Clarion Magazine

Clarion News

- » iQ-Notes 3.61
- » Year End Tax Sale
- » Data Equity 50% Sale
- » Lodestar Software Moving Sale
- » Australian Clarion Users Group Web Site
- » Clarion 6.3.9058 Released
- » SetupBuilder 6.7 Build 2083
- » CHT 11D1.04
- » J-Media 1.15
- » TDC Version Control, Issue Tracking
- » 1st Logo Design Three Day Sale
- » Free Goodies From 1st Icon Design
- » StrategyOnline Christmas Schedule
- » Organize365 7.04
- » J-Cal 2.06
- » GTL 6.36
- » SetupBuilder Build 2077
- » HyperBrowZe
- » Data Interoperabilty Community of Interest Handbook, 2nd Edition
- » Free Metabase 6.9.6
- » DMC 1.102
- » DMC Price Reduction
- » Clarion Mappoint Templates 2.0.300
- » SV Updates Coming
- » Handy Tools Clarion# Zip Application
- » iQ-XML 1.27d
- » DynaLib Source 4.0.9
- » Data Management Center Updated
- » EasyExcel 4.04
- » New Icon Set
- » Free Organize365 SDK
- » J-Spell 1.53
- » African CHT Dealer
- » vuLimiter Limits Concurrent Users
- » 1st Logo Design New Look
- » Organize365 7.0
- » Clarion-Made Text Analytics Suite
- » Clarion.NET Game Of Life
- » Free Beginning SQL Server 2005 for Developers
- » Keystone Sample Clarion# Apps
- » J-Media 1.10
- » CHT Free Zip'N EMail Utility
- » Free Wallpaper
- » .NET Reflection App
- » AFE + RPM Discount Sweetened and Extended

[More news]

Latest Subscriber Content

Testing Clarion# Libraries With NUnit

So you've started building your Clarion# code library (or libraries). How do you ensure that everthing's working the way it should? With automated unit testing, of course.

Posted Thursday, December 27, 2007

A Simple Clarion# PDA Application

Skip Williams shows how easy it is to create a simple PDA application in Clarion#.

Posted Wednesday, December 26, 2007

Merry Christmas, a Happy New Year, and a Holiday Schedule!

A Merry Christmas to all who celebrate the season! Clarion Magazine will be mostly closed between now and January 2, but we still have some articles on the way...

Posted Friday, December 21, 2007

Designing Clarion# Libraries

Creating code libraries is considerably easier in Clarion# than in Clarion, in part because you no longer need to declare prototypes to use library code. And .NET's namespaces make it easy to organize your code into a meaningful hierarchy.

Posted Thursday, December 20, 2007

Clarion# And The Google Calendar API

Randy Rogers shows how to use the Google Calendar API to add public and private calendars to your Clarion# applications.

Posted Monday, December 17, 2007

Clarion# Language Comparison

Mike Hanson has prepared a cross-reference showing Clarion# equivalents to VB.NET and C# statements. Topics include program structure, comments, data types, constants, enumerations, operators, choices, loops, arrays, functions, strings, exceptions, namespaces, classes, interfaces, construtors, using object, structs, properties, delegates, events, and I/O. Based on a VB.NET/C# document by Frank McCown. Updated Dec 17 with latest USING/NAMESPACE syntax.

Posted Monday, December 17, 2007

Should Clarion# Drop Automatic Instantiation?

Dave Harms argues that Clarion# would be a clearer, easier-to-use language if it no longer permitted automatic instantiation of classes. This article is available to subscribers and to anyone who has registered at ClarionMag.com.

Posted Thursday, December 13, 2007

The Clarion.NET FAQ - Updated Dec 13

A list of frequently-asked questions about Clarion.NET/Clarion#, and some hopefully informative answers. Latest update: "Why should I choose Clarion# over VB.NET or C#?"

Posted Wednesday, December 12, 2007

Understanding Clarion# Strings

There are some significant differences between Clarion strings and .NET strings. Clarion# introduces the ClaString class, which for the most part allows Clarion# to use strings the same way classic Clarion uses strings. Here's what you need to know about this new data type.

Posted Monday, December 10, 2007

Source Code Library 2007.11.30 Available

The Clarion Magazine Source Code Library has been updated to include the November source. Source code subscribers can download the Jan-November 2007 update from the My ClarionMag page. If you're on Vista please run Lindersoft's Clarion detection patch first.

Posted Friday, December 07, 2007

[Last 10 articles] [Last 25 articles] [All content]

Source Code

The ClarionMag Source Code Library

Clarion Magazine is more than just a great place to learn about Clarion development techniques, it's also home to a massive collection of Clarion source code. Clarion subscribers already know this, but now we've made it easier for subscribers and non-subscribers alike to find the code they need.

The Clarion Magazine Source Library is a single point download of all article source code, complete with an article cross-reference.

More info • Subscribe now

Printed Books & E-Books

- » Clarion.NET FAQ
- » Clarion# Language Comparison
- » Testing Clarion# Libraries With NUnit
- » A Simple Clarion# PDA Application
- » Designing Clarion# Libraries
- » Clarion# And The Google Calendar API
- » Clarion# Language Comparison

[More Clarion & .NET]

[More Clarion 101]

Latest Free Content

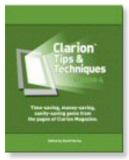
- » Merry Christmas, a Happy New Year, and a Holiday Schedule!
- » Clarion# Language Comparison
- » Should Clarion# Drop Automatic Instantiation?
- » The Clarion.NET FAQ - Updated Dec 13
- » Source Code Library 2007.11.30 Available

[More free articles]

Clarion Sites

- » Clarion Australian Users Group

Clarion Blogs



From The Publisher

About Clarion Magazine

Clarion Magazine is your premier source for news about, and in-depth articles on Clarion software development. We publish articles by many of the leading developers in the Clarion community, covering subjects from everyday programming tasks to specialized techniques you won't learn anywhere else. Whether you're just getting started with Clarion, or are a seasoned veteran, Clarion Magazine has the information *you* need.

Subscriptions

While we do publish some free content, most Clarion Magazine articles are for subscribers only. Your subscription not only gets you premium content in the form of new articles, it also includes all the back issues. Our search engine lets you do simple or complex searches on both articles and news items. Subscribers can also post questions and comments directly to articles.

Satisfaction Guaranteed

For just pennies per day you can have this wealth of Clarion development information at your fingertips. Your Clarion magazine subscription will more than pay for itself - you have my personal guarantee.

Dave Harms

ISSN

Clarion Magazine's ISSN

Clarion Magazine's International Standard Serial Number (ISSN) is 1718-9942.

About ISSN

The ISSN is the standardized international code which allows the identification of any serial publication, including electronic serials, independently of its country of publication, of its language or alphabet, of its frequency, medium, etc.

Clarion Magazine

Clarion News

Search the news archive

PostgreSQL Security Releases

The PostgreSQL Global Development Group has released updated versions which patch five security vulnerabilities. These releases update all current PostgreSQL versions, including 8.2, 8.1, 8.0, 7.4 and 7.3. They are considered CRITICAL and PostgreSQL DBAs and sysadmins should install the update as soon as they reasonably can. All security fixes will be included in the upcoming version 8.3 release candidate.
Posted Monday, January 07, 2008

PiFolio Word Reporter In Clarion.NET

Hanspeter Stutz has a PiFolio Word Reporter example built with Clarion.NET. You don't need to have Clarion.NET to run it but you need MS Word (of course) and the .NET Framework 2.0 installed. If you don't have clarion.net yet but you're interested in the source code open Main.cln and/or Mainform.cln and/or PiFolio.cln in an editor.
Posted Monday, January 07, 2008

J-Skype 2.0

J-Skype 2.0 has been released. If you already own J-Skype 1, you can upgrade to J-Skype 2 for an extra $29. J-Skype 1 will still be supported, but no new features will be added to it. If you don't own J-Skype 1, you can purchase J-Skype 2 at a discounted price of $108 until the end of January, at which time the price will increase to $149. ($108 is what it would have cost if you had purchased J-Skype 1, and then upgraded to J-Skype 2.)
Posted Monday, January 07, 2008

WindowID 2.00

WindowID 2.00 is now available. This release not only identifies where you are, but also how you got there. You will see the menu call sequence plus the buttons pressed afterwards in the WindowID information. The upgrade to this version is free of charge for users with a current valid maintenance plan for WindowID 1.x.
Posted Monday, January 07, 2008

iQ-Notes 3.61

iQ-Notes 3.61 is now available. Changes include: Ability to switch/change the printer when printing notes; Ability to set next Sync time when working offline for long periods of time; Ability to recover automatically from a corrupted notes file; Added Dutch Language Set; When notes were Password Protected, an orphan Checkbox would appear in the popup menu item.
Posted Monday, December 31, 2007

Year End Tax Sale

Yes, Lee White is having a tax sale... as in he'd love to pay MORE income taxes for 2007. Only a few days left. 25%

discount: Fax enable your App and save $100.00; 28% discount: Preview your reports and save $79.00; 100% discount: Get PNet free with any new order of AFE or RPM. Discount coupon for PNet will be sent by email within 1-2 business days after your order is processed.
Posted Sunday, December 30, 2007

Data Equity 50% Sale

Data Equity (www.DataEquity.com) hopes that their fellow Clarion developers have had an awesome year and have much to be thankful for. It is in the spirit of the season that they would like to take this opportunity to offer a 50% discount on all of their products. This is a limited time offer; take advantage by using coupon code HOLIDAY07. Merry Christmas and Happy Holidays from the Stockstill Family to Yours!
Posted Saturday, December 22, 2007

Lodestar Software Moving Sale

No, Lodestar Software is not moving. But our web servers have moved. Due to a change in policy by our ISP (no more high speed service in our area!), we've had to relocate our web sites to offsite servers. To offset this increased cost we're moving product! Through the end of 2007 you can get a 25% discount on AFE and fax enable your application or a 28% discount on previewing your reports with RPM. And with any new purchase you get a 100% discount on PNet so you can add postnet barcodes to your reports.
Posted Friday, December 21, 2007

Australian Clarion Users Group Web Site

The new Australian Clarion Users Group web site by is an initiative Geoff Spillane (current Australian Distributor of all things Clarion and Lindersoft) and Tony York (who has run the last couple of DevCons).
Posted Thursday, December 20, 2007

Clarion 6.3.9058 Released

The latest release of Clarion 6.3 is now shipping.
Posted Thursday, December 20, 2007

SetupBuilder 6.7 Build 2083

Lindersoft has released maintenance build 2083 of SetupBuilder 6.7. This build has been successfully tested on Windows Server 2008 RC1. Get the latest version by selecting "Check for Updates" from within the SetupBuilder 6 IDE or the "Check for SetupBuilder 6 Updates" shortcut found in the Start Menu for SetupBuilder 6.
Posted Thursday, December 20, 2007

CHT 11D1.04

Clarion Handy Tools has released its fourth-quarter build, sub-update 11D1.04. It has been download-tested and compile-tested in both Windows XP and VISTA. Subscribers with current subscriptions please use your WEBUPDATER installer to download and install all new and revised classes, templates, applications and utilities.
Posted Thursday, December 20, 2007

J-Media 1.15

J-Media 1.15 is now available. Several users have reported problems with certain video formats under Windows Vista, and this release should fix those issues.

Posted Thursday, December 20, 2007

TDC Version Control, Issue Tracking

TDC lets you keep all your valuable information and resources in a centralized repository, and generates historical information of all changes not only for developers but also for customers. Includes issue tracking (bugs, improvements, new functionality, queries, tasks, etc) to ease software project management. Fully functional demo version of TDC available.

Posted Thursday, December 20, 2007

1st Logo Design Three Day Sale

1st Logo Design has a variety of products on sale, including the Mallorca Collection (Reg. $349 - 3 day sale $99), the LNS Collection (Reg. $199 - 3 day sale $99), the Webmaster Collection (Reg. $69 - 3 day sale $49) and Bundle Collections. This offer expires on Saturday December 22, 2007

Posted Thursday, December 20, 2007

Free Goodies From 1st Icon Design

1st Icon Design has made all the pixel icon collections available free as well as some screen savers and wallpapers (1680x1050).

Posted Thursday, December 20, 2007

StrategyOnline Christmas Schedule

The StrategyOnline office will be closed from Dec 20-26, 2007.

Posted Thursday, December 20, 2007

Organize365 7.04

Organize365 version 7.04 has been released. A 60-day trial version is available, as is a free SDK.

Posted Thursday, December 20, 2007

J-Cal 2.06

J-Cal 2.06 supports English, Afrikaans, Danish, Dutch, French, German and Italian. If you would like other languages added to J-Cal, please send Gary the days of the week and the months of the year in your language of choice.

Posted Thursday, December 20, 2007

GTL 6.36

GTL 6.36 is now available from the Par2 download page. This release offers a "Generate Only" option and revised docs (such as they are).

Posted Wednesday, December 19, 2007

SetupBuilder Build 2077

Lindersoft has released build 2077 of SetupBuilder 6.7. Get the latest version by selecting "Check for Updates" from within the SetupBuilder 6 IDE or the "Check for SetupBuilder 6 Updates" shortcut found in the Start Menu for SetupBuilder 6.

Posted Friday, December 14, 2007

HyperBrowZe

HyperBrowZe is a new paradigm for browsing relational databases from Enabling Simplicity, publishers of UltraTree. Demo available.
Posted Friday, December 14, 2007

Data Interoperabilty Community of Interest Handbook, 2nd Edition

The Data Interoperability Community of Interest Handbook has been released in a second edition. Added to the book are Problems and Exercises at the end of every chapter. Also added is a comprehensive Return on Investment model that shows that the COI approach costs 50% of the Stove Pipe approach, and when adding one additional community and/or significant database to the interoperability environment, the cost is only about 10% the cost of the Stove Pipe approach. Also, check out a good review of the COI book.
Posted Friday, December 14, 2007

Free Metabase 6.9.6

Metabase Version 6.9.6 has been released. If you want a free copy please go to the Whitemarsh website and ask for one. There is also a new short paper, Quality Data-Centric Engineering and Management.
Posted Friday, December 14, 2007

DMC 1.102

DMC 1.102 has been released. Changes include: Added process cancellation on data transfers; Changed main frame to resizable and updated inner screens to 1024x768 compatibility; Fixed transfer to XLS bug.
Posted Friday, December 14, 2007

DMC Price Reduction

The price of DMC has been reduced during the beta period. Those who have already purchased will receive lifetime updates. The beta price is now €69.
Posted Friday, December 14, 2007

Clarion Mappoint Templates 2.0.300

WC Software Development Inc. has released the Clarion Mappoint Templates 2.0.300. Clarion for Mappoint Templates allows developers to add MS Mappoint 2004/2006 to their applications. Full Source code, No DLLs, supports Clarion 5.5, 6.x, ABC and Legacy. Current users of the Clarion Mappoint templates can download the update for free. Complied Demo available for download, requires Mappoint(tm) 2004 or Mappoint(tm) 2006 to be installed.
Posted Wednesday, December 12, 2007

SV Updates Coming

SoftVelocity has a numer of updates staged for release next week. Clarion 6.3 9058 will be released to third party vendors. Clarion 7 and Clarion.Net beta refreshes will go out, with lots of new features and fixes. There are also updates and new lessons coming for the Clarion# by Example Course.
Posted Friday, December 07, 2007

Handy Tools Clarion# Zip Application

The zip file containing this Clarion# project contains all the necessary DLLs, .EXE and source files in order to both run the example application, even if you don't own Clarion.NET, and of course, to compile it if you do own Clarion.NET.

Posted Friday, December 07, 2007

iQ-XML 1.27d

iQ-XML 1.27d contains Corrections to some online help examples, and writer functions were made more efficient. This includes a fix to a regression in 1.27c.

Posted Friday, December 07, 2007

Clarion Magazine

Testing Clarion# Libraries With NUnit

by Dave Harms

Published 2007-12-27

In Designing Clarion# Libraries I explained how the structure of code libraries has changed in Clarion#. The need for INCLUDE statements and separate prototypes is gone, thanks to .NET reflection. And NAMESPACE and USING make it easy to organize your code into a meaningful hierarchy.

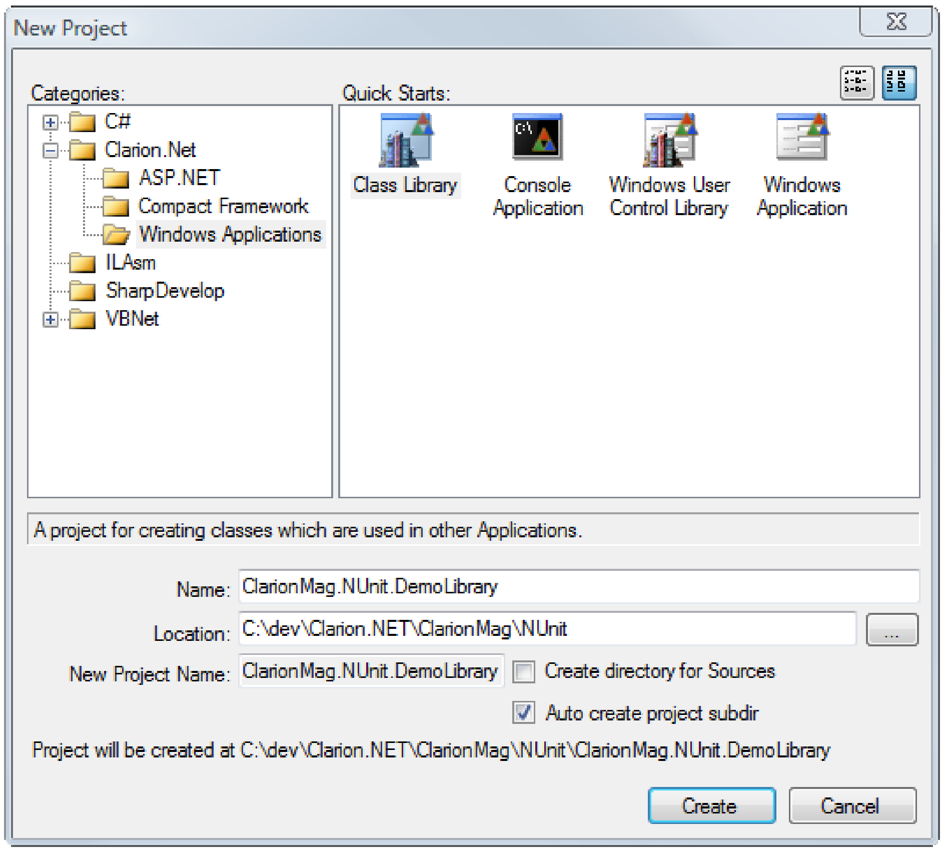
Creating and using class libraries is much easier in Clarion# than it is in Clarion; not only that, but you can also quite easily create comprehensive test suites to make sure your libraries are doing exactly what they're supposed to do. In this article I'll walk you through the process of creating a small library and testing its methods with NUNit.

Creating a library

Clarion.NET comes with a number of preset project types for Clarion#, including one called Class Library. Choose File| New Solution, select Clarion.NET and Windows Applications, then choose Class Library.

Navigate to a suitable directory on your hard drive (you can set the default in Tools | Options | General | Projects and Solutions). My base directory for Clarion Magazine library code is (at least for now) c:\dev\Clarion.NET\ClarionMag. As I indicated in Designing Clarion# Libraries, you have a lot of flexibility in how you can structure your directories to align them with your namespaces. I'm experimenting with creating directories to match the first two levels, but I'm naming each solution after the complete namespace.

For example, the source accompanying this article demonstrates creating a library and testing it with the NUnit utility. I'll call the solution ClarionMag.NUnit.DemoLibrary, and I'll place it in ClarionMag\NUnit. Figure 1 shows the new solution settings.


New Project

Categories:
- C#
- Clarion.Net
  - ASP.NET
  - Compact Framework
  - Windows Applications
- ILAsm
- SharpDevelop
- VBNet

Quick Starts:
Class Library
Console Application
Windows User Control Library
Windows Application

A project for creating classes which are used in other Applications.

Name: ClarionMag.NUnit.DemoLibrary

Location: C:\dev\Clarion.NET\ClarionMag\NUnit

New Project Name: ClarionMag.NUnit.DemoLibrary

[ ] Create directory for Sources

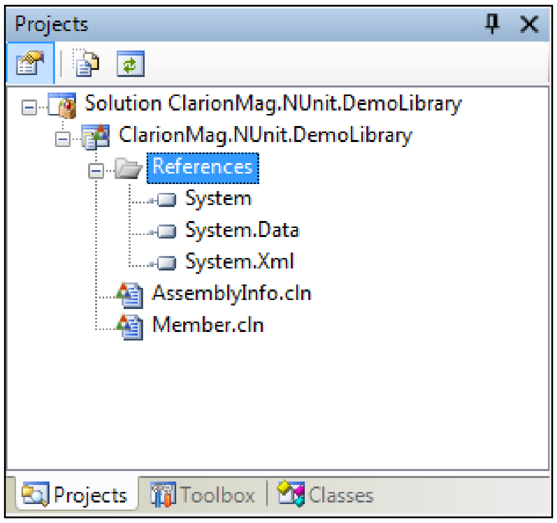
[✓] Auto create project subdir

Project will be created at C:\dev\Clarion.NET\ClarionMag\NUnit\ClarionMag.NUnit.DemoLibrary

Create | Cancel

Figure 1. Creating the solution.

Figure 2 shows the project pad for the just-created class library, with the default set of files.


Projects

```
Solution ClarionMag.NUnit.DemoLibrary
  ClarionMag.NUnit.DemoLibrary
    References
      System
      System.Data
      System.Xml
    AssemblyInfo.cln
    Member.cln
```

Projects | Toolbox | Classes

Figure 2. The project pane

Note that the References node lists three .NET namespaces: System, System.Data, and System.Xml. System contains support for core application functionality, various basic data types, writing to the console, and the like. System.Data contains ADO.NET classes (the .NET successor to ADO, which itself replaced ODBC some years ago), and System.Xml contains standard XML handling classes. You can remove the references you don't need, and add any new references you do need. I'll leave this section untouched for now.

There are two source files: AssemblyInfo.cln and Member.cln, which I'll cover in that order.

AssemblyInfo.cln

AssemblyInfo.cln is preconfigured and looks like this:

```
                MEMBER('')
                NAMESPACE('ClarionMag.NUnit.DemoLibrary')
       USING('System.Reflection')
       USING('System.Runtime.CompilerServices')
       USING('System.Runtime.InteropServices')

       ! Information about this assembly is defined by the following
       ! attributes.
       !
       ! change them to the information which is
      ! associated with the assembly you compile.

       [assembly: AssemblyTitle('ClarionMag.NUnit.DemoLibrary')]
       [assembly: AssemblyDescription('')]
       [assembly: AssemblyConfiguration('')]
       [assembly: AssemblyCompany('')]
       [assembly: AssemblyProduct('ClarionMag.NUnit.DemoLibrary')]
       [assembly: AssemblyCopyright('')]
       [assembly: AssemblyTrademark('')]
       [assembly: AssemblyCulture('')]

       ! This sets the default COM visibility of types in the
      ! assembly to invisible.
       ! If you need to expose a type to COM, use [ComVisible(true)]
      ! on that type.
       [assembly: ComVisible(false)]

       ! The assembly version has following format :
       !
       ! Major.Minor.Build.Revision
       !
       ! You can specify all values by your own or you can
      ! build default build and revision
       ! numbers with the '*' character (the default):

       [assembly: AssemblyVersion('1.0.*')]
```

All those lines enclosed in square brackets are attributes, and represent attribute values that will be added to the *assembly*. In .NET an assembly is either an EXE or a DLL which contains code and data (in IL format), information about the assembly (attributes) and can contain resources.

The attribute fields in AssemblyInfo.cln are pretty much free form, with the notable exception of AssemblyCulture. In most cases AssemblyCulture will be empty, indicating a neutral culture; if you specify a culture, such as 'en' for English or 'de' for German, this assembly will be seen as a satellite assembly containing alternate resources for that culture, rather than executable code.

It's a good idea to fill in the other attribute fields with your company info etc, but none of this is necessary to make the assembly function. I'll have a bit more to say about attributes later on, when it comes to unit testing.

Member.cln

Member.cln is the default source file for your class library, and you should probably rename it right away to something more meaningful. I right-clicked on Member.cln in the Project pad and renamed it to DemoLibrary.cln.

DemoLibrary.cln (formerly Member.cln) looks like this:

```
!~
~ Created by Clarion.
~ User: Dave
~ Date: 17/12/2007
~ Time: 7:53 PM
~!

        MEMBER()

        NAMESPACE('ClarionMag.NUnit.DemoLibrary')
        MAP
        END
```

The first line of DemoLibrary.cln is the familiar MEMBER statement, which has a couple of new attributes in Clarion#: INTERNAL and PUBLIC. These indicate the default access modifier for any structures declared in this source file. These don't appear to be supported by the compiler in the first beta.

INTERNAL vs PUBLIC

As I mentioned in Designing Clarion# Libraries, Classes in .NET have two possible visibilities. A CLASS with no modifier (and no overriding MEMBER attribute) has INTERNAL visibility; it can be seen by any code anywhere else in the assembly (in other words, by any other code in the project). While it's possible you'll want all your declarations to be public, it's more likely you'll have a mix. Although you can use MEMBER(),PUBLIC to default to public, it's generally safer to begin with the assumption that everything is INTERNAL (the default state) and only expose those classes, methods and properties you really want to expose.

The next line is the NAMESPACE directive; all classes I declare in this file will automatically have ClarionMag. NUnit.DemoLibrary. prepended to their name.

Finally, there's a MAP/END statement. In Clarion, MAP implicitly includes the Clarion runtime library definitions, but here it doesn't seem to have that effect, since you can use Clarion# RTL statements even if there is no MAP.

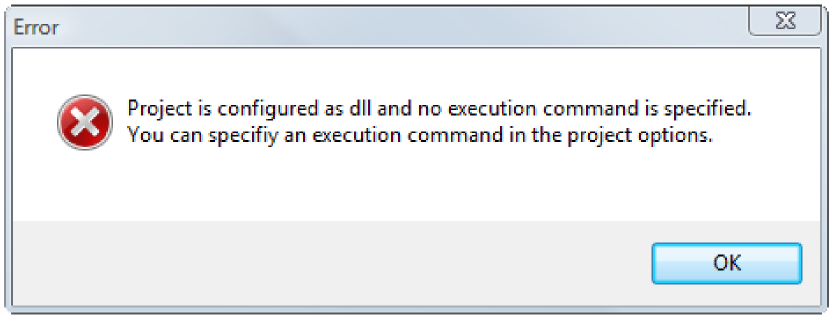
A test class

I've created a simple library class to do simple math operations. It's a pointless class because it doesn't add any useful functionality, so I've called it PointlessMathClass. It's only purpose is to demonstrate a library and provide a basis for testing with NUnit.

Here's the complete source listing for DemoLibrary.cln:

```
                  MEMBER()

                  NAMESPACE('ClarionMag.NUnit.DemoLibrary')
                  MAP
                  END

PointlessMathLibrary  class,public,type
Add                   procedure(long a,long b),long
Subtract              procedure(long a,long b),long
DoMath                procedure(long a,ClaString oper,long b),long
                  end

PointlessMathLibrary.Add       procedure(long a,long b)
  code
  return(a-b)

PointlessMathLibrary.Subtract  procedure(long a,long b)
  code
  return(a-b)

PointlessMathLibrary.DoMath    procedure(long a,ClaString oper,long b)
  code
  if oper = '+'
    return(self.add(a,b))
  elsif oper = '-'
    return(self.subtract(a,b))
  else
    throw new System.AccessViolationException
  End
```

If at this point you hit the compile and run button, the class will compile, successfully, but you'll see the error message in Figure 3.



Project is configured as dll and no execution command is specified.
You can specifiy an execution command in the project options.

Figure 3. Attempting to run a DLL.

The reason for the error is that this is a DLL, not an EXE. Instead of pressing the green compile and run button, choose Build|Build Solution or press F8. The IDE will compile the project without trying to run it as an application.

Now, how do you go about testing this class, short of creating an application? Simply, you create a test class, and then you run that test class using a free tool unit testing tool called NUnit.

Unit testing

The idea behind unit testing is that you exhaustively test all of the individual pieces of your application, rooting out bugs, and then when you stitch those pieces together you're much less likely to run into problems. Not only that, but you maintain a test suite, and any time you introduce some new functionality into the system, or upgrade some component, you run the entire suite of tests. If anything's happened to affect your code, it should show up in the test results.

The first thing you'll need is a unit testing framework. Perhaps the most-used unit testing tool for .NET is NUnit, which is a port of the JUnit Java tool. There are many tools in the xUnit family, including specialized versions of NUnit for Windows Forms and ASP.NET.

You can download zipped binaries or MSI installers for both .NET 1.1 and 2.0 (you'll want the 2.0 version for Clarion. NET). Installation follows the typical process; when complete, you can run NUnit from your Start Menu. But before you run NUnit you'll need to set up a test class.

