Visual COBOL and COBOL for .NET
Language Introduction
## Contents

*Visual COBOL and COBOL for .NET Language Introduction* ...........................................4

- Introduction .........................................................................................................................4
- Visual Studio Overview ........................................................................................................4
- Starting Visual COBOL .........................................................................................................5
- Projects and Solutions .........................................................................................................6

*A First Console Application* ...............................................................................................6

- Creating a Console Application ...........................................................................................6
- Building and Executing an Application from Visual Studio .............................................7
- Building and Executing an Application from a Command Prompt ....................................7

*A First Graphical Application* .............................................................................................8

- Creating a Graphical Application .......................................................................................8
- Adding a Label Control .......................................................................................................8
- Adding a Button Control ....................................................................................................9
- Adding a Click Event Handler ...........................................................................................9
- Adding a Text Box Control ..............................................................................................10
- Tying it all Together ..........................................................................................................10

*Overview of Visual COBOL Debugging* ...............................................................................11

- Building for Debug .........................................................................................................11
- Executing with Debug .......................................................................................................12
- Breakpoints ......................................................................................................................12
- Debug Navigation ............................................................................................................13

*Migrating to Managed Code* ..............................................................................................14

- The CblDemo1 Sample Application ..................................................................................14
- Working with Data Types and Namespaces ......................................................................17
- Error Capture ....................................................................................................................22
- Working With Classes .......................................................................................................23

*Further Application Enhancements* ....................................................................................26

*Appendix* ..............................................................................................................................28

- Changing the Startup Object ...........................................................................................28
- Data Type Reference .........................................................................................................29
- Working with IntelliSense .................................................................................................30
Visual COBOL and COBOL for .NET
Language Introduction

Introduction

This document provides a short introduction to Visual COBOL for Visual Studio and the COBOL for .NET language. It introduces basic parts of the development environment, and shows some basic application benefits, features, and enhancements of COBOL for .NET.

The document is split into two parts. In the first part, you create a simple application and get a feel for the Visual COBOL for Visual Studio development environment. The first part ends with an overview of the debugging tools available with Visual COBOL. The second part starts with a small legacy application and progressively introduces Visual COBOL concepts and features into the application. This gives helps you understand how existing code can be migrated to take full advantage of Visual COBOL and COBOL for .NET.

Visual Studio Overview

Visual Studio is an integrated development environment (IDE) used to develop a wide variety of console and graphical user interface (GUI) applications. You can also use it to develop web sites, applications and services in both native and managed code.

The graphical code editor includes numerous built-in tools and accepts plug-ins for additional functionality, from both Microsoft and third party vendors. It supports different programming languages, some built-in, some installed separately. Individual language-specific plug-ins are also available providing language support for a more dedicated development environment. Visual COBOL is one such plug-in.

The COBOL for .NET Language

Visual COBOL for Visual Studio offers COBOL for .NET, which is COBOL with extensions that support the .NET managed framework. The extensions allow Object Oriented (OO) syntax support and access to available .NET class libraries.

Managed and Native Code

Native code is code that is created and compiled to low-level executable code for a specific microprocessor and operating system. It is not portable to systems with different microprocessors and operating systems.

Managed code is code that executes in a controlled runtime environment. Machines with microprocessor types different from the source code machine cannot execute the code natively without a system to manage code execution. On Windows systems, the controlled environment is the .NET Framework Common Language runtime (CLR).
On windows systems, source code is encoded in the Microsoft Intermediate Language (MSIL). The CLR manages the encoding to MSIL. Any programming language that targets the CLR produces the correct encoding for that language. Before execution, the CLR compiles the MSIL into machine dependent, or native executable code. Because the compiling is done in a managed environment, that managed code can be used by any host machine.

Starting Visual COBOL

Start Visual COBOL for Visual Studio to load the standard libraries and tools used to develop robust and reusable applications using COBOL.

- Click Start menu > All Programs > Micro Focus Visual COBOL 2010 > Visual COBOL for Visual Studio.

If prompted to set the environment, choose General development environment. The General development environment customizes Visual Studio to work with COBOL projects.

Visual COBOL starts within Visual Studio, and the Visual Studio Start page appears with up-to-date Visual Studio information as follows.
Use the File menu to open solutions and projects. The Solution Explorer appears in the right-hand pane. The Solution Explorer displays the structure of the opened solution or project.

In the Solution Explorer, double click a file name to display its contents in a separate Code Editor tab.

The Output window displays below the Code Editor tab. This is where messages from the Visual Studio Integrated Development Environment (IDE) and compiler appear. The Error List tab in the Output window shows any error messages, warnings and other messages that occur during code development or when compiling a solution or project.

**Projects and Solutions**

A solution is a container holding one or more projects that work together to create an application. A solution file has a .sln extension and is a readable text file.

