Visual COBOL for Visual Studio Getting Started with Solutions, Projects and the COBOL CALL Statement

I) Overview

1. Solutions and Projects

In Visual COBOL for Visual Studio, the main unit of work is called a solution. Solutions can contain multiple projects. These projects can be managed code COBOL projects or native code COBOL projects or can be C# projects or VB.NET projects, etc. Visual COBOL projects can contain only COBOL programs or classes but these programs and classes can interact with the programs or classes contained within projects written in a different language like C#.

There are two basic types of projects, Application projects and Library projects. Normally, a solution would contain a main Application project like a Windows Forms Application, WPF Application or a Console Application. Application projects generate an output file with the .EXE extension and contain the main entry point of an application. Library projects, like a Class Library or a Link Library typically contain programs and classes that are called by the main application project. Library projects generate an output file with the .DLL extension.

Each project can contain one or more source programs or class programs. In managed code, each project is compiled into a single output file called an assembly. In native code COBOL Application and Library projects you can also select to have multiple output files. In this case, each individual program within the project will be compiled into its own .EXE or .DLL.

2. Problems with Calling Programs Located in Different Projects

Each project specifies an output folder into which its generated output files will be stored. The default name of this folder varies depending on the project CPU settings and which build type you are using such as DEBUG or RELEASE. The default location is in a subfolder which is relative to the projects main folder, i.e., .\bin\x86\debug. This default name of the output folder is configurable under the COBOL tab of the Project Properties page.

There are two issues that need to be addressed when a program in one project calls a program in another project.

1. Programs that are called cannot be found.

When an application is started in Visual Studio the output folder in which the main application resides will become the current folder. Programs that are called must either be placed in this startup folder or all programs must be placed in a different folder or they must reside in a folder that is locatable via environment variable PATH.

2. Entry points that are called that are different from the name of the .DLL cannot be found.

When the name of the program in the call statement matches the name of the .DLL on disk then it will be found as long as the conditions in 1 above are true. But if calling an entry point which is the name of another program within the .DLL or the name of an entry point specified in an ENTRY statement within a program in the .DLL, the .DLL containing the program to be called must be preloaded in order to make its entry points visible to the run-time system. This can be done using one of the following methods.

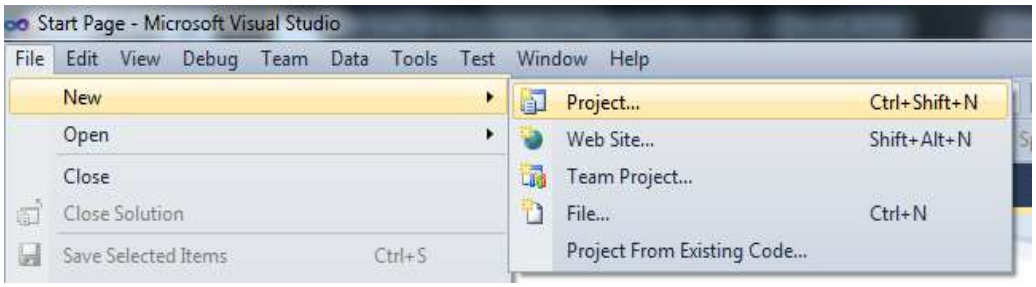
- set proc-pointer to entry “dllname”
- Micro Focus Entry Name Mapper (MFENTMAP)
- Interop Preload section of app.config file (managed code only)

All of these scenarios will be covered in the tutorials that follow.

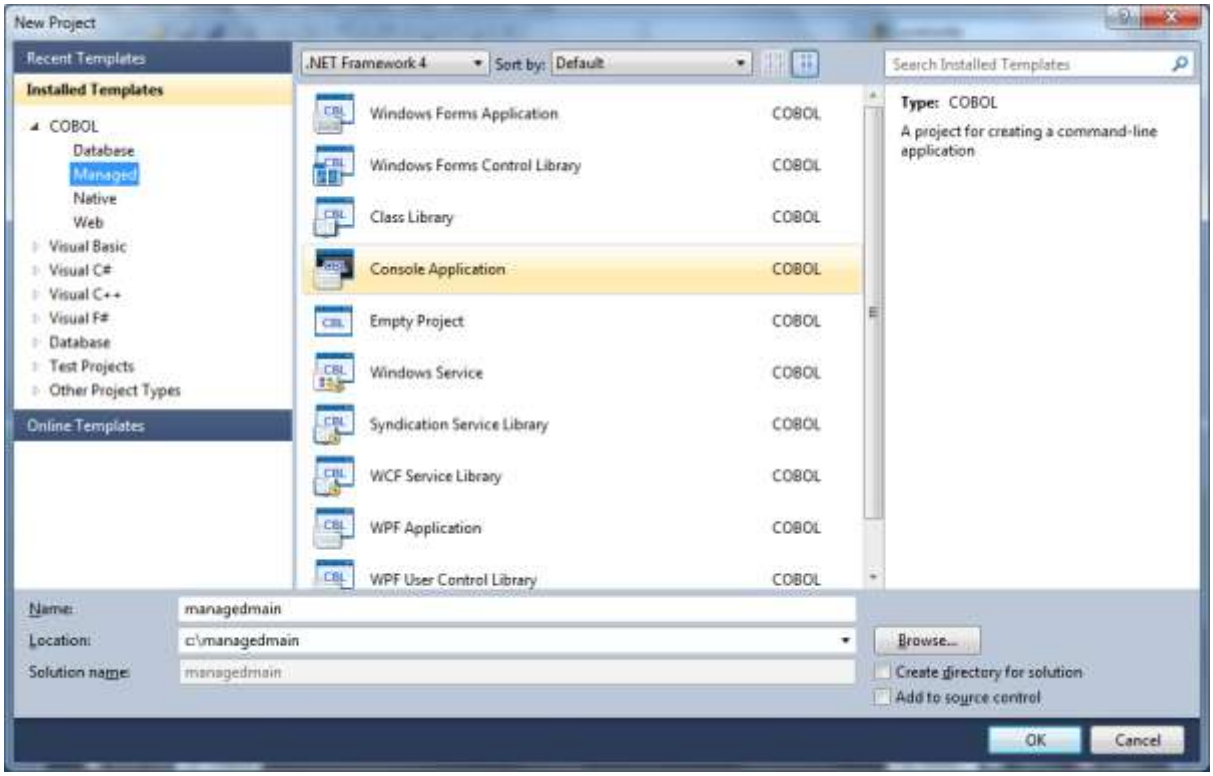
II) Working with Managed COBOL Projects

In this tutorial you will be shown how to setup and use a Visual COBOL solution containing a main application project and a Class Library project containing a program that will be called.

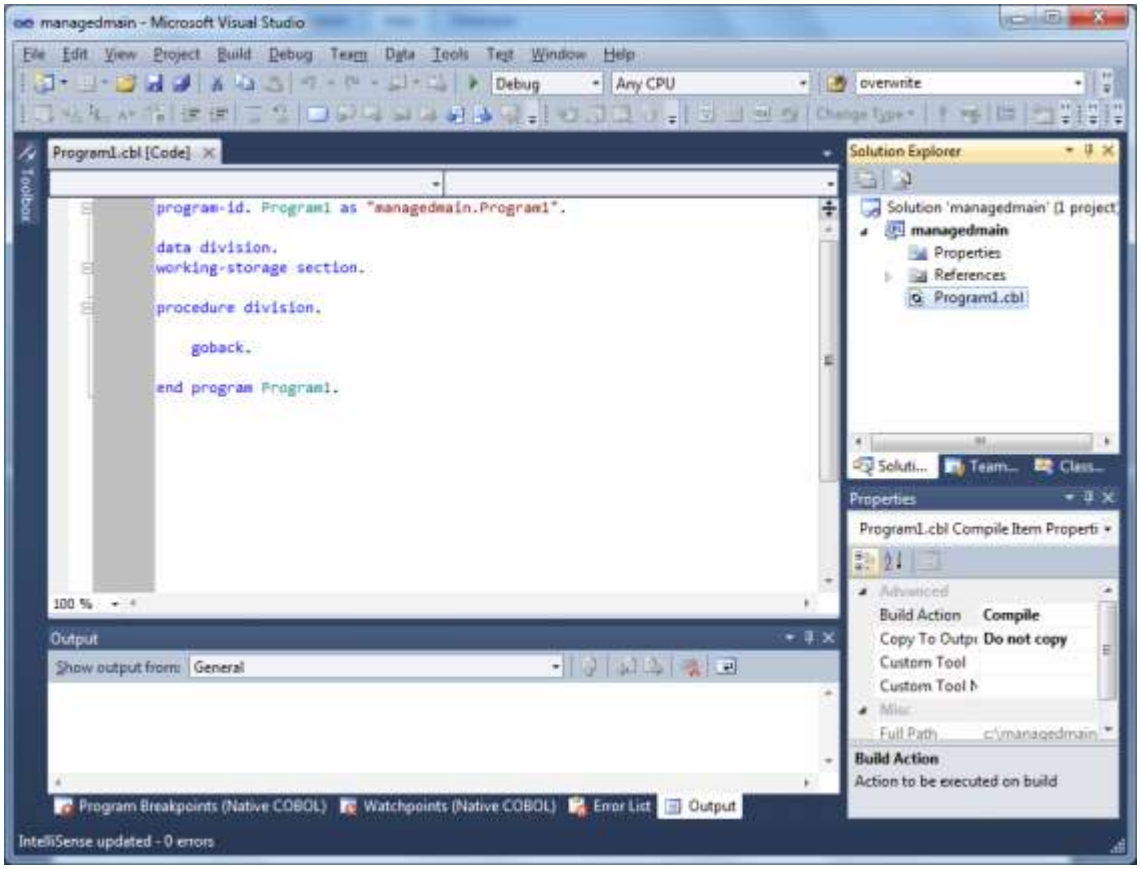
Start Visual COBOL and from the main menu select New→Project as shown below:



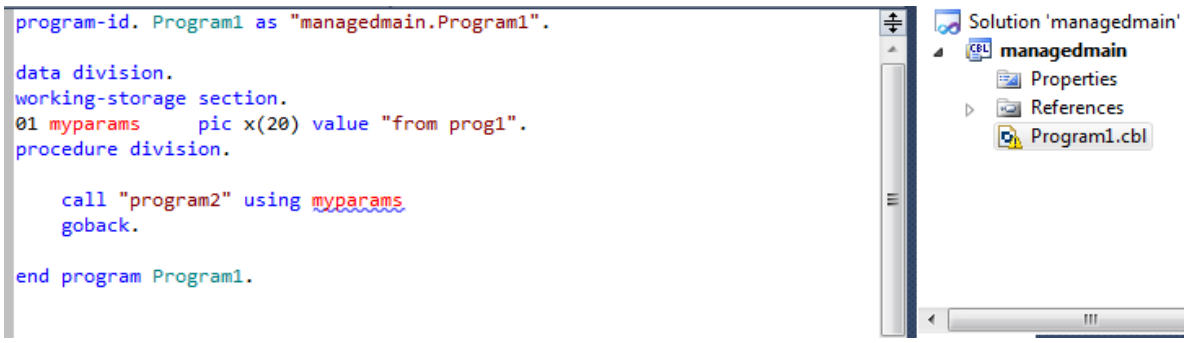
On the New Project Dialog select Managed under COBOL, highlight Console Application and then change the Project Name and Location to managedmain and C:\managedmain respectively. Also uncheck the option for Create Directory for Solution so that your project will have the same folder structure as shown in this tutorial. Click OK to create the new project.



Visual COBOL will automatically create a solution with the same name as your project file and will add a new Program1.cbl file to the project. If you do not see the Solution Explorer Window or the Properties Window you can select to display them under the View menu item.

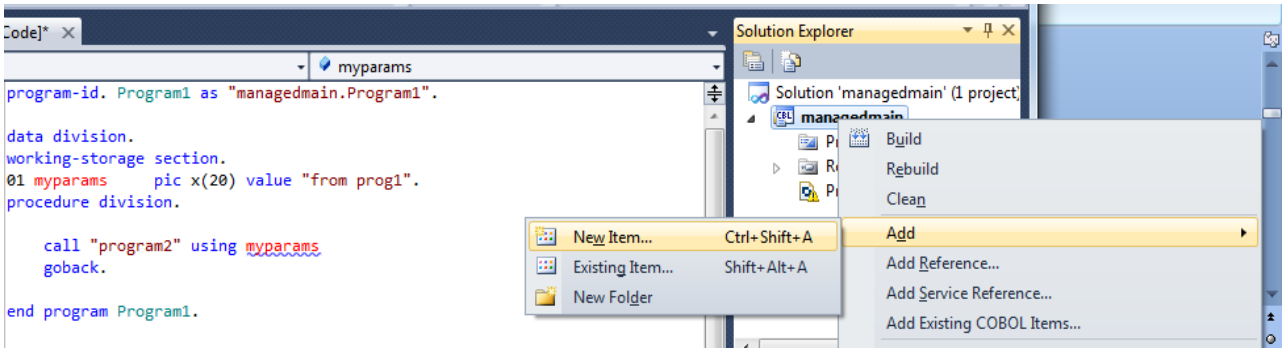


Modify the source code to Program1.cbl in the editor so that it looks exactly like the image below:

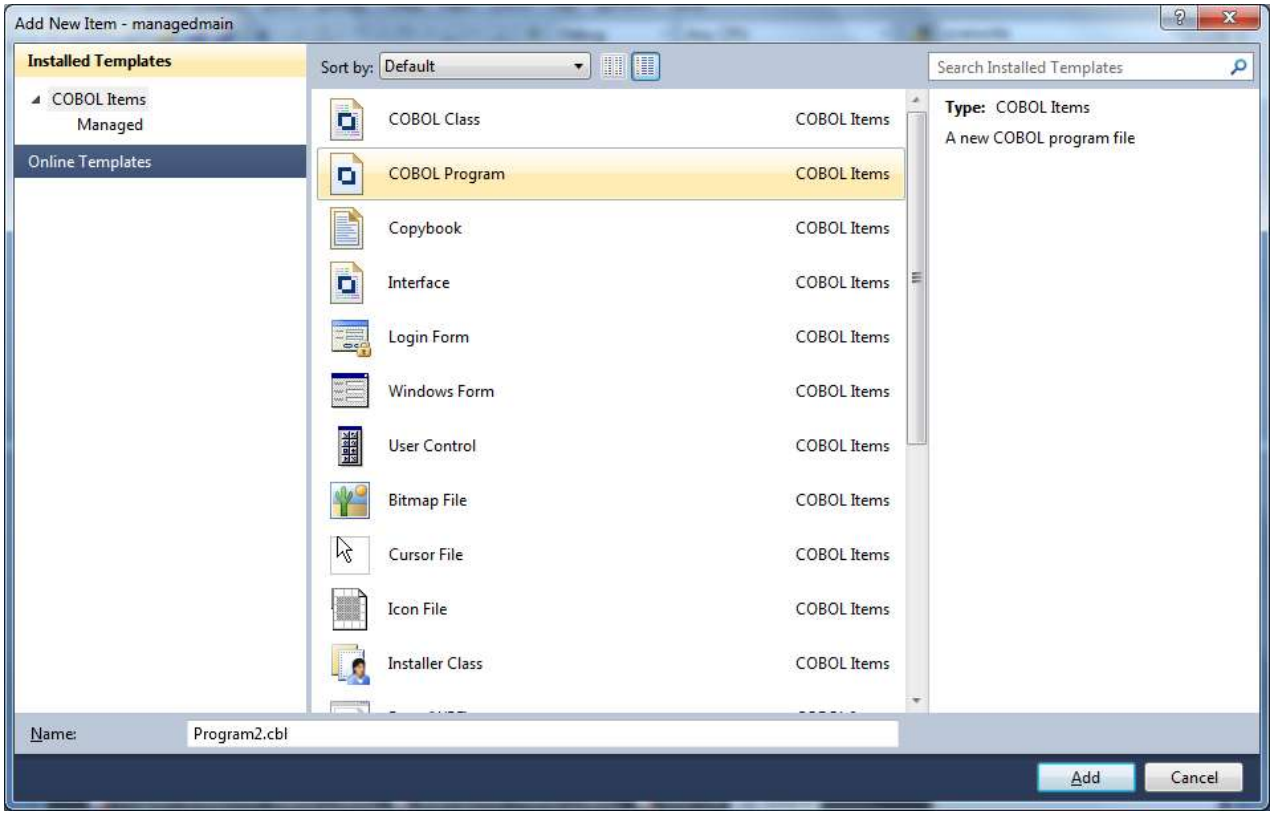


Notice that there is a blue squiggle underneath myparams in the call statement. This is Intellisense in action. If you position the mouse over this squiggle you will see an error message because there currently no program named “program2” exists in the solution. This can be ignored.