Creating a test class

When setting up a test class you need to consider whether how your class will typically be used. If the class will be used only from within its own assembly, you'll want to set up a test class that is part of that assembly, and you won't need to be concerned with making the library classes public. But if your class will expose some of its methods to other assemblies, you may want to create your test class in a different project so the test code is duplicating the typical environment.

In this case I'll simply add a test class to my current project. I right-click on the Project name in the Project pad and choose Add | New Item (Figure 4).

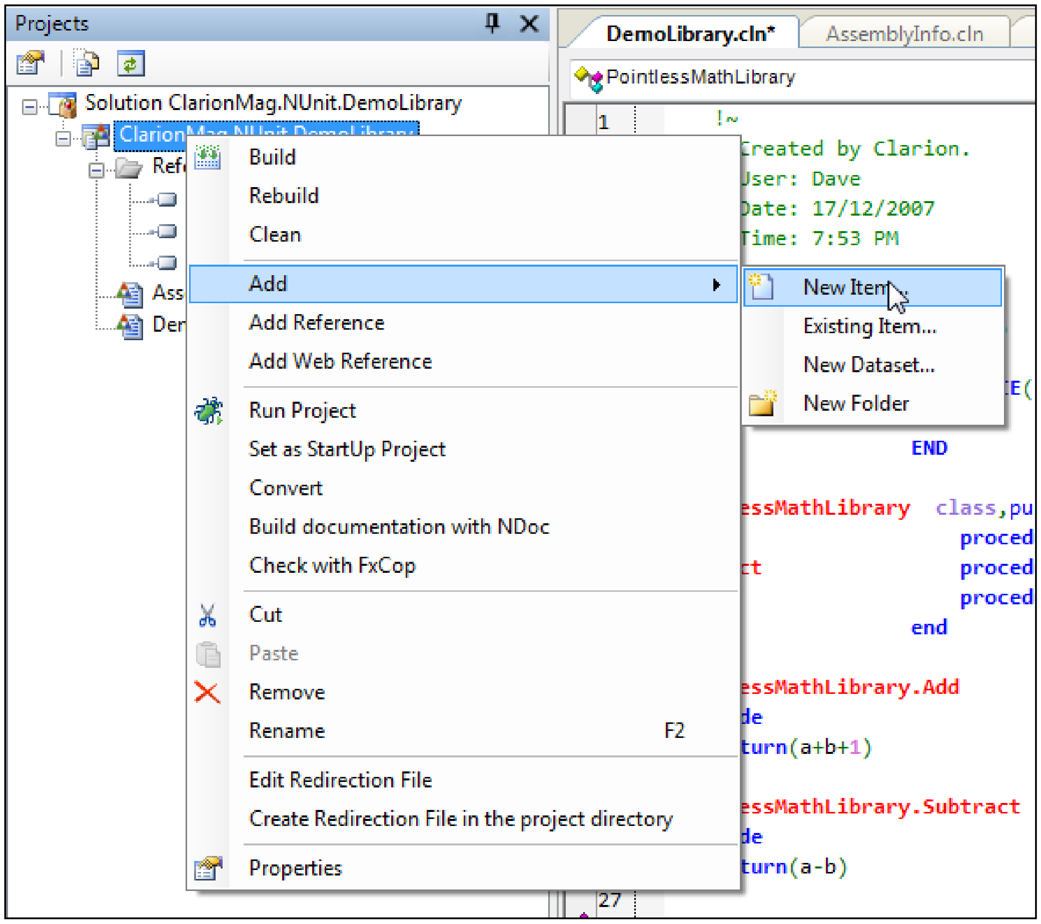
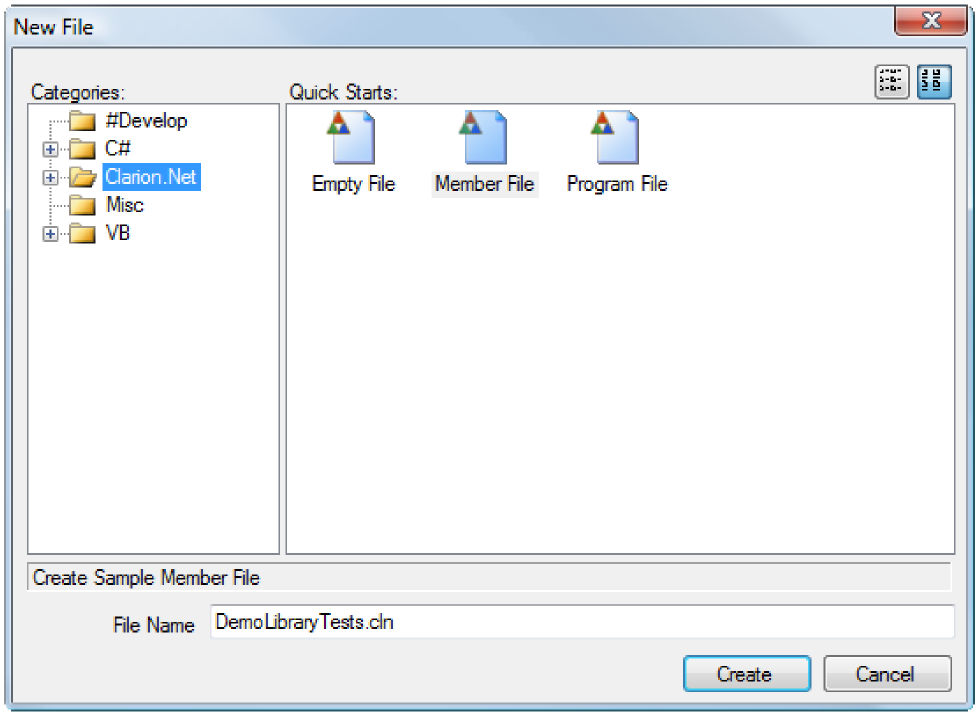


Figure 4. Adding a new item

Figure 5 shows the New File dialog. I've chosen the Clarion.Net category and the Member File quick sStart.



Create Sample Member File

File Name DemoLibraryTests.cln

Figure 5. Adding a new member file

The new member file, which I've called DemoLibraryTests.cln, looks the same as the original Member.cln file (which I renamed to DemoLibrary.cln). I want to make one change, however, before adding my test class, and that's to change the NAMESPACE directive from

```
NAMESPACE('ClarionMag.NUnit.DemoLibrary')
```

to

```
NAMESPACE('ClarionMag.NUnit.DemoLibrary.Tests')
```

Now, both of these files are in the same assembly, so they classes they contain are going to be visible to each other. The only reason I'm changing the namespace is to indicate more clearly the purpose of the new class. Your namespaces should always say something meaningful about the classes they contain.

The code for my DemoLibraryTests class starts off in the usual fashion. Note the two USING statements, one for the DemoLibrary code, and the other for the NUnit.Framework library.

```
                MEMBER()

                NAMESPACE('ClarionMag.NUnit.DemoLibrary.Tests')
                MAP
                END

    Using('ClarionMag.NUnit.DemoLibrary')
    using('NUnit.Framework')
```

The importance of attributes

I mentioned earlier that .NET assemblies can contain a variety of attributes, and NUnit relies on these attributes to know which methods it should call and in what order. Think for a moment about how you'd go about setting up a testing framework for your applications. You'll need at least two things; test code, and a way to call that test code.

In the simplest arrangement you can create an EXE with one or more menu items calling test procedures, whether in that EXE or in another DLL. But you'll also need some standardized way of reporting back the results of those tests. Creating test suites this way is a bit tedious, and if you're calling test procedures in a DLL you not only need to create the DLL, you need to update your test EXE with the right calls to the DLL. And what if you forget to include a test in your EXE? In a large application, you may have hundreds of tests; it's easy for one to be forgotten.

Wouldn't it be nice if you could use just one standardized testing utility that would examine all your EXEs and DLLs, automatically extract any tests, allow you to run them all at once or selectively, and report the errors in a standardized way? That is, of course, exactly what NUnit does, and the secret to using NUnit is to selectively place attributes, which are contained in square brackets, at suitable locations in your source.

For example, the [TestFixture] attribute tells NUnit that DemoLibraryTests is a test class. You have to mark this class as PUBLIC, along with any methods you want NUnit to call.

```
    [TestFixture(Description = 'A simple test class')]
```

```
DemoLibraryTest     class,public
math                &PointlessMathLibrary
```

Any attributes you add to your source code are included in the compiled code; other code, in the same application or in another application, an examine these attributes at runtime. And that's just what NUnit does; you tell it which DLLs or EXEs you want it to test, and NUnit loads up that code and searches for the TestFixture attribute on all the public classes; when it finds that attribute it looks for other attributes that will tell it how to run the actual tests.

DemoLibraryTests contains a reference to an instance of the PointlessMathLibrary class. This reference has to be initialized, and while I could have done it with a constructor I've chosen to use the [Setup] attribute, which tells NUnit to call this method first before any test methods.

```
                    [Setup]
Init                 procedure,public
```

The remainder of the methods have the [Test] attribute, and will be listed by NUnit as runnable tests. The final method also has an [ExpectedException] attribute, which means that it will fail if the wrong exception (or no exception) is thrown.

```
                    [Test]
AddTest                procedure,public
                    [Test]
SubtractTest           procedure,public
                    [Test]
DoMathTest              procedure,public
                    [Test]
                    [ExpectedException('System.ArithmeticException',|
                      'System.ArithmeticException')]
DoMathExceptionTest     procedure,public
                 end
```

Here's the source for the individual methods. I'm using NUnit assertions to test for various conditions. If the assertion fails, NUnit reports an error. NUnit lets you test equality, identity, conditions, comparisons, expected types, thrown exceptions, strings and substrings, collections, and more. In this example I'm using a simple equality test to determine if a returned value is the expected value. Also note that I'm using the classic syntax, which employs a different Assert class method for each type of test. There is also a newer constraint-based syntax which uses just one method, Assert. That, combined with different constraints.

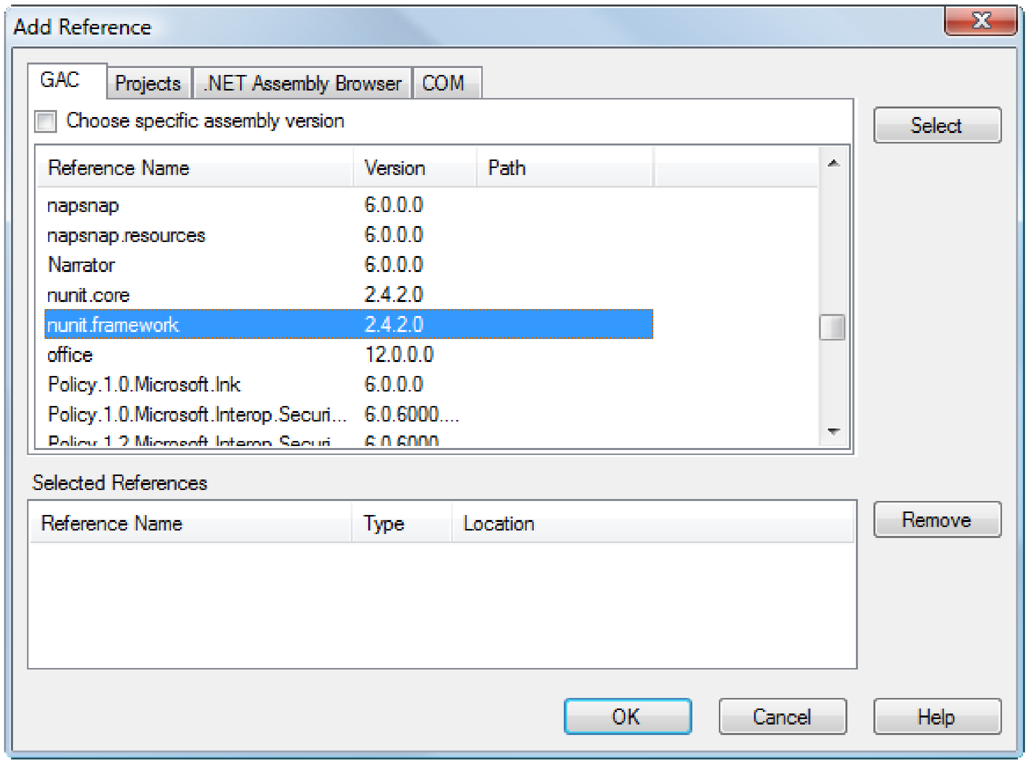
```
DemoLibraryTest.AddTest     procedure
  code
  NUnit.Framework.Assert.AreEqual(5,|
    self.math.Add(2,3),'Error adding')

DemoLibraryTest.SubtractTest       procedure
  code
  NUnit.Framework.Assert.AreEqual(2,|
        self.math.Subtract(15,13),'Error subtracting')

DemoLibraryTest.DoMathTest  procedure
    code
    NUnit.Framework.Assert.AreEqual(2,|
        self.math.doMath(15,'-',13),|
        'Invalid result with - operator')
    NUnit.Framework.Assert.AreEqual(5,|
        self.math.doMath(2,'+',3),|
        'Invalid result with + operator')

DemoLibraryTest.DoMathExceptionTest procedure
i    long
    code
    i = self.math.doMath(15,'/',23)

DemoLibraryTest.Init        procedure
    code
    self.math &= new PointlessMathLibrary
```

Clearly this is pretty basic testing; even on a simple class like this you'd want to something a bit more exhaustive. Think of all the possible ways the methods could be called. What happens when you use negative numbers? When you pass null values? Think about unusual usage and boundary conditions, and don't assume that your methods will only be called the way they should be called. Your job is to make the class as bulletproof as possible, and able to respond gracefully to errors.

Running the tests

If you attempt to compile the class at this point you'll get numerous errors, as the compiler can't resolve either the NUnit attributes or the NUnit.Framework classes. You'll need to find a way to add the necessary assembly reference.

In the Project pad right-click on References and choose Add Reference. You'll see the window in Figure 6.



☐ Choose specific assembly version

Reference Name	Version	Path
napsnap	6.0.0.0	
napsnap.resources	6.0.0.0	
Narrator	6.0.0.0	
nunit.core	2.4.2.0	
nunit.framework	2.4.2.0	
office	12.0.0.0	
Policy.1.0.Microsoft.Ink	6.0.0.0	
Policy.1.0.Microsoft.Interop.Securi...	6.0.6000....	

Selected References

Reference Name	Type	Location

Figure 6. Adding the nunit.framework reference

When you installed NUnit, the install program placed a copy of the DLL containing the NUnit.Framework namespace in the global assembly cache. On the GAC tab, navigate to the nunit.framework line and click on Select, then OK. You will now see nunit.framework listed in the Project pad, under the References node. You should now be able to compile the source.

You have a library, and you have a test class, and you're ready to do some actual unit testing. Fire up NUnit and from the File menu choose Open Project. Navigate to DemoLibrary's build directory; if you've compiled the application in Debug mode, as I have, the EXE will be in bin\debug. Select ClarionMag.NUnit.DemoLibrary.dll and click Open. NUnit will load the class and show you something like Figure 7.

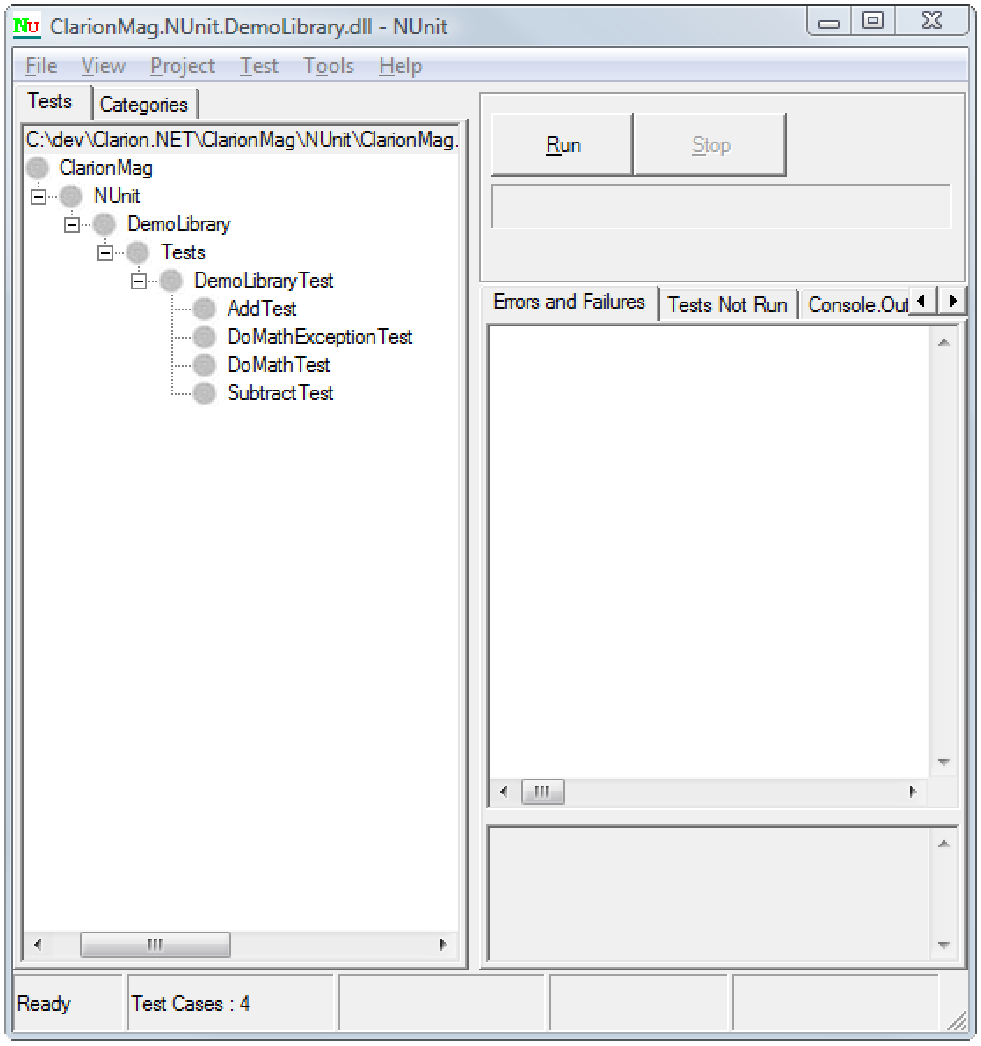
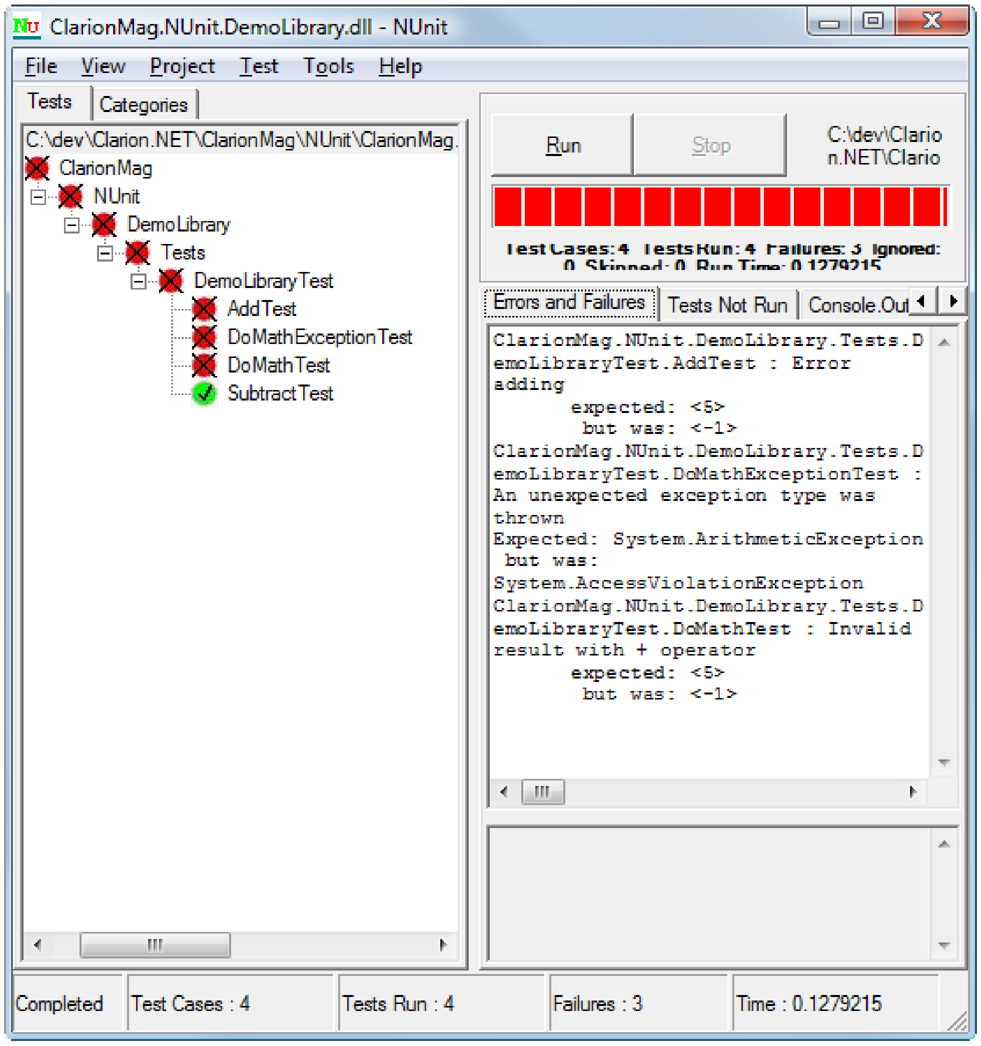


Figure 7. The test class loaded in NUnit.

Now click on the Run button. NUnit will run whichever of the test methods you've selected; click on DemoLibraryTest or any higher node to run all the tests. You'll see output similar to Figure 8.


ClarionMag.NUnit.DemoLibrary.dll - NUnit
File View Project Test Tools Help
Tests Categories
C:\dev\Clarion.NET\ClarionMag\NUnit\ClarionMag.
ClarionMag
NUnit
DemoLibrary
Tests
DemoLibraryTest
AddTest
DoMathExceptionTest
DoMathTest
SubtractTest
Run Stop
C:\dev\Clarion.NET\Clario
Test Cases: 4 Tests Run: 4 Failures: 3 Ignored: 0 Skipped: 0 Run Time: 0.1279215
Errors and Failures
Tests Not Run
Console.Out
Completed
Test Cases : 4
Tests Run : 4
Failures : 3
Time : 0.1279215

```
ClarionMag.NUnit.DemoLibrary.Tests.D
emoLibraryTest.AddTest : Error
adding
       expected: <5>
        but was: <-1>
ClarionMag.NUnit.DemoLibrary.Tests.D
emoLibraryTest.DoMathExceptionTest :
An unexpected exception type was
thrown
Expected: System.ArithmeticException
 but was:
System.AccessViolationException
ClarionMag.NUnit.DemoLibrary.Tests.D
emoLibraryTest.DoMathTest : Invalid
result with + operator
       expected: <5>
        but was: <-1>
```

Figure 8. NUnit test results

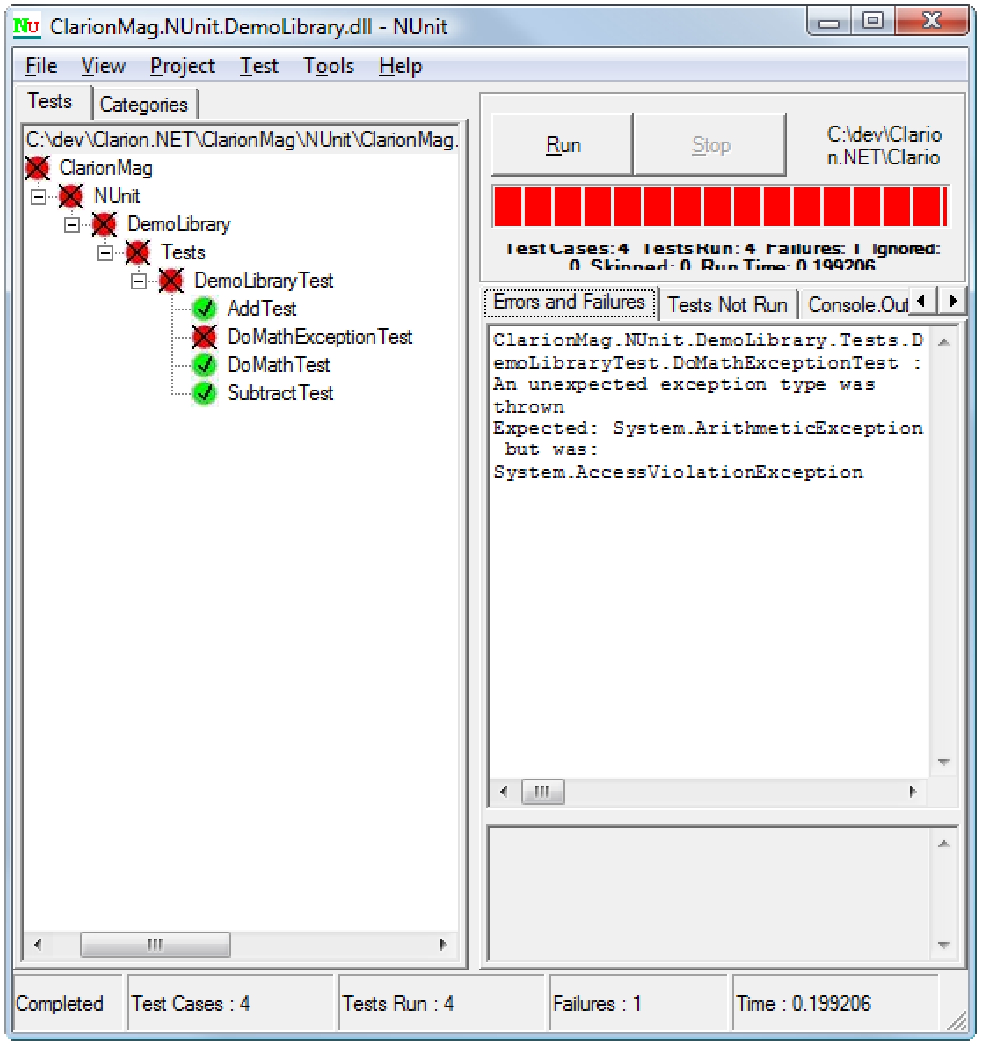
Obviously there are a couple of problems with my class. Three of the four methods have failed! Looking back through the source code I see that some lazy programmer has the add math all wrong. This code:

```
return(a-b)
```

should be:

```
return(a+b)
```

I make the changes and compile. Conveniently, NUnit notices that the DLL has changed and it reloads the file from disk. This time when I press Run I get a better result. Since the DoMath method called the Add method, I've fixed two errors at once (Figure 9).


ClarionMag.NUnit.DemoLibrary.dll - NUnit
File View Project Test Tools Help
Tests Categories
C:\dev\Clarion.NET\ClarionMag\NUnit\ClarionMag.
ClarionMag
NUnit
DemoLibrary
Tests
DemoLibraryTest
AddTest
DoMathExceptionTest
DoMathTest
SubtractTest
Run Stop
C:\dev\Clario n.NET\Clario
Test Cases: 4 Tests Run: 4 Failures: 1 Ignored: 0 Skipped: 0 Run Time: 0.199206
Errors and Failures Tests Not Run Console.Out
Completed Test Cases : 4 Tests Run : 4 Failures : 1 Time : 0.199206

```
ClarionMag.NUnit.DemoLibrary.Tests.D
emoLibraryTest.DoMathExceptionTest :
An unexpected exception type was
thrown
Expected: System.ArithmeticException
 but was:
System.AccessViolationException
```

Figure 9. After fixing the Add error

There's still one problem. NUnit is telling me that the DoMathExceptionTest is returning a different value than expected. Actually this is due to a bug in the attribute implementation in the first beta release of Clarion#. My DoMathException method is in fact throwing an ArthmeticException, but I'm not testing for the type of exception, I'm testing for the actual exception message. Testing for the type of an exception requires this kind of test attribute:

```
[ExpectedException( typeof(System.ArithmeticException),|
  'Expected System.ArithmeticException')]
```

The typeof syntax is fixed for the second Clarion.NET release, but even without that capability the test case is easy to fix: I change

```
throw new System.ArithmeticException
```

to

```
throw new System.ArithmeticException('System.ArithmeticException')
```

and run the test again. As Figure 10 shows, all tests now pass.