A project is a packaging unit that typically builds a single library or executable. A Managed COBOL project file has a .cblproj extension and is a readable text file.

Both solution and project files should not be updated outside of Visual Studio.

**A First Console Application**

This section shows how to create a managed COBOL console application from supplied templates and extend it to create a traditional Hello World application. It also shows how to build and execute the application with Visual COBOL and how to execute the application from a command prompt.

**Creating a Console Application**

Follow the steps to create a new solution and new project. The new solution displays in the Solution Explorer. The COBOL program created from a template for the project displays in the Code Editor tab.
1. Click File > New > Project.
   The New Project dialog box appears.
2. Click COBOL > Managed in the Installed Templates pane.
   Available COBOL templates appear in the templates list.
3. Select Console Application from the list.
4. Enter ConsoleHelloWorld as the name of the project in the Name field, and click OK.
   Visual Studio creates a new solution and a project in the solution named ConsoleHelloWorld, and a
   new program template named Program1.cbl. Program1.cbl displays in a Code Editor tab. The
   current code for Program1.cbl is the template code, as shown in the following.

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

   data division.
   working-storage section.

   procedure division.
       goback.
   end program Program1.
   ```
5. Add a statement to this program to display the text "Hello World" when executed.
   To do this, place the cursor after the procedure division. line and press Tab to move the cursor to
   the start of Area B (column 12). Add the display statement into the program between the procedure
   division and the goback statement so the program looks as follows.

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

   data division.
   working-storage section.

   procedure division.
       display "Hello World"
       goback.
   end program Program1.
   ```
6. Click File > Save Program1.cbl to save the code update.

**Building and Executing an Application from Visual Studio**

Building an application in Visual Studio converts the code to MSIL. You can execute an application in
Visual Studio or from a command prompt. When executing an application, the MSIL is compiled by the
CLR into the machine code required by the host machine.

1. Click Build > Build ConsoleHelloWorld.
   ConsoleHelloWorld is the current project name. The Output window shows messages generated during
   the build.
2. Once the application has built, click Debug > Start without debugging.
   A console prompt displays the text Hello World. You are prompted to press any key to continue. This
   closes the console prompt.

**Building and Executing an Application from a Command Prompt**
To execute an application from a command prompt instead of with Visual COBOL, open a Visual COBOL command prompt and enter the command to execute the application.

1. Click **Start menu > All Programs > Micro Focus Visual COBOL 2010 > Visual COBOL Tools > Visual COBOL Command Prompt (32-bit)**. This opens a command prompt to your document directory by default. From this directory, navigate to the directory containing the application. This directory is \Visual Studio 2010\Projects\ConsoleHelloWorld\ConsoleHelloWorld\bin\debug by default. Verify that the application is in this directory.

2. To navigate to the directory containing the application, type `cd \[default document directory\] \Visual Studio 2010\Projects\ConsoleHelloWorld\ConsoleHelloWorld\bin\debug` at the command prompt and press **Enter**.

3. Type **ConsoleHelloWorld** and press **Enter**. The application starts, displays the text **Hello World** and then ends. The text **Press any key to continue...** does not appear. This text only appears if the application runs in Visual COBOL to allow you to see the console application output after completion.

**A First Graphical Application**

This section shows how to create a managed COBOL application from a template and use it to create the traditional Hello World application. It also shows how to extend the application by adding user input and button control.

**Creating a Graphical Application**

Follow these steps to create a new solution for the graphical application.

1. Click **File > New > Project**. The **New Project** dialog box displays.
2. Click **COBOL > Managed** in the Installed Templates pane. Available COBOL templates appear in the templates list.
3. Select **Windows Forms Application** from the list.
4. Type **WindowsHelloWorld** as the name of the project in the **Name** field and click **OK**. Visual Studio creates a new solution and a project in the solution named **WindowsHelloWorld**, and a new program template called **Main.cbl**. It also creates a new form and associated program named **Form1.cbl** and **Form1.Designer.cbl**. The project structure displays in the Solution Explorer, and the **Form1.cbl** form displays in a Code Editor tab.

**Adding a Label Control**

Use labels to place text on a form that names or titles another control on the form. Label text cannot be edited by a user. This includes labels for text fields, instructions and some warning indicators. Follow these steps to create and place a label on the form.

1. Click **View > Toolbox** to display the Toolbox.
2. Drag and drop a **Label** control onto the form. This creates a Label control named **label1** containing the text **label1**.
3. In the Solution Explorer, highlight the **Text** property value in the **Appearance** section of the **Properties** pane. The **Properties** pane is in the bottom right of the Visual Studio IDE under the Solution Explorer pane by default.
4. Type Hello World to change the property value and press Enter.
   The property value for the Label control updates.

Use instructions in Building and Executing an Application from Visual Studio to build and execute the application. This opens a form called form1 that displays a label named Hello World. Click on the x in the top-right corner to close the form and stop the application.