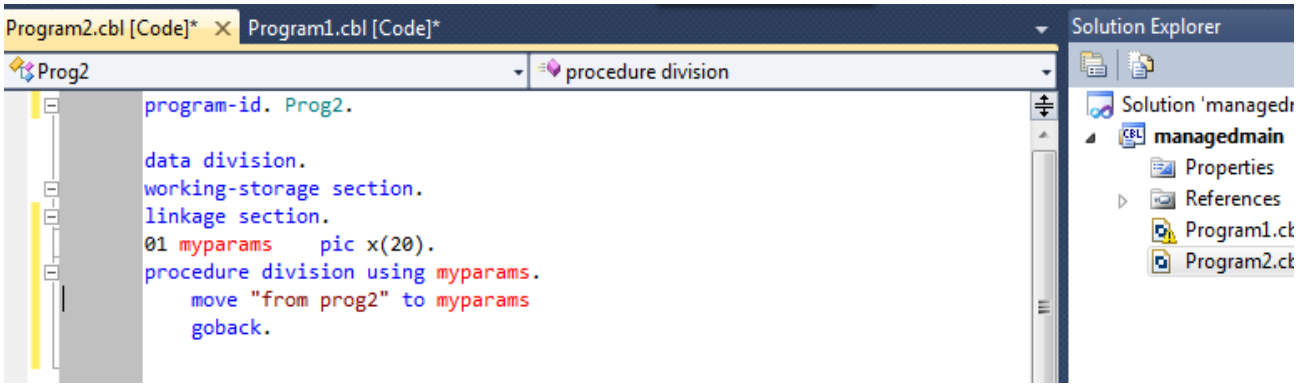
Now we will add a second program to our project. Right click on the Project name in Solution Explorer, which is the managed**main** in bold with the CBL icon next to it and select Add→New Item.



Highlight COBOL Program from the list and then click the Add button at the button to accept the default name of Program2.cbl.

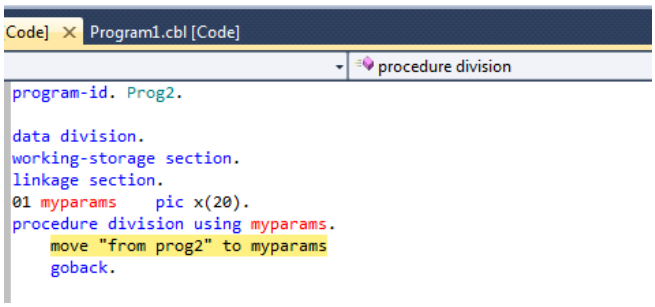


Edit Program2.cbl so that it looks like below. Make sure that you change the program-id from Program2 to Prog2 and then delete the end program statement at the bottom.



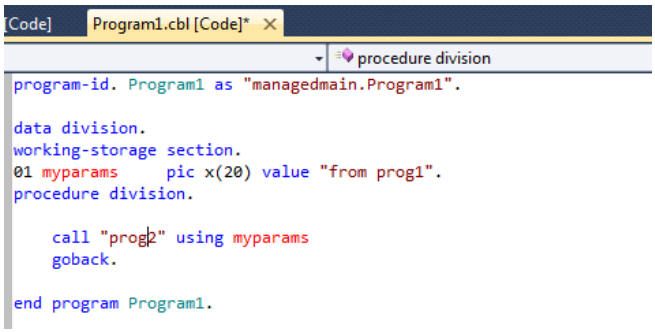
Press the F11 key to build the project and start debugging. The current statement should be highlighted as shown below. Press the F11 key again to execute the current line.

Control should now be given to the called program, Program2 as shown below: Continue to press F11 to step through the rest of the statements and return control to Program1. Press F11 on the goback statement in Program1 to exit the debugger.



Notice that the call statement was referencing program2 which is the name of the program on disk and not the name of the program in the program-id.

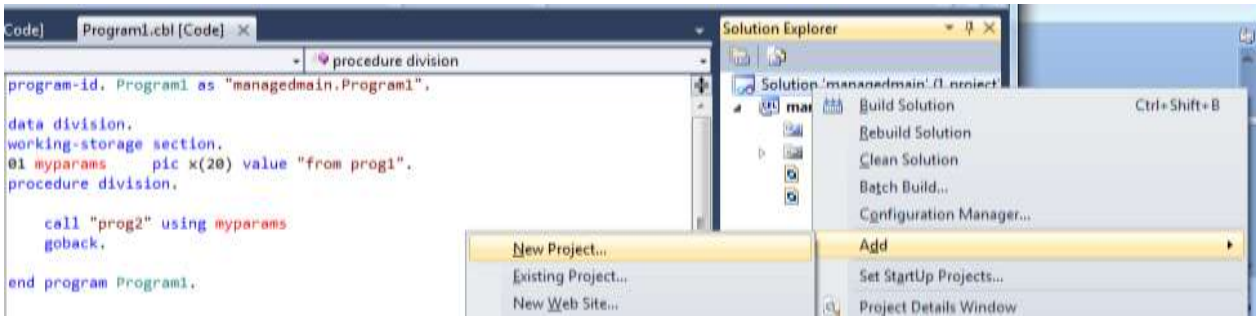
Now change the name in the call statement from program2 to prog2 as show below and then press F11 to start debugging again.



Press F11 to step through the call statement and into program2. Complete the debugging by pressing F5 to run the rest of the program. This is the behavior of the CALL statement in Visual COBOL. You can call a program by its name on disk or by its program-id, if the two happen to differ.

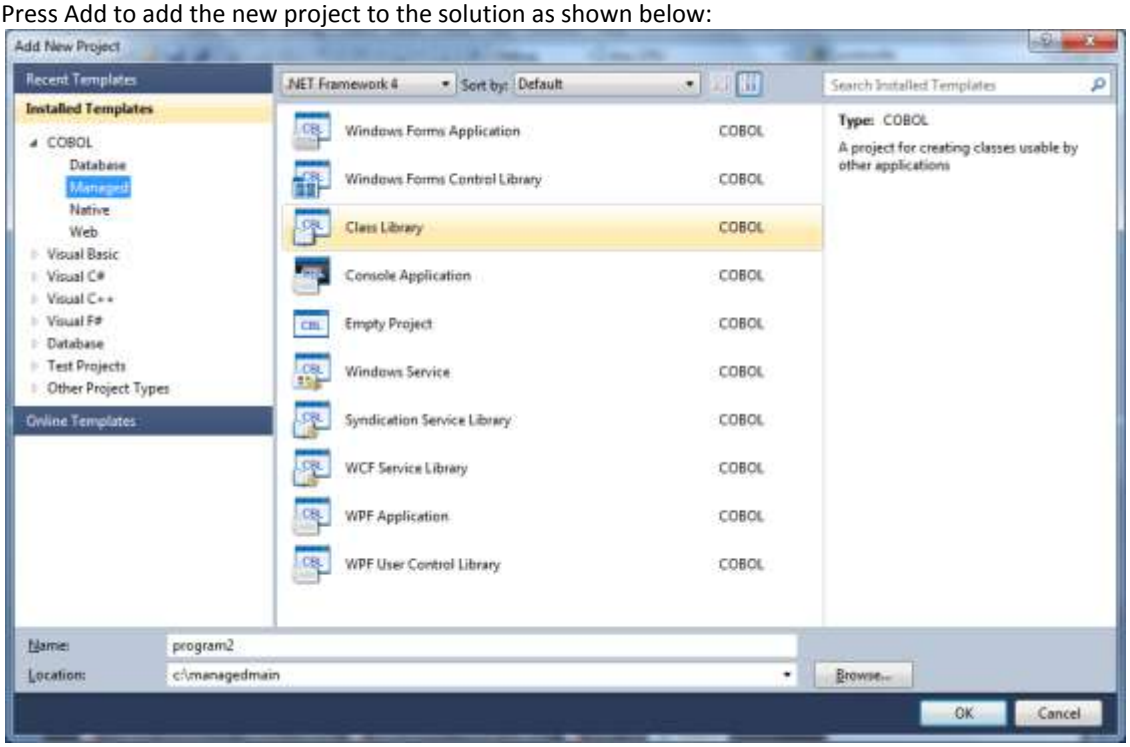
This works fine in this example because both the calling program and the called program both exist in the same project. We will now place the two programs in separate projects to demonstrate a common scenario.

Right click on the Solution name in the Solution Explorer window and then Select Add→New Project as shown below. Make sure that you right click on the Solution name which will be at the top and not the Project name which will be under it.