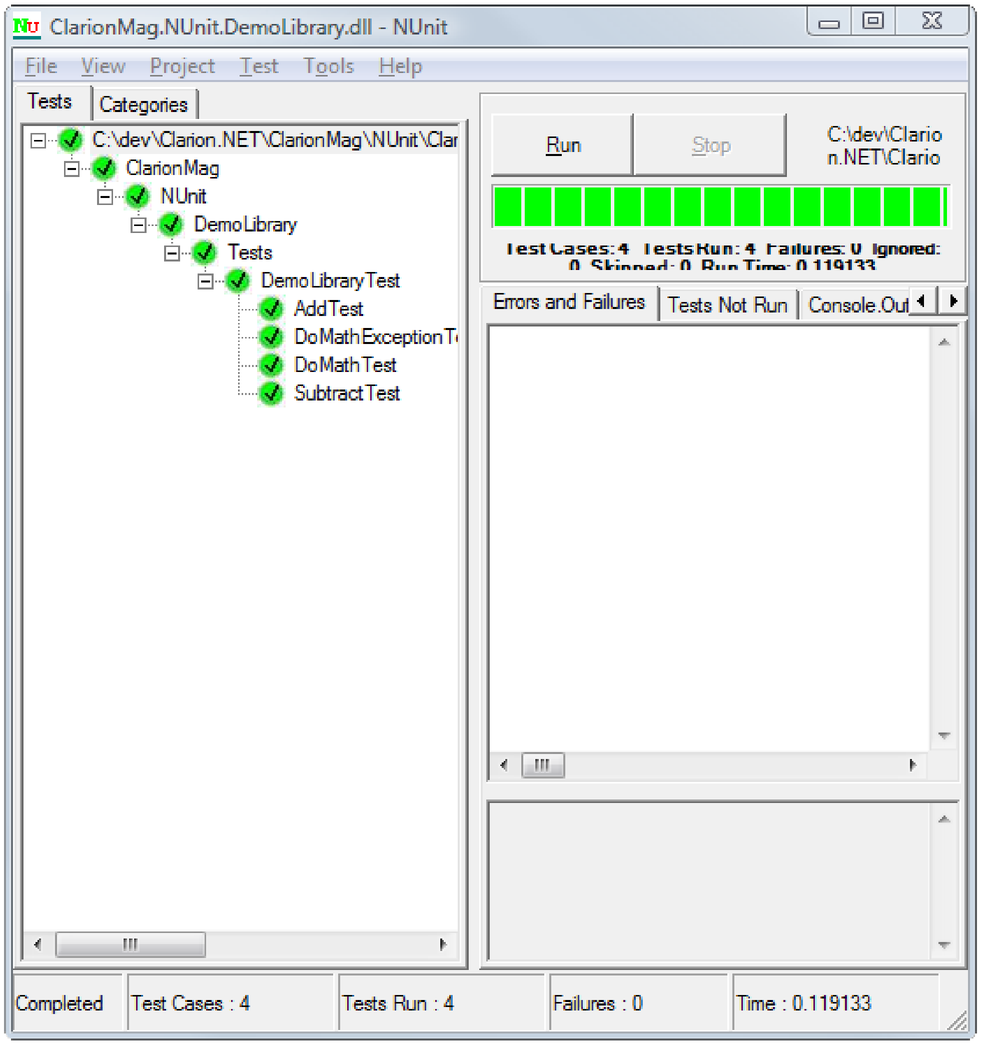

ClarionMag.NUnit.DemoLibrary.dll - NUnit
File View Project Test Tools Help
Tests Categories
C:\dev\Clarion.NET\ClarionMag\NUnit\Clar
ClarionMag
NUnit
DemoLibrary
Tests
DemoLibraryTest
AddTest
DoMathExceptionT
DoMathTest
SubtractTest
Run Stop
C:\dev\Clario n.NET\Clario
Test Cases: 4 Tests Run: 4 Failures: 0 Ignored: 0 Skipped: 0 Run Time: 0.119133
Errors and Failures Tests Not Run Console.Out
Completed Test Cases : 4 Tests Run : 4 Failures : 0 Time : 0.119133

Figure 10. Success!

Testing for a specific exception type is generally more useful than testing for an exception string, since there is no type checking on the exception string. If this were production code, I'd change it to typeof as soon as possible. But in any case, NUnit not only informed me of the error in my code, it alerted me to a problem with my test case declaration.

Summary

The process of creating class libraries has become simpler and more effective in Clarion#. You no longer need prototypes to use library code, and namespaces make it easier to create and maintain a useful class hierarchy.

But creating a class library is only half the battle; you also want it to be as bug-free as possible, particularly when you make changes to core classes or receive new code from other sources. Unit testing is a methodology for ensuring that the code you write not only is bug-free, but stays bug-free. Regular, comprehensive testing can pay big dividends in the long run.

Download the source

Download NUnit

David Harms is an independent software developer and the editor and publisher of Clarion Magazine. He is also co-author with Ross Santos of *Developing Clarion for Windows Applications*, published by SAMS (1995), and has written or co-written several Java books. David is a member of the American Society of Journalists and Authors (ASJA).

Reader Comments

Posted on Tuesday, January 01, 2008 by Mark Sarson

Great article Dave, and now Russ has something to get his teeth into :)

Happy New Year all Clarion Mag Readers.

Mark

Posted on Wednesday, January 02, 2008 by Dave Harms

Thanks Mark, and a Happy New Year to you too!

Dave

Add a comment

Clarion Magazine

A Simple Clarion# PDA Application

by Skip Williams

Published 2007-12-26

I have been intrigued with PDAs for quite a while. I bought a Palm many years ago and did a little programming for it using NSBasic, which worked pretty well. Five years ago, Microsoft was giving away Viewsonic V37 PDAs which had Compact Framework (CF) 1.1 built in to the ROM to entice developers into the .NET Compact Framework world. I was able to get one and tried writing a few programs for it using Visual Studio. My programs worked okay. Still my main interest was Clarion, and I never got into VS in a big way. I really wanted Clarion queues, ISAM files, and the many other benefits of the Clarion language over what the Microsoft .Net languages offered.

Then Soft Velocity introduced their pre-release version of Clarion.Net which supports Compact Framework 2.0, so I had to at least give it a try.

This article chronicles my build of a very simple example Clarion# program for a PDA using CF 2.0.

All the necessary prerequisites

As I mentioned, my old Viewsonic PDA had CF 1.1 installed and PPC 2002 as the operating system. Clarion# requires CF 2.0 which in turn requires the PPC 2003 operating system. Fortunately, a couple of years ago, I had purchased the PPC 2003 firmware for the Viewsonic on Ebay, but had never installed it. It was now time.

After the PDA firmware upgrade, the next step was to upgrade Activesync to version 3.7 and install CF 2.0 on the PDA. I already had .Net Framework 2.0 installed on my PC, but I also had to install the .Net Compact Framework 2.0 in order to do CF development. It also helps to have the.Net Framework 2.0 SDK installed on your PC.

Here's a quick recap of what you need:

On the PDA:

- At least PPC 2003
- Compact Framework 2.0 redistributable (I used CF2.0 sp2)

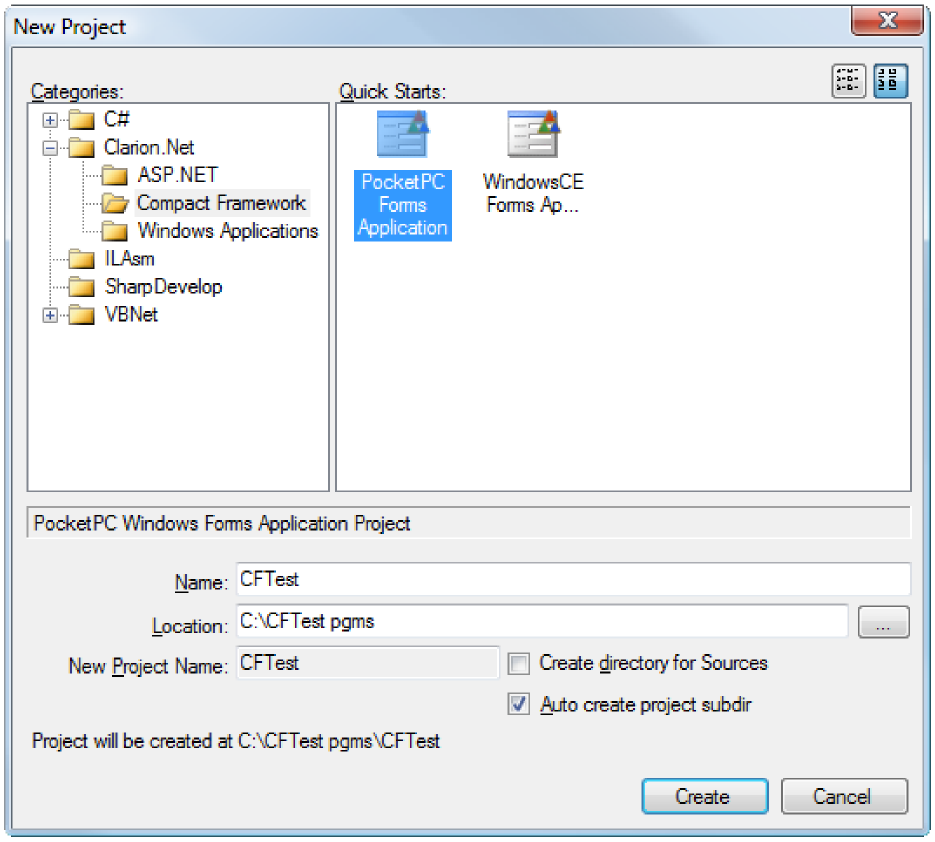
On the PC:

- .Net Framework 2.0 redistributable
- ,Net Framework 2.0 SDK
- Activesync 3.7 minimum
- And of course, Clarion.Net

I found that all of the prerequisites were the most time consuming part of the whole process. I ended up having to completely uninstall Compact Framework on my PC before I could install the updated SP2 version. Building the test app was pretty easy.

Now to build the application

Launch Clarion.Net and select New/Solution from the menu. Select Compact Framework from the list and highlight PocketPC Forms Application. Give the application a name, CFTEST, and a location, C:\CFTest pgms (see Figure 1).



PocketPC Windows Forms Application Project

Name: CFTest

Location: C:\CFTest pgms

New Project Name: CFTest

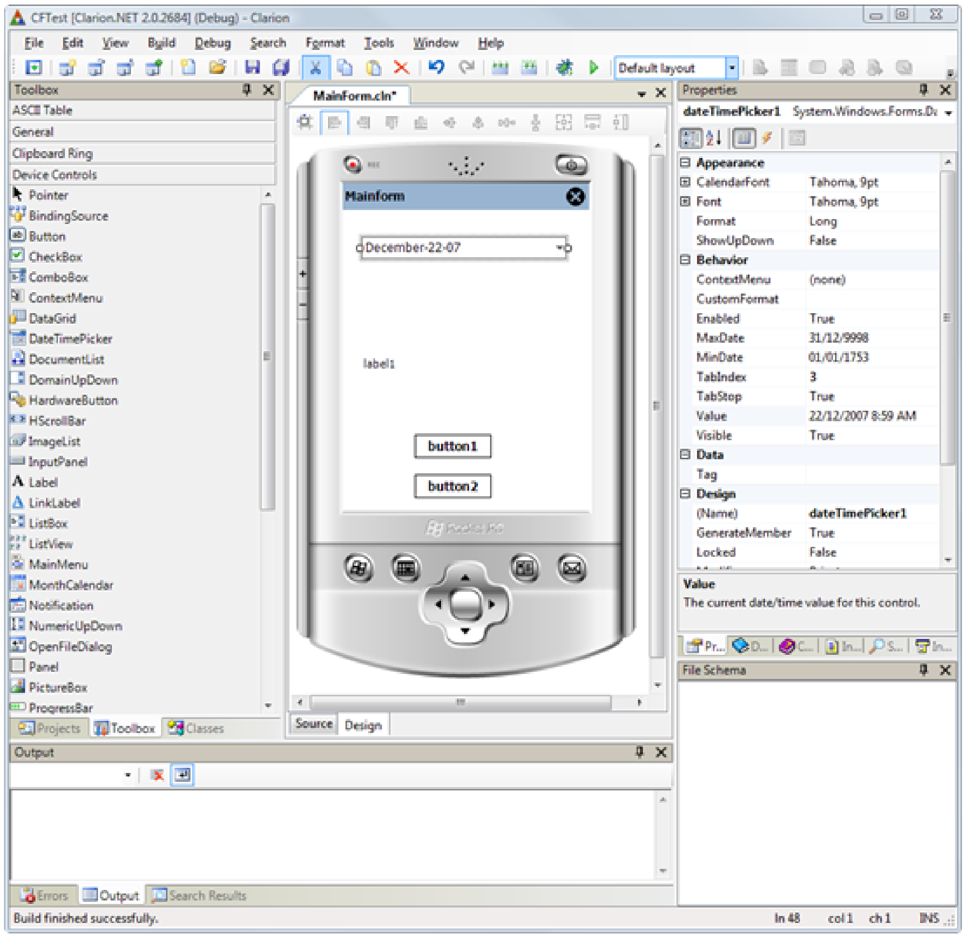
Create directory for Sources

[✓ Auto create project subdir]

Project will be created at C:\CFTest pgms\CFTest

Figure 1. Creating a new project

Press the Create button for the magic to happen. A blank PDA screen appears along with a toolbox and a property box (Figure 2).



dateTimePicker1 System.Windows.Forms.D

```
Appearance
  CalendarFont     Tahoma, 9pt
  Font             Tahoma, 9pt
  Format           Long
  ShowUpDown       False
Behavior
  ContextMenu      (none)
  CustomFormat
  Enabled          True
  MaxDate          31/12/9998
  MinDate          01/01/1753
  TabIndex         3
  TabStop          True
  Value            22/12/2007 8:59 AM
  Visible          True
Data
  Tag
Design
  (Name)           dateTimePicker1
  GenerateMember   True
  Locked           False
```

Value
The current date/time value for this control.

Build finished successfully.   ln 48   col 1   ch 1   INS

Figure 2. The new application in the IDE (view full size image)

While you weren't looking, I added a DateTime Picker control, a label, and two buttons by simply dragging and dropping the controls on the screen (Figure 3). Using the mouse, the controls can be resized and moved around quite easily, just like the familiar Clarion windows formatter.

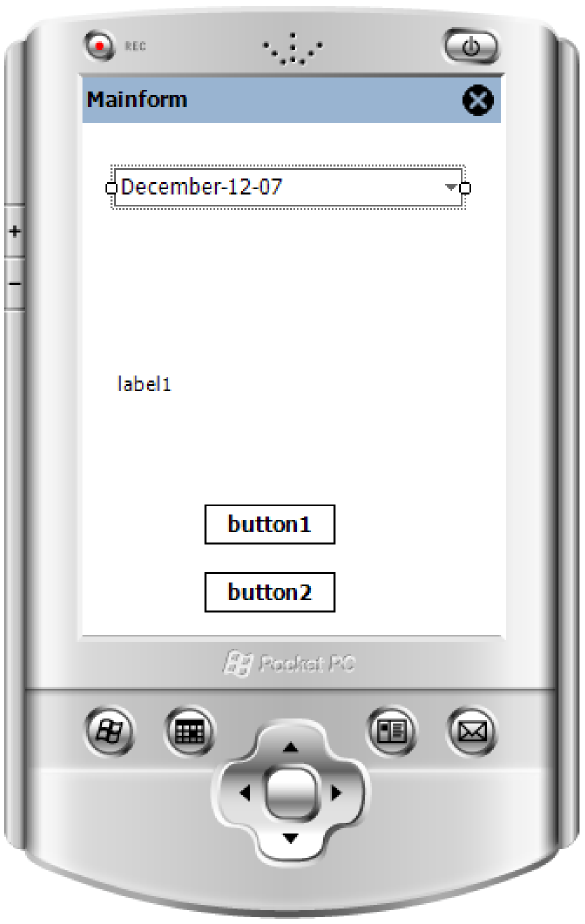


Figure 3. Adding the controls

If you have Clarion 7, or have used Visual Studio or #Develop, you will be familiar with the properties window on the right side of the Clarion.Net IDE. This is where the control's properties are modified.

Using the properties window, I quickly changed the screen name, the text in the label field and both buttons (Figure 4).

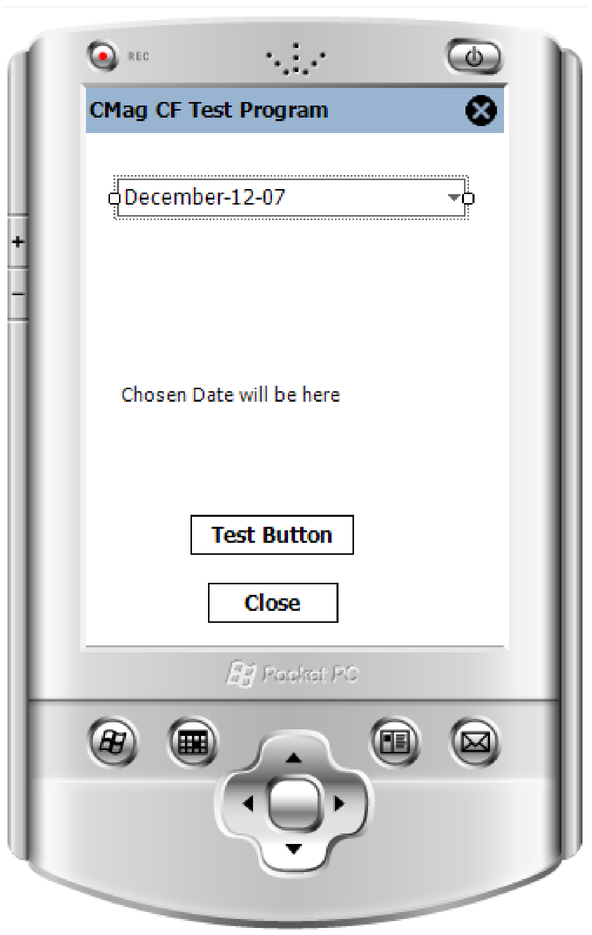


Figure 4. The updated controls

Notice the two tabs below the PDA screen on Figure 2: Source and Design. Selecting the Source tab allows you to see the underlying code for the window, which, at this point isn't very exciting.

```
!~
~ Created by Clarion.
~ User: Skip Williams
~ Date: 12/11/2007
~ Time: 8:12 AM
~!

                    MEMBER('')
                    NAMESPACE('CFTest')

USING('System')
USING('System.Data')
USING('System.Drawing')
USING('System.Text')
USING('System.Windows.Forms')
```

```
        !!!
        !!! Description of ${ClassName}.
        !!!
    MainForm           CLASS(Form), TYPE, PUBLIC, PARTIAL
    CONSTRUCT              PROCEDURE(), PUBLIC
                            END
```

Now it's time to make the controls actually do something useful. I will use the Date Picker to choose a date and place it in the label field. The Test Button will display a message that the button has been pressed, and the Close button will do what all good close buttons do: close the application.

To add underlying code to each control, return to the designer mode and simply double click on each control. Clarion. Net inserts the following code for each control.

```
    MainForm.DateTimePicker1_ValueChanged PROCEDURE(System.Object sender,|
                                System.EventArgs e)
      CODE



    MainForm.Button1_Click     PROCEDURE(System.Object sender,|
                            System.EventArgs e)
      CODE



    MainForm.Button2_Click     PROCEDURE(System.Object sender,|
                            System.EventArgs e)
      CODE
```

Now all I have to do is add code to make the controls work.

```
    MainForm.DateTimePicker1_ValueChanged  PROCEDURE(System.Object sender, |
                                 System.EventArgs e)
      CODE
      str" = SELF.dateTimePicker1.text
      MessageBox.Show('The new date is ' & str")
      Self.label1.Text = str"



    MainForm.Button1_Click     PROCEDURE(System.Object sender,|
                            System.EventArgs e)
      CODE
      Message('You pressed the button','Hello')

    MainForm.Button2_Click     PROCEDURE(System.Object sender,|
      CODE
```

```
        self.Close()
```

When a date is chosen, I display a message box displaying the date and move it to the Label1 field. I know, I know, implicit variables are bad. I only used the str" implicit variable to see if it still worked in Clarion#.

For button1 I used the Clarion Message statement, both to see if it still worked and to see the difference between the two ways of presenting a message. Button2 closes the application.

Now I was ready to do a "compile and go" for this new CF application. Pressing the little green arrow on the toolbar causes the application to be built and executed. The result is shown in Figure x:

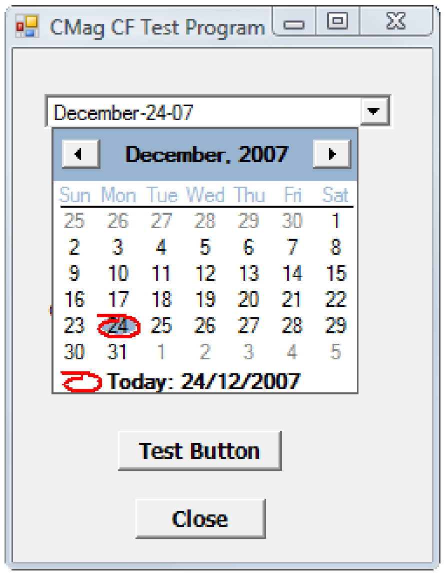
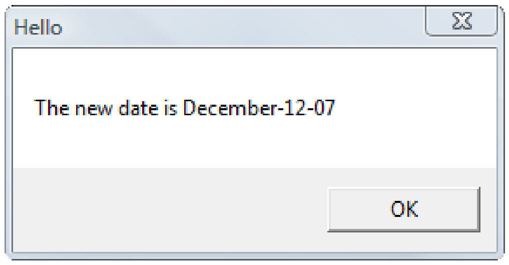


Figure 5.

Before this screen shot, I selected the DatePicker control that caused the calendar to open. When I pick a date, a .Net message box is opened, indicating the date that I picked.



The new date is December-12-07

Figure 6.

Press OK to close the message box, and then press Test Button; the following Clarion message box opens. There is a slight difference between the two.

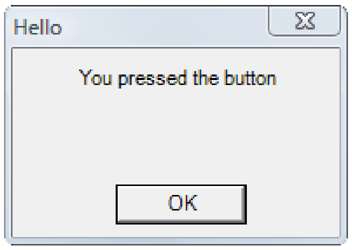
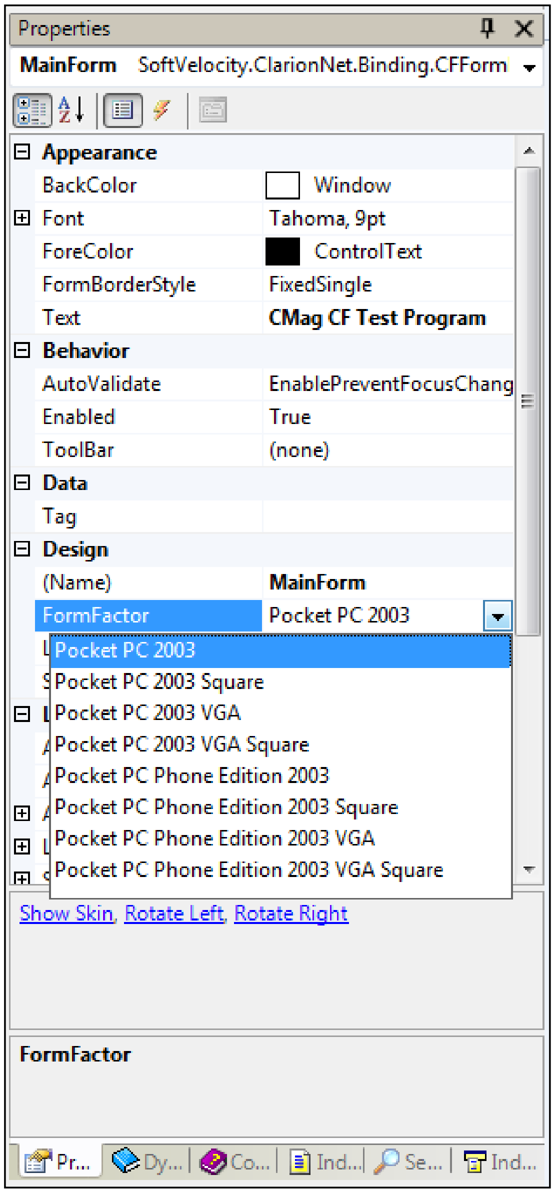


Figure 7

Pressing the Close button ends the application.

So far I have been running the application under the IDE. I understand that there are emulators available that can be installed to enhance the testing experience. You also have several options in the IDE to match your emulator screen more closely to your device. These options are available for your selection from the properties window, under Design | Form Factor.


Properties

MainForm SoftVelocity.ClarionNet.Binding.CFForm

Appearance	
BackColor	⬜ Window
Font	Tahoma, 9pt
ForeColor	⬛ ControlText
FormBorderStyle	FixedSingle
Text	**CMag CF Test Program**
Behavior	
AutoValidate	EnablePreventFocusChang
Enabled	True
ToolBar	(none)
Data	
Tag	
Design	
(Name)	**MainForm**
FormFactor	Pocket PC 2003

Pocket PC 2003
Pocket PC 2003 Square
Pocket PC 2003 VGA
Pocket PC 2003 VGA Square
Pocket PC Phone Edition 2003
Pocket PC Phone Edition 2003 Square
Pocket PC Phone Edition 2003 VGA
Pocket PC Phone Edition 2003 VGA Square

Show Skin, Rotate Left, Rotate Right

FormFactor

Figure 8. Changing the form factor

You can also enable/disable the display of the skin, and set the width and height of the screen to specific values.

Now comes the real test.

Deployment to the PDA!

If the application is working, deployment should be pretty simple, right? Actually, in my case it was, but there are instances of others having problems assembling all of the necessary components to make it happen. I understand that Soft Velocity is in the process of writing a white paper on the process, and that SetupBuilder version 7 from Lindersoft will support deployment to the Compact Framework devices.

I simply wanted to copy the executable to my PDA and see it work.

As part of the build process, Clarion.Net creates a \bin directory under your project. Here you'll find the executable, along with three runtime DLLs that need to be copied to the Activesync directory as well.

```
24/12/2007  11:42 AM           10,752 CFTest.exe
24/12/2007  11:42 AM           28,160 CFTest.pdb
14/11/2007  07:00 PM           40,448 SoftVelocity.Clarion.FileIOCF.dll
14/11/2007  07:00 PM          152,576 SoftVelocity.Clarion.Runtime.ClassesCF.dll
14/11/2007  07:00 PM          176,128 SoftVelocity.Clarion.Runtime.ProceduresCF.dll
```

PPC2003 has a directory called Programs which seemed like a good location. In the directory that Activesync uses to send stuff to the PDA I created a subdirectory of Programs called CFTest, and copied my executable there.

I ran Activesync, which created the /PCTest subdirectory on the device and copied my program and DLLS. I didn't create a program icon or shortcut on the device, so it was necessary to use the PDA's File Explorer to navigate to the \Programs\CFTest directory and select CFTest.exe with the stylus. I then selected the DatePicker control; you can see the result in Figure 9.

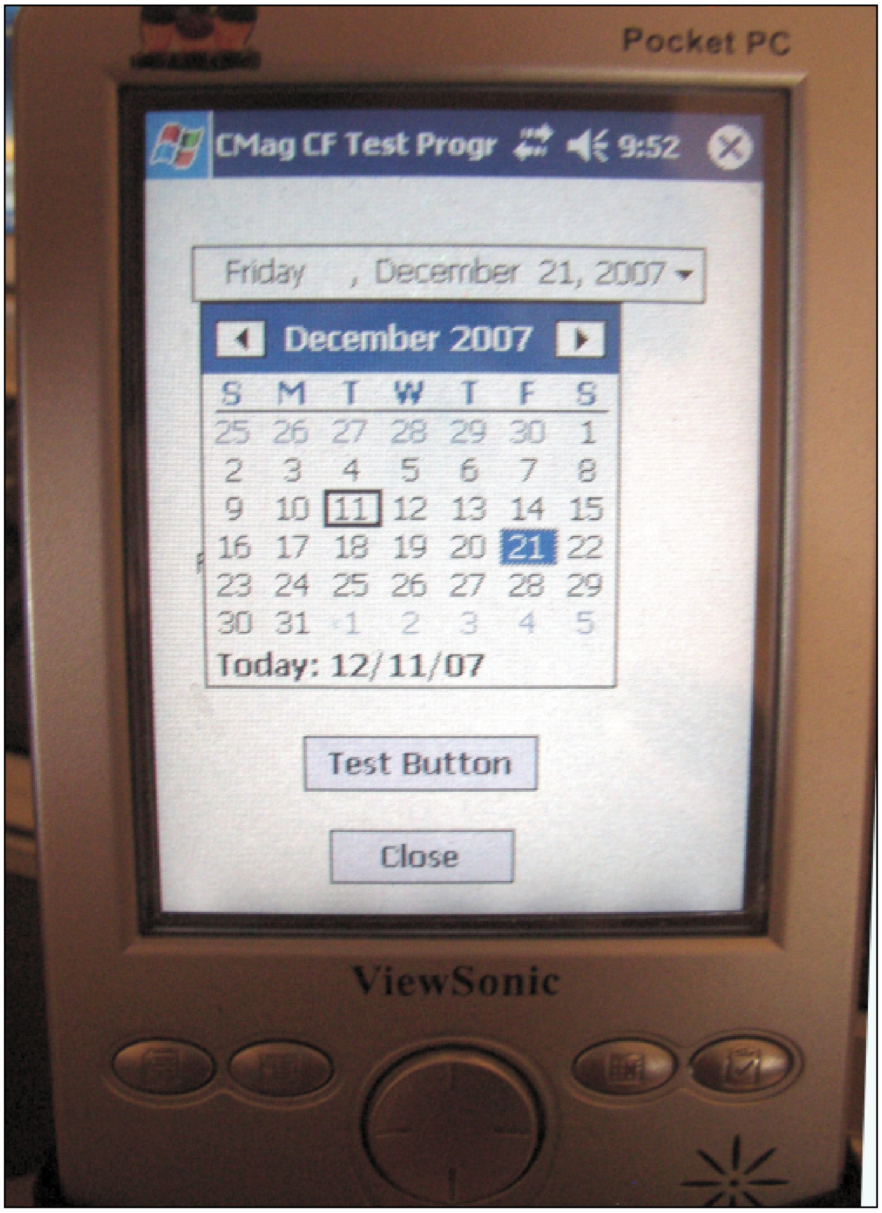


Figure 9. Running on a PDA

Now, for several final observations.