Adding a Button Control

A button is a control on a form that a user can click to perform a predetermined task. A click event handler defines the task the button performs. Use the following steps to place a button onto the Form1.cbl form.

1. Click View > Toolbox to display the Toolbox.
2. Drag and drop a Button control onto the form.
   - Tip: Visual Studio displays line guides when controls are aligned on the page.
3. In the Solution Explorer, highlight the Text property value in the Appearance section of the Properties pane.
4. Type Say Hello to change the property value and press Enter.
   The property value for the Button control updates.

Use instructions in Building and Executing an Application from Visual Studio to build and execute the application. The button displays on the form. However clicking the button has no effect as there is currently no click event handler written for it. The next exercise adds the click event handler for the button.

Adding a Click Event Handler

A click event handler is code that executes when a control is clicked. The code can be a single action or several actions, including navigation to a different page. The following steps add the click event handler to the button control.

1. Double-click the button.
   The code for Form1.cbl appears. Code for a method named button1_click generates and appears in a separate Code Editor tab. The method is given the same name as the button with _click appended. This method contains the code to be executed when clicking the button, as shown in the following.

```visualcobol
working-storage section.
method-id NEW.
procedure division.
   invoke self::InitializeComponent
go back.
end method.

method-id button1_Click final private.
procedure division using by value sender as object e as type System.EventArgs.
end method.
end class.
```
2. Add a statement to move Hello World into the text property of the label named label1 previously created. Currently label1 already contains a value, so this is removed. To do this, place the cursor after the procedure division line and press Enter.
   This creates a new line and puts the cursor to the start of Area A (column 8).
3. Press the Tab key to move the cursor to the start of Area B and type the event handler code as follows.
   ```visualcobol
   set self::label1::Text to "Hello World"
   ```
This code moves the text **Hello World** to the Text property of the previously created label1 label. This removes the current label1 value.

**Tip:** Visual COBOL uses IntelliSense to suggest text completions as you type. See *Working With IntelliSense* for more information.

The code should be as follows.

```cobol
working-storage section.
method-id NEW.
procedure division.
    invoke self::InitializeComponent
goback.
end method.

method-id button1_Click final private.
procedure division using by value sender as object e as type System.EventArgs.
    set self::label1::Text to "Hello World"
end method.
end class.
```

4. Click on the tab labeled **Form1.cbl [Design]** at the top of the code editor to display the form in the Code Editor tab.

5. Click on the **Label** control.
   The control's properties appear in the **Properties** pane.

6. Clear the **Text** property value and press **Enter**.
   The property value updates.

Use instructions in *Building and Executing an Application from Visual Studio* to build and start the application. When started, the form opens and displays a button with the text *Say Hello*. Click the button to cause **Hello World** to display on the form. Click the x in the top-right corner to close the form and stop the application.

### Adding a Text Box Control

A text box is a control that lets you enter textual content. Follow these steps to place a text box onto the form.

1. Click on **Form1.cbl [Design]** tab to display the form designer window.
2. Click **View > Toolbox** to display the Toolbox.
3. Drag and drop a **Text box** control onto the form.
   A text box control is added to the form.

Use instructions in *Building and Executing an Application from Visual Studio* to build and execute the application, which displays a text box and a button. At this point, no matter what you enter into the text box, nothing is done with it.

### Tying it all Together

The application now has **Label**, **Button** and **Text box** controls and a click event handler. All that remains is to display a text label for the entry field and add an event to the button to do something with the entry field. To do this, perform the following steps.

1. **Add a Label** control to the form.
   Place the label near the text box to give the user an indication of what to type into the text box.
2. In the Solution Explorer, highlight the Text property value in the Appearance section of the Properties pane.

3. Type First Name to change the property value and press Enter. The property value for the Label control updates.

4. Double-click the Button control. The code for Form1.cbl appears. This code contains the click event handler for the button that was updated in the exercise to add a click event handler, but with an additional method for the new label as shown in the following.

```
working-storage section.
method-id NEW.
procedure division.
   invoke self::InitializeComponent
goback.
end method.

method-id button1_Click final private.
procedure division using by value sender as object e as type System.EventArgs.
   set self::label1::Text to "Hello " & self::textBox1::Text
end method.

method-id label2_Click final private.
procedure division using by value sender as object e as type System.EventArgs.
end method.
end class.
```

5. Replace the previously added code so it reads as follows:

```
set self::label1::Text to "Hello " & self::textBox1::Text
```

This concatenates the constant text Hello and the contents of the text box textBox1, and places the results into the label1 label when clicking the button.

Build and execute the application to display a text box labeled First Name and a button. Type Fred into the text box and click the button. The text Hello Fred appears.

**Overview of Visual COBOL Debugging**

Writing and building an application like the Hello World example is very simple. The chance of creating an error by spelling a word incorrectly is low, especially with the dynamic checking being done by IntelliSense within Visual COBOL.