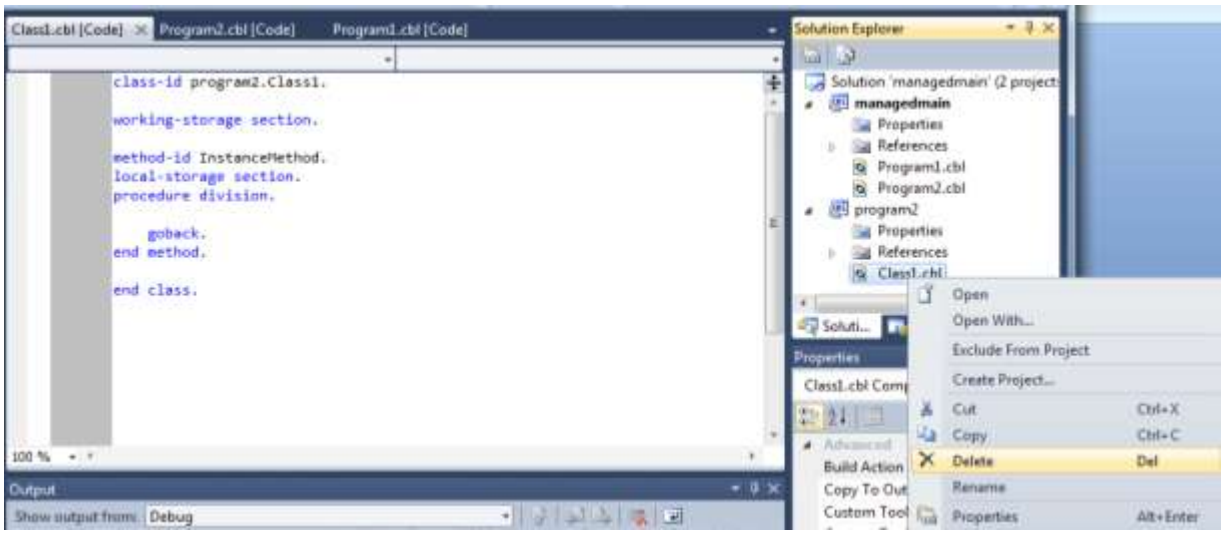
Select Managed under COBOL and then Class Library as the project type. Change the name of the project to program2 and leave the Location set to c:\managedmain so that the new project will be in a subfolder of the main solution.

Press Add to add the new project to the solution as shown below:



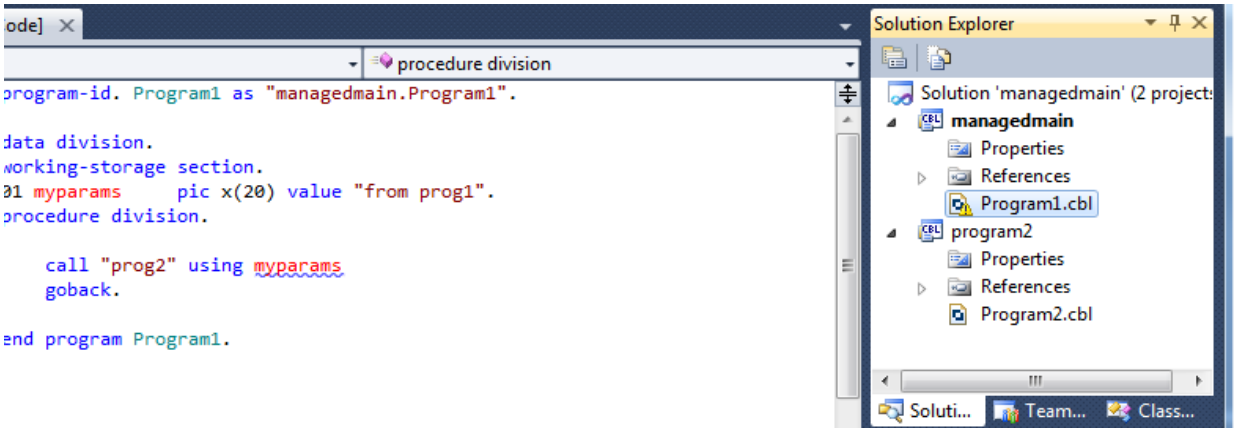
The new project program2 will be created and the default program Class1.cbl will be added to it.

Notice that the default when adding a native Link Library was to add a program named Program1.cbl to the new project. In managed code a Class is the default type in a Class Library. In this example we are not using a class so we will delete it from the project. Right click on Class1.cbl in the program2 project and select Delete as shown below.



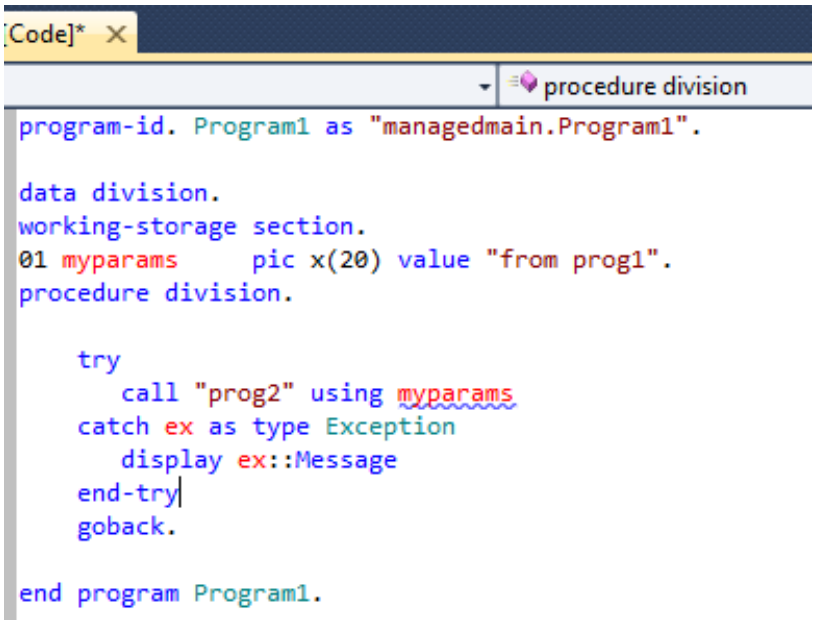
Confirm the deletion when prompted to do so.

Now move the program2.cbl source from project managedmain to project program2 by dragging it from managedmain to the project name program2 (with CBL project icon next to it) You could also do this by right clicking on Program1.cbl in managedmain and selecting Cut and then right clicking on project name program2 and selecting Paste. Your solution should then look like the following:

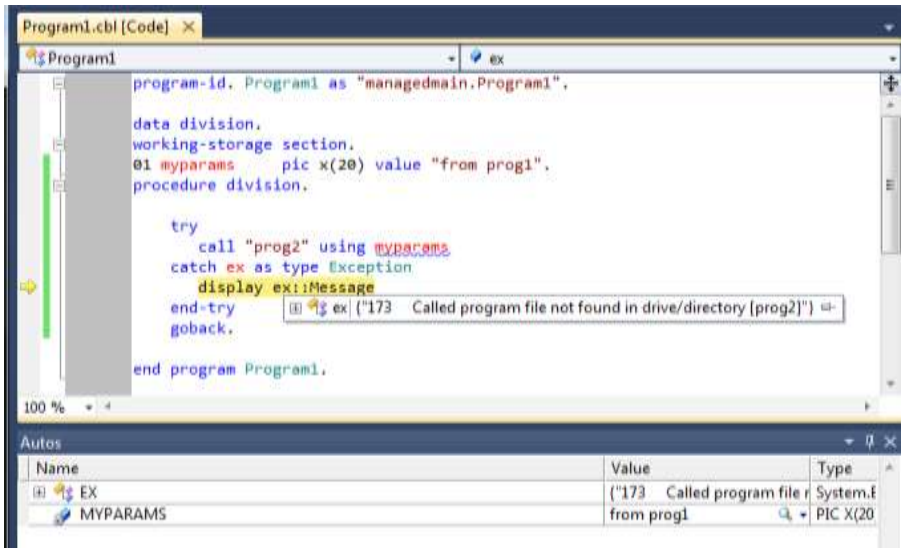


Now Press F11 to rebuild the solution and start debugging again.

This time when you step the call statement it will fail and the debugger will stop. The call is generating an Exception. It trap this Exception so that we can see the error message add exception handling code and make the program look as follows:



Now press F11 to build the project and start debugging. When the call statement is executed the catch code will be executed to handle the exception. Hover the mouse over the ex variable and you will see the contents of the exception message. If you have the Autos window opened you will also see it displayed there.