Remember that in the code I used two different message statements, the Clarion one and the .Net one? Both worked as expected in the emulator on the desktop. On the PDA, however, the Clarion message statement opened a full window, covering the main window completely. The .Net message statement opened a normal message box, but, because the calendar was still dropped down, the message box opened behind it. Remember that an emulator is no substitute for testing on the real device.

Also, please note that this application will not work with Windows Mobile devices such as SmartPhones. This example and the example program provided by Soft Velocity both throw a System.NotSupportedException. The problem is that Windows Mobile (at least as of version 5) does not support button controls; you are supposed to use the device's soft keys instead. You can create Clarion# applications that run on SmartPhones, you just have to keep to the allowed set of controls.

Summary

I've demonstrated a very simple Clarion# application for .Net Compact Framework. It only has a few controls and doesn't do much, but it demonstrates how easy it can be to create Compact Framework applications in hand code; SoftVelocity's upcoming templates will make mobile development that much simpler.

Download the source

Skip Williams' first job was programming mainframes for a bank. He did that for 10+ years then became a VP of Technical Services and moved into a management job. After several bank mergers, Skip became a programmer again and helped start Global Trade Information Services in Columbia, SC, using Clarion programming tools. Skip has been using Clarion since 1986, when he was persuaded by Bruce Barrington, at Comdex, to try this new language Bruce had developed.

Reader Comments

Posted on Monday, December 31, 2007 by Devan Sabaratnam

Great article Skip! Keep them coming. This is the sort of thing that motivates me to forge ahead and experiment with Clarion7/Clarion.Net myself. Cheers, Devan.

Add a comment

Clarion Magazine

Designing Clarion# Libraries

by Dave Harms

Published 2007-12-20

Code re-use is essential to maintaining high productivity, as almost all applications use a lot of the same code (or the same kind of code) over and over. Clarion programmers accomplish code re-use two ways: with code libraries, and with templates. As SV hasn't yet shipped any Clarion# templates, it's not possible to say with certainty how much the template approach to code re-use will change. But .NET class libraries *are* a known quantity, and in this article I'll look at how to structure your own Clarion# class libraries (I'll save the specifics of creating class libraries for another time).

But why talk about libraries? Wouldn't it make more sense to talk about full-blown Clarion# applications first?

It's certainly possible to start building Clarion# applications, but because the AppGen isn't done yet, and there are no shipping Clarion# templates, all Clarion# application development at present must be done by hand. Okay, there's one exception: Clarion.NET ships with a C6 wizard that will generate a rudimentary Clarion# app from your dictionary. But that code really isn't (by SoftVelocity's own admission) a good example of how to write .NET applications; it's more of a convenience to get developers used to .NET concepts.

Writing Clarion# applications by hand isn't a bad thing, and it's certainly a topic that will get coverage in Clarion Magazine in the future. But most Clarion developers will use the AppGen for the bones of their applications.

So what can you do while you're waiting for AppGen? Start writing and testing libraries, of course.

Now, that may seem like an odd statement. After all, most of us who write code libraries test them in context; we plug the code into our applications and we see if it works. But in .NET you don't always have to build your application to test the library code; you can use a process called *unit testing* to ensure that your code works properly even before you plug it into your application.

In this article I'll cover some of the concepts you need to have in hand before you start building your .NET libraries, and next time I'll show how to use the freely available NUnit utility to test those libraries.

But first a bit of history....

The Clarion approach

In Clarion (not Clarion#) you have two kinds of code libraries, procedural and object-oriented.

Procedural libraries are simply collections of PROCEDUREs; typically you'll compile these into LIBs or DLLs. Clarion Magazine has a number of articles on creating libraries; two you might want to check out are Alan Telford's My First Function Library and Jeff Slarve's DLLs and Reusable Code: Divide and Simplify. You don't necessarily have to compile procedural library source ahead of time - you can also keep it as source code and just include it as necessary in your application, but that adds to compile time. In most cases using a LIB or DLL is preferable.

Object-oriented libraries are much like procedural libraries, except that all the code is contained inside classes. If you're new to object-oriented programming, I suggest you take a moment to read The ABCs of OOP. Although it's possible to create procedural code in Clarion#, for the most part you'll want to be conversant with object-oriented programming.

While procedural libraries are often precompiled, in Clarion (not Clarion#) it's quite common to see object-oriented code that's compiled each time along with the application source. The ABC class library, for instance, is all source code; Clarion compiles in the classes referenced by your application's source code.

In Clarion, the norm is to use class libraries as source; in .NET, however, the norm is to use compiled class libraries.

The need for prototypes and declarations

Whether you're talking procedural or OOP Clarion code, any time you use a compiled library you need some sort of source code that describes what's in the library. In the case of procedural code, you need a *prototype* of the method. In particular, Clarion developers run into this when using the Windows API. Here's an example of a prototype for a WinAPI function contained in Kernel32.DLL:

```
CloseHandle(HANDLE hObject),BOOL,PASCAL,PROC
```

In this example, the prototype I use for CloseHandle has to be compatible with the code that the author of the CloseHandle function used.

A similar situation exists for compiled libraries. Clarion programmers don't encounter this very often, since Clarion classes tend to be supplied as source code. For instance, I have a utility class called CCIAddressClass, and whenever I want to use it in one of my apps I need to add this line of code at an appropriate data embed point:

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

In this example CCIAddrINC contains a CLASS definition and a QUEUE declaration, while the CLASS's source code is in CCIAddr.clw. Here's the source for CCIAddr.INC:

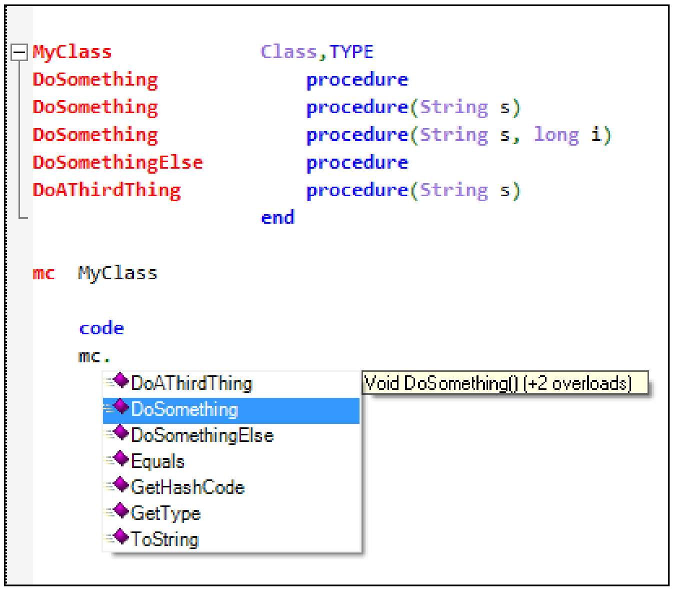
```
!ABCIncludeFile

OMIT('_EndOfInclude_',_cciAddressPresent_)
_cciAddressPresent_ EQUATE(1)

AddressQueue      queue,type
Line              cstring(100)
                  end


cciAddressClass CLASS,TYPE,MODULE('cciaddr.clw'),|
                  LINK('cciaddr.clw',_ABCLinkMode_),DLL(_ABCDllMode_)
AddressQ          &AddressQueue
AddLine           procedure(string Part1,<string Part2>,|
                   <string Part3>,<string Part4>,<string Part5>)
AddOneLine        procedure(string Line)
AppendToLine      procedure(*cstring cs,string s),private
Construct         procedure
Destruct          procedure
GetAddress        procedure,string
Reset             procedure

                end

_EndOfInclude_
```

If I were to compile this class into a library, and add that library to my application's project data, that wouldn't be enough for me to use the class in my application. I would still need some sort of class declaration. There would be some changes: the CLASS would need to be declared as EXTERNAL,DLL. And I wouldn't have to include everything in the CLASS declaration, just the methods and properties I needed.

The Clarion# approach

Code libraries change radically in Clarion#. Although the INCLUDE mechanism is still there (as is the ability to write procedural code), most code reuse is accomplished using standard .NET mechanisms. And specifically when it comes to using already-compiled code, the key is something called *reflection*.

Figure shows an example of reflection in action. I've created a class called MyClass, and an instance called mc. In the code section I've just typed mc. and the editor's Intellisense capability is showing a list of available methods.



```
MyClass             Class,TYPE
DoSomething           procedure
DoSomething           procedure(String s)
DoSomething           procedure(String s, long i)
DoSomethingElse       procedure
DoAThirdThing         procedure(String s)
                    end

mc  MyClass

    code
    mc.
```

Figure 1. The IDE showing available methods

Okay, what's the big deal? That's just Intellisense, the IDE helping you figure out what your options are, right? Right, except that it's also an example of reflection at work. The editor is examining MyClass's structure using reflection, and presenting Intellisense options based on that information.

> **Aside:** Okay, I can't absolutely guarantee that the IDE is using reflection in this exact case because I haven't asked SV, and I haven't seen the underlying source code. I suppose it's possible the editor is parsing my class's source. But Intellisense also works for classes where the source isn't present, and it makes more sense to only have one mechanism for any given feature.

Reflection is possible for two reasons. First, .NET classes contain not just code and data, but metadata, which is data about the class itself. Second, the .NET framework includes a number of classes to help in the examination of that metadata.

No more library prototypes or declarations

Because .NET classes contain this readily-accessible metadata, you no longer need to create declarations of compiled classes or other structures before you use them. Let me repeat that. *You no longer need to create declarations of compiled classes or other structures before you use them.* That's a huge change, and a great advantage. If you don't need to create declarations, you don't have to worry about making mistakes on those declarations, and you also don't have to worry about keeping your declarations in sync if you end up changing the library.

> **NOTE:** You can still use source code libraries in Clarion#; you simply include the source file in the project. If the library classes are in the same namespace as the code you're writing, you don't have to use the fully qualified class names. If the library classes are in a different namespace, you must either use the fully qualified name or add a USING directive specifying the namespace. You will also need to use the fully qualified name if you have same-named classes in two namespaces, since the compiler won't otherwise know which class you mean. In general, however, it's better to reuse compiled source; you shouldn't have to keep compiling the same library code over and over.

Assemblies and visibility

When you create a Clarion# library, your project is compiled to a DLL, and in .NET a DLL, like an EXE, has a fancy new name: it's called an *assembly*. Assemblies are a little more sophisticated than EXEs or DLLs, but for purposes of this discussion you can think of an assembly the same way you think of a Clarion EXE or DLL.

In .NET, class methods visibility attributes are similar to those we're used to in Clarion. These include PUBLIC, PRIVATE, and PROTECTED, as well as a new attribute, INTERNAL, which means the method is visible anywhere within its own assembly. CLASSes are either INTERNAL (the default) or PUBLIC.

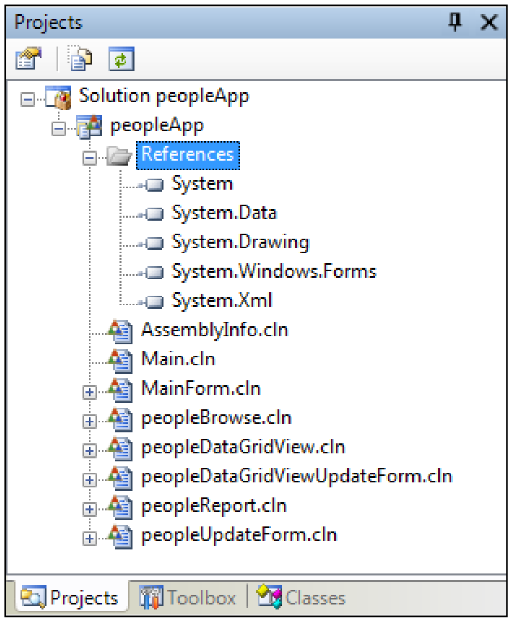


Figure 2. The project pane

Scope in .NET is also determined by *namespaces*. I've gone into this topic in another article, so I won't cover it exhaustively here. Essentially, namespaces are a way of grouping similar code, which can be spread across many source files.

When you create a new Clarion# application, the IDE creates a default namespace based on the project name. Each source file you create via the IDE will have that namespace added. Let's say you create a new project called TestNamespaces. The IDE will add this directive to each source file created (I'm using the forthcoming syntax *without* the quotes):

```
NAMESPACE(TestNamespaces)
```

You can create multi-level namespaces like this:

```
NAMESPACE(TestNamespaces.ASubNamespace)
```

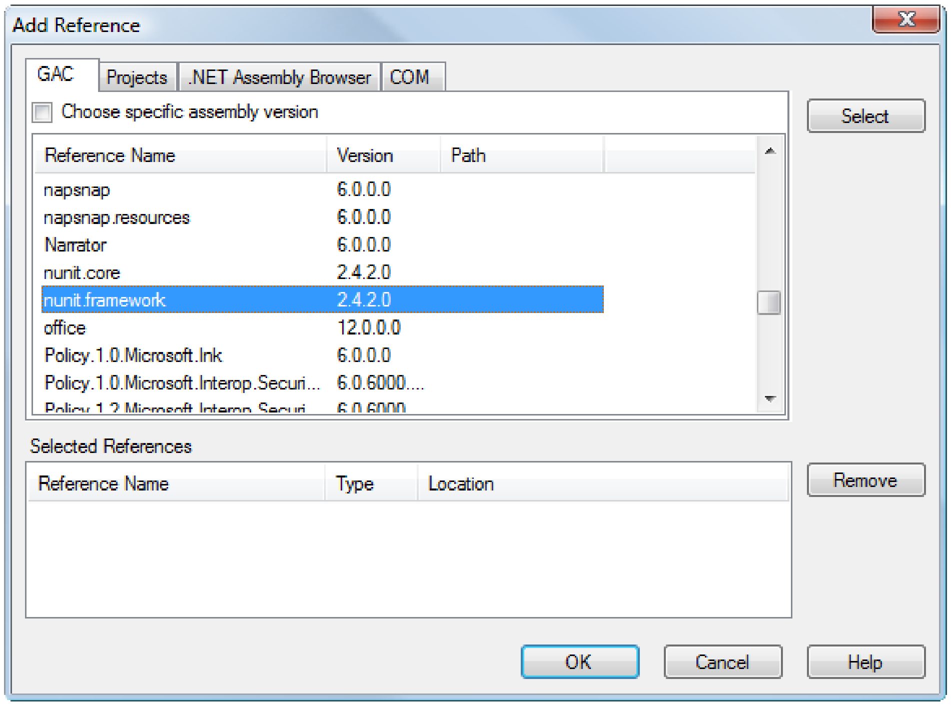
All classes you create using the first example will effectively have TestNamespaces. prepended. If your class is called MyClass, its full name will be TestNamespaces.MyClass. If you use the second example, the full name will be TestNamespaces.ASubNamespace.MyClass

Under the covers, all classes have these fully qualified names - that's how the compiler sees them. But for you and I, who may not like to type that much, there are a number of shortcuts. When you're working on a class in its own source file, you don't need the prefix, because the NAMESPACE is implicit. If someone else is using your class and doesn't want to type TestNamespaces. all the time, they can insert a USING(TestNamespaces) directive and the compiler will try out that namespace along with the current namespace as it attempts to resolve the labels you've typed into the actual fully qualified values.

Adding references

To use a compiled class library in your application you creating a *reference* to the library's assembly (typically a DLL, but it could also be an EXE).

Have a look again at the project pane in Figure 2. To add a new reference you right-click on References and choose Add Reference. You'll see the window in Figure 3.



Reference Name	Version	Path
napsnap	6.0.0.0	
napsnap.resources	6.0.0.0	
Narrator	6.0.0.0	
nunit.core	2.4.2.0	
nunit.framework	2.4.2.0	
office	12.0.0.0	
Policy.1.0.Microsoft.Ink	6.0.0.0	
Policy.1.0.Microsoft.Interop.Securi...	6.0.6000....	

Reference Name	Type	Location

Figure 3. Adding a reference, GAC tab

GAC stands for the Global Assembly Cache, which is a central location for commonly-used, version-managed .NET DLLs. In general you don't want to go adding your own DLLs to the GAC, but you may find that various third party add-ons install their DLLs here. You can manage the GAC with the utility program gacutil, which you can find in your . NET framework bin directory.

If the DLL you want to use isn't in the GAC, you can locate it manually. Click on the .NET Assembly Browser, and then on the Browse... button shown in Figure 4. Navigate to the DLL or EXE you want to use and select it. Click OK to add the reference.

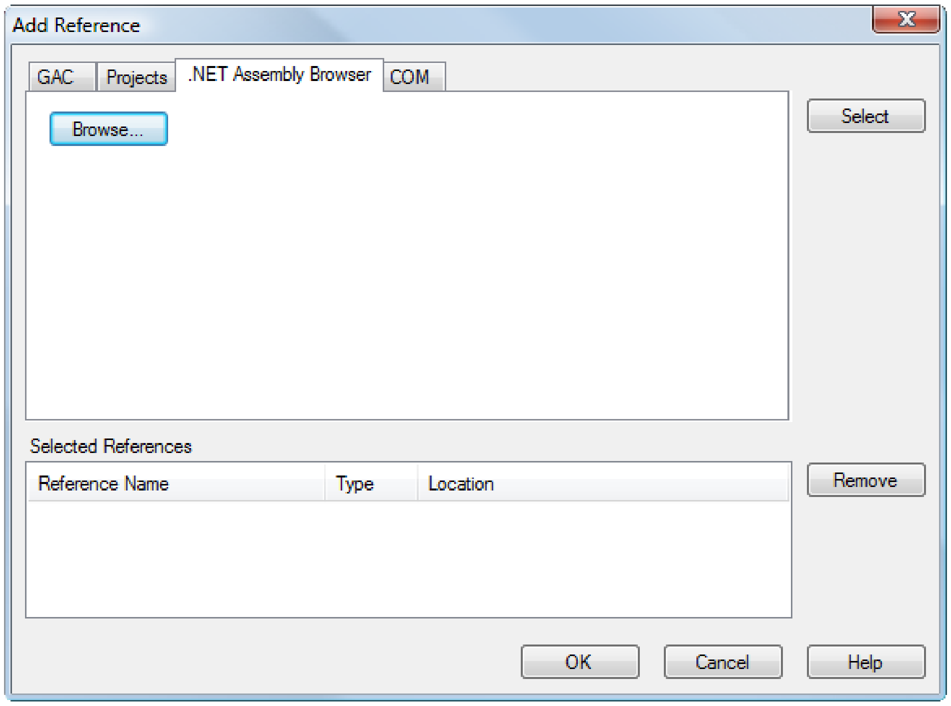


Figure 4. Adding a DLL or EXE reference

Note that after you add the reference, it isn't the DLL's or EXE's name that appears in the project list, but the namespace (s) contained in that DLL or EXE.

The same rules regarding fully qualified names and USING apply to references as to classes included as source code.

Choosing namespaces

Namespaces are clearly an important part of Clarion# and .NET, and a very useful way to organize your own code. But how do you go about choosing your namespaces?

There are two goals you should have for your namespaces: they should be descriptive, and they should be unique. I've always liked the Java approach to uniquely naming namespaces (which Java calls *packages*); at the top level you use your domain name, reversed. Namespaces for ClarionMag.com classes would begin with com.clarionmag, thereby absolutely guaranteeing uniqueness.

The .NET namespace conventions are less restrictive and potentially more readable, but also less certain to result in unique names. Here's what Microsoft has to say on the subject:

From MSDN:

> .NET Framework types use a dot syntax naming scheme that connotes a hierarchy. This technique groups related types into namespaces so they can be searched and referenced more easily. The first part of the full name - up to the rightmost dot - is the namespace name. The last part of the name is the type name. For example, System.Collections.ArrayList represents the ArrayList type, which belongs to the System.Collections namespace. The types in System.Collections can be used to manipulate collections of objects.
>
> This naming scheme makes it easy for library developers extending the .NET Framework to create hierarchical groups of types and name them in a consistent, informative manner. It is expected that library developers will use the following guideline when creating names for their namespaces:
>
> CompanyName.TechnologyName
>
> For example, the namespace Microsoft.Word conforms to this guideline.

> The use of naming patterns to group related types into namespaces is a very useful way to build and document class libraries. However, this naming scheme has no effect on visibility, member access, inheritance, security, or binding. A namespace can be partitioned across multiple assemblies and a single assembly can contain types from multiple namespaces. The assembly provides the formal structure for versioning, deployment, security, loading, and visibility in the common language runtime.

Starting with the company name or domain name is a good idea. After that, think about grouping classes by functionality. Here are a few suggestions for code related to the basic Clarion procedure types:

- *CompanyName*.Windows.Forms
- *CompanyName*.Windows.Forms.Menus
- *CompanyName*.Windows.Forms.Browses
- *CompanyName*.Windows.Forms.Forms

You might want to create a similar hierarchy for web applications (*CompanyName*.Web) or compact framework apps (*CompanyName*.Compact). And how about business objects? Perhaps you'd want to group these by business type: *CompanyName*.Business.Shipping, or just *CompanyName*.Shipping.

It's important to note that you don't have to have all your classes for one namespace in a single DLL. As noted in the MS quote, you can spread a namespace across multiple DLLs, and one assembly (typically a DLL) can contain multiple namespaces.

Directory structure

You'll also have to decide how you want to structure your project directories. You probably want your directory structure to mimic your namespaces, but do you want to do it by folders or by dot notation, or by some combination? The Java namespace specification requires each level of the hierarchy to have its own directory, but the .NET specification isn't as restrictive.

In .NET you are certainly free to specify your directories in Java style:

```
CompanyName
  Windows
    Forms
  Windows
    Forms
      Menus
```

Or you could do it like this:

```
CompanyName.Windows.Forms
CompanyName.Windows.Forms.Menus
```

Or you can use a hybrid, where you only go down a few levels in the directory structure, even if the namespace structure goes deeper:

```
CompanyName
  Windows.Forms
  Windows.Forms.Menus
```

Summary

Thanks to reflection, class declarations are no longer required when using compiled .NET libraries. Instead, the compiler will search the referenced assemblies (DLLS and/or EXEs) looking for matching classes and other structures.

The new NAMESPACE and USING directives make it easy to group code by functionality; a little thought now will go a long way to organizing your code into a usable and maintainable namespace hierarchy.

Next time I'll show how to create a small library, and do unit testing with NUnit.

David Harms is an independent software developer and the editor and publisher of Clarion Magazine. He is also co-author with Ross Santos of *Developing Clarion for Windows Applications*, published by SAMS (1995), and has written or co-written several Java books. David is a member of the American Society of Journalists and Authors (ASJA).

Reader Comments

Add a comment

Clarion Magazine

Clarion# And The Google Calendar API

by Randy Rogers

Published 2007-12-17

Over the past couple of months I have been exploring various way to allow my municipal customers to make their Facilities Scheduling calendars available to their rate payers via the Internet. In my research I came across the Google Calendar API which allows programmatic access to Google's on-line calendaring application. You can have both private and public calendars, and the best part is that you have access to the calendar anywhere you have an Internet connection.

I thought this might be a good solution for my customers, and I began trying to implement access to the Google Calendar API from Clarion 6. I've written a lot of these types of interfaces, but eventually decided that creating a NetTalk webserver-based calendar application would be much easier than the work I would need to do to interface with the Google Calendar API.

Then SoftVelocity released the first public beta of Clarion#. I held out for a week after the announcement from Softvelocity, but curiosity finally got the better of me and I bought into the beta program. I spent a bit of time familiarizing myself with the new IDE and working through the examples before embarking on my first hand-coded Clarion# application. Since I had recently been looking at interfacing to the Google Calendar API, I thought that might make a good first project. I set out to convert some sample C# code to Clarion#.

The Google APIs

The Google Data API libraries, documentation, and sample code can be found at http://code.google.com/apis/calendar/
I downloaded and installed the Google tools; to make sure the supplied code was good I used Visual Studio to compile and run an example C# application. Then I set out to create my Clarion# application.

One thing I discovered later in this process was that all of the Google API libraries were not registered in the Global Assembly Cache (GAC), and Clarion# (at least in the first beta) seems to have trouble using a mixture of global and local assemblies from the same namespace. If you find this happens to you, you'll need to use gacutil to add the missing DLLS to the cache. Here's the syntax:

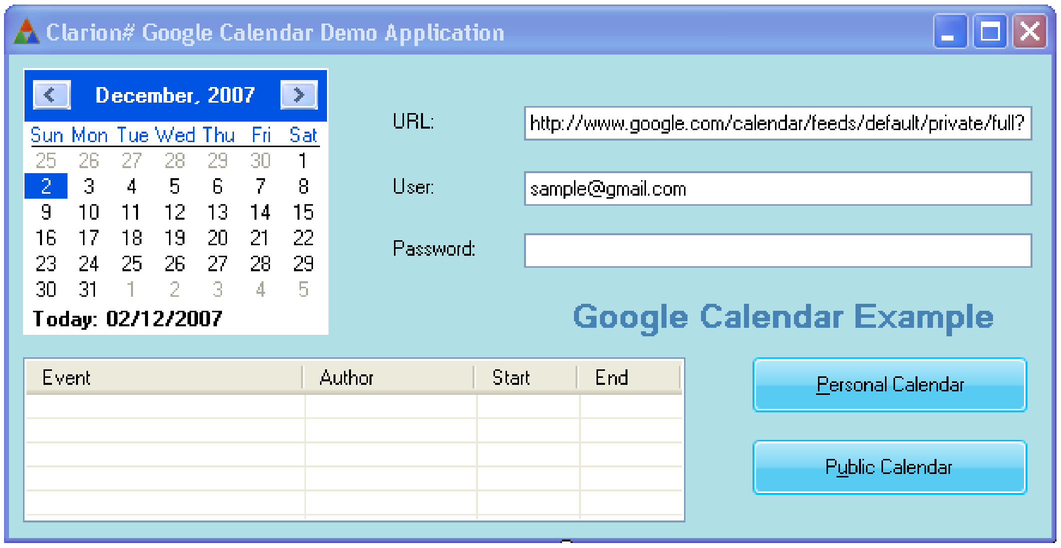
```
gacutil /i dllname
```

You can find gacutil in the .NET SDK's bin directory (e.g. C:\Program Files\Microsoft.NET\SDK\v2.0\Bin)

The Clarion# application

I began with a new Clarion# Windows (that is, Windows Forms) project, which supplied me with a MainForm.cln file containing a form. The Clarion# Windows Designer also creates a *formname*.Designer.cln file (MainForm.Designer.cln in this case) that contains all the information and code needed to intialize the form. You must be very careful to make screen changes using the Window Designer. Directly editing the designer.cln file is discouraged and it is very easy to ruin your application; I know because I have done just that.

Figure 1 shows the Google Calendar demo in the Window Designer.



URL: http://www.google.com/calendar/feeds/default/private/full?

User: sample@gmail.com

Figure 1. The Window Designer

The elements of the screen in Figure 1 are, from left to right and top to bottom, a MonthCalendar control, some labels and TextBoxes, a ListView control showing events, and a couple of SoftVelocity's new GlassButton controls.

The source code for the MainForm starts with the familiar member statement and is followed by a new namespace compiler directive that is used to declare a scope in the .NET framework. The USING directive lets you reference objects in those namespaces without having to specify the namespace in your code. (Editor's note: As of the next build of Clarion# the single quotes will no longer be needed for NAMESPACE and USING.)

```
MEMBER('')
NAMESPACE('SampleApp')
USING('System')
USING('System.Drawing')
USING('System.Collections')
USING('System.ComponentModel')
USING('System.Windows.Forms')
USING('Google.GData.Client')
USING('Google.GData.Extensions')
USING('Google.GData.Calendar')
```

One new concept that is ubiquitous in Clarion# is that pretty well everything is a class, including Forms. In this example, the MainForm is a typed class with three private properties and a half dozen methods. The EntryList and ArrayList are used to hold the calendar events, m_calendarURI is the Uniform Resource Identifier (URI) for the calendar to be accessed, and m_selectedDate holds the date that the user selects via the MonthCalendar control.