As an application gets more complex the likelihood of an application error gets higher very quickly. Visual COBOL provides tools to help the cause of these errors should they occur. This section shows how to build and run using debugging and how to use the debugging features.

**Building for Debug**

To debug an application, ensure that the debug configuration is selected. The Solution Configurations listbox on the Visual Studio toolbar shows the current configuration type. To change the configuration, select the appropriate option from the list box.
Executing with Debug

Visual COBOL provides a wide variety of application debugging tools that let you to track down any application errors. These tools are available during the application execution when execution is configured for debugging. To configure for debugging:

1. Click **Debug > Start Debugging**.
   Visual COBOL automatically rebuilds the application if any source has changed since the previous build, and the application starts.
2. (optional) Set breakpoints to make the application stop at specific points during execution.

Breakpoints

A breakpoint is a placeholder placed in the source code. When a breakpoint is encountered during application execution, the application pauses and the line containing the breakpoint highlights. While the application is paused you can retrieve information about current variables, set additional breakpoints or interrogate application behavior.

Tip: Pausing the application during debug execution is also known as break mode.

Setting a Breakpoint

Perform the following steps to set a breakpoint:

1. Click the line where you want to pause the application execution.
2. Click **Debug > Toggle Breakpoint**.
   A breakpoint is created. A red dot displays in the left margin next to the code line where execution is set to pause.
**Note:** If a breakpoint is set on a non-executing code line, the application pauses at the next executing code line.

**Disabling a Breakpoint**

While debugging you might want to ignore a set breakpoint, but keep it in place for later use. This cannot be done by using the menu options at the top of the Visual COBOL window, but instead via the context menu as follows:

1. Right-click on the line of code beside the breakpoint to display the context menu.
2. Select **Breakpoint > Disable Breakpoint**.
   The selected breakpoint disables as indicated by the red circle in the left margin. To re-enable a breakpoint select **Breakpoint > Enable Breakpoint** from the context menu.

**Removing a Breakpoint**

Once you have no further use for a breakpoint, remove it by doing the following.

1. Select the breakpoint by clicking on the corresponding code line.
2. Click **Debug > Toggle Breakpoint**.
   The selected breakpoint is removed. Repeating these steps alternatively sets and removes a breakpoint.

**Debug Navigation**

Dynamically interrogating and monitoring data movement during application execution is a powerful and useful feature within the Visual COBOL environment. The current line of execution displays in the Code Editor tab during application debugging. As the application executes, the display steps into or over the methods as directed, highlighting the next line of code to be executed. The list below shows the navigational options available during debugging.

- **Stepping Over** (**Debug > Step Over**)  
  Executes the current line and progresses to the next line of code in the current code block. Any methods the debugger encounters execute completely. If the end of the code block is reached, control returns to the calling block.

- **Stepping Into** (**Debug > Step Into**)  
  Executes one code command at a time. For a simple numeric operation, the debugger behaves as if **Step Over** is being used. For more complex code, the debugger passes control to any called method and executes its first code line, steps through the rest of the method before returning to the calling method. The Code Editor tab shows the executing code line.

- **Stepping Out** (**Debug > Step Out**)  
  Completes the current method and stops at the next line of the calling method.

- **Restarting the debugger** (**Debug > Restart**)  
  Restarts debugging at the first line of the application.

- **Stopping the debugger** (**Debug > Stop Debugging**)  
  Terminates the debugger.

- **Setting the next statement.**
  Allows you to set the next statement to be executed. To do this, right-click the line of code to be executed and select **Set Next Statement**.
Migrating to Managed Code

This part of the Language Introduction illustrates some of the major benefits of using COBOL with Windows forms when the application is built as managed code. It shows how to take advantage of tools and techniques to speed up your software development and produce efficient and powerful platform-independent applications.

Using a simplified procedural console application, the next sections show the capabilities and benefits of using COBOL as part of a managed solution. As you progress through the sections, the code adds supported Object Oriented (OO) syntax to access available extensions.

The sections use a sample solution named CblDemoSolution, which contains a project named CblDemoProject. CblDemoProject contains programs which relate to the following sections, starting with CblDemo1. The project also includes a class definition named Person, which is used in the final section. To execute the programs, ensure that the startup object is set correctly for the project. To change the startup object, see Changing the Startup Object in the appendix.

The CblDemo1 Sample Application

CblDemo1 is a console application that asks for name, date of birth and today's date and displays the number of days until your next birthday. It is a complete application in that it can be executed.