The message is the same error that we saw in native code; Runtime Error 173. The Run-time System error 173 means that the name in the program name referenced in the call statement could not be found. Continue to press F11 to step through the rest of program1 until it ends.

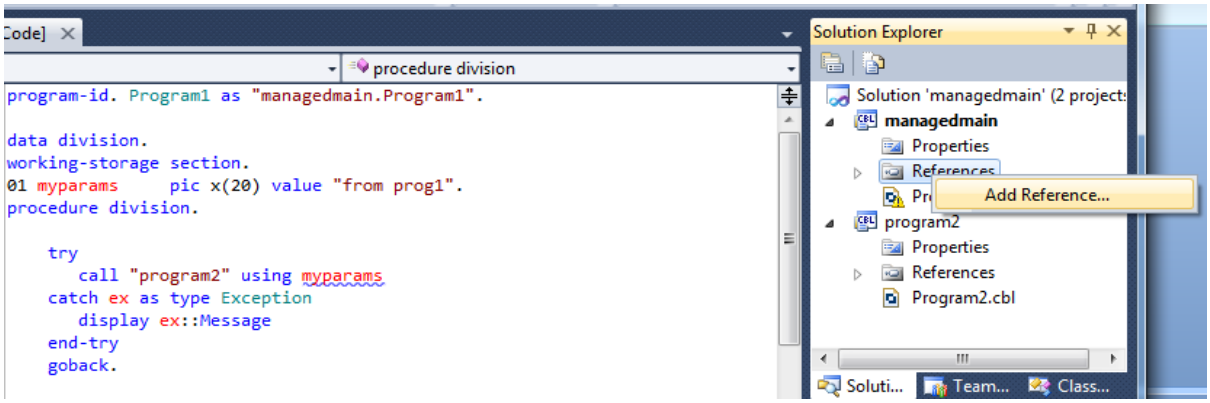
Change the name in the call statement from prog2 to program2 and debug again by pressing F11. The same error occurs. So what has changed?

The difference is that program2 is now in a different project which has a different output folder than the calling project.
managedmain.exe is in C:\managedmain\managedmain\bin\Debug
and
program2.dll is in C:\managedmain\program2\bin\Debug

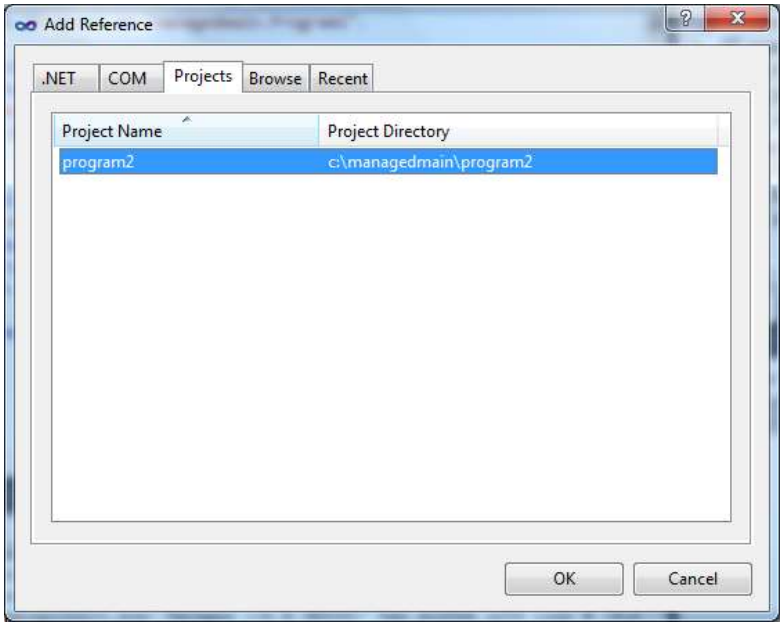
When the application is started the folder containing the startup program becomes the current folder so any programs that it calls, such as program2.dll must either also be in the startup folder or they must be in a folder which is referenced in the PATH environment variable.

In managed code it s much easier to resolve the call as you can simply add a project reference from the main project to the project that contains the program that you wish to call.

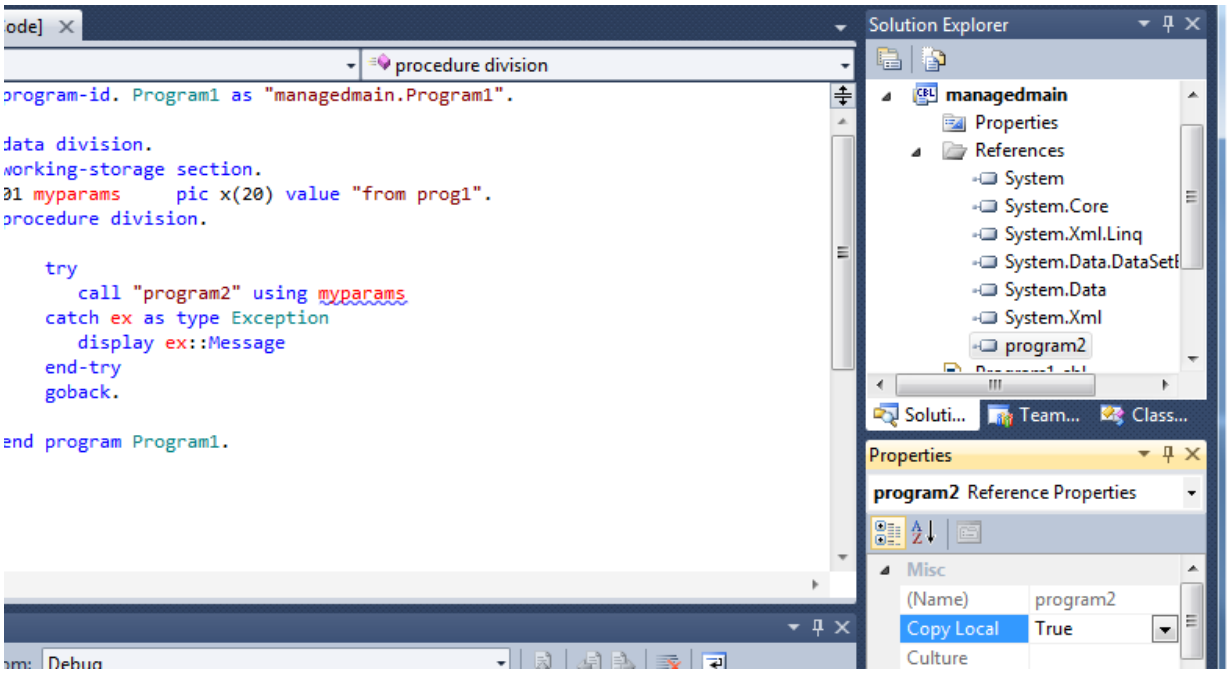
In Solution Explorer, right click on the References folder under the managedmain project and select Add.Reference.



On the Add Reference Dialog, click the Projects tab and then highlight program2 and click Add OK.



If you highlight program2 in the References Folder you will see that it has the property called Copy Local set to True. This means that when program2.dll is built it will automatically be copied into the output folder of the project containing the reference which in this case is managedmain.



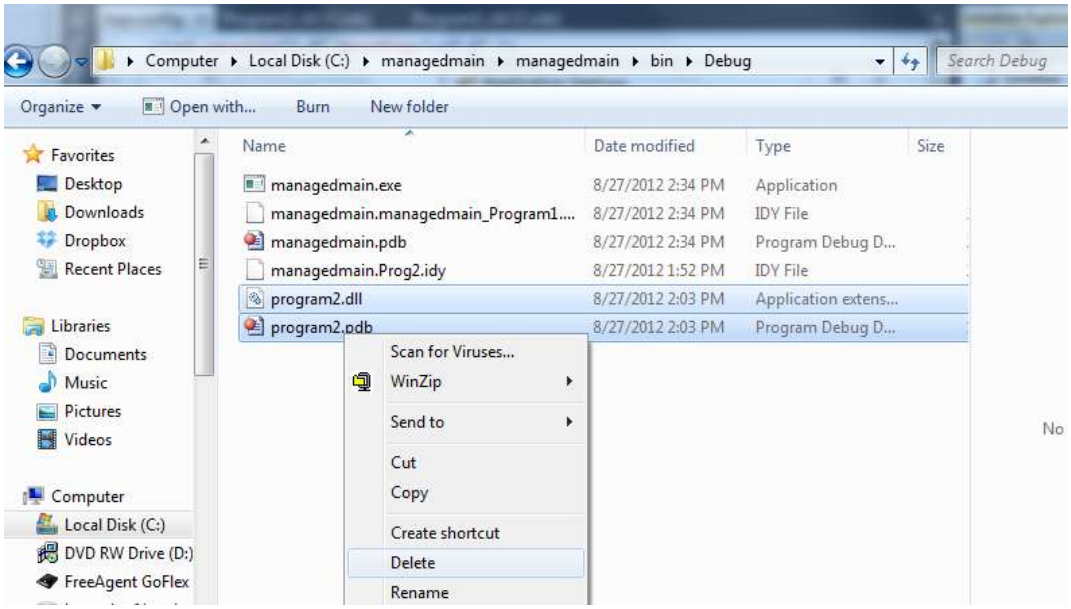
Press F11 to rebuild and start debugging again. Notice that the squiggle underneath myparams in the call statement now disappears because the program is found at compile time. Keep pressing F11 to step into program2 and then back into program1 until finished.