The Construct method is generated automatically by the Clarion.Net IDE, and the other methods are ones that I created (via the Designer) to handle the various event responses. I also created a reference, this, to a MainForm class to make it a bit easier for me to port the code from C# to Clarion#. I could have changed all instances of this in the sample code to self, but I am a lazy programmer.

```
MainForm       CLASS(System.Windows.Forms.Form),TYPE,NETCLASS,PARTIAL
entryList     &ArrayList,PRIVATE
```

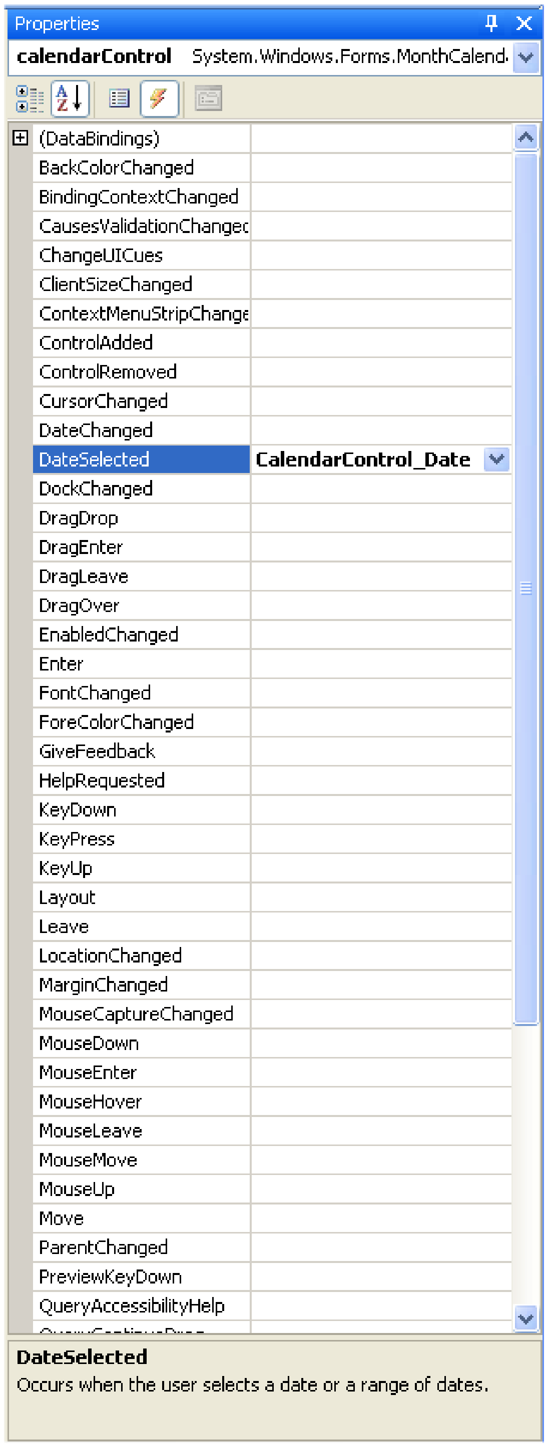
```
m_calendarURI         &String,PRIVATE
m_selectedDate        DateTime,PRIVATE
CONSTRUCT             PROCEDURE(),PUBLIC
CalendarControl_DateSelected  |
                      PROCEDURE(System.Object sender, |
                        System.Windows.Forms.DateRangeEventArgs e),PUBLIC
Go_Click              PROCEDURE(System.Object sender, System.EventArgs e),PUBLIC
GoPublic_Click        PROCEDURE(System.Object sender, System.EventArgs e),PUBLIC
RefreshFeed           PROCEDURE(),PRIVATE
Password_KeyPress     PROCEDURE(System.Object sender, |
                        System.Windows.Forms.KeyPressEventArgs e),PUBLIC
                    END

this        &MainForm
```

The Construct method contains the call to InitializeComponent method which is declared and defined in the MainForm.designer.cln file (the one you do not want to edit as source code). Notice that MainForm has the PARTIAL attribute, indicating it is a *partial class*, spread across multiple files. In this case, the part you can modify is MainForm.cln, and the part you must not modify is in MainForm.Designer.cln.

After the InitializeComponent call, I initialize the m_selectedDate and this properties.

```
MainForm.CONSTRUCT  PROCEDURE()
  CODE
  !
  ! The InitializeComponent() call is required for
  ! Windows Forms designer support.
  !
  SELF.InitializeComponent()
  SELF.m_selectedDate = NEW DateTime(TODAY())
  this &= SELF
```

The CalendarControl_DateSelected event handler is called when the user clicks on a date in the MonthCalendar control. To see the available methods via the Designer, select the calendar control and click on the Events button in the Properties pane. You can type a new method name for any event, or select from the existing methods. The Designer will create a stub method in MainForm.cln, and will add the necessary code to MainForm.Designer.cln to register the event with the control.



Properties

calendarControl System.Windows.Forms.MonthCalend

(DataBindings)	
BackColorChanged	
BindingContextChanged	
CausesValidationChangec	
ChangeUICues	
ClientSizeChanged	
ContextMenuStripChange	
ControlAdded	
ControlRemoved	
CursorChanged	
DateChanged	
DateSelected	CalendarControl_Date
DockChanged	
DragDrop	
DragEnter	
DragLeave	
DragOver	
EnabledChanged	
Enter	
FontChanged	
ForeColorChanged	
GiveFeedback	
HelpRequested	
KeyDown	
KeyPress	
KeyUp	
Layout	
Leave	
LocationChanged	
MarginChanged	
MouseCaptureChanged	
MouseDown	
MouseEnter	
MouseHover	
MouseLeave	
MouseMove	
MouseUp	
Move	
ParentChanged	
PreviewKeyDown	
QueryAccessibilityHelp	

DateSelected
Occurs when the user selects a date or a range of dates.

Figure 2. Calendar control events

At the start of the procedure I declare references to the various classes I will need later on. Several of these are lists of objects, and I'll process these, not with a LOOP, but with the new ForEach looping statement.

```
MainForm.CalendarControl_DateSelected  |
    PROCEDURE(System.Object sender, |
     System.Windows.Forms.DateRangeEventArgs e)
results     &ArrayList
```

```
evtEntry    &EventEntry
evtTimes    &WhenCollection
w           &When
lvItem      &ListViewItem
```

The first conditional after the code statement checks to see if the DateRangeEventArgs are null. Normally, when invoked because of an event, the DateRangeEventArgs are not null, but I also make calls to this method from a few other places and pass a null reference for the DateRangeEventArgs. If I got here from an actual event I save the selected date and clear the ListView control, which is called DayEvents. Note that this is the actual object name, not a field equate. You won't see any field equates in this example.

```
CODE
IF ~e &= NULL
  this.m_selectedDate = e.Start
  this.DayEvents.Items.Clear()
END
```

The entryList object contains the events received from the Google Calendar API. entryList gets filled in the RefreshFeed method which I discuss later.

This sample application only shows five events in the ListView control. You would need to make the events array larger to show more events.

If entryList has some entries in it, I create a new ArrayList to hold the events I am going to display. I then iterate through the collection of EventEntries using the new ForEach statement. Each event has a WhenCollection of event dates; I iterate through that collection looking for dates that match the date that the user clicked on. Matching entries are then added to the results array.

```
IF ~this.entryList &= NULL AND this.entryList.Count > 0
  results &= NEW ArrayList(5)
  FOREACH evtEntry IN this.entryList
    ! let's find the entries for that date
    evtTimes &= (evtEntry.Times)
    IF evtTimes.Count > 0
      FOREACH w IN evtTimes
        IF this.m_selectedDate.Date.CompareTo(w.StartTime.Date) = 0 OR |
          this.m_selectedDate.Date.CompareTo(w.EndTime.Date) = 0
          results.Add(evtEntry)
          BREAK
        END
      END
    END
  END
```

Finally, I iterate through the results array and for each entry I create a ListViewItem object, setting the event title, author name, and start and end times. I add the ListViewItem object to the ListView control for display.

```
  FOREACH evtEntry IN results
```

```
      lvItem &= NEW ListViewItem(evtEntry.Title.Text)
      lvItem.SubItems.Add(evtEntry.Authors[0].Name)
      evtTimes &= (evtEntry.Times)
      IF evtTimes.Count > 0
        lvItem.SubItems.Add(evtTimes[0].StartTime.TimeOfDay.ToString())
        lvItem.SubItems.Add(evtTimes[0].EndTime.TimeOfDay.ToString())
        this.DayEvents.Items.Add(lvItem)
      END
    END
```

That's how calendar data is displayed, but before you can display the data you have to retrieve or add the data.

Retrieving calendar data

The Go_Click method is called when the user clicks on the Personal Calendar button. I set the URI for the user's private calendar, specifying that I want the data sorted by ascending date order. Next I call the RefreshFeed method to retrieve data from Google, and then I call the CalendarControl_DateSelected method with a null EventArgs parameter, as discussed previously, to display the events for the currently selected date.

```
MainForm.Go_Click   PROCEDURE(System.Object sender, System.EventArgs e)
  CODE
  SELF.m_calendarURI &= |
    'http://www.google.com/calendar/feeds/default/private/
     full?orderby=starttime&sortorder=ascending'
  SELF.CalendarURI.Text := SELF.m_calendarURI
  SELF.RefreshFeed()
  SELF.CalendarControl_DateSelected( sender, NULL)
```

The GoPublic_Click method is called when the user clicks on the Public Calendar button. This bit was not in the original sample application, but I wanted to figure out how to access my public calendar. It took a bit of research, but I eventually found what I was looking for. The URI is pretty cryptic to say the least.

```
MainForm.GoPublic_Click PROCEDURE(System.Object sender, System.EventArgs e)
  CODE
  SELF.m_calendarURI &= | ! line breaks added below
    'http://www.google.com/calendar/feeds/d882sjdnar12kk80c1lc1h9flk@
    group.calendar.google.com/private/full
    ?orderby=starttime&sortorder=ascending'
  SELF.CalendarURI.Text := SELF.m_calendarURI
  SELF.RefreshFeed()
  SELF.CalendarControl_DateSelected( sender, NULL)
```

To discover the URI for your public calendar, do the following after signing in to your Google account:

- Click the My Account and then My Services Calendar links.
- On the left side of the screen you should find a small link that says Manage Calendars; click this link (Figure 3)

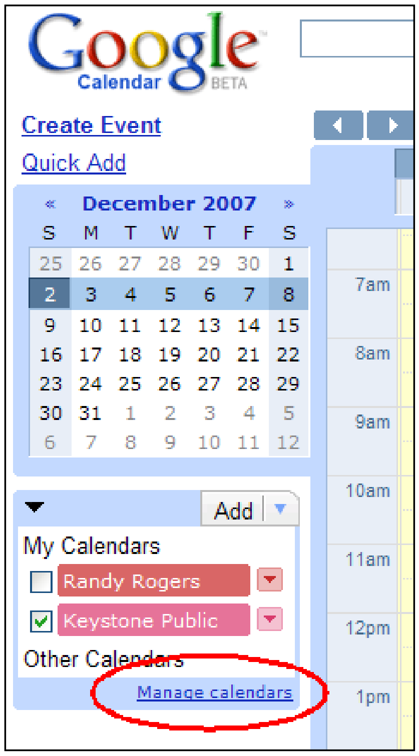


Figure 3. The Manage calendars link

- You should then see something similar to Figure 4:

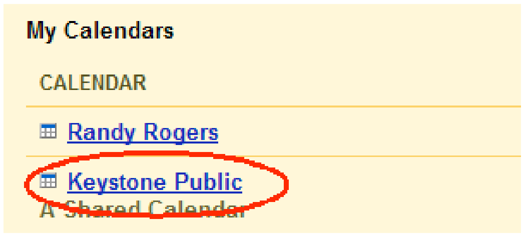
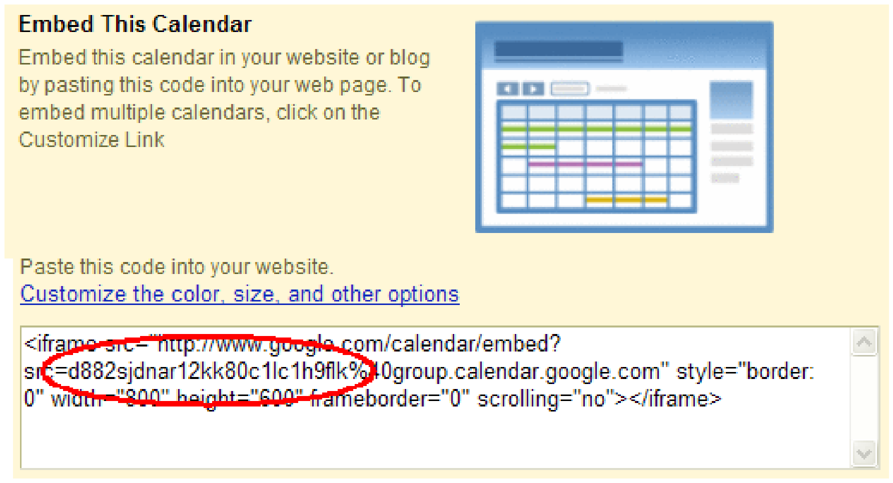


Figure 4. The public calendar link

- Click on the link for your public calendar and you will find the odd bit that you need to construct the URI for your public calendar (Figure 5).



Embed This Calendar

Embed this calendar in your website or blog by pasting this code into your web page. To embed multiple calendars, click on the Customize Link

Paste this code into your website.

Customize the color, size, and other options

```
<iframe src="http://www.google.com/calendar/embed?src=d882sjdnar12kk80c1lc1h9flk%40group.calendar.google.com" style="border: 0" width="800" height="600" frameborder="0" scrolling="no"></iframe>
```

Figure 5. The calendar URL

I call the RefreshFeed method whenever I want to retrieve calendar data from Google. First, I declare references to the classes I am going to need to use:

```
MainForm.RefreshFeed    PROCEDURE()
userName              &string
passWord              &string
dates                 &ArrayList
query                 &EventQuery
service               &CalendarService
calFeed               &EventFeed
evtEntry              &EventEntry
evtTimes              &WhenCollection
w                     &When
aDates                DateTime[]
i                     LONG
d                     DateTime
exType                &System.Type
tryAgain              BOOL
populateCalendar      BOOL
```

I disable the Go and GoPublic buttons to prevent the user from clicking on them while the code is busy. Then I initialize the references and verify the users credentials with the setUserCredentials method of the Calendar Service.

```
CODE
this.Go.Enabled = FALSE
this.GoPublic.Enabled = FALSE
userName      &= this.UserName.Text
passWord      &= this.Password.Text
dates         &= NEW ArrayList(50)
query         &= NEW EventQuery
service       &= NEW CalendarService('SampleApp')
```

```
IF ~userName &= NULL AND userName.Length > 0
  service.setUserCredentials(userName, passWord)
END
```

Now I set up the query parameters and try to get the Calendar data feed from Google. If I am successful I display the calendar title on the screen, clear the ListView control, and create a new ArrayList for fifty entries.

```
! only get event's for today - 1 month until today + 6 months
query.Uri &= NEW Uri(SELF.m_calendarURI)
query.StartTime = DateTime.Now.AddMonths(-1)
query.EndTime = DateTime.Now.AddMonths(6)
calFeed &= NULL
populateCalendar = TRUE
LOOP
  tryAgain = FALSE
  TRY
    calFeed &= service.Query(query) AS EventFeed
    this.CalendarTitle.Text = calFeed.Title.Text & ' Calendar'
    this.DayEvents.Items.Clear()
    this.entryList  &= NEW ArrayList(50)
```

If there is an error, I catch the exception and try to display an intelligent error message.

```
  CATCH (Exception ex)
    exType &= ex.GetType()
    CASE exType.Name.ToString()
    OF 'AuthenticationException'
      MESSAGE('Are your user name and password correct?', |
        exType.Name.ToString() ,ICON:HAND)
    OF 'GDataRequestException'
      CASE MESSAGE('An error occurred trying to retrieve the calendar events.'|
        & '|Would you like to retry?', exType.Name.ToString() ,|
        ICON:QUESTION, BUTTON:YES+BUTTON:NO, BUTTON:YES)
    OF BUTTON:YES
      tryAgain = TRUE
    OF BUTTON:NO
      populateCalendar = FALSE
    ELSE
      MESSAGE('An Unexpected Error Has Occurred!', exType.Name.ToString(), |
        ICON:HAND)
      populateCalendar = FALSE
    END
  FINALLY
  END
```

```
      IF tryAgain = FALSE
        BREAK
      END
    END
```

TRY, CATCH, and FINALLY make up Clarion#'s new structured exception handling mechanism. If service.Query encounters an error it will throw an exception; that means that execution inside the TRY block immediately terminates and resumes at the CATCH statement where the exception can be processed. An optional FINALLY clause will execute when CATCH is done.

I loop until there are no more entries in the calendar feed; the loop also repeats if there was a connection error and the user wishes to retry. The query results are iterated through using foreach and the event start times are stored in the dates array.

```
    IF populateCalendar = TRUE
      ! now populate the calendar
      LOOP WHILE (~calFeed &= NULL AND calFeed.Entries.Count > 0)
        FOREACH evtEntry IN calFeed.Entries
          this.entryList.Add(evtEntry)
          evtTimes &= (evtEntry.Times)
          IF evtTimes.Count > 0
            FOREACH w IN evtTimes
              dates.Add(w.StartTime)
            END
          END
        END
        ! just query the same query again.
        IF (~calFeed.NextChunk &= NULL)
          query.Uri &= NEW Uri(calFeed.NextChunk)
          calFeed &= service.Query(query) AS EventFeed
        ELSE
          calFeed &= NULL
        END
      END
```

Next I create an array of DateTime objects, one for each event date. I assign that array to the calendar control's BoldedDates property and the control automatically displays those dates in bold. Finally I re-enable the buttons on the screen.

```
      aDates &= NEW DateTime[dates.Count]
      i = 0
      FOREACH d in dates
        i += 1
        aDates[i] = d
      END
      this.calendarControl.BoldedDates &= aDates
    END
    this.Go.Enabled = TRUE
```

```
this.GoPublic.Enabled = TRUE
```

When I was testing the program, I would quite often enter my username and password and then simply press Enter. I was expecting some default action that didn't occur, so I initially added a method to watch for the Enter key, and programmatically click the Personal Calendar button. I later discovered that I could achieve the effect I really wanted by assigning the Personal Calendar Button to the form's AcceptButton property.

To run the program you need a Google account. Enter your username and password and then click on one of the buttons, or just press Enter for your personal calendar. If you have put events in your Google calendar, you should see them displayed on the screen. There are only a couple of entries in the Keystone Public calendar for November 2007. The example demonstrates how to retrieve data; it is also possible with the Google Calendar API to manage calendars and maintain events programmatically.

I certainly had fun putting this example together and I am enjoying having access to all the code available in the . Net framework. An executable version of this program and other Clarion# examples can be found on my web site.

Download the source

Randy Rogers is a data processing professional with over 35 years of experience in a wide variety of industries including accounting, municipal government, insurance, printing, and pharmacoeconomics. He has a degree in Mathematics from Florida State University and is the president of Keystone Computer Resources. Randy is the author of ClassViewer, a utility for browsing the Clarion class hierarchies. He is also the creator of NetTools, Queue Edit-in-Place, and Screen Capture Tools for Clarion application developers.

Reader Comments

Add a comment

Comparison of C#, Clarion# and VB.NET

This is a quick reference guide to highlight some key syntactical differences between C#, Clarion# and VB.NET (version 2).

NOTE: Clarion.NET and the Clarion# language are currently in alpha test, and various areas of the documentation are incomplete and in a state of flux. Therefore, it's very likely that some of the entries will change as new information becomes available.

Program Structure

C#

```
using System;

namespace Hello {
   public class HelloWorld {
      public static void Main(string[] args) {
         string name = "C#";

         //See if an argument was passed from the command line
         if (args.Length == 1)
            name = args[0];

         Console.WriteLine("Hello, " + name + "!");
      }
   }
}
```

Clarion#

```
  PROGRAM
  NAMESPACE(Hello)
  USING(System)
  MAP
  END

Name &STRING

  CODE
  Name = 'Clarion#'
  !See if an argument was passed from the command line
  IF COMMAND('1') <> ''
    Name = COMMAND('1')
  END
  Console.WriteLine('Hello, '& Name & '!')
```

VB.NET

```
Imports System

Namespace Hello
   Class HelloWorld
      Overloads Shared Sub Main(ByVal args() As String)
         Dim name As String = "VB.NET"

         'See if an argument was passed from the command line
         If args.Length = 1 Then name = args(0)

         Console.WriteLine("Hello, " & name & "!")
      End Sub
   End Class
End Namespace
```

Comments

C#

```
//Single line


/* Multiple
   lines */

///<summary>XML comments on single line</summary>

/** <summary>
  XML comments on multiple lines
</summary> */
```

Clarion#

```
!Single line


!~ Multiple
   lines ~!

!!!<summary>XML comments on single line</summary>

!!!<summary>
!!!  XML comments on multiple lines
!!!</summary>
```

VB.NET

```
'Single line only
REM Single line only



'''<summary>XML comments</summary>
```

Data Types

C#	Clarion#	VB.NET

// Value Types
`bool`
`byte, sbyte`
`char`
`short, ushort, int, uint, long, ulong`
`float, double`
`decimal`
`DateTime    //not a built-in C# type`
	! Value Types
	`BOOL`
	`BYTE, SBYTE`
	`CHAR, CSTRING, PSTRING, CLASTRING`
	`SHORT, USHORT, SIGNED, UNSIGNED, LONG, ULONG, CALONG`
	`SREAL, REAL, BFLOAT4, BFLOAT8`
	`DECIMAL, PDECIMAL, CLADECIMAL`
	`DATE, TIME, CLADATE, CLATIME`
		' Value Types
		`Boolean`
		`Byte, SByte`
		`Char`
		`Short, UShort, Integer, UInteger, Long, ULong`
		`Single, Double`
		`Decimal`
		`Date`

// Reference Types
`object`
`string`
	! Reference Types
	`&OBJECT`
	`&STRING`
		' Reference Types
		`Object`
		`String`

// Initializing
`bool correct = true;`
`byte b = 0x2A;   //hex`
 	
`object person = null;`
`string name = "Mike";`
`char grade = 'B';`
`DateTime today = DateTime.Parse("12/31/2007 12:15:00");`
`decimal amount = 35.99m;`
`float gpa = 2.9f;`
`double pi = 3.14159265;`
`long lTotal = 123456L;`
`short sTotal = 123;`
`ushort usTotal = 123;`
`uint uiTotal = 123;`
`ulong ulTotal = 123;`
	! Initializing
	`Correct  BOOL(True)`
	`H        BYTE(02Ah)     !hex`
	`O        BYTE(052o)     !octal`
	`B        BYTE(01101b)   !binary`
	`Person   &OBJECT`
	`Name     &STRING`
	`  Name = 'Mike'`
	`Grade    CHAR('B')`
	`Today    DATE`
	`  Today = DATE(12,31,2007)`
	`Amount   DECIMAL(35.99)`
	`GPA      SREAL(2.9)`
	`Pi       REAL(3.14159265)`
	`lTotal   LONG(123456)`
	`sTotal   SHORT(123)`
	`usTotal  USHORT(123)`
	`uiTotal  UNSIGNED(123)`
	`ulTotal  ULONG(123)`
		' Initializing
		`Dim correct As Boolean = True`
		`Dim b As Byte = &H2A   'hex`
		`Dim o As Byte = &O52   'octal`
		`Dim person As Object = Nothing`
		`Dim name As String = "Mike"`
		`Dim grade As Char = "B"c`
		`Dim today As Date = #12/31/2007 12:15:00 PM#`
		`Dim amount As Decimal = 35.99@`
		`Dim gpa As Single = 2.9!`
		`Dim pi As Double = 3.14159265`
		`Dim lTotal As Long = 123456L`
		`Dim sTotal As Short = 123S`
		`Dim usTotal As UShort = 123US`
		`Dim uiTotal As UInteger = 123UI`
		`Dim ulTotal As ULong = 123UL`

// Type Information
`int x;`
`Console.WriteLine(x.GetType());          //Prints System.Int32`
`Console.WriteLine(typeof(int));          //Prints System.Int32`
`Console.WriteLine(x.GetType().Name);   //Prints Int32`
	! Type Information
	`X SIGNED`
	`  Console.WriteLine(X.GetType())         !Prints System.Int32`
	`  Console.WriteLine(TYPEOF(SIGNED))     !Prints System.Int32`
	`  Console.WriteLine(X.GetType().Name)   !Prints Int32`
		' Type Information
		`Dim x As Integer`
		`Console.WriteLine(x.GetType())          'Prints System.Int32`
		`Console.WriteLine(GetType(Integer))   'Prints System.Int32`
		`Console.WriteLine(TypeName(x))          'Prints Integer`

// Type Conversion
`float d = 3.5f;`
`int i = (int)d;    //set to 3  (truncates decimal)`
	! Type Conversion
	`D        SREAL(3.5)`
	`I        SIGNED`
	`  I = D              !Implicitly truncate to 3`
	`  I = D AS SIGNED   !Explicitly truncate to 3`
		' Type Conversion
		`Dim d As Single = 3.5`
		`Dim i As Integer = CType(d, Integer)  'set to 4 (Banker's rounding)`
		`i = CInt(d)   'same result as CType`
		`i = Int(d)    'set to 3 (Int function truncates the decimal)`

Constants

C#

```
const int MAX_STUDENTS = 25;

// Can set to a const or var; may be initialized in a constructor
readonly float MIN_DIAMETER = 4.93f;
```

Clarion#

```
! CONST and READONLY unsupported.
! Use EQUATEs or regular data types instead.
MAX_STUDENTS EQUATE(25)    !Instead of CONST; will auto-convert
MIN_DIAMETER SREAL(4.93)   !READONLY is unsupported
```

VB.NET

```
Const MAX_STUDENTS As Integer = 25

'Can set to a const or var; may be initialized in a constructor
ReadOnly MIN_DIAMETER As Single = 4.93
```

Enumerations

C#

```
enum Action {Start, Stop, Rewind, Forward};






enum Status {
   Flunk = 50,
   Pass  = 70,
   Excel = 90
};

Action a = Action.Stop;
if (a != Action.Start)
   Console.WriteLine(a + " is " + (int) a); //Prints "Stop is 1"

Console.WriteLine((int) Status.Pass);       //Prints 70
Console.WriteLine(Status.Pass);             //Prints Pass
```

Clarion#

```
Action   ENUM                  Action   ENUM   !Alternate syntax
Start      ITEM                           Start
Stop       ITEM                           Stop
Rewind     ITEM                           Rewind
Forward    ITEM                           Forward
         END                              END

Status   ENUM
           Flunk(50)
           Pass (70)
           Excel(90)
         END

A        Action(Action.Stop)
  IF (A <> Action.Start)
    Console.WriteLine(A.ToString() &'is '& A)  !Prints "Stop is 1"
  END
  Console.WriteLine(Status.Pass + 0)           !Prints 70
  Console.WriteLine(Status.Pass)               !Prints Pass
```

VB.NET

```
Enum Action
  Start
  [Stop]   'Stop is a reserved word
  Rewind
  Forward
End Enum

Enum Status
  Flunk = 50
  Pass  = 70
  Excel = 90
End Enum

Dim a As Action = Action.Stop
If a <> Action.Start Then _
   Console.WriteLine(a.ToString & " is " & a)  'Prints "Stop is 1"

Console.WriteLine(Status.Pass)                 'Prints 70
Console.WriteLine(Status.Pass.ToString())      'Prints Pass
```

Operators

C#

```
// Comparison
==  <  >  <=  >=  !=

// Arithmetic
+  -  *  /
%              //mod
/              //integer division if both operands are ints
Math.Pow(x, y) //raise to a power

// Assignment
=  +=  -=  *=  /=   %=  &=  |=  ^=  <<=  >>=  ++  --


// Bitwise
&  |  ^  ~  <<  >>
```

Clarion#

```
! Comparison
=  <  >  <=  >=  ~=  <> &=

! Arithmetic
+  -  *  /
%    !mod
/    !integer division
^    !raise to a power

! Assignment
=  +=  -=  *=  /=  %=  &=
:=   !"Smart" replacement for = and &=

! Bitwise
BAND(val,mask) BOR(val,mask) BXOR(val,mask) BSHIFT(val,count)
```

VB.NET

```
' Comparison
=  <  >  <=  >=  <>

' Arithmetic
+  -  *  /
Mod
\    'integer division
^    'raise to a power

' Assignment
=  +=  -=  *=  /=  \=  ^=  <<=  >>=  &=


' Bitwise
And   Or   Xor   Not   <<   >>
```

```
// Logical
&&   ||   &   |   ^   !

// Note:  && and || perform short-circuit logical evaluations

// String Concatenation
+
```

```
! Logical
AND OR XOR NOT

! Note:  AND and OR perform short-circuit logical evaluations

! String Concatenation
&
```

```
' Logical
AndAlso   OrElse   And   Or   Xor   Not

' Note:  AndAlso and OrElse perform short-circuit logical evaluations

' String Concatenation
&
```