Use this sample application to complete exercises in this section. The solution has been left deliberately poor in design so it can be extended with your own functionality. The application is reproduced below in its entirety for ease of reference without complete code formatting. Comments are excluded to simplify maintenance and readability.

```cobol
IDENTIFICATION DIVISION.
PROGRAM-ID. CblDemo1.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
  01 today.
    03 today-month PIC 99.
    03 today-day PIC 99.
  01 dob.
    03 dob-month PIC 99.
    03 dob-day PIC 99.
  01 test-date.
    03 test-month PIC 99.
    03 test-day PIC 99.
  01 max-month PIC 99.
  01 max-days PIC 99.
  01 total-days PIC 999.
  01 full-name PIC X(30).
  01 display-output PIC X(60).
  01 greeting-output PIC X(60).
  01 show-non-usage PIC X(12).
  01 txt1 PIC X(10) value "Hello Mr ".
  01 diff-output.
    03 txt2 PIC X(11) value " you have ".
    03 output-days PIC ZZ9.
    03 txt3 PIC X(40) value " days until next birthday".
PROCEDURE DIVISION.
  DISPLAY "Enter your full name"
  ACCEPT full-name
  DISPLAY "Enter your date of birth in mmd format"
  ACCEPT test-date
```

Visual COBOL and COBOL for .NET Language Introduction
PERFORM validate-test-date
MOVE test-date TO dob
DISPLAY "Enter the date today in mmdd format"
ACCEPT test-date
PERFORM validate-test-date
MOVE test-date TO today
PERFORM add-day-difference
PERFORM add-mth-difference UNTIL (max-month = DOB-month)
MOVE total-days TO output-days
STRING txt1, full-name DELIMITED BY SIZE INTO greeting-output
DISPLAY greeting-output
STRING full-name, diff-output DELIMITED BY SIZE INTO display-output
DISPLAY display-output
STOP RUN
.

validate-test-date SECTION.
  IF (test-day = 0)
    STOP "Invalid day field"
    STOP RUN
  END-IF.
  IF ((test-month = 0) OR (test-month > 12))
    STOP "Invalid month field"
    STOP RUN
  END-IF.
  MOVE test-month TO max-month.
  PERFORM set-max-days.
  IF (test-day > max-days)
    STOP "Invalid day field"
    STOP RUN
  END-IF.

add-day-difference SECTION.
  MOVE today-month TO max-month
  PERFORM set-max-days
  IF (DOB-day = today-day)
    MOVE ZERO TO total-days
  ELSE
    IF (DOB-day > today-day)
      COMPUTE total-days = DOB-day - today-day
    ELSE
      COMPUTE total-days = (max-days - today-day) + DOB-day
    END-IF
  END-IF

add-mth-difference SECTION.
  PERFORM set-max-days
  ADD max-days TO total-days
  PERFORM increment-test-month
  .

increment-test-month SECTION.
  ADD 1 TO max-month
  IF (max-month > 12)
    MOVE 1 TO max-month
  END-IF
  .

unused SECTION.
EXIT SECTION.
Native Code Support

Existing native COBOL code can be built and executed in Visual COBOL without additional changes. This means rapid migration of existing application code to the new environment, since an application can quickly run alongside original applications.

With Visual COBOL, some procedural COBOL coding syntax is no longer required for applications to run. Redundant lines are still recognized but are ignored during the application build. In CblDemo1, some redundant lines of code have been included as an illustration and are listed below, but will be removed elsewhere in this document.

- IDENTIFICATION DIVISION
- ENVIRONMENT DIVISION
- DATA DIVISION
- WORKING-STORAGE SECTION

COBOL Margins

Visual COBOL displays the set COBOL margins. A gray margin for the sequence number and the indicator displays when the format is set as Variable. An additional margin displays to the right when the format is set as Fixed.

Changing COBOL Margins

The following steps show you how to change COBOL margins in Visual COBOL:

1. Right-click on the project in the Solution Explorer and select Properties. The project properties display in the Properties page.
2. In the Properties page, select the COBOL tab. The COBOL tab properties appear.
3. Use the Source Format list box to select the appropriate COBOL margin type. This sets the displayed margins for COBOL code in Visual COBOL.
Use the Source Format setting to specifying margin settings instead of the SOURCEFORMAT compiler directive.

**Unused Member Identification**

Visual COBOL helps you quickly find unused variables and sections in COBOL code.

By default, unused variable names in Visual COBOL appear in gray text. The text color changes to red by default for variable names used in the application.

Variable name text color can help to determine which variables can be removed completely. CblDemo1 includes an example of an unused variable name called show-non-usage.

Identify an unused section by hovering the cursor over the section name. This displays a ToolTip indicating the number of references. If No References displays in the ToolTip, then the section is a candidate for removal. CblDemo1 includes the section unused with no references.

**Working with Data Types and Namespaces**

This section describes available managed code data types and briefly discusses what namespaces are and how they are used. This section also:

- Uses CblDemo1 to show the DateTime data type and how to optimize DateTime functionality.
- Uses CblDemo3 to show the String data type and how to optimize String functionality.