Change the call statement to reference “prog2” instead of “program2” and then step through again. The call statement can reference either the program-id name or the program name on disk and both are resolved by adding a reference.

Although, adding a project reference is the easiest way to resolve program names when calling between projects the other methods demonstrated in the section “Working with Native Projects” will also work with managed code. You may want to use one of the following methods if you have a large number of projects or if you wish to call programs in assemblies that are not part of the current solution.

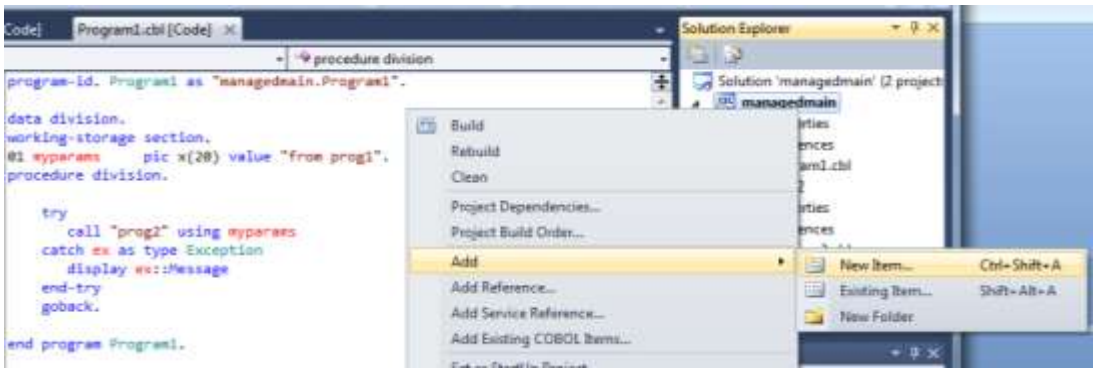
Let’s reset the project to try the other methods. Open the References folder under the managedmain project. Right click on program2 in the References list and select Remove to remove the reference.

Using Windows Explorer, navigate to the folder c:\managedmain\managedmain\bin\debug and delete the files program2.dll and program2.pdb.

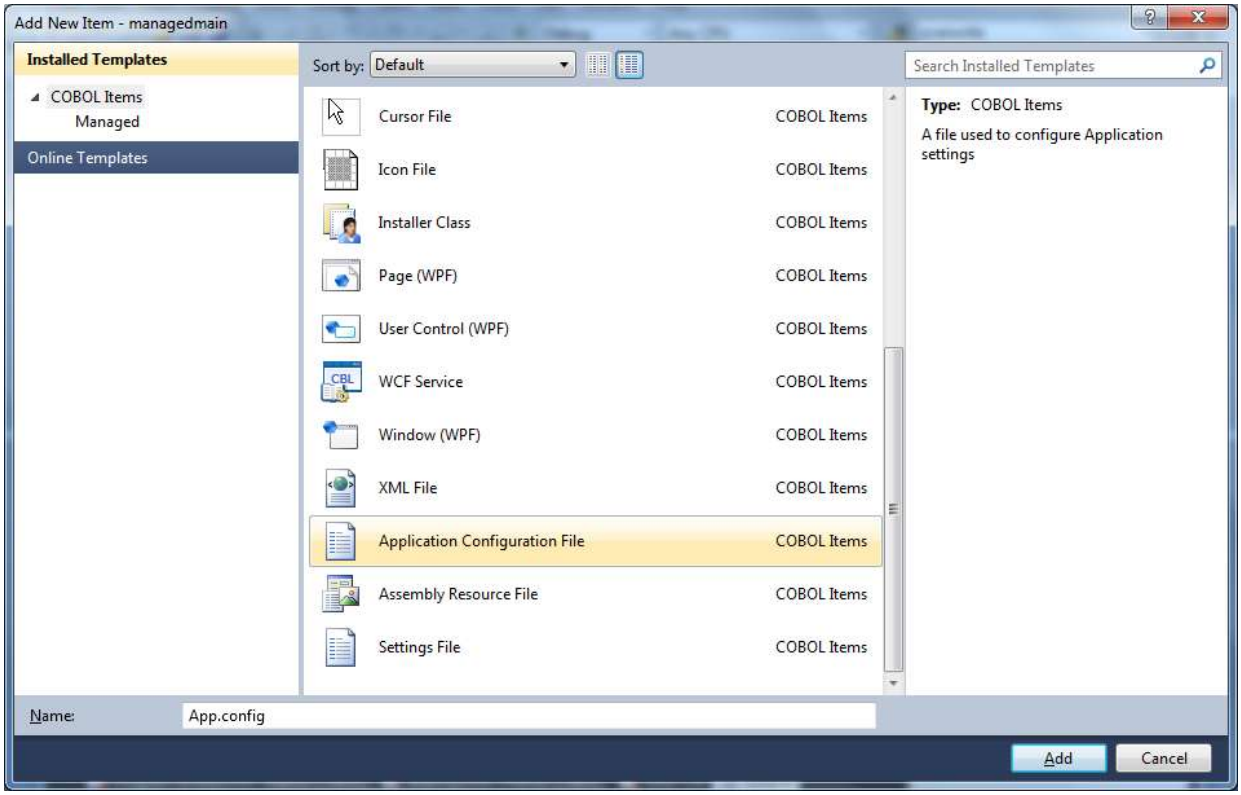


Change the name in the call statement of program1.cbl from “prog2” to “program2”.

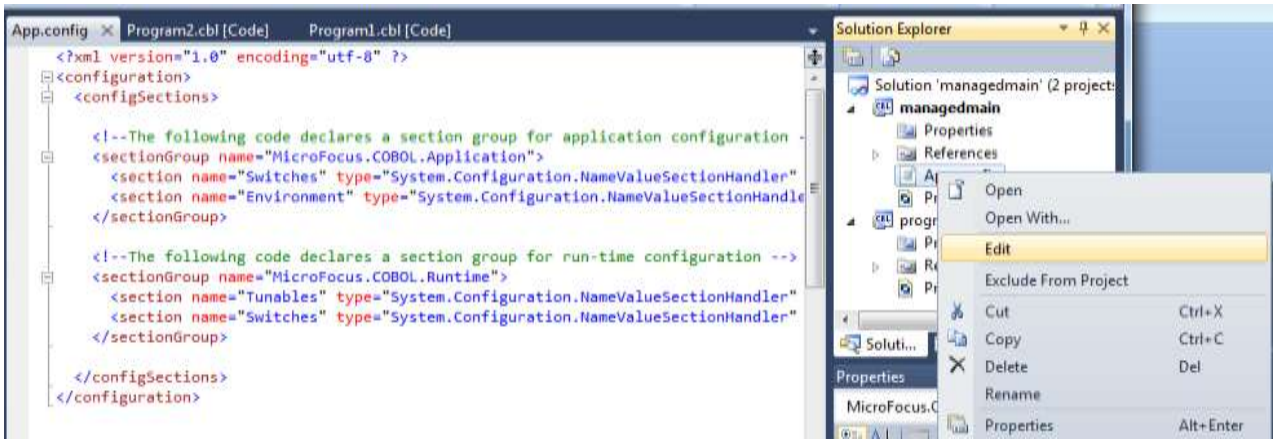
Right click on the project name managedmain in Solution Explorer and select Add→New Item as shown below:



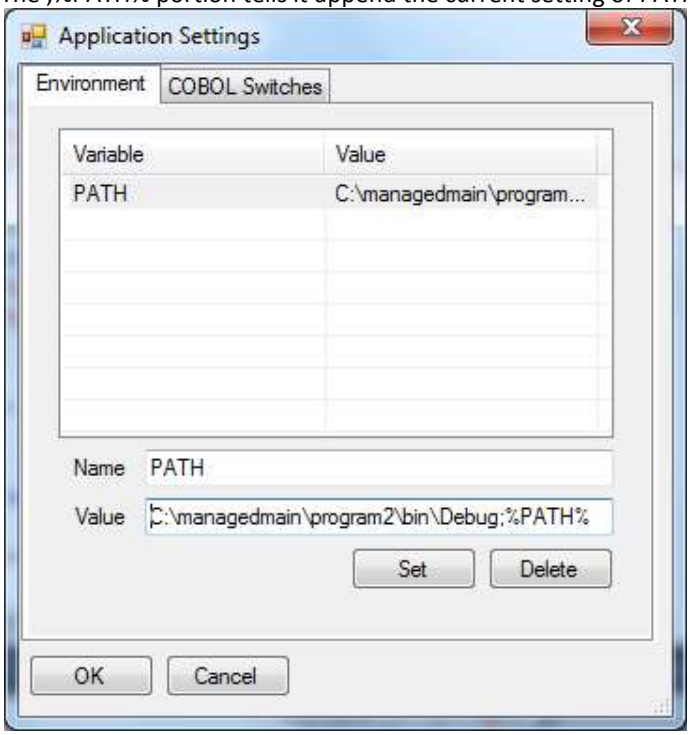
Select Application Configuration file from the list and accept the default name of App.config by clicking on Add.



The file will be added to the managedmain project and loaded into the editor. Close the XML version of this file by clicking on the X in the tab next to app.config name. Then right click on App.config in Solution Explorer and select Edit.

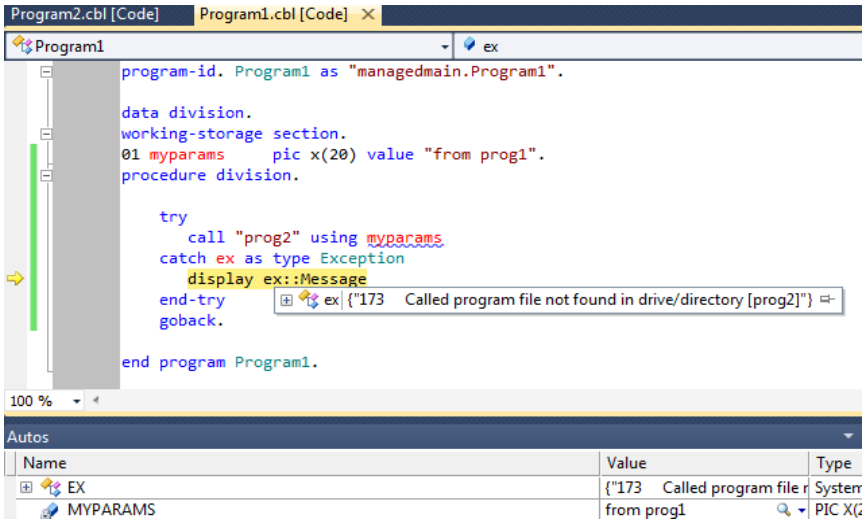


In the popup editor that appears add the value PATH in the name field and enter the location where the program2.dll resides followed by ";%PATH% in the VALUE field and press SET. Then Press OK to save this. The ;%PATH% portion tells it append the current setting of PATH to the new one.



Press F11 to start debugging again and when you execute the call "program2" statement it will now work, although the squiggle error under myparams in the call statement remains because the call is being treated as dynamic, i.e. resolved at run-time instead of compile time.

Stop debugging and change the name in the call statement from program2 to prog2 which is the program-id of the program to be called. Now press F11 to step through the call statement again. It fails when trying to call prog2.



The reason that it worked when calling “program2” is that program2 is also the name of the .dll file on disk, program2.dll. When the call statement is executed the run-time system first checks to see if an entry point named “program2” has already been loaded. If it has then it will call that one. If it hasn’t been loaded, then it next tries to find a program with that name on disk in the current folder. If that search fails it will search through the folders specified in PATH, looking for a program called “program2”. In the case of this tutorial “program2.dll” will be found and loaded and then “program2” will be called.

The reason why it failed when calling “prog2” instead of “program2” is that there is no program called prog2.dll available on disk.

In this case where you wish to call an entry point of a program that resides within a .dll that has a name other than the name of the .dll itself then the .dll must be preloaded in order for the call statement to find the entry point. This is true when calling by program name of by the program-id name if they differ. This also applies to programs that have multiple entry points by using the COBOL ENTRY statement. Of course, if you have already called the main entry point of the .dll, in this case “program2” then the .dll will already be loaded and its entry points made available.

There are a couple of methods that can be used to preload a .dll whose main entry point has not yet been called when working with managed code.

First is by setting a procedure-pointer variable to the entry of the .dll name.

Add a variable called pp to the working-storage section of program1.cbl and then add the set statement as show below before the existing call statement.

```
program-id. Program1 as "managedmain.Program1".

data division.
working-storage section.
01 myparams     pic x(20) value "from prog1".
01 pp           procedure-pointer.
procedure division.
    set pp to entry "program2"
    try
       call "prog2" using myparams
    catch ex as type Exception
       display ex::Message
    end-try
    goback.

end program Program1.
```

The set statement will preload “program2.dll” and make any entry points in it visible to the COBOL run-time system. Remember that this will only work if the PATH in the app.config file includes the folder where program2.dll resides.

Press F11 to start debugging and step through the call statement to show that it now works correctly.

The second method to preload a .dll is to use the Micro Focus Entry Point Mapper or MFENTMAP. This is more complicated to configure than simply using a procedure-pointer but we will include it here for the sake of completeness.

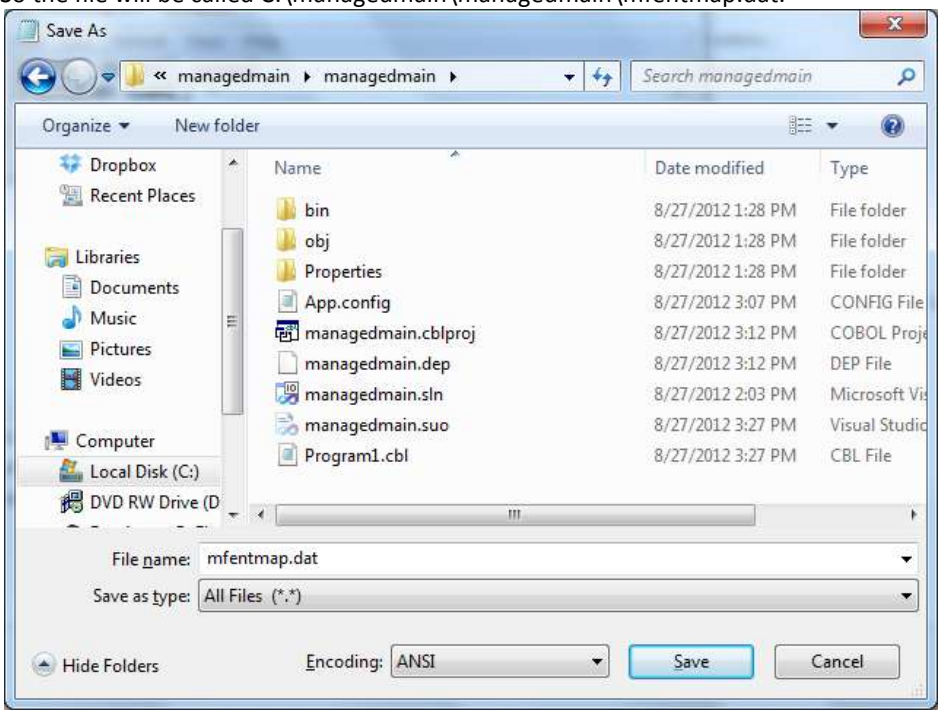
First, comment out the set statement in our current program by placing an asterisk in column 7 of its source line as shown below so that the program2.dll will not be preloaded.

```
program-id. Program1 as "managedmain.Program1".

data division.
working-storage section.
01 myparams     pic x(20) value "from prog1".
01 pp           procedure-pointer.
procedure division.
*     set pp to entry "program2"
      try
         call "prog2" using myparams
      catch ex as type Exception
         display ex::Message
      end-try
      goback.

end program Program1.
```

Open up Notepad or any text editor and create a file containing the following three lines:

[ENTRY-POINT] prog2
[PROGRAM-NAME] *
[SUBPROGRAM-NAME] program2

Save this file in your managedmain project folder using the name “mfentmap.dat”.
If using Notepad, ensure you change the file type to All Files so that it will not add the extension .txt to the file.
So the file will be called C:\managedmain\managedmain\mfentmap.dat.