Choices

C#

```
greeting = age < 20 ? "What's up?" : "Hello";

if (age < 20)
  greeting = "What's up?";
else
  greeting = "Hello";

// Semi-colon ";" is used to terminate each statement,
// so no line continuation character is necessary.

// Multiple statements must be enclosed in {}
if (x != 100 && y < 5) {
  x *= 5;
  y *= 2;
}

if (x > 5)
  x *= y;
else if (x == 5)
  x += y;
else if (x < 10)
  x -= y;
else
  x /= y;
```

Clarion#

```
Greeting = CHOOSE(Age < 20, 'What''s up?', 'Hello')

! One line requires THEN (or ;)
IF Age < 20 THEN Greeting = 'What''s up?' END
IF Age < 20; Greeting = 'What''s up?'ELSE Greeting = 'Hello' END

! Use semi-colon (;) to put two commands on same line.
! A period (.) may replace END in single line constructs,
! but it is discouraged for multi-line constructs.
IF X <> 100 AND Y < 5 THEN X *= 5; Y *= 2.   !Period is OK here

! Multi-line is more readable (THEN is optional on multi line)
IF X <> 100 AND Y < 5 THEN
  X *= 5
  Y *= 2
END

IF X <> 100 AND Y < 5
  X *= 5
  Y *= 2
.   !Period is hard to see here

! To break up any long single line use | (pipe)
IF WhenYouHaveAReally < LongLine |
AND ItNeedsToBeBrokenInto2 > Lines
  UseThePipe(CharToBreakItUp)
END

IF X > 5
  X *= Y
ELSIF X = 5
  X += Y
ELSIF X < 10
  X -= Y
ELSE
  X /= Y
END
```

VB.NET

```
greeting = IIf(age < 20, "What's up?", "Hello")

' One line doesn't require End If
If age < 20 Then greeting = "What's up?"
If age < 20 Then greeting = "What's up?" Else greeting = "Hello"

' Use colon (:) to put two commands on same line
If x <> 100 AndAlso y < 5 Then x *= 5 : y *= 2

' Multi-line is more readable
If x <> 100 AndAlso y < 5 Then
  x *= 5
  y *= 2
End If

' To break up any long single line use _
If whenYouHaveAReally < longLine AndAlso _
    itNeedsToBeBrokenInto2 > Lines Then _
    UseTheUnderscore(charToBreakItUp)

If x > 5 Then
  x *= y
ElseIf x = 5 Then
  x += y
ElseIf x < 10 Then
  x -= y
Else
  x /= y
End If
```

```
// Every case must end with break or goto case
switch (color) {                      //Must be integer or string
  case "pink" :
  case "red"  : r++;      break;
  case "blue" : b++;      break;
  case "green": g++;      break;
  default:      other++; break;   //break necessary on default
}
```

```
CASE Color                  !Any data type or expression
OF 'pink' OROF 'red'
  R += 1
OF 'blue'
  B += 1
OF 'green'
  G += 1
ELSE
  Other += 1
END

CASE Value
  OF     0.00 TO   9.99;  RangeName = 'Ones'
  OF    10.00 TO  99.99;  RangeName = 'Tens'
  OF   100.00 TO 999.99;  RangeName = 'Hundreds'
  !etc.
  ELSE                 ;  RangeName = 'Zillions'
END

EXECUTE Stage          !Integer value or expression
  Stage1               ! expression equals 1
  Stage2               ! expression equals 2
  Stage3               ! expression equals 3
ELSE
  StageOther           ! expression equals some other value
END
```

```
Select Case color        'Must be a primitive data type
  Case "pink", "red"
    r += 1
  Case "blue"
    b += 1
  Case "green"
    g += 1
  Case Else
    other += 1
End Select
```

Loops

C#

```
// Pre-test Loops
while (c < 10)
  c++;

// no "until" keyword

for (c = 2; c <= 10; c += 2)
  Console.WriteLine(c);

// Post-test Loop
do
  c++;
while (c < 10);

// Untested Loop
for (;;) {
  //break logic inside
}
```

Clarion#

```
! Pre-test Loops
LOOP WHILE C < 10
  C += 1
END
```

```
LOOP UNTIL C = 10
  C += 1
END

LOOP C = 2 TO 10 BY 2
  Console.WriteLine(C)
END
```

```
! Post-test Loops
LOOP
  C += 1
WHILE c < 10
```

```
LOOP
  C += 1
UNTIL C = 10
```

```
! Untested Loops
LOOP
  !Break logic inside
END
```

```
LOOP 3 TIMES
  Console.WriteLine
END
```

VB.NET

```
' Pre-test Loops
While c < 10
  c += 1
End While

Do While c < 10
  c += 1
Loop
```

```
Do Until c = 10
  c += 1
Loop

For c = 2 To 10 Step 2
  Console.WriteLine(c)
Next
```

```
' Post-test Loops
Do
  c += 1
Loop While c < 10
```

```
Do
  c += 1
Loop Until c = 10
```

```
' Untested Loop
Do
  //break logic inside
Loop
```

```
// Array or collection looping
string[] names = {"Fred", "Sue", "Barney"};
foreach (string s in names)
  Console.WriteLine(s);
```

```
! Array or collection looping
Names &STRING,DIM(3)
S      &STRING
       CODE
       Names[1] := 'Fred'; Names[2] := 'Sue'; Names[3] := 'Barney'
       FOREACH S IN Names
         Console.WriteLine(S)
       END
```

```
' Array or collection looping
Dim names As String() = {"Fred", "Sue", "Barney"}
For Each s As String In names
  Console.WriteLine(s)
Next
```

```
// Breaking out of loops
int i = 0;
while (true) {
  if (i == 5)
    break;
  i++;
}
```

```
! Breaking out of loops
I      SHORT(0)
       CODE
       LOOP
         IF I = 5 THEN BREAK.
         I += 1
       END
```

```
' Breaking out of loops
Dim i As Integer = 0
While (True)
  If (i = 5) Then Exit While
  i += 1
End While
```

```
// Continue to next iteration
for (i = 0; i < 5; i++) {
  if (i < 4)
    continue;
  Console.WriteLine(i);   //Only prints 4
}
```

```
! Continue to next iteration
       LOOP I = 0 TO 4
         IF I < 4 THEN CYCLE.
         Console.WriteLine(I)
       END
```

```
' Continue to next iteration
For i = 0 To 4
  If i < 4 Then Continue For
  Console.WriteLine(i)   'Only prints 4
Next
```

Arrays

C#

```
int[] nums = {1, 2, 3};
for (int i = 0; i < nums.Length; i++)
  Console.WriteLine(nums[i]);
```

```
// 5 is the size of the array
string[] names = new string[5];
names[0] = "David";
names[5] = "Bobby";   //Throws System.IndexOutOfRangeException
```

```
// C# can't dynamically resize arrays, so copy into new array
string[] names2 = new string[7];
Array.Copy(names, names2, names.Length);
// or
names.CopyTo(names2, 0);
```

```
float[,] twoD = new float[rows, cols];
twoD[2,0] = 4.5f;
```

Clarion#

```
Nums    SIGNED,DIM(3)
I       SIGNED
        CODE
        Nums[1] = 1; Nums[2] = 2; Nums[3] = 3
        LOOP I = 1 TO Nums.Length
          Console.WriteLine(Nums[I])
        END
```

```
! 5 is the size of the array
Names   &STRING,DIM(5)
        CODE
        Names[1] := 'David'
        Names[6] := 'Bobby'  !Caught by compiler
        I = 6
        Names[I] := 'Bobby'  !Throws System.IndexOutOfRangeException
```

```
! Clarion# can't dynamically resize arrays, so copy into new array
Names2 &STRING[]
        CODE
        Names2 := NEW STRING[7]
        Array.Copy(Names, Names2, Names.Length)  !or
        Names.CopyTo(Names2, 0)
```

```
TwoD    SREAL[,]
        CODE
        TwoD := NEW SREAL[Rows, Cols]
        TwoD[3,1] = 4.5
```

VB.NET

```
Dim nums() As Integer = {1, 2, 3}
For i As Integer = 0 To nums.Length - 1
  Console.WriteLine(nums(i))
Next
```

```
' 4 is the index of the last element, so it holds 5 elements
Dim names(4) As String
names(0) = "David"
names(5) = "Bobby"   'Throws System.IndexOutOfRangeException
```

```
' Resize the array, keeping existing values (Preserve is optional)
ReDim Preserve names(6)
```

```
Dim twoD(rows-1, cols-1) As Single
twoD(2, 0) = 4.5
```

```
// Jagged arrays
int[][] jagged = new int[3][] {
  new int[5], new int[2], new int[3] };
jagged[0][4] = 5;
```

```
! Jagged arrays unsupported
```

```
' Jagged arrays
Dim jagged()() As Integer = { _
  New Integer(4) {}, New Integer(1) {}, New Integer(2) {} }
jagged(0)(4) = 5
```

Functions

C#

```
// Pass by value(in,default), reference(in/out), and reference(out)
void TestFunc(int x, ref int y, out int z) {
  x++;
  y++;
  z = 5;
}


int a = 1, b = 1, c;   //c doesn't need initializing
TestFunc(a, ref b, out c);
Console.WriteLine("{0} {1} {2}", a, b, c);   //1 2 5
```

```
// Accept variable number of arguments
int Sum(params int[] nums) {
  int sum = 0;
  foreach (int i in nums)
    sum += i;
  return sum;
}

int total = Sum(4, 3, 2, 1);    //returns 10
```

```
/* C# doesn't support optional arguments/parameters.
   Just create two different versions of the same function. */
void SayHello(string name, string prefix) {
  Console.WriteLine("Greetings, " + prefix + " " + name);
}
void SayHello(string name) {
  SayHello(name, "");
}
```

Clarion#

```
! Pass by value(in,default), reference(in/out), and reference(out)
TestFunc PROCEDURE(SIGNED X, *SIGNED Y, *SIGNED Z)
  CODE
  X += 1
  Y += 1
  Z = 5
  RETURN  !Optional, if not returning a value


A UNSIGNED(1)
B UNSIGNED(1)
C UNSIGNED      !C doesn't need initializing
  CODE
  TestFunc(A, B, C)
  Console.WriteLine('{{0} {{1} {{2}', A, B, C)  !1 2 5
```

```
! Accept variable number of arguments
Sum PROCEDURE(PARAMS UNSIGNED[] Nums),UNSIGNED
Result UNSIGNED(0)
I      UNSIGNED
  CODE
  FOREACH I IN Nums
    Result += I
  END
  RETURN Result

  Total# = Sum(4, 3, 2, 1)    !returns 10
```

```
! Optional parameters without default value
! (When omitted, value parameters default to 0 or an empty string)
! (Use OMITTED to detect the omission)
SayHello PROCEDURE(STRING Name, <STRING Prefix>)

! Optional parameters with default value
! (Valid only on simple numeric types)
! (OMITTED will not detect the omission — the default is passed)
SayHello PROCEDURE(STRING Name, BYTE Age = 20)
```

VB.NET

```
' Pass by value(in,default), reference(in/out), and reference(out)
Sub TestFunc(ByVal x As Integer, ByRef y As Integer, _
             ByRef z As Integer)
  x += 1
  y += 1
  z = 5
End Sub


Dim a = 1, b = 1, c As Integer    'c set to zero by default
TestFunc(a, b, c)
Console.WriteLine("{0} {1} {2}", a, b, c)   '1 2 5
```

```
' Accept variable number of arguments
Function Sum(ByVal ParamArray nums As Integer()) As Integer
  Sum = 0
  For Each i As Integer In nums
    Sum += i
  Next
End Function    'Or use Return statement like C#

Dim total As Integer = Sum(4, 3, 2, 1)    'returns 10
```

```
' Optional parameters must be listed last and have a default value
Sub SayHello(ByVal name As String, Optional ByVal prefix As String = "")
  Console.WriteLine("Greetings, " & prefix & " " & name)
End Sub

SayHello("Strangelove", "Dr.")
SayHello("Madonna")
```

Strings

C#	Clarion#	VB.NET

// Escape sequences
`\r    //carriage-return`
`\n    //line-feed`
`\t    //tab`
`\\    //backslash`
`\"    //quote`
	! Special Characters
	`<13>` *!carriage-return*
	`<10>` *!line-feed*
	`<9>` *!tab*
	`<n>` *!character with the ASCII value=n (see above)*
	`<<` *!less-than*
	`{{` *!left-curly-brace*
	`''` *!single-quote*
	`{n}` *!Repeat previous character "n" times*
		' Special Character Constants
		`vbCrLf, vbCr, vbLf, vbNewLine`
		`vbNullString`
		`vbTab`
		`vbBack`
		`vbFormFeed`
		`vbVerticalTab`
		`""`

// String concatenation
`string school = "Harding\t";`
`school = school + "University";`
// school is "Harding(tab)University"
	! String concatenation
	`School &STRING`
	`Univ   CLASTRING('University')` *!Clarion string class*
	`       CODE`
	`       School = 'Harding<9>'`
	`       School = School & Univ`
	!School is "Harding(tab)University"
		' String concatenation (use & or +)
		`Dim school As String = "Harding" & vbTab`
		`school = school & "University"`
		'school is "Harding(tab)University"

// Chars
`char letter = school[0];` *//letter is H*
`letter = Convert.ToChar(65);` *//letter is A*
`letter = (char)65;` *//same thing*
`char[] word = school.ToCharArray();` *//word holds Harding*
	! Chars
	`Letter CHAR`
	`Word   CHAR[]`
	`       CODE`
	`       Letter = School[1]` *!Letter is H*
	`       Letter = Convert.ToChar(65)` *!Letter is A*
	`       Letter = '<65>'` *!Same thing*
	`       Word   = School.ToCharArray` *!Word holds Harding*
		' Chars
		`Dim letter As Char = school.Chars(0)` *'letter is H*
		`letter = Convert.ToChar(65)` *'letter is A*
		`letter = Chr(65)` *'same thing*
		`Dim word() As Char = school.ToCharArray()` *'word holds Harding*

// String literal
`string msg = @"File is c:\temp\x.dat";`
// same as
`string msg = "File is c:\\temp\\x.dat";`
	! No string literal operator
	`Msg    &STRING`
	`       CODE`
	`       Msg = 'File is c:\temp\x.dat'`
		' No string literal operator
		`Dim msg As String = "File is c:\temp\x.dat"`

// String comparison
`string mascot = "Bisons";`
`if (mascot == "Bisons")` *//true*
`if (mascot.Equals("Bisons"))` *//true*
`if (mascot.ToUpper().Equals("BISONS"))` *//true*
`if (mascot.CompareTo("Bisons") == 0)` *//true*
	! String comparison
	`Mascot &STRING`
	`       CODE`
	`       Mascot = 'Bisons'`
	`       IF Mascot = 'Bisons'` *!true*
	`       IF Mascot.Equals('Bisons')` *!true*
	`       IF Mascot.ToUpper().Equals('BISONS')` *!true*
	`       IF UPPER(Mascot) = 'BISONS'` *!true*
	`       IF Mascot.CompareTo('Bisons') = 0` *!true*
		' String comparison
		`Dim mascot As String = "Bisons"`
		`If (mascot = "Bisons") Then` *'true*
		`If (mascot.Equals("Bisons")) Then` *'true*
		`If (mascot.ToUpper().Equals("BISONS")) Then` *'true*
		`If (mascot.CompareTo("Bisons") = 0) Then` *'true*

// Substring `Console.WriteLine(mascot.Substring(2, 3));` *//Prints "son"*	*! Substring* `Console.WriteLine(Mascot.Substring(2, 3))` *!Prints "son"* `Console.WriteLine(SUB(Mascot, 3, 3))` *!Prints "son"* `Console.WriteLine(Mascot[3 : 6])` *!Prints "son"*	*' Substring* `Console.WriteLine(mascot.Substring(2, 3))` *'Prints "son"*

```csharp
// String matching
// No exact equivalent to Like - use regular expressions

using System.Text.RegularExpressions;
Regex r = new Regex(@"Jo[hH]. \d:*");
if (r.Match("John 3:16").Success)   //true
```

```
! String matching
! No exact equivalent to Like - use regular expressions or MATCH

  USING(System.Text.RegularExpressions)
R &Regex
A &STRING
B &STRING
C &STRING
  CODE
  R := NEW Regex('Jo[hH]. \d:*')
  IF R.Match('John 3:16').Success          !true
  A = 'Richard'
  B = 'RICHARD'
  C = 'R*'
  IF MATCH(A,B,MATCH:Simple+Match:NoCase)  !true: case insensitive
  IF MATCH(A,B,MATCH:Soundex)              !true: soundex
  IF MATCH(A,C)                            !true: wildcard (default)
  IF MATCH('Fireworks on the fourth', '{{4|four}th', |
           MATCH:Regular+Match:NoCase)     !true: RegEx
  IF MATCH('July 4th fireworks', '{{4|four}th', |
           MATCH:Regular+Match:NoCase)     !true: RegEx
```

```vb
' String matching
If ("John 3:16" Like "Jo[Hh]? #:*") Then     'true

Imports System.Text.RegularExpressions       'More powerful than Like
Dim r As New Regex("Jo[hH]. \d:*")
If (r.Match("John 3:16").Success) Then       'true
```

```csharp
// My birthday: Sep 3, 1964
DateTime dt = new DateTime(1964, 9, 3);
string s = "My birthday: " + dt.ToString("MMM dd, yyyy");
```

```
! My birthday: Sep 3, 1964
DT DateTime
S  &STRING
  CODE
  DT = NEW DateTime(1964, 9, 3)
  S  = 'My birthday: '& DT.ToString('MMM dd, yyyy')
```

```vb
' My birthday: Sep 3, 1964
Dim dt As New DateTime(1964, 9, 3)
Dim s As String = "My birthday: " & dt.ToString("MMM dd, yyyy")
```

```csharp
// Mutable string
System.Text.StringBuilder buffer =
    new System.Text.StringBuilder("two ");
buffer.Append("three ");
buffer.Insert(0, "one ");
buffer.Replace("two", "TWO");
Console.WriteLine(buffer);     //Prints "one TWO three"
```

```
! Mutable string
Buffer System.Text.StringBuilder('two ')
  CODE
  Buffer.Append('three ')
  Buffer.Insert(0, 'one ')
  Buffer.Replace('two', 'TWO')
  Console.WriteLine(Buffer)     !Prints "one TWO three"
```

```vb
' Mutable string
Dim buffer As New System.Text.StringBuilder("two ")
buffer.Append("three ")
buffer.Insert(0, "one ")
buffer.Replace("two", "TWO")
Console.WriteLine(buffer)     'Prints "one TWO three"
```

Exception Handling

C#

```csharp
// Throw an exception
Exception ex = new Exception("Something is really wrong.");
throw ex;
```

Clarion#

```
! Throw an exception
Ex &Exception('Something is really wrong.')
   CODE
   THROW Ex
```

VB.NET

```vb
' Throw an exception
Dim ex As New Exception("Something is really wrong.")
Throw  ex
```

```
// Catch an exception
try {
  y = 0;
  x = 10 / y;
}
catch (Exception ex) {   //Argument is optional, no "When" keyword
  Console.WriteLine(ex.Message);
}
finally {
  //Requires reference to the Microsoft.VisualBasic.dll
  //assembly (pre .NET Framework v2.0)
  Microsoft.VisualBasic.Interaction.Beep();
}
```

```
! Catch an exception
  TRY
    y = 0;
    x = 10 / y;
  CATCH(Exception Ex)    !Argument is optional, no "When" keyword
    Console.WriteLine(Ex.Message);
  FINALLY
    BEEP
  END
```

```
' Catch an exception
Try
  y = 0
  x = 10 / y
Catch ex As Exception When y = 0 'Argument and When is optional
  Console.WriteLine(ex.Message)
Finally
  Beep()
End Try

' Deprecated unstructured error handling
On Error GoTo MyErrorHandler
...
MyErrorHandler: Console.WriteLine(Err.Description)
```

Namespaces

C#

```
namespace Harding.Compsci.Graphics {
  ...
}

// or

namespace Harding {
  namespace Compsci {
    namespace Graphics {
      ...
    }
  }
}

using Harding.Compsci.Graphics;
```

Clarion#

```
NAMESPACE(Harding.Compsci.Graphics)




!Progressive, nested namespaces unsupported







USING(Harding.Compsci.Graphics)
```

VB.NET

```
Namespace Harding.Compsci.Graphics
  ...
End Namespace

' or

Namespace Harding
  Namespace Compsci
    Namespace Graphics
      ...
    End Namespace
  End Namespace
End Namespace

Imports Harding.Compsci.Graphics
```

Classes / Interfaces

C#

```
// Accessibility keywords
public
private
internal
protected
protected internal
static

// Inheritance
class FootballGame : Competition {
  ...
}
```

Clarion#

```
! Accessibility keywords
PUBLIC
PRIVATE
INTERNAL
PROTECTED
PROTECTED INTERNAL
STATIC

! Inheritance
FootballGame CLASS(Competition)
  ...
             END
```

VB.NET

```
' Accessibility keywords
Public
Private
Friend
Protected
Protected Friend
Shared

' Inheritance
Class FootballGame
  Inherits Competition
  ...
End Class
```

```
// Interface definition
interface IAlarmClock {
  ...
}

// Extending an interface
interface IAlarmClock : IClock {
  ...
}


// Interface implementation
class WristWatch : IAlarmClock, ITimer {
   ...
}
```

```
! Interface definition
IAlarmClock  INTERFACE
  ...
             END

! Extending an interface
IAlarmClock  INTERFACE(IClock)
  ...
             END


! Interface implementation
WristWatch   CLASS,IMPLEMENTS(IAlarmClock),IMPLEMENTS(ITimer)
   ...
             END
```

```
' Interface definition
Interface IAlarmClock
  ...
End Interface

' Extending an interface
Interface IAlarmClock
  Inherits IClock
  ...
End Interface

' Interface implementation
Class WristWatch
  Implements IAlarmClock, ITimer
   ...
End Class
```

Constructors / Destructors

C#

```
class SuperHero {
  private int _powerLevel;

  public SuperHero() {
     _powerLevel = 0;
  }

  public SuperHero(int powerLevel) {
    this._powerLevel= powerLevel;
  }

  ~SuperHero() {
    //Destructor code to free unmanaged resources.
    //Implicitly creates a Finalize method
  }
}
```

Clarion#

```
SuperHero   CLASS
_PowerLevel   UNSIGNED,PRIVATE
Construct     PROCEDURE,PUBLIC
Construct     PROCEDURE(UNSIGNED PowerLevel),PUBLIC
Destruct      PROCEDURE
            END

SuperHero.Construct PROCEDURE
  CODE
  SELF._PowerLevel = 0

SuperHero.Construct PROCEDURE(UNSIGNED PowerLevel)
  CODE
  SELF._PowerLevel = PowerLevel

SuperHero.Destruct PROCEDURE
  CODE
  !Destructor code to free unmanaged resources.
```

VB.NET

```
Class SuperHero
  Private _powerLevel As Integer

  Public Sub New()
    _powerLevel = 0
  End Sub

  Public Sub New(ByVal powerLevel As Integer)
    Me._powerLevel = powerLevel
  End Sub

  Protected Overrides Sub Finalize()
    'Destructor code to free unmanaged resources
    MyBase.Finalize()
  End Sub
End Class
```

Using Objects

C#

```
SuperHero hero = new SuperHero();






// No "With" construct
hero.Name = "SpamMan";
hero.PowerLevel = 3;
```

Clarion#

```
Hero  &SuperHero
Hero2 &SuperHero
Obj   &Object
  CODE
  Hero &= NEW SuperHero

  ! No "With" construct
  hero.Name = 'SpamMan'
  hero.PowerLevel = 3
```

VB.NET

```
Dim hero As SuperHero = New SuperHero
'or
Dim hero As New SuperHero


With hero
  .Name = "SpamMan"
  .PowerLevel = 3
End With
```

```csharp
hero.Defend("Laura Jones");
SuperHero.Rest();                //Calling static method


SuperHero hero2 = hero;          //Both reference the same object
hero2.Name = "WormWoman";
Console.WriteLine(hero.Name);    //Prints WormWoman

hero = null ;                    //Free the object

if (hero == null)
  hero = new SuperHero();


Object obj = new SuperHero();
if (obj is SuperHero)
  Console.WriteLine("Is a SuperHero object.");


// Mark object for quick disposal
using (StreamReader reader = File.OpenText("test.txt")) {
  string line;
  while ((line = reader.ReadLine()) != null)
    Console.WriteLine(line);
}
```

```clarion
Hero.Defend('Laura Jones')
SuperHero.Rest()               !Calling static method


Hero2 &= Hero                  !Both reference the same object
Hero2.Name = 'WormWoman'
Console.WriteLine(Hero.Name)   !Prints "WormWoman"

Hero &= NULL                   !Free the object

IF Hero &= NULL
  Hero &= NEW SuperHero
END

Obj &= NEW SuperHero();
IF Obj IS SuperHero
  Console.WriteLine('Is a SuperHero object.')
END

! No equivalent to USING(Resource) to mark for quick disposal
```

```vbnet
hero.Defend("Laura Jones")
hero.Rest()    'Calling Shared method
'or
SuperHero.Rest()

Dim hero2 As SuperHero = hero   'Both reference the same object
hero2.Name = "WormWoman"
Console.WriteLine(hero.Name)    'Prints WormWoman

hero = Nothing                  'Free the object

If hero Is Nothing Then _
  hero = New SuperHero


Dim obj As Object = New SuperHero
If TypeOf obj Is SuperHero Then _
  Console.WriteLine("Is a SuperHero object.")


' Mark object for quick disposal
Using reader As StreamReader = File.OpenText("test.txt")
  Dim line As String = reader.ReadLine()
  While Not line Is Nothing
    Console.WriteLine(line)
    line = reader.ReadLine()
  End While
End Using
```

Structs

C#

```csharp
struct StudentRecord {
  public string name;
  public float gpa;

  public StudentRecord(string name, float gpa) {
    this.name = name;
    this.gpa = gpa;
  }
}


StudentRecord stu = new StudentRecord("Bob", 3.5f);
StudentRecord stu2 = stu;

stu2.name = "Sue";
Console.WriteLine(stu.name);     //Prints Bob
Console.WriteLine(stu2.name);    //Prints Sue
```

Clarion#

```clarion
StudentRecord STRUCT
Name            &STRING,PUBLIC
GPA             SREAL,PUBLIC
Construct       PROCEDURE(STRING Name,SREAL GPA),PUBLIC
                  INLINE
                    CODE
                    SELF.Name = Name
                    SELF.GPA = GPA
                  END
              END

Stu  StudentRecord('Bob', 3.5)
Stu2 StudentRecord
     CODE
     Stu2.Name = 'Sue'
     Console.WriteLine(Stu.Name)    !Prints Bob
     Console.WriteLine(Stu2.Name)   !Prints Sue
```

VB.NET

```vbnet
Structure StudentRecord
  Public name As String
  Public gpa As Single

  Public Sub New(ByVal name As String, ByVal gpa As Single)
    Me.name = name
    Me.gpa = gpa
  End Sub
End Structure


Dim stu As StudentRecord = New StudentRecord("Bob", 3.5)
Dim stu2 As StudentRecord = stu

stu2.name = "Sue"
Console.WriteLine(stu.name)     'Prints Bob
Console.WriteLine(stu2.name)    'Prints Sue
```

Properties

C#

```
private int _size;

public int Size {
  get {
    return _size;
  }
  set {               //parameter is always "value"
    if (value < 0)
      _size = 0;
    else
      _size = value;
  }
}

// Usage
foo.Size++;
```

Clarion#

```
_Size UNSIGNED,PRIVATE
Size  PROPERTY,UNSIGNED,PUBLIC
        INLINE
        GETTER
          CODE
          RETURN SELF._Size
        SETTER    !parameter is always "Value"
          CODE
          IF Value < 0
            SELF._Size = 0
          ELSE
            SELF._Size = Value
          END
        END

! Rather than use INLINE, you may define Get_PropertyName
! and Set_PropertyName methods separately.
ClassName.Get_Size PROCEDURE
  CODE
  RETURN SELF._Size
ClassName.Set_Size PROCEDURE(UNSIGNED pValue)
  CODE
  SELF._Size = CHOOSE(pValue < 0, 0, pValue)

! Usage
  Foo.Size += 1
```

VB.NET

```
Private _size As Integer

Public Property Size() As Integer
  Get
    Return _size
  End Get
  Set (ByVal Value As Integer)
    If Value < 0 Then
      _size = 0
    Else
      _size = Value
    End If
  End Set
End Property

' Usage
foo.Size += 1
```

Delegates / Events

C#

```
delegate void MsgArrivedEventHandler(string message);


event MsgArrivedEventHandler MsgArrivedEvent;

// Events must use explicitly-defined delegates in C#


MsgArrivedEvent += new
  MsgArrivedEventHandler(My_MsgArrivedEventCallback);
MsgArrivedEvent("Test message");  //Throws exception if obj is null
MsgArrivedEvent -= new
  MsgArrivedEventHandler(My_MsgArrivedEventCallback);
```

