. If that overridden method is called inside the parent class, which well might be, it continues to behave as coded in the parent class, paying no heed to what's been done in the derivative class.

The PRIVATE methods in our parent class DVCNSampleStarter are not visible or reachable by the derivative class. As you can see we've added a new constructor in this class which executes purely in the context of this derivative class. It does not override the parent class constructor (since it's private, after all) nor does it replace the parent class constructor. In the parent class, the private constructor declared and implemented there continues to do what it was coded to do, with absolutely no consideration or knowledge of the constructor in the derivative class.

Our derivative class introduces a new method called Method 4, again a VIRTUAL which could be derived by a second derivative class that uses this one as its parent.

Derivative Classes - Further Generations

Classes already derived from other classes may still themselves be further derived as needed by the context, in order to modify behaviours or introduce new behaviours.

If we were to do just that, I mean create a derivative class from an already derived class, the VIRTUAL methods in the original parent class may still be derived by this new derivative class, in order to introduce still newer behaviours if required. The qualifier, if required is important. A derivative class as you've already seen from our earlier embedding experiment can use a virtual method AS IS or it can add to its behaviours, or it can completely replace its behaviours.

In other words, classes are by design and definition, more abstract than pure, NON-OOP procedural code. To do what we're able to do here with these few lines of OOP code that define our derivative class **DVCNStarterDerivation**, would, in all likelihood, require us to clone and modify the parent code in a NON-OOP, procedural setting.

Demonstration: Now, open the DVCNTEST.APP application and navigate to the same test procedure called StarterSampleTemplate (). Right click and select Extensions and touch the StarterSample (Devcon) template in the Extension and Control Templates list presented by the Clarion IDE. Next click the Classes button and uncheck the "Use Default ABC: DVCNStarterSample" checkbox and check the "Use Application Builder Class" checkbox. In the dropdown control that becomes enabled when you do this select DVCNStarterDerivation.

The SampleStarter (Devcon) template applied to this procedure, as you can see attaches by default the **DVCNStarterSample** class which we've shown you in some detail. These ABC classes tabs make it easy to force the template to use another class instead of the default insisted on by the template. I can ask it to use another Application Builder Class (which must, by the way, be ABC compliant

or it won't appear in this drop-down list). I'm going to ask it to use our derivative class **DVCNSampleDerivation**.

Save those changes and exit back to the procedure tree. Right click the SampleStarterTemplate () procedure and select SOURCE to enter the Embeditor and slide down to where the STRT class is instantiated.

You'll see now that the STRT instance which was earlier declared by the template as a CLASS (**DVCNStarterSample**) has now been declared as a CLASS (**DVCNStarterDerivation**).

That means we're potentially about to create a derivation of an already derived class - if we embed into any of the listed methods. And you can see the effects right here inside the Clarion embeditor. This class now sports three methods:

```
STRT   CLASS(DVCNStarterDerivation)
  Method1    PROCEDURE(*CSTRING xText,<STRING xOptText>,  |
                        LONG xVal=100),BYTE,PROC,DERIVED
  Method2    PROCEDURE(*PropertyGroup xGrp,BYTE xVal=0),BYTE,PROC,DERIVED
  Method4    PROCEDURE(),DERIVED
END
```

Two of the three methods are derived from what we might now call the grand parent class (**DVCNSampleStarter**) while Method 4 is derived directly from the parent class (**DVCNSampleDerivation**).

Unless I embed behavioural changes into these methods by introducing new code, the procedure (called SampleStarterTemplate ()) still behaves exactly as it did before, with one exception. I showed you earlier an OMIT/COMPILE structure that tested for the presence of **DVCNStarterDerivation** using the omit flag, **_DVCNStarterDerivation_**. That code attached to the method 4 button on our test procedure now executes the code available in the Method4() of the DVCNStarterDerivation class.

Demonstration: At this point exit the embeditor, compile and run the app and start the procedure called SampleStarterTemplate (). Click the buttons one by one to indicate all the functionality is exactly as it was with the exception of the MESSAGEBOX title which we've programmed to name the actual parent class in use. The button named Button 4 also indicates that it has successfully called Method 4 which is only available in the DVCNStarterDerivation class.