When using MFENTMAP you would create the three entries shown in the file for each of the entry points that you would like to make known to the run-time system.

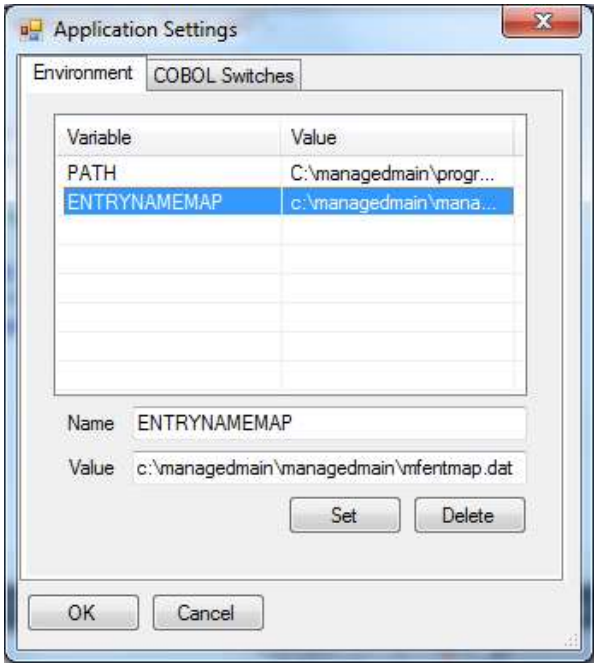
[ENTRY-POINT] prog2 - This is the name of the entry point used in the call statement.
[PROGRAM-NAME] * - This is the name of the calling program. Use * to mean any program.
[SUBPROGRAM-NAME] program2 - This is the name of the program that contains the entry point.

In our case when calling “prog2” the run-time system will first load “program2” if required in order to find “prog2”.

To complete the setup we must set the environment variable ENTRYNAMEMAP to point to the location of the mfentmap.dat file. In managed code it is not necessary to set COBCONFIG so we can ignore that step.

Right click on the app.config file in Solution Explorer and select Edit.

Add the new environment variable ENTRYNAMEMAP with the value of the mfentmap.dat file that we saved previously. Press Set and then OK to Save it.



Start debugging by pressing F11 and step through the call statement. “prog2” will now be found via mfentmap.dat.

In managed code there is a third method that can be used to preload a class library assembly.
To reset the project so mfextmap.dat will not be used, right click on app.config and select Edit again.

Remove the ENTRYNAMEMAP environment variable by highlighting it and clicking Delete. Press OK to save it.

Double click on app.config in Solution Explorer to open the XML file up in the Editor.
We need to add two new sections to the existing file. They are highlighted below:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!--The following code declares a section group for application configuration -->
    <sectionGroup name="MicroFocus.COBOL.Application">
      <section name="Switches" type="System.Configuration.NameValueSectionHandler" />
      <section name="Environment" type="System.Configuration.NameValueSectionHandler" />
      <sectionGroup name="Interop">
        <section name="PreLoad" type="System.Configuration.NameValueSectionHandler" />
      </sectionGroup>
    </sectionGroup>
    <!--The following code declares a section group for run-time configuration -->
    <sectionGroup name="MicroFocus.COBOL.Runtime">
      <section name="Tunables" type="System.Configuration.NameValueSectionHandler" />
      <section name="Switches" type="System.Configuration.NameValueSectionHandler" />
    </sectionGroup>
  </configSections>
  <MicroFocus.COBOL.Application>
    <Switches />
    <Environment>
      <add key="PATH" value="C:\managedmain\program2\bin\Debug;%PATH%" />
    </Environment>
    <Interop>
      <PreLoad>
        <add key="program2.dll" value="managed"/>
      </PreLoad>
    </Interop>
  </MicroFocus.COBOL.Application>
</configuration>
```

Save the file using the Save icon on the toolbar and then close app.config.

Press F11 to step thru the application and the call to “prog2” should now be resolved by using the Interop PreLoad section of app.config.

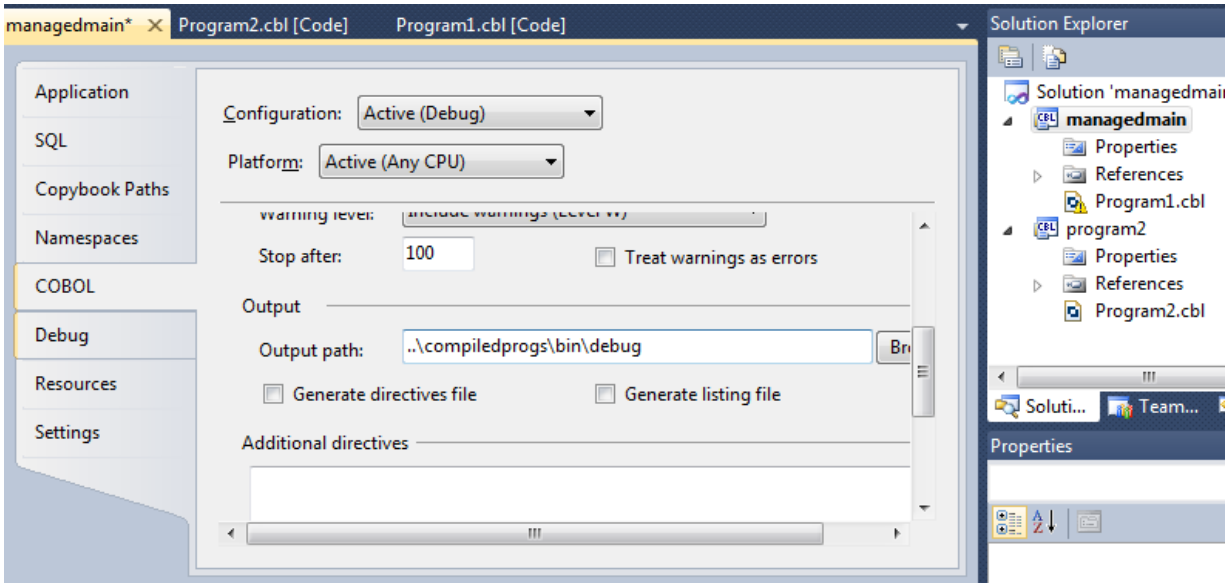
If you have a large number of Class Library projects in your solution it may become a hassle to have to set the PATH to include the output folders of every project. In this case it may be advantageous to change the output folders of all projects to point to a common location such as the output folder of the main application or a new common folder. You must remember that when doing so you must change the output folder for each build type as these specify different locations.

Let’s give this a try.

First right click on the app.config file under Solution Explorer and select Delete to remove it from the project. Uncomment the set pp to entry “program2” statement in program1.cbl so that it will again be executed.

In Solution Explorer double-click on Properties under the managedmain project heading to display the Properties page below. Click on the COBOL tab to the left and scroll down until you see the entry for Output Path:

Change the current value to ..\compiledprogs\bin\debug. This will place the project output into folder c:\managedmain\bin\debug. It is best to use the relative paths like "..\" instead of hardcoding the names in case the solution is moved to another folder. Click on the save icon to save the changes.



Close the managedmain property page and open up the property page for the program2 project and make the same changes that you made to managedmain using the same Output Folder name of ..\compiledprogs\bin\debug

Save this and start debugging again by pressing F11 and "program2.dll" will be loaded by the run-time without the need for the PATH to be set because "program2.dll" now resides in the same folder as the startup program managedmain.exe.

We have now completed the section on Managed Code development. The next section will go through the same exercise using a managed code project that calls programs in a native code project.

III) Summary

We have covered a number of different scenarios here in the preceding tutorials, some of which may or may not be applicable to your particular application.

The chart below summarizes the techniques that we covered in these pages and outlines under which scenarios each can be used.

	All Native	INT/GNT	All Managed	Managed to Native P/Invoke	Native to Managed CCW
Common Output Folder	X	X	X	X	NA
PATH in app.config	X		X	X	NA
COBPATH in app.config		X			NA
MFENTMAP	X	X	X	X	NA
Cobconfig required for MFENTMAP	X	X			NA
Preload section in app.config			X		NA
Add reference to projects			X		NA
Add reference to .dlls			X	X	NA
Multiple Output Projects	X	X		X	NA
SET PROC-POINTER TO ENTRY	X	X	X	X	NA