Clarion#

```
  MAP
MsgArrivedEventHandler PROCEDURE(STRING Message),DELEGATE
  END

MsgArrivedEvent EVENT,MsgArrivedEventHandler

! Events must use explicitly-defined delegates in Clarion#


  MsgArrivedEvent += My_MsgArrivedEventCallback

  MsgArrivedEvent('Test message')
  MsgArrivedEvent -= My_MsgArrivedEventCallback
```

VB.NET

```
Delegate Sub MsgArrivedEventHandler(ByVal message As String)


Event MsgArrivedEvent As MsgArrivedEventHandler

' or to define an event which declares a delegate implicitly
Event MsgArrivedEvent(ByVal message As String)

AddHandler MsgArrivedEvent, AddressOf My_MsgArrivedCallback
' Won't throw an exception if obj is Nothing
RaiseEvent MsgArrivedEvent("Test message")
RemoveHandler MsgArrivedEvent, AddressOf My_MsgArrivedCallback
```

```
using System;
using System.Windows.Forms;

public class MyClass {
  Button MyButton;



  public void Init() {
    MyButton = new Button();
    MyButton.Click += new EventHandler(MyButton_Click);
  }

  private void MyButton_Click(object sender, EventArgs e) {
    MessageBox.Show("Button was clicked", "Info",
        MessageBoxButtons.OK, MessageBoxIcon.Information);
  }
}
```

```
  USING(System)
  USING(System.Windows.Forms)

MyForm         CLASS,TYPE,NETCLASS,PARTIAL
MyButton        &Button
Init            PROCEDURE
MyButton_Click  PROCEDURE(Object Sender, EventArgs E)
               END

MyForm.Init PROCEDURE
  CODE
  MyButton &= NEW Button
  SELF.MyButton.Click += SELF.MyButton_Click

MyForm.MyButton_Click PROCEDURE(Object Sender, EventArgs E)
  CODE
  MessageBox.Show('Button was clicked', 'Info', |
      MessageBoxButtons.OK, MessageBoxIcon.Information)
```

```
Imports System
Imports System.Windows.Forms

Public Class MyForm
  Dim WithEvents MyButton As Button



  Public Sub Init
    MyButton = New Button
  End Sub


  Private Sub MyButton_Click(ByVal sender As Object, _
      ByVal e As EventArgs) Handles MyButton.Click
    MessageBox.Show(Me, "Button was clicked", "Info", _
        MessageBoxButtons.OK, MessageBoxIcon.Information)
  End Sub
End Class
```

Console I/O

C#

```
Console.Write("What's your name? ");
string name = Console.ReadLine();
Console.Write("How old are you? ");
int age = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("{0} is {1} years old.", name, age);
// or
Console.WriteLine(name + " is " + age + " years old.");


int c = Console.Read();   //Read single char
Console.WriteLine(c);     //Prints 65 if user enters "A"
```

Clarion#

```
Name  &STRING
Age   UNSIGNED
C     UNSIGNED
  CODE
  Console.Write('What's your name? ')
  Name = Console.ReadLine()
  Console.Write('How old are you? ')
  Age = Convert.ToInt32(Console.ReadLine())
  Console.WriteLine("{{0} is {{1} years old.", Name, Age);
  ! or
  Console.WriteLine(Name + 'is '+ age + 'years old.');


  C = Console.Read()        !Read single char
  Console.WriteLine(C)      !Prints 65 if user enters "A"    0
```

VB.NET

```
Console.Write("What's your name? ")
Dim name As String = Console.ReadLine()
Console.Write("How old are you? ")
Dim age As Integer = Val(Console.ReadLine())
Console.WriteLine("{0} is {1} years old.", name, age)
' or
Console.WriteLine(name & " is " & age & " years old.")

Dim c As Integer
c = Console.Read()        'Read single char
Console.WriteLine(c)      'Prints 65 if user enters "A"
```

File I/O

C#

```
using System.IO;

// Write out to text file
StreamWriter writer = File.CreateText("c:\\myfile.txt");
writer.WriteLine("Out to file.");
writer.Close();
```

Clarion#

```
  USING(System.IO)

! Write out to text file
Writer &StreamWriter
  CODE
  Writer &= File.CreateText('c:\myfile.txt')
  Writer.WriteLine('Out to file.')
  Writer.Close()
```

VB.NET

```
Imports System.IO

' Write out to text file
Dim writer As StreamWriter = File.CreateText("c:\myfile.txt")
writer.WriteLine("Out to file.")
writer.Close()
```

```
// Read all lines from text file
StreamReader reader = File.OpenText("c:\\myfile.txt");
string line = reader.ReadLine();
while (line != null) {
  Console.WriteLine(line);
  line = reader.ReadLine();
}
reader.Close();
```

```
! Read all lines from text file
Reader &StreamReader
Line   &STRING
  CODE
  Reader &= File.OpenText('c:\myfile.txt')
  Line = Reader.ReadLine()
  LOOP WHILE NOT Line &= NULL
    Console.WriteLine(Line)
    Line = Reader.ReadLine()
  END
  Reader.Close()
```

```
' Read all lines from text file
Dim reader As StreamReader = File.OpenText("c:\myfile.txt")
Dim line As String = reader.ReadLine()
While Not line Is Nothing
  Console.WriteLine(line)
  line = reader.ReadLine()
End While
reader.Close()
```

```
// Write out to binary file
string str = "Text data";
int num = 123;
BinaryWriter binWriter =
   new BinaryWriter(File.OpenWrite("c:\\myfile.dat"));
binWriter.Write(str);
binWriter.Write(num);
binWriter.Close();
```

```
! Write out to binary file
Str       &STRING
Num       SIGNED(123)
BinWriter &BinaryWriter
  CODE
  Str = 'Text data'
  BinWriter &= NEW BinaryWriter(File.OpenWrite('c:\myfile.dat'))
  BinWriter.Write(Str)
  BinWriter.Write(Num)
  BinWriter.Close()
```

```
' Write out to binary file
Dim str As String = "Text data"
Dim num As Integer = 123
Dim binWriter As New BinaryWriter(File.OpenWrite("c:\myfile.dat"))
binWriter.Write(str)
binWriter.Write(num)
binWriter.Close()
```

```
// Read from binary file
BinaryReader binReader =
   new BinaryReader(File.OpenRead("c:\\myfile.dat"));
str = binReader.ReadString();
num = binReader.ReadInt32();
binReader.Close();
```

```
! Read from binary file
BinReader &BinaryReader
  CODE
  BinReader &= NEW BinaryReader(File.OpenRead('c:\myfile.dat'))
  Str = BinReader.ReadString()
  Num = BinReader.ReadInt32()
  BinReader.Close()
```

```
' Read from binary file
Dim binReader As New BinaryReader(File.OpenRead("c:\myfile.dat"))
str = binReader.ReadString()
num = binReader.ReadInt32()
binReader.Close()
```

Clarion Magazine

Should Clarion# Drop Automatic Instantiation?

by Dave Harms

Published 2007-12-13

Earlier I published an article titled Understanding Clarion# Strings, which I concluded with a bit of code demonstrating the difference between how Clarion# declares Clarion.ClaString variables and System.String variables, and I suggested that those differences would lead to some interesting questions (and answers) on how Clarion declares classes. If you haven't read that article you might want to do so now.

In this article I'll argue that Clarion# would be a clearer, easier-to-use language if it did not permit the automatic instantiation of classes, which is a standard feature of the Clarion language. I'll also argue that the costs of this change are less than the costs of continuing to permit automatic instantiation in Clarion#.

Here's the snippet of code from the Strings article:

```
s       ClaString(20)
t       &String

      CODE
      s = 'abcdef'
      t = s
      s = sub(s,1,3)
      t = t.Substring(1,3)
```

The important point about this code is that while String is declared with a leading &, indicating it is a reference, ClaString is declared without the &. So what's the big deal about that? One is a reference, and one isn't right?

As it turns out, there is a problem, not with the declaration per se, but with the code. Look at the source listing and you won't see a NEW statement for &String.

That seems odd. Isn't NEW required for all reference variables? In Clarion, yes. In Clarion#, it isn't always the case.

If I add a & to ClaString, as in the following code, I will in fact need to NEW the &ClaString before I can use it:

```
s       &ClaString(20)
t       &String
  CODE
  s &= new ClaString(20)
  s = 'abcdef'
  t = s
  s = sub(s,1,3)
  t = t.Substring(1,3)
```

So the rule seems to be that ClaString reference variables have to be NEWed, but .NET String reference variables do *not* have to be NEWed. And that's a surprising inconsistency in a language valued for its readability.

In fact, if you look at some of the Clarion# sample applications, you'll quickly realize that many if not all .NET objects are declared with the & syntax, while some Clarion objects are declared with & and some are not. Although the String example is particularly confusing, the whole concept of sometimes needing & and sometimes not needing it is a bit odd when you compare Clarion# code to other .NET languages. And the reason for the oddness and the confusion comes down to one important concept: automatic instantiation.

Automatic instantiation

If you have a bit of experience with object-oriented programming, you're probably familiar with the formal distinction between a *class* and an *object*. In general, the term class refers to the type (or definition, if you prefer) of an object; there can be one or more *instances* of the class, and each of those instances is an object. So class=definition, and instance=object.

In Clarion (and, at least for now, in Clarion#) you have several ways of declaring a CLASS. Here's what I think of as a "normal" class, which is to say it's a type, and is not allocated any memory:

```
NameClass   CLASS,TYPE
Name            String(20),Private
SetName     Procedure(String newName)
GetName     Procedure,String
            END



NameClass.SetName       Procedure(String newName)
  code
  self.Name = NewName



NameClass.GetName       Procedure
  code
  return self.Name
```

Note that the class definition begins with CLASS and ends with END. Following the class you have the source code for the methods (although these can also be in a separate source file).

The TYPE attribute tells the compiler *not* to allocate memory to this class. To create an instance of the class you need to do something like this in your code:

```
MyClass &NameClass
     code
     MyClass &= new (NameClass)
```

MyClass is declared as a reference variable; that is, it's a null value until the NEW operator allocates memory for an instance of NameClass, and assigns a reference to that object to MyClass.

However, you can also declare NameClass without the TYPE attribute:

```
NameClass   CLASS
!... properties and methods
             END
! methods here
```

In this case you don't need to declare a MyClass reference variable; you can just go ahead and use NameClass directly (provided it's in scope, of course):

```
code
NameClass.SetName('Dave')
```

This idea of declaring a class and automatically instantiating it is, for the most part, foreign to .NET. (Actually there is such a thing as a *static class*, but that's a slightly different animal and beyond the scope of this article.) The norm, in .NET, is that all classes are implicitly TYPEd, and you have to create a NEW instance whenever you want to use a class. Special exceptions are made for some data types, however; you don't need to NEW String objects, you just assign a value to them and the String's constructor is called automatically, passing in the value.

Automatic instantiation has the potential to cause much confusion for Clarion# developers because the *default* in Clarion is automatic instantiation. You have give the CLASS the TYPE attribute to make it a type, and you have to modify a variable with & to make it a reference variable. By and large, in the world of OOP, things are the other way around. TYPEd classes and reference variables are the default, and you have to mark something as *static* if you want it available without explicit object creation. These conflicting standards can make mixing Clarion objects and .NET objects just a little confusing.

But if automatic instantiation isn't the norm for object-oriented languages, why is it the norm for Clarion?

The value of automatic instantiation

Automatic instantiation in Clarion (not Clarion#) has a very important value: it makes memory management easier. Up until Clarion grew OO extensions, it was just about impossible to create a memory leak in a Clarion application. Oh, you could forget to empty a queue, but really there weren't any nasty memory-related problems waiting to happen because Clarion programmers didn't explicitly allocate memory.

And then came OOP, and with OOP came those two ways to declare classes. Conveniently, the default implementation (without the TYPE attribute) meant Clarion programmers could continue to code just as before, without any concern for cleaning up memory. As soon as a CLASS (declared without TYPE) went out of scope the runtime library freed up the associated memory. But to allow the full flexibility afforded by object-oriented programming, the compiler had to permit the runtime creation of objects. That meant using NEW to allocate memory, and if you allocated memory with NEW you had to free the memory with DISPOSE, or your application would have a memory leak.

Is automatic instantiation still needed?

Making CLASSes default to automatic instantiation no doubt saved many of us from a host of memory leaks. In Clarion#, however, these kinds of memory leaks are no longer a concern. That's because the .NET platform employs something called automatic garbage collection.

Clarion developers have enjoyed automatic garbage collection from the beginning. For the longest time we didn't (couldn't!) allocate memory - the runtime did that for us - but we also never had to clean up memory. .NET takes that a step further; you can NEW objects to your heart's content, and when they are no longer referenced by any active code, the .NET garbage collector swoops in and scoops them up.

That means that the primary reason for automatically instantiating Clarion CLASSes is no longer valid in .NET.

Reference variables

A byproduct of Clarion's automatic instantiation is the concept of a reference variable. Clarion makes a distinction between a CLASS, a TYPEd CLASS, and a reference variable. A reference variable is a variable that starts out as a null value and is assigned a reference to an object of its type.

In most of the OOP world there are only classes and objects; what we think of as a CLASS (without the TYPE attribute) and a reference variable are really one and the same. In .NET object variables are references to object instances, or references to NULL. In Clarion, CLASSes without the TYPE attribute are a special entity; they're like TYPEd classes except they are automatically instantiated, and you can't use them as reference variables.

The point of confusion

Clearly if you write only Clarion# code, and particularly if you're familiar with Clarion syntax, none of the above need concern you. It's really only when you start mixing .NET objects and Clarion# objects that you need to remember to always decorate the .NET declarations with & and instantiate those objects with NEW or assign those references to existing objects. (System.String, of course, gets special treatment and doesn't need to be explicitly NEWed.) And if you're porting, say, some C# code to Clarion, and your Clarion version uses some automatically instantiated objects, you'll have to remember that the undecorated class in C# is the opposite of the undecorated (with TYPE or &) Clarion class.

A modest proposal

Obviously it would be nice not to have this conflicting style of declarations between Clarion# and .NET. If Clarion# were more closely aligned to .NET it would make it easier to translate code between Clarion# and other .NET languages, and it would actually simplify Clarion#'s syntax. But what would that syntax look like?

This subject has been discussed extensively in the dotnet.general newsgroup (restricted to .NET beta participants, at least for now), and in particular Mike Hanson and Dennis Evans made some excellent points. I'm doing my best to summarize, and hopefully Mike and Dennis will excuse me if I missed a point or got something wrong.

If Clarion# were to abandon automatic instantiation, the TYPE attribute would become implicit, as would the & prefix on variables. In other words, all Clarion# classes would be TYPEs, and you would have to manually instantiate a reference variable before you could use the class (strings, like System.String, being a notable exception). The above NameClass example would look like this:

```
NameClass   CLASS
!... properties and methods
            END
! methods here
```

To use NameClass you would need to declare an instance (reference) variable like this:

```
MyClass    NameClass
```

and in code, create an instance like this:

```
MyClass = new NameClass
```

Although you could declare a reference variable with & (as in &NameClass), in general the & would be ignored by the compiler. Mike expressed a preference for retaining reference variables for simple data types. (I think it's worth pointing out that .NET languages like C# do allow you to have references to simple data types, which results in an implicit conversion from simple data type to object - search for the terms boxing and unboxing for more information.)

Mike's reference variables for simple data types notwithstanding, removing & as a meaningful term would mean that any class variable is a reference variable, and could be either instantiated as a new object or assigned to an existing object. That would get rid of the special status of automatically instantiated CLASSes, which cannot be used as reference variables at present, and make Clarion#'s object handling syntax simpler and more .NET-like.

Other uses of TYPE

The TYPE attribute is also used in procedure prototyping and deriving structures such as GROUPs and QUEUEs. In .NET all complex data types are classes, so I would expect that an implicit TYPE would essentially mean no code would have to change, except for addition of code to NEW as needed.

The downside

Removing automatic instantiation would obviously break any code that relied on non-TYPEd class declarations. It also raises some questions about the = operator, which currently is used both to assign values and to test for equality. Asking it to assign references as well might cause problems.

Aside from these issues, for the most part the solution would be quite simple: just add an instance variable and NEW it. And based on what I've seen so far of Clarion#, I think it's likely that most developers will choose to write new code in any case, rather than attempt to port existing APP files, even if a migration tool is available. Although Clarion and Clarion# are both Windows languages, the differences between the WinAPI and the .NET environments are significant enough that the change is not unlike moving from DOS to Windows. And very few developers chose porting to Windows over rewriting for Windows.

Oh, and one more thing...

There's one other platform compatibility issue that's raised its head lately, and that's array indices. Clarion has always had 1-based arrays; that is, the first element in any array has an index of 1, followed by 2, etc. In .NET, however, arrays are zero-based. If you're mixing Clarion# and .NET objects, you can have a situation where sometimes you need to be zero-based, and other times one-based. Here's an example with System.String:

```
a    &String[]
  CODE
  a &= new String[5]
  a[1] = 'abcdefg'
  System.Console.WriteLine(a[1].Substring(1,3))
```

The variable a is an array of System.String objects, and because that array was created in Clarion it is 1-based, and the first string in the array is set to 'abcdefg'. But when you run this code the output is

```
bcd
```

which is probably not what you expect. That's because System.String is a .NET object, and the Substring method expects a zero-based argument, not a 1-based argument. You have to use this code:

```
  System.Console.WriteLine(a[1].Substring(0,3))
```

to get the first three characters. You'll encounter similar problems when dealing with arrays of objects created by .NET classes.

Some Visual Basic users have had to deal with this issue, since VB6 allows you to create non-zero-based arrays. The solution, naturally enough, was a custom array datatype for VB.NET that lets you specify a different base for arrays. And since Clarion# already supports 1-based arrays, there's obviously no problem achieving this feature. The question, as with instantiation, is what should be the default?

The arguments

Here are the main points in the argument in favor of leaving Clarion#'s automatic instantiation in place:

1. Familiarity - Clarion developers can continue to use objects in the way to which they've become accustomed
2. Portability - It will be easier to move existing Clarion code to Clarion#

Here are the main points in favor of getting rid of Clarion's automatic instantiation:

1. Clarity - having two ways to instantiate objects is more confusing than having one way (in this respect, getting rid of automatic instantiation is truer to the spirit of Clarion).
2. Alignment with .NET - it's easier to port C#/VB.NET examples to Clarion# without the &.
3. Ease of implementation - the compiler could probably treat & as a valid, but ignored, part of the declaration; this way existing variable declarations using & would not be broken.
4. Less disruption - arguable, but I think having to add NEW where needed is less trouble than untangling the confusion caused when the undecorated (no &) style means something completely different in Clarion# than in other .NET languages
5. Automatic instantiation had the great benefit of not requiring the developer to DISPOSE; under most circumstances, DISPOSE is now automatic in .NET so the programmer no longer needs to be protected this way.
6. Marketability - any significant difference in how classes are declared is going to negatively affect the marketability of Clarion# to non-Clarion developers.
7. And the best argument, I think: The users most affected by the loss of automatic instantiation are developers who already know how to write classes, and they can cope. The Clarion developers who are trying to learn .NET OOP are the ones who are most likely to be tripped up by this difference.

Clearly I'm biased in favor of dropping automatic instantiation and making Clarion# more closely aligned with the .NET platform. Your view may be different.

Take the survey

When I first ran these ideas past Bob Zaunere, he suggested it would make a good poll for Clarion Magazine, and I agreed. For the first two weeks after this article goes live, you can go to the Clarion Magazine home page and cast your vote for or against dropping automatic instantiation. I don't know whether SoftVelocity will change object instantiation in Clarion# if there's a strong vote in favor, but I believe if a large number of developers are comfortable with dropping automatic instantiation, that change is certainly more likely. Depending on the results of that poll I may run another one on zero-based arrays.

David Harms is an independent software developer and the editor and publisher of Clarion Magazine. He is also co-author with Ross Santos of *Developing Clarion for Windows Applications*, published by SAMS (1995), and has written or co-written several Java books. David is a member of the American Society of Journalists and Authors (ASJA).

Reader Comments

Posted on Thursday, December 13, 2007 by Garry Anderson

What about using the ,AUTO attribute to not auto instantiate?

Posted on Thursday, December 13, 2007 by Dave Harms

Garry,

That did occur to me as well. Unfortunately, I think there are two strikes against using ,AUTO

1. It means you have to add an attribute to achieve non-instantiation, so it's still out of sync with the .NET default, and

2. The term AUTO, to the uneducated coder, implies automatic instantiation, not a lack of automatic instantiation. So not only is it an additional step, it communicates exactly the opposite of its meaning.

Dave

Posted on Thursday, December 13, 2007 by Stephen Ryan

Technicaly your right, breaks a lots of code that could be ported or dual compiled. Dual compiling will require special tools. So id go so far as to say that porting would then be a waste of time. That leaves mix and match. I now favour mix and match since its easier to leave code in win32 and call it from dot net if you need to.

clarion 7 and 8 should provide look and feel that means code in dot net and win 32 does not look to dif.

so forget porting

Posted on Thursday, December 13, 2007 by John Dunn

These Clarion.NET and Clarion# articles are great! I've really enjoyed them. My vote would be to drop automatic instantiation and provide for a mechanism to specify zero based arrays.

John

Posted on Thursday, December 13, 2007 by Douglas E. Johnson

While probably not a popular idea, what about forcing code to be explicit and clear? In other words, a declaration MUST have an attribute to indicate automatic instantiation or not. If the darned AUTO attribute was not contrary to itself, IMO it would be an easy choice between TYPE and AUTO, otherwise it's TYPE or INSTANT.<g>

Posted on Thursday, December 13, 2007 by Angel L. Bermudez, Jr.

Dave,

I agree. Clarion# should align with other .Net languages. SV should get rid Clarion's automatic instantiation and if necessary use AUTO to signify automatic instantiation.

Posted on Friday, December 14, 2007 by Garry Anderson

Hi Dave,

Yeah, I thought after I had posted that using AUTO wasn't the smartest idea

ok, so here's another one...

If you declare a class without the & then auto instantiate.
If the class is declared with the & then don't...
that way the behavior is closer to the way Clarion currently works and also closer to .NET

Posted on Friday, December 14, 2007 by Dave Harms

Steve,

I don't think dual compile has much of a future - .NET and the WinAPI are just too different. But I can definitely see some libraries being ported. And ClaString is immensely important for anyone who has specialized string handling code.

Dave

Posted on Friday, December 14, 2007 by Dave Harms

Douglas,

That's an interesting suggestion - it would definitely make things clear. The only downside is it would seem odd to have to specify a non-instantiated state, when compared to that being the default in other languages.

Maybe a CREATE attribute?<g>

Dave

Posted on Friday, December 14, 2007 by Dave Harms

Garry,

> If you declare a class without the & then auto instantiate.
> If the class is declared with the & then don't...

That's how it works now, for the most part. If Clarion# was a language unto itself, that would be fine. The problem is that the current default in Clarion# (declaring an object with the class name only, without the &) is instantiated, which is the opposite of how it's done in .NET where the default state (class name only) is an uninstantiated reference variable.

Dave

Posted on Friday, December 14, 2007 by Douglas E Johnson

>> Maybe a CREATE attribute?<g>

That's Clarion enough for me.

The discussion does make one wonder how AUTO came about. Any DAB's around here?

Posted on Monday, December 17, 2007 by Bjarne Havnen

I must agree with the overall point of this article. The confusion on when to use a reference and/or NEW is what has made me make most errors in my first project. Clarion should line up with .NET in this matter

Add a comment

Clarion Magazine

The Clarion.NET FAQ

by Dave Harms

Published 2007-11-17

In this Frequently Asked Questions (FAQ) page I'll attempt to answer at least some of the questions that have been raised about Clarion.NET. If you log in you can post your own comments and questions below.

Terminology!

One of the problems in discussing Clarion.NET is finding a meaningful term for traditional (non-.NET) Clarion applications. Some developers use the term *Win32*, but that isn't always helpful since .NET applications can also be Win32 (or Win64).

As noted below, one of the key differences between the Clarion we now know and Clarion.NET is that the former is ultimately built on top of the Windows API. For that reason I will refer to "traditional" Clarion Windows applications as WinAPI apps, to differentiate them from .NET apps.

What did you just say?

I said that when I use the term *WinAPI app* I'm referring to a traditional Clarion (Windows) application.

But I don't use the Windows API in my applications.

You might not, but without the Windows API there would be no Clarion (for Windows) as we know it. All your apps use it all the time.

Is Clarion.NET available now?

Yes, the first beta was released to subscription program participants on Saturday, Nov 17, 2007.

What is Clarion.NET?

Clarion.NET is a version of the Clarion development environment specifically designed for Microsoft's .NET platform.

Okay, what is .NET?

From Wikipedia:

> The Microsoft .NET Framework is a software component that can be added to or included in the Microsoft Windows operating system. It provides a large body of pre-coded solutions to common program requirements, and manages the execution of programs written specifically for the framework. The .NET Framework is a key Microsoft offering, and is intended to be used by most new applications created for the Windows platform.

I'll just add (for now) that .NET is an object-oriented framework which accommodates a great variety of programming languages.

Isn't .NET all about web development?

You might think so, given the name of the platform. While .NET contains many web/Internet-related classes, it also has extensive support for desktop development. As well, there's a version of the framework for mobile devices.

.NET is a comprehensive development and runtime environment suitable for most kinds of programming, but not usually for low level hardware-oriented stuff (although you *can* get an 80386 assembler that generates .NET code).

What's the difference between Clarion.NET applications and the Clarion WinAPI applications I write now?

There are a number of differences. One of the biggest is that the Clarion language (for Windows) is highly dependent on the Windows API. For instance, when you open a window, that window is created by the Clarion runtime library via calls to the Windows API. When you create a new variable or class instance, memory is allocated via the Windows API. This is procedure-oriented coding, and procedure-oriented applications (and operating systems) tend to be hardwired - they have only set ways of doing things. For instance, the Clarion window formatter only has a specific set of controls available, and you can't add your own custom widgets. You can use add-on components like ActiveX controls, but as anyone who's tried it in Clarion can tell you, it isn't all that easy either.

In a WinAPI application you often have to work hard to make different functional units of code work together because the Windows API isn't really geared to the concept of components.

Applications written for .NET, however, use the extensive .NET Framework class library rather than the Windows API. In .NET it's all about objects. You can assemble an application out of different classes and components much more easily because .NET provides a congenial framework for all these things to work together.

Is the .NET Framework that much better than the Windows API?

Yes, in almost every way. That's not toadying to Microsoft - it's just the evolution of programming. The Windows API is disorganized - you need to know what you're looking for or you'll get lost very quickly. The .NET Framework library is organized into namespaces. For instance, security classes are grouped together, as are diagnostic classes, data access classes, etc.

The .NET framework library is much more massive than the API and is growing larger. It provides a whole lot of code that you'd otherwise have to write yourself. There are many other benefits to the library but many of these are better described in the context of the Common Language Runtime (CLR).

Okay, I'll bite. What's the Common Language Runtime?

The CLR is an essential part of .NET. It's the layer that sits between you and the hardware, and which provides important services to your applications including, but not limited to: memory management; thread management; exception handling; garbage collection; security; introspection and reflection. The CLR means many tasks are easier in .NET. For instance, if you NEW something you don't (usually) have to DISPOSE it - the CLR's garbage collector will detect when the object is no longer in use and will clean it up automatically.

Are there any other weird acronyms I should know?

Well, there's CIL, the Common Intermediate Language (often just called IL). The CLR doesn't actually run the code you write; instead, all .NET language compilers translate their code into this CIL (IL) bytecode, and that's what the CLR executes.

So .NET is a giant interpreter? This seems like a step back from true compiled languages - didn't CPD generate pseudocode which had to be interpreted?

You're right - CPD did in fact generate pseudocode, and it's a fair analogy. The advantage of IL is that since all .NET languages compile to an intermediate language, they all share a common platform. You can take some classes compiled in, say, Fortran.NET (no, I'm not kidding) and easily use them in C# or Boo or Clarion.NET.

What is Clarion#?

Clarion# is the .NET version of the Clarion programming *language*. The Clarion IDE for .NET is, at least at present, called Clarion.NET. But a lot of developers already use the terms Clarion# and Clarion.NET interchangeably.

Clarion is both a procedural and an object-oriented language, but .NET is an OO framework - can I still write procedure code in Clarion.NET?

Yes, you can still write procedural code in Clarion.NET. Your procedures are converted to object-oriented code when they're compiled to IL code, but you don't need to know or care about that if procedural code is what makes you happy.

Are Clarion 7 and Clarion.NET the same product?

No, they are separate products, but they share the same integrated development environment (IDE).

Does that mean if I buy Clarion 7 and Clarion.NET I'll end up with just one installed IDE able to work with both platforms?

Most likely, although there may be some versioning issues that make that more difficult, at least during the beta process. So for a time you may have two IDEs installed. I really don't know.