As you can see our new class, which has introduced a new method and a new message box title, still derives all of the behaviours built into the parent class **DVCNStarterSample**. We did not change that parent class. We derived it and introduced changes to it using the derivation of classes and derivation of

VIRTUAL methods. I've also shown you that derivation can run deeper than one generation.

IF TIME PERMITS DO THIS:

Just to prove that our original DVCNStarterSample class still works exactly as it did before, I'm going to create a copy of this procedure and have it use **DVCNStarterSample** class directly. Once I recompile this app you'll see that things in that class have not changed at all and I have both implementations running on the same app.

END OF IF TIMER PERMITS

The Wrapper Template

We've provided an ABC compliant wrapper template for our sample ABC compliant class. To make it hook another ABC Compliant Class only one, small change is required.

Demonstration: Close the application and navigate now to:
Setup ->Template Registry -> SampleStarter -> Edit Definition

Indicate that this is the wrapper template in use on the application procedure called SampleStarterTemplate () in our demonstration application DVCNTEST.APP. Begin with the following introduction…

While I've spent a lot of time harping about getting to know Clarion code, I'm not going to spend great amounts of time on the Wrapper template. The reason for that should be relatively obvious by now. The "inventing", the "creativity", the "solution" you're trying to address with your Class/Template combination is done at the Clarion Code level when you build and test your new OOP class. For a wrapper template to do anything useful, it must have something to wrap.

So the OOP class must exist, be working, and have been tested before you ever get to designing a wrapper for it. The interface of your template is consequently determined by the nature of your OOP class. Your template must provide information that your OOP class needs to know in order to do its job.

Template code is basically script. Its procedural and it sure as heck is not OOP. I can't teach you in an hour or two what I've learned about Clarion templates in the last 13 or 14 years since Clarion introduced the template language in Clarion Data Base Developer - not that it takes all that long to learn how it works.

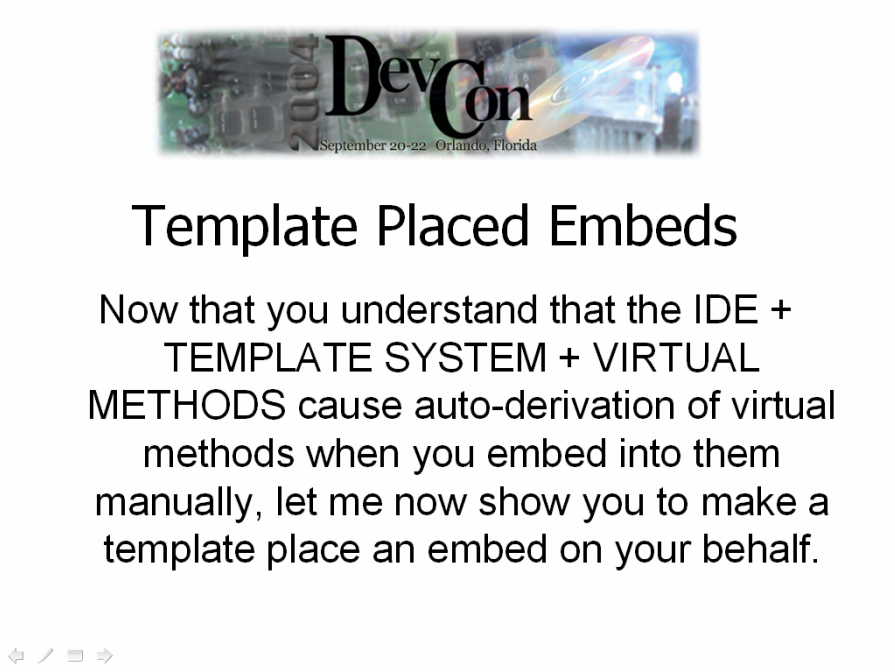
The best way to learn to write templates is to experiment and write some templates! Start with simple ones and graduate to more complicated ones as you

get better. In that regard, it's no different than writing programs by hand. You learn the language; how it uses variables and constants; how it uses subroutines; how it uses control constructs like LOOP, CASE and IF; how you access the file definitions in the dictionary and, of course, how you access the class definitions read into memory when Clarion runs its class registration process.

If you were to create an ABC Compliant Class of your own, modelled on the starter class I've provided, and call it say, **MyFirstClass**, you could create your first wrapper template for that class in about 30 seconds by copying and pasting the sample template I've provided, called **SampleStarter**, giving it a new name and making one small change in Line 7 to insert the name of your class where I presently have inserted **DVCNStarterSample**.

Here in the realm of templates, unfortunately things are not OOPified and we're stuck with having to clone and modify in order move forward.

Demonstration: At this stage you should demonstrate that by creating a new template called **SampleStarterDerivation** using copy and paste. Change line 7 to incorporate the name of our sample derivation class **DVCNSampleDerivation**. Close the template editor, reload your application, (point out that the template registration process is running) and go to the procedure StarterSampleTemplate () and remove the StarterSample (Devcon) template.



Now that you understand that the IDE + TEMPLATE SYSTEM + VIRTUAL METHODS cause auto-derivation of virtual methods when you embed into them manually, let me now show you to make a template insert an embed on your behalf, automatically.

Before you do that of course you will have decided some things:

1) Where you want to place the embed
2) What the embed should do
3) Write the embed code manually first and test it
4) Drop the **EmbedInformation** Template provided
5) Copy its output and your embed code into the template
6) Adjust the code to be output by the template
7) Add to the template interface any prompts required to collect information
8) Add any conditionals to the output
9) Identify your embed as having come from your template
10) Test

Demonstration: At this stage add a new method to the DVCNStarterClass called AutoInit. Make this a "PlaceHolder" meant simply as a repository for template generated initialization code.

Refresh the classes so that this is read in, and drop the EmbedInformation template on it and bring that back to our test template. Make the embed initalize the SELF.WhoAmI property with some easily recognizable value. Compile and test.

Template Language

Template language strongly resembles Clarion and if I've helped convinced you of the merits of spending more time on your Clarion language skills, no doubt you're prepared to do the same with template language.

I've already admitted that I can't teach you template writing in an hour or even in a day. Template language is a real language – a script language – that's interpreted not compiled. If you're going to write templates, you're going to have to learn to read template script. I think I've given you enough to get started here with this demonstration.

Fortunately, template language strongly resembles Clarion and since I've convinced you of the merits of spending more time on your Clarion language skills, no doubt you're prepared to do the same with template script.

Template commands are distinguished from Clarion code with a # (pound) sign as you've already seen. Template variables, called Symbols start with a % (percent sign). Control structures include a Clarion-like #LOOP as well as well as a #FOR loop like C or C++, C# and JavaScript.

Screen design in the template system is done blind. There is no Screen Designer tool for templates. I've often talked about writing such a tool but haven't gotten to it. Here's an opportunity for budding tool designers to create something really useful that hasn't been done by somebody else.

Two things you must do in order to become a good template writer is get to know all the symbols that:

1) Represent the files, keys, fields and so forth coming from the dictionary
2) Represent the OOP headers coming from the class registration process

For the user, ABC Compliant templates do a lot of the grunt work of implementing classes in your ABC application that developers would otherwise have to do themselves. Unless, of course, you plan to do what we've been discussing here for the last number of hours.

While the tool-maker still has to do the grunt work outlined in the ten steps discussed earlier, he/she only has to do this once to begin to reap the benefits of application development automation as provided for in the domain of ABC Compliancy.

I hope I've been able to show you in a repeatable way how you can create ABC Compliant Classes and ABC Compliant Wrapper Templates. Thank you. That's a wrap!

Your Homework Is...

The complete text from this talk is available
for download from this web link:
http://devcon.getaloadathis.com/devconooptemplates.zip

The complete text from this talk will be available for download - as soon as I put it up - from the link http://devcon.getaloadathis.com/devconooptemplates.zip

Pages 209 to 279 of the training guide that you were given on day one of the pre-conference training seminar, contain the same basic information conveyed here from a slightly different angle using different demos and a much larger set of practice templates.

Training guide pages 209 to 279 also explain how to use the other demonstration templates provided that I touched on briefly today. This material is a great follow-up and refresher course of what I've shown you today.

Please Note: The Training Guide, Classes and Templates discussed in the Training Guide are not included in this zip. They are the property of Soft Velocity and are not mine to give away. Those of you who attended the Devcon pre-conference training already have that material in the form of a 300 page manual including a CD.