In the sample solution, CblDemo2 contains the completed code based on DateTime data type exercises completed in this section. CblDemo3 contains the completed code based on String data type exercises.
completed in this section. See the Changing the Startup Object appendix for information on starting the programs.

**COBOL Data Types**

Managed COBOL supports both .NET data types as well as traditional COBOL data types. Frequently used types like Strings and Integers have synonyms in COBOL. You can reference other types using the keyword `TYPE` followed by the type name as shown.

```
01 dob                  type DateTime.
01 display-output            String.
```

See the appendix for a data type reference chart.

**.NET Data Types**

The .NET framework has strong data types. This means objects of one type cannot be freely exchanged with another type, and implicit and explicit conversions allow for data conversion. All data types derive from a `System.Object` data type, which can be assigned any object or variable.

**Namespaces**

A namespace is a grouping containing related data types.

A namespace references its available classes and methods. For example, the field `dob` is defined as being of type `DateTime` which is grouped into the namespace `System`:

```
01 dob                type System.DateTime.
```

The .NET Framework includes a variety of namespaces such as Accessibility, System.Data, System, System.IO, and System.Runtime. All namespaces include related classes and methods.

Visual COBOL imports the namespace `System` by default, so there is no need to prefix classes from this namespace. This means the example above can also be written as:

```
01 dob                type DateTime.
```

**Removing Namespace Prefixing**

Add namespaces directly to your project properties so that you don't need to type the fully qualified namespace when developing code. The `System` namespace is included by default. To add other namespaces to your project properties:

1. Right-click on the project name in the Solution Explorer and select Properties. The project properties display in the Properties page.
2. In the Properties page, select the References tab. The References tab properties appear.
3. Scroll down to the Imported Namespaces list. This list contains all available namespaces.
4. Select the check box next to each namespace in the list to directly import into the project build. This indicates whether the namespace is directly imported in the project build. Namespaces not selected require the full namespace prefixing in code.

Simplify your code by removing namespace prefixes included in the Imported Namespaces list. Use the Imported Namespaces project setting to add namespace to a project, as it supercedes the `$SET ILUSING` compiler directive.

**DateTime Data Type**

In the `CblDemo1` sample application, the `today` and `dob` fields in the data definition can be replaced by the `System.DateTime` data type to provide standard date and time functionality. Since the `System`
namespace is included in the imported namespaces list by default, the data definition for `today` and `dob` can be replaced with simply `DateTime`. Types derived from `System.Object` must be declared at level 01. Since `DateTime` does not have a synonym in COBOL, the `type` keyword must precede it.

In the sample application, the variable `today` contains the current date. The value is initialized using the built-in `DateTime` method `Today()`.

The following shows the changed variable definitions:

```cobol
01 today type DateTime value DateTime::Today().
01 dob type DateTime.
01 test-date.
  03 test-month PIC 99.
  03 test-day PIC 99.
01 max-month PIC 99.
01 max-days PIC 99.
01 total-days PIC 999.
01 full-name PIC X(10).
01 display-output PIC X(60).
01 greeting-output PIC X(60).
01 txt1 PIC X(10) value "Hello Mr ".
01 diff-output.
  03 txt2 PIC X(11) value " you have ".
  03 output-days PIC ZZ9.
  03 txt3 PIC X(40) value " days until next birthday".
```

In this example, a variable derived from `System.Object` provides the contents of the variable by default.

You can use data types of a namespace to replace standard variable data definitions without having to make further updates to existing procedure division code.

Since retrieving the current date for input is no longer required, the lines of code in the sample application that previously defined the `today` and `dob` fields can be removed.

**DateTime Object Initialization**

The `dob` field is assigned the `DateTime` object value using validated variables as part of a constructor. A constructor is a method that is called when it's object is initialized, and is typically used to initialize the object's parameters.

In the sample application, the year parameter is provided by the `year` field from the `today` object as shown below:

```cobol
set dob to new type DateTime(today::Year, test-month, test-day)
```

**Note:** Commas separating the parameters are optional.

**DateTime Object Comparison**

`DateTime` objects provide a range of methods allowing for date and time manipulation and comparison. Using the difference between the `today` and `dob` fields can be obtained in a much simpler way.

To ensure that negative results are not returned, compare dates to ensure that the `dob` field contains a value that is not less than the value in `today`. The `DateTime` object provides a `CompareTo()` method which returns zero if the dates are identical and +1 or -1 to indicate a difference.

The `DateTime` object provides methods which add and subtract individual elements of a date, automatically adjusting the other elements to ensure that the date remains valid.

In the demonstration application the year is incremented by one if the `dob` date is less than today's date to ensure that it is in the future. The method to be used is `AddYears()` which accepts an integer as a parameter.
In the following line, the if statement equates to true if today is greater than dob. If true, the year element of dob increments by one.

```cobol
if today::CompareTo(dob) > 0
    set dob to dob::AddYears(1)
end-if.
```

Use CompareTo() method as a condition to a perform statement to write a single section to determine the number of days between two DateTime objects, as shown in the following.

```cobol
perform add-to-total until today::CompareTo(dob) = 0
```

The add-to-total section increments the variable total-days and today. The add statement increments the total-days variable. The AddDays() method increments the day element of the today variable.

```cobol
add-to-total SECTION.
    add 1 to total-days
    set today to today::AddDays(1).
```

With the add-to-total section in place, the sections add-day-difference, dd-mth-difference and increment-test-month are now redundant and can be removed.

### String Data Type

You can change the sample application to take advantage of more managed COBOL features. Change the PIC X variable definition for display-output to use the string data type to improve string manipulation.

#### String Data Type Additions

In the CblDemo3 sample application, you can replace the display-output variable in the data definition with the System.String data type to provide standardized text manipulation functionality. You can also convert the constant text variables to show that initial values have not changed. Types derived from System.Object must be declared at level 01, so the group variable diff-output can be removed. You can also remove the output-days variable created to display the zero suppressed number of days as it is no longer required.

Defining variables as type String removes the need to know the length of the constant. The object automatically sizes to store the variable.

Note: String has a synonym in COBOL so does not need to be preceded with the type keyword.

The following shows the changed variable definitions, with temp1 and greeting-output added for later use:

```cobol
01 today             type DateTime value type DateTime::Today().
01 dob               type DateTime.
01 test-date.
    03 test-month    PIC 99.
    03 test-day      PIC 99.
01 max-month         PIC 99.
01 max-days          PIC 99.
01 total-days        PIC 999.
01 templ             PIC 99.
01 full-name         PIC X(30).
01 display-output    String.
01 greeting-output   String.
01 txt1              String value "Hello Mr ".
01 txt2              String value " you have ".
01 txt3              String value " days until next birthday".
```

Defining variables as type String does not require updates to procedure division code. This is because a variable derived from System.Object provides the contents of the variable by default.
The sample application copies text captured from the console into the `String` field `display-output` to take advantage of available methods.

**String Functionality**

Defining the variables as type `String` does not require amendments to procedure division code as default behavior of a variable derived from `System.Object` provides the contents of the variable.

The sample application copies the text captured from the console into the `String` field `display-output` to take advantage of available methods.

**Displaying The Surname**

The application displays an informal salutation. This exercise shows how to change this to display a surname.

The application displays the contents of the field `txt1` followed by the surname and lastly a comma (,). Assuming that the surname is the last part of the full name preceded by a space character, the position of the space just before the surname needs to be found.

The `String` object lets you use a method to retrieve a substring. To set up the code to take advantage of this method, first set up the `display-output` variable to receive the substring. The `Trim()` method removes all leading and trailing spaces from the string. Use the statement shown below to populate the `greeting-output` variable with the substring.

```plaintext
set display-output to full-name
set greeting-output to display-output(???::Trim())
```

The example still needs to determine the start position of the surname. Use the `String` object to determine the last occurrence of a provided Unicode character. To do this, use the `LastIndexOf()` method with the first position of the string set to zero. The method takes two parameters separated by a colon (:). The first parameter is the start position of the substring, with one being the first position in the original string. The second parameter is the length of the string to retrieve. If not provided, the remainder of the string is returned.

Use this to populate the `temp1` variable so that the final space character position is just before the surname. Increment the `temp1` variable value by one to give the surname start position.

```plaintext
set display-output to full-name
set temp1 to display-output::Trim()::LastIndexOf(' ')
set greeting-output to display-output(temp1 + 1::Trim())
```

To display the information, use the `display` keyword followed by the variables in sequence:

```plaintext
set display-output to full-name
set temp1 to display-output::Trim()::LastIndexOf(' ')
set greeting-output to display-output(temp1 + 1::Trim())
display txt1 greeting-output ","
```

**Displaying the First Name**

Now that the code is updated to display the surname, this exercise shows how to update the code with managed COBOL to display the following in order:

- The first name
- The `txt2` variable value
- The number of days until the next birthday
- The `txt3` variable value.

It is assumed that no leading spaces have been entered.

The previous exercise uses the `substring` method to display the surname as part of the salutation. This exercise uses it to retrieve the first name from the beginning of the string. The exercise assumes no leading spaces are entered in the string.
With the length of the first name string to be determined, the display statement is as follows:

display display-output(1:???) txt2 total-days txt3

Use the `IndexOf()` method to determine the position of the first provided Unicode character, with the first position of the string being zero. Locating the string's first space position gives the length of the first name. The following populates the `temp1` variable with the first space in the `display-output` string:

```
set temp1 to display-output::IndexOf(' ')
```

Putting these two statements together results in displaying the first name with the contents of `txt2`, `total-days` and `txt3` as shown below:

```
set temp1 to display-output::IndexOf(' ')
display display-output(1:temp1) txt2 total-days txt3
```

The display statement concatenation accepts numeric and string values.

## Error Capture

This section looks at the basic error capturing in the sample application and shows how use the `Try` statement to enhance error capturing. Deciding what to do once an error is captured typically depends on the application. This exercise displays a message when capturing an error.

> **Note:** CblDemo4 contains completed code once for changes discussed in this section. To execute the program see Changing the Startup Object appendix.

The `validate-test-date` section in the sample application validates that the supplied birth day and month are in range. It validates that the birth day is not zero and that the birth month is a number from 1 to 12. The `set-max-days` then retrieves the maximum days in the birth month and validates that the entered birth day is within range:

```
validate-test-date SECTION.
    IF (test-day = 0)
        STOP "Invalid day field"
        STOP RUN
    END-IF
    IF ((test-month = 0) OR (test-month > 12))
        STOP "Invalid month field"
        STOP RUN
    END-IF
    MOVE test-month TO max-month.
    PERFORM set-max-days.
    IF (test-day > max-days)
        STOP "Invalid day field"
        STOP RUN
    END-IF.
```

The section `set-max-days` is straightforward code. However, it contains no code to manage leap year determination:

```
set-max-days SECTION.
    EVALUATE max-month.
        WHEN 1 MOVE 31 TO max-days
        WHEN 2 MOVE 28 TO max-days
        WHEN 3 MOVE 31 TO max-days
        WHEN 4 MOVE 30 TO max-days
        WHEN 5 MOVE 31 TO max-days
        WHEN 6 MOVE 30 TO max-days
        WHEN 7 MOVE 31 TO max-days
        WHEN 8 MOVE 31 TO max-days
        WHEN 9 MOVE 30 TO max-days
        WHEN 10 MOVE 31 TO max-days
        WHEN 11 MOVE 30 TO max-days
        WHEN 12 MOVE 31 TO max-days
```

---

Visual COBOL and COBOL for .NET Language Introduction
This is an acceptable way to handle errors. However, the DateTime data type also validates the entered values as the data type is created, including leap year determination, and throws an exception if they are in error.

**Exception Handling With The DateTime Data Type**

The following code line in the sample application creates a DateTime instance from user-supplied information:

```cobol
set dob to new type DateTime(today::Year test-month test-day)
```

In the preceding code line, `validate-test-date` validates that the supplied information is in range. Replace this line using data returned from the DateTime data type. The validation is stored as an ArgumentException data type, with zero indicating success. The code includes a line that sets the ArgumentException data type to a variable access the validation result.

```cobol
01 argException     type ArgumentException.
```

Wrapping the constructor in a try statement informs the application to check the validation result. If any exceptions are raised, the first catch statement matching the thrown exception executes. The catch statement below uses the variable result to tell the application what to do when the user supplied data causes an exception.

```cobol
try
  set dob to new type DateTime(today::Year test-month test-day)
  catch argException
    display "I couldn't understand your date entry"
    display argException::Message
    stop run
end-try
```

The above code replaces the functionality provided by `validate-test-date` and `set-max-days` so that code can be removed. The redundant variables `max-month` and `max-days` can also be removed.

**Working With Classes**

The sample application contains functions and information useful to other applications. But rather than copying code everywhere it's functionality is required, create a class containing the functionality. The class can then be used by other applications.

**Note:** In the sample solution, `CblDemo5` and the class `Person` contain the completed code based on exercises completed in this section. See the Changing the Startup Object appendix for information on starting the programs.

**Creating a Class**

A class is a type definition containing fields and methods, usually representing a tangible thing or action. Since the sample application includes code that creates, stores and displays information about an individual, create the `Person` class for that information.

1. Right-click on the project name in the Solution Explorer window and select **Add > New Item**.
   The Add New Item window appears.
2. Click **COBOL Items > Managed** in the Installed Templates pane, then select **COBOL Class** from the list.
3. Type `Person.cbl` in the Name field. The Name indicates both the name of the file and class to create. This denotes the name of the file and class being created.
4. Click **Add** to create a file called *Person.cbl* containing the empty *Person* class template. The class template is very similar to the program template and contains an empty method.

```cobol
class-id CblSolution.Person.
working-storage section.
method-id InstanceMethod.
local-storage section.
procedure division.
  goback.
end method.
end class.
```

5. Remove *CblSolution* from the class identifier located on the first line of the class template. The solution name prefixes to the class identifier by default to ensure unique class names across the application. This is not required for the sample application and can be removed.

**Populating the Class**

Copy the relevant fields and functionality from existing code to the *Person* class. Ensure only to copy relevant code to avoid confusion when using the class in other applications.

1. Copy the relevant variables into the class.
   In the COBOL code, *dob* and *full-name* directly relate to a person. Copy these into the class, converting the full name to a *String* data type to ensure correct sizing, as shown in the following.

```cobol