What can I do with Clarion.NET/Clarion# beta in the first release?

The beta includes a template wizard you can use to generate a .NET application, but since those templates run in C6, not the new IDE (AppGen isn't ready yet), you can't yet create and maintain a .NET APP file in the same way you do in C6. No word yet on when the AppGen will be ready, but meanwhile you can hand code not just desktop applications, but also ASP.NET and mobile applications.

There are a number of example solutions shipped with the beta. These include demonstrations of connecting to a database with ADO.NET, displaying a BLOB image onscreen, a mobile app, data binding, drag and drop, a FOREACH with QUEUE example, glass buttons, new listbox features, the SCHOOL application, mixing Clarion# and C#, a simple web service, a "Hello world" ASP.NET web application, a .NET remoting example, the PEOPLE app with data grid and browse procedures, and a screen capture utility.

Since Clarion# is a full .NET producer/consumer language, you will have full access to all of the .NET framework library as well as the rich supply of third party .NET tools.

What will I be able to do with Clarion.NET/Clarion# in the long run?

When complete Clarion.NET will include templates to generate desktop (WinForms) applications, web (ASP.NET) applications, and mobile (Compact Framework) applications.

What is the Compact Framework?

The Compact Framework is .NET for mobile devices, and includes about 30% of the full framework plus classes specific to mobile devices, and takes up about 1/10 of the space, mostly due to file compression.

What is ASP.NET?

ASP.NET is Microsoft's Active Server Pages web application framework for .NET. You create ASP.NET pages using a development approach similar to desktop development: you place controls on a page, and write code for those controls; ASP.NET then renders the pages accordingly and executes the code on the server side when data comes back from the browser.

How hard is it to write .NET applications?

Writing .NET applications can be both easier and harder than writing WinAPI apps, and the comparison between the two brings to mind the differences between DOS and Windows API development. You could argue that Windows development was a lot harder because you had to do so much more to create even a simple Hello World application (at least in C, if not in Clarion). On the other hand, Windows provided standard capabilities like a windowing library; in DOS you had to write or otherwise obtain a windowing library to achieve a consistent, accessible user interface. In DOS you had to know how to talk to every printer you wanted to use; in Windows you talked to the printer driver, and the printer driver sorted out the back end. Similarly .NET does a lot of the heavy lifting you now have to code in WinAPI apps, leaving newer and more complex tasks to your wily programming brain.

One big difference between WinAPI and .NET is that the latter is exclusively object-oriented. To get the most out of Clarion.NET apps you will definitely want some basic OO programming knowledge. With that knowledge in hand, I think you'll find .NET easier and safer to use than the Windows API. If you've had to deal with ActiveX or (heaven help you) directly call COM functions you'll truly find .NET an easier place to code.

.NET introduces some new language concepts that may take some getting used to, such as delegates, which are a sort of type-safe object-oriented callback mechanism.

Is .NET open source?

.NET is not open source, but Microsoft is in the process of releasing source code for some parts of the framework.

Will my third party tools work in .NET?

The majority of third party tools will need to be ported to Clarion.NET. Templates that don't generate any source code and do not depend on a particular template chain are likely to work without modification, but there aren't many that fall into that category. You're probably best off assuming you'll need new versions until you hear otherwise from the vendor.

I've often said that .NET is a two-edged sword for third party vendors. If what you provide is readily available as part of the .NET framework, then zing!! off goes your head as you step into .NET land. On the other hand, if you have a great product and you can port it to .NET, you're ready to carve a swath through a programming market that numbers in the millions of coders.

Will my customers want .NET and if so why?

Some customers will want .NET just because it's a buzzword. That's an easy sale.

Other customers may or may not know whether they want .NET. Clearly what they want is software that does what they need it to. If their needs run to eye candy or very unique user interfaces, then .NET presents some distinct advantages. There are a bazillion custom controls out there for .NET, all of which you can use with Clarion.NET. And there are many class libraries that handle important behind-the-scenes tasks as well.

I find it difficult to overstate the importance of ready access to all of that existing code. With Clarion WinAPI apps you have to be concerned about prototyping functions, register passing conventions, the arcana of COM, etc. etc. With .NET you just drop in the library and start to use it, no matter what language it's written in.

.NET offers programming benefits as well. For instance, your code compiles to IL code which is run under the watchful eye of the CLR. That means the CLR can detect problems with your code and present far more detailed information to you than you get from, say, a GPF in a WinAPI application. This makes debugging easier and faster.

Can I run .NET apps on Linux or the Mac? What is Mono?

Mono is a Novell-sponsored project to port the .NET platform's functionality to multiple platforms, including Linux, Mac OS X, Solaris, BSD, and Windows. Mono necessarily lags behind Microsoft's efforts, and currently has completed support for .NET 1.1 and mostly-complete support for .NET 2.0.

Versions? What are all these .NET versions?

Microsoft released .NET 1.0 in 2002, and 1.1 in 2003. You may see some computers with 1.1 as the latest version, (and some computers without .NET installed at all) but the standard at present is .NET 2.0, released in 2005. Clarion.NET targets 2.0 apps - there was talk in the early days of 1.1 being supported as well but the only reason I can see for supporting 1.1 is for Mono compatibility, given that Windows Forms 2.0 support is now scheduled for Mono 2.5, which does not have a release target.

For most of us, .NET 2.0 will be just where we want to be.

What about .NET 3.0? Or 3.5?

The first thing to keep in mind about .NET 3.0 is that it is not a replacement for .NET 2.0. It's a bunch of new stuff added to 2.0, including Windows Presentation Foundation, Windows Workflow Foundation, Windows Communication Foundation, and Windows Card Space. The base class library is unchanged from 2.0.

.NET 3.5 uses the same CLR as 2.0 but it adds some new stuff to the base class library, in particular support for the LINQ query language. 2.0 apps will still run fine on 3.5.

Will I need to learn C# or VB.NET?

You will not need to learn C# or VB.NET or any other .NET language, but you may want to. In particular there's a lot of C# source code out there, and you may want to adapt some of it to your own uses. If you can read object-oriented Clarion code you won't have much trouble reading C#.

What is ADO.NET?

ADO.NET is Microsoft's data access layer for .NET, and consists of data providers (i.e. drivers) and DataSets. A DataSet is a set of objects that model the database elements (tables, views, columns, rows, relations, etc.).

Will my Clarion (.clw) programs compile in Clarion#?

While much of the Clarion language is unchanged in Clarion#, it's unlikely that any single WinAPI application will compile as Clarion# code without modification.

What will I have to do to port my apps to .NET?

Porting applications to .NET is a bit of a gray area at the moment. Theoretically it can be done; the question is, is it worth the work? Clarion WinAPI apps are built on a traditional, client-server model. Is this a good approach to take into the .NET world? Do we really want ABC.NET? Perhaps a multi-tier design would be more appropriate, particularly one where you could easily reuse your business logic in desktop, web, and mobile versions of your application. SV has alluded to this kind of design but it isn't clear yet what kind of desktop application templates will be included with Clarion.NET.

Can we mix and match .NET objects with Win32 API objects easily?

You can include WinAPI code in a .NET app and vice versa. Easy is a relative term. See Wade Hatler's series of articles.

Can I get my Clarion 7 and earlier programs to use .Net components I create using Clarion.Net or other .Net languages?

Most likely you could (see the article series above) but I think this would be a stopgap measure at best.

How secure is .NET code? Can it be easily decompiled?

Code security is a legitimate concern in .NET and yes, IL code can be decompiled much more easily than native Windows executable code. .NET obfuscators alter label names and use various code scrambling approaches to make it very difficult for anyone to make sense of the decompiled result. Or you could just hire someone who's a natural at writing unreadable code.

I've heard .NET programs run slower than native code. Will I notice the difference?

.NET uses just-in-time (JIT) compiler technology, meaning that IL code is compiled to native code the first time that IL code is needed by the application. That means there is a small startup penalty but once the code is compiled it runs just like any other native code. Theoretically code produced by a JIT compiler can outperform code issued by a standard compiler because the JIT compiler can tailor the code to the hardware.

Can I include a .NET runtime with my apps so I don't have to require my customers to install .NET?

There is a tool called the Salamander .NET Linker, Native Compiler and Mini-Deployment Tool that will do just this. It's a bit expensive, and I don't know how well it works. Apparently Thinstall will also create self-contained .NET installs.

If both new Clarions deliver desktop applications - why should I buy both? Would Clarion# not be enough?

I'll assume here you mean *after* AppGen is released for C7 and Clarion.NET. While you can create real desktop apps already with Clarion.NET, most developers will want to use AppGen for larger apps.

So yes, you can create desktop apps with both. Why would you still want C7? Here are a few reasons:

- Better productivity with the new IDE
- Ability to work with different versions of Clarion within the same IDE
- C7's runtime improvements including eye candy and Unicode/ClearType support.
- Stability - templates are well established and the runtime is solid
- Potentially smaller installs - no need for the .NET runtime

Can a Clarion# class inherit from a C# class?

Definitely. And vice versa. The same goes for all .NET languages.

What are .NET's minimum requirements?

According to Microsoft, the minimum requirements for the .NET 2.0 redistributable are:

- 400 Mhz processor (800 Mhz recommended)

- 96-128 Mb memory (256 or better recommended)
- 280 Mb hard disk space (610 for 64 bit), 1 gig recommended
- 800x600 256 color, 1024x768 high color recommended

As with most Microsoft platform requirements, you're probably not going to be very happy at the low end of the spectrum. I suggest you take the "recommended" values as the minimum values.

Supported x86-based operating systems:

- Microsoft Windows 98
- Microsoft Windows 98 Second Edition
- Microsoft Windows 2000 Professional with SP4
- Microsoft Windows 2000 Server with SP4
- Microsoft Windows 2000 Advanced Server with SP4
- Microsoft Windows 2000 Datacenter Server with SP4
- Microsoft Windows XP Professional with SP2
- Microsoft Windows XP Home Edition with SP2
- Microsoft Windows XP Media Center Edition 2002 with SP2
- Microsoft Windows XP Media Center Edition 2004 with SP2
- Microsoft Windows XP Media Center Edition 2005
- Microsoft Windows XP Tablet PC Edition with SP2
- Microsoft Windows XP Starter Edition
- Microsoft Windows Millennium Edition
- Microsoft Windows Server 2003 Standard Edition
- Microsoft Windows Server 2003 Enterprise Edition
- Microsoft Windows Server 2003 Datacenter Edition Microsoft Windows Server 2003 Web Edition

x64-bit based systems

- Microsoft Windows XP Professional x64 Edition
- Microsoft Windows Server 2003, Standard x64 Edition
- Microsoft Windows Server 2003, Enterprise x64 Edition
- Microsoft Windows Server 2003, Datacenter x64 Edition

Itanium-based systems

- Microsoft Windows Server 2003 with SP1, Enterprise Edition for Itanium-based Systems
- Microsoft Windows Server 2003 with SP1, Datacenter Edition for Itanium-based Systems

Should I be learning C# or VB.NET, or some other .NET language?

Although Clarion is a full-fledged .NET language, it probably will be to your advantage to learn at least one other . NET language. There's a wealth of .NET programming information out there, and the vast majority of books and articles deal with either C# or VB.NET. So which language should you learn?

VB.NET in general has more Clarion-like syntax; there are certainly differences, but VB.NET is more of a "plain English" programming language. C# on the other hand has C-like syntax which isn't to everyone's liking. It's also easier to make non-obvious mistakes with C#. On the other hand, C# is the .NET reference language, so you can expect it to support all the latest .NET features, and it's generally a better source of programming examples.

If you have C, C++, or Java experience, choose C#. If you've never worked in a language with C-like syntax then you'll probably be better off with VB.NET. Carl Barnes recommends Programming VB .NET: A Guide For Experienced Programmers, which is also available as a free download - look for the Free eBook Download link on that page.

Why should I choose Clarion# over VB.NET or C#?

Clarion developers have enjoyed the benefits of code generation since the days of CPD 2.0. And when the Clarion# AppGen and templates are ready, I think it'll be easy to see the productivity advantage in Clarion#. But the Clarion. NET AppGen isn't ready yet, and there are no shipping Clarion# templates. Until that happens, why should you choose Clarion# over VB.NET or C#?

First, let me deal with the reasons to use VB.NET or C# instead of, or in addition to, the Clarion# beta. Obviously both those languages are available in gold release, and have been for some years, while Clarion# is, well, in beta. So you can expect fewer bugs in VB.NET and C#, and more complete support for many .NET features. And Visual Studio is a more evolved hand-coder's environment, at least at the moment, with extensive add-in support.

There are, however, some important reasons for choosing the Clarion.NET beta over (or at the very least in addition to) VB.NET and/or C#:

- Language familiarity - you can start getting up to speed on .NET using a language with which you are familiar
- QUEUEs - although .NET has extensive support for collections, queues are still a dead-easy way to manage lists in memory, and a terrific feature of the language.
- Reports - The report designer isn't yet feature complete, but the report structure in Clarion# is basically the same as it is in Clarion. Reporting is one of Clarion's great strengths. Creating reports in other .NET languages typically means buying add-on products.
- File access - you have access to all the file drivers, including TopSpeed and Clarion files.
- File processing - you can still use Clarion's file access grammar (SET/NEXT etc.) if you want to.
- String handling - Strings in .NET are massively different from Strings in Clarion. The ClaString class provides compatibility with Clarion string handling code.
- Preparation - although SV has indicated a C# and/or VB.NET template chain is a future possibility, you can be sure that the first template sets will be for Clarion#. As with Clarion, the better you know the Clarion# language, the better you'll be able to take advantage of the templates and the corresponding class libraries.

Are there bugs in the beta? Sure, it's a beta. Don't buy in if you're not ready to deal with that fact. But you can already do some pretty cool stuff with the first beta, and the compiler is pretty solid.

Additional reading

- Clarion Magazine articles on Clarion.NET
- All Clarion Magazine articles related to .NET
- Clarion Magazine articles on mixing Clarion 6 and .NET
- How to subscribe to Clarion Magazine
- CapeSoft's Clarion.NET FAQ
- Randy Rogers' Clarion# Examples

David Harms is an independent software developer and the editor and publisher of Clarion Magazine. He is also co-author with Ross Santos of *Developing Clarion for Windows Applications*, published by SAMS (1995), and has written or co-written several Java books. David is a member of the American Society of Journalists and Authors (ASJA).

Reader Comments

Posted on Saturday, November 17, 2007 by Wolfgang Orth

If both new Clarions deliver desktop applications - why should I buy both? Would Clarion# not be enough? It seems to be the most evolved as it covers Desktop, somehow Web-applictions and mobile devices. So the traditional Clarion 7 with its WinAPI-programming is oldfashioned history from now on, even before it got gold? Or am I wrong on this?

Posted on Sunday, November 18, 2007 by Dave Harms

Thanks Wolfgang - see answer above.

Dave

Posted on Wednesday, November 21, 2007 by Carl Barnes

A concern I don't think I see addressed in the FAQ are the system requirements for the End User computer for a Clarion# application.

A "classic" Clarion Win32 app will typically run on Windows 98 or newer. Probably even Windows 95. No patches are needed. Nothing to download. 3rd party tools can change this.

A Clarion# .Net app requires the 2.0 .Net Framework. That is more picky about systems. Here's a list I found:
Windows 2000 SP3; Windows 98; Windows 98 SE; Windows ME; Windows Server 2003; Windows Vista; Windows XP SP2

Note that XP users must have SP2. I don't see NT 4 or 95.

Some (or many) users will have to download and install 2.0. That's a 23MB file that takes a 280MB of diskspace. I wonder if the 23MB is just the installer and it downloads more.

In summary if you are trying to reach the maximum users, that may be running older hardware, the .Net way might cost you a few.

Posted on Friday, November 23, 2007 by Dave Harms

Thanks Carl - I've updated the FAQ.

Posted on Thursday, December 13, 2007 by Robert Wright

If one would have to more or less re-write ones apps (and learn clarion#) and it looks like C# and VB.Net are high as recommemded languages. What advantage or disadvantage does clarion.net have over visual studio 2005/8? Does any one have a feature comparisson.

Posted on Thursday, December 13, 2007 by Dave Harms

Robert - I've added a response to the FAQ.

Dave

Posted on Wednesday, December 19, 2007 by Robert Wright

How does the tree control in clarion# compare with C6 Clarion, C# and VB.net?

Posted on Wednesday, December 19, 2007 by Dave Harms

Robert,

I've had a look at the declarations in the Clarion.Windows.Forms namespace and don't see anything indicating there's a Clarion-specific tree control. I might have missed it, or something may be in development, or it may be that you'll just use the standard WinForms control or any of the third party products out there.

Dave

Add a comment

Clarion Magazine

Understanding Clarion# Strings

by Dave Harms

Published 2007-12-10

Clarion.NET brings numerous changes to the Clarion language (now called Clarion#). I discussed a number of these changes in Clarion.NET Language Changes: What's Gone and Clarion.NET Language Changes: What's New. In this article I'll take an in depth look at how strings change in Clarion#. I'll also lay the groundwork for an important follow-up discussion on automatic instantiation, which you won't want to miss.

In most applications strings account for the majority of memory allocation. And unfortunately for Clarion developers, there's a big difference between how Clarion has always handled strings and how they are handled in .NET.

Traditional Clarion strings

I'm still struggling a bit with terminology when it comes to differentiating between the Clarion we know (which I've been calling Clarion WinAPI) and Clarion#, the .NET version of the language. A few people have suggested to me that Clarion and Clarion# are difference enough, so that's the terminology I'll use in this article. Post a reader comment below and let me know what you think.

Here's the formal syntax for STRING in Clarion:

```
                    | length |
label   STRING(     | string constant |)     [,DIM( )][,OVER( )] [,NAME( )] [,EXTERNAL] [,DLL] [,STATIC]
                    | picture |              [,THREAD] [,AUTO] [,PRIVATE] [,PROTECTED]
```

But that's pretty complicated. Most STRING declarations are simply a label and a length:

```
S       STRING(20)
```

The most important point is that STRING is a fixed length string. You specify how much space you need, and that memory is automatically allocated. You don't need to clean up this kind of string after you're done with it.

You also, however, have the option of creating strings dynamically, at runtime. In this case you declare the string as a reference variable without a specified length:

```
S       &STRING
```

Until you do something with this reference it has a null value; you have to create the string using the NEW operator:

```
S &= NEW(String(20))
```

And when you're done you have to dispose of the string this way:

```
DISPOSE(S)
```

Failure to dispose means the memory is not freed until the application terminates; this unfreed memory is called a memory leak, and memory leaks are Not A Good Thing.

Value and reference types

Before I get into .NET strings I want to explain a bit of terminology. In Clarion, and in Clarion# (as in .NET generally) there are two types of variables: value types and reference types. The code

```
Sval      STRING(30)
```

is an example of a value type. The variable is the 30 byte string. On the other hand, the code

```
Sref      &STRING(30)
```

is an example of a reference type. Sref is not the string, it is the address of the string. That's an important and subtle distinction.

Value string types can be accessed faster because the variable contains the string; reference types contain the address of the string, so there's an extra step involved in locating the actual string in memory. (I'm using the example of a string, but reference types can be just about any kind of data.)

The benefit of using a value type is speed; the benefit of a reference type is flexibility. Although most Clarion developers use value type strings, there are situations where using a reference type is an advantage, as when you'll be dealing with strings whose size you won't know until runtime.

.NET strings

.NET strings are quite a different animal from Clarion strings, although they bear some resemblance to string references. The .NET string datatype is System.String. Here's a System.String declaration in Clarion# (the USING SYSTEM directive is implicit, so the System. namespace doesn't have to be specified);

```
s            &String
  code
  s = 'abcdef'
```

Strings in .NET are instances of the System.String class. That is, each string is an instance of a class, a.k.a. an object, with properties and methods. And System.String is a reference type, not a value type.

The second important point is that .NET strings are fixed length and immutable. When you assign a value to a System. String, the string is exactly as long as the data. There's no padding, no empty space. When you assign a new value, or append a value to a string, the old string data is discarded and a new string is automatically created.

Along with a Length property, System.String has numerous methods including:

- Clone
- Compare
- CompareOrdinal
- CompareTo
- Concat
- Copy
- CopyTo
- EndsWith
- Equals
- Format
- GetEnumerator

- GetHashCode
- GetType
- GetTypeCode
- IndexOf
- IndexOfAny
- Insert
- Intern
- IsInterned
- Join
- LastIndexOf
- LastIndexOfAny
- PadLeft
- PadRight
- Remove
- Replace
- Split
- StartsWith
- Substring
- ToCharArray
- ToLower
- ToString
- ToUpper
- Trim
- TrimEnd
- TrimStart

As you can guess from that lengthy list, most of Clarion's string handling functions (SUB, UPPER, INSTRING etc) have direct equivalents in System.String methods. And since SUB, UPPER etc. all still exist in Clarion#, it also seems plausible that these methods could simply call System.String methods as needed. But there's at least one big problem with just using System.String in place of STRING in Clarion# code, and that has to do with STRING's fixed length. There are also some issues with string slicing and automatic type conversion.

Needed: A .NET string of arbitrary length

When you declare a STRING in Clarion you give it a length, but you don't do that with System.String. You can give a System.String an initial value if you want, but the length of a System.String is always the length of its data (sort of like a CSTRING).

SoftVelocity's solution is Clarion.ClaString, a completely new class which is evidently *not* derived from System.String. Here's a code snippet (note that the USING Clarion directive is implicit, just like the USING System directive):

```
s          ClaString(20)
 code
 s = 'abcdef' ! String is 20 characters long
```

Interestingly, if you leave off the length specification ClaString behaves just like System.String, and is only as long as the data it contains:

```
s      ClaString
```

```
code
s = 'abcdef' ! String is six characters long
```

ClaString's methods include:

- AssignString
- Clone
- Dispose
- Equals
- GetBufferSize
- GetBytes
- GetHashCode
- GetReferenceToSlice
- GetSlice
- GetType
- IsBytesDeposit
- RefRequal
- ReplaceSlice
- ToBoolean
- ToByte
- ToChar
- ToDateTime
- ToDecimal
- ToDouble
- ToInt16
- ToInt32
- ToInt64
- ToSByte
- ToSingle
- ToType
- ToUInt16
- ToUInt32
- ToUInt64

Properties include

- Kind
- Length
- RefType
- Value

You can see all of the To... methods that accommodate Clarion's automatic type conversion - similarly there are a bunch of constructors that make it possible to assign numeric values to strings.

The upshot of all of this is that you can still do string handling in Clarion# almost exactly as you do in Clarion, provided you use ClaString instead of System.String.

ClaString vs. STRING

Does all of this mean that you should always use ClaString in place of System.String? Probably not.

There are at least two issues here. One is that you'll have to change all of your STRING definitions to ClaString. Presumably a conversion tool could be created to take care of this task relatively painlessly.

> **NOTE:** You may be wondering why Clarion.ClaString isn't Clarion.String. The problem is name collisions. Every .NET application (at least, every one I can think of) needs classes declared in the System namespace; every Clarion# application that uses Clarion-specific functionality is going to need to use classes declared in the Clarion namespace as well as the System namespace. If you have identical labels in two namespaces, and you have USING directives for both those namespaces, the compiler is going to get confused. It won't know if String means System.String or Clarion.String, so you'll have to use the fully qualified name instead of the much shorter class name. And that's a pain.

The second issue is that ClaString is a completely different class from System.String, and if you write code that other .NET languages can use you'll have to stick to System.String anyway if you want those languages to call methods in your classes and pass or receive strings.

Finally, since ClaString is not derived from System.String, you have to use the Clarion RTL functions for string manipulation, which may make it more difficult to port sample code from, say, C#. Happily you can assign a String to a ClaString (and vice versa) so that isn't necessarily a huge problem, as long as your code makes it relatively clear which strings are .NET strings and which are Clarion strings.

Here's a little console application that illustrates some of the differences between ClaString and String manipulation:

```
s       ClaString(20)
t       &String

    CODE
    s = 'abcdef'
    t = s
    s = sub(s,1,3)
    t = t.Substring(1,3)
    System.Console.WriteLine(s)
    System.Console.WriteLine(t)
    System.Console.ReadKey()
```

The ClaString s is assigned a value; then the System.String t is assigned a copy of that value. The Clarion SUB function extracts a substring and assigns it to s, just as the SubString method does. But the output may not be what you expect:

```
abc
bcd
```

Strings are basically arrays of characters, and in .NET arrays are zero based, so the t.Substring method has to be called this way

```
t = t.Substring(0,3)
```

to get the same result as Clarion's SUB function.

Array indexes are just one of the subtle ways Strings are different from ClaStrings. Another has to do with how references are assigned.

Clarion has the &= operator which you must use when assigning references (although in Clarion# you can also use the := "smart" operator, which will do an = or &= as the situation requires).

This code creates two references to the same string:

```
s        ClaString(20)
s2        &ClaString

         CODE
         s = 'abcdef'
         s2 &= s
         s2 = 'ghijkl'
         System.Console.WriteLine(s)
         System.Console.WriteLine(s2)
         System.Console.ReadKey()
```

The output is

```
ghijkl
ghijkl
```

because both s and s2 point to the same string object.

Now, look at some similar code in C#:

```
String t = "abcdef";
String t2 = t;
t2 = "ghijkl";
Console.WriteLine(t);
Console.WriteLine(t2);
Console.ReadKey();
```

The output is

```
abcdef
ghijkl
```

Okay, what just happened? You may think the problem is that the = operator creates a copy, but this is not the issue. You can use the System.String.Clone method, which returns the string instance, and you will still get the same result.

Remember what I said earlier about the immutability of .NET strings: when you assign a new value to a string variable, the old string is disposed and a new string is created. So even though t2 may start off with a reference to t, by changing the value of t2 you're also removing that reference and creating a reference to the new string, rather than changing the value of t.

Clearly there are enough differences in string handling between Clarion and .NET to warrant the use of ClaString in many situations. If you do wish to use System.String instead of ClaString you'll need to pay special attention to zero based indices and the immutability of .NET strings.

Happily, you can easily mix ClaString and String, and all you need is an = sign to assign a string value from one type to the other.

But wait, there's more...

Let me take you back to this bit of code:

```
s        ClaString(20)
t        &String

      CODE
      s = 'abcdef'
      t = s
      s = sub(s,1,3)
      t = t.Substring(1,3)
      System.Console.WriteLine(s)
      System.Console.WriteLine(t)
      System.Console.ReadKey()
```

Notice anything odd? The ClaString s is declared in the manner of a normal Clarion string - ClaString(20). On the other hand, the String t is declared as &String. Why does String need the & while ClaString doesn't?

Watch what happens if I put a matching & in front of ClaString and take away the (20):

```
s        &ClaString
t        &String

      CODE
      s = 'abcdef'
      t = s
      s = sub(s,1,3)
      t = t.Substring(1,3)
      System.Console.WriteLine(s)
      System.Console.WriteLine(t)
      System.Console.ReadKey()
```

The code compiles, but the application gets a runtime Null Pointer exception. That's because I haven't allocated any memory for s. If I add the line

```
s &= new ClaString(20)
```

before I reference s, then all is well. (I can also leave off the (20) if I want a variable length string.)

How about this declaration:

```
t        String
```

If you try to compile that code, you'll get this error:

Cannot find constructor for class 'System.String' that receives specified set of arguments (Wrong number of parameters) (CLA00074)

When your application first creates an instance of any string object, whether it's a ClaString or a System.String, it uses the appropriate constructor for either the declaration or the NEW statement, as appropriate. The problem here is that the compiler needs to create an instance of System.String, but it doesn't have anything to pass to the constructor, and there's no matching parameter-less constructor. If, on the other hand, you declare the String as

```
t        &String
```

and then in your code do something like

```
t = 'abcdef'
```

the compiler will create the String instance and pass the value 'abcdef' to the constructor. There is a matching constructor so the code compiles.

There are, then, three legal ways to declare strings in Clarion#:

```
s        &ClaString ! requires NEW
s        ClaString  ! optional parameters, use as is
t        &String    ! use as is
```

Clearly there's a significant difference between how Clarion# handles its own classes, and how it handles .NET classes. There are some interesting historical reasons for this, but even more importantly, it may be that the question of how much Clarion# should look like Clarion, and how much it should look like other .NET languages, isn't completely settled.

Earlier I asked why, in the absence of a NEW statement, ClaStrings *do not* need the & and .NET strings *do* need the &. There is an answer to that question, and the answer raises some an important (and perhaps still open) question as to how Clarion# should handle class instantiation. More on that next time...

David Harms is an independent software developer and the editor and publisher of Clarion Magazine. He is also co-author with Ross Santos of *Developing Clarion for Windows Applications*, published by SAMS (1995), and has written or co-written several Java books. David is a member of the American Society of Journalists and Authors (ASJA).

Reader Comments

Posted on Wednesday, December 12, 2007 by Gregg Matteson

Dave,

Appreciate this article. Keep em coming!

Thank you,

Gregg Matteson

Posted on Wednesday, December 12, 2007 by Dave Harms

Thanks Gregg - definitely lots more where this came from.

Dave

Add a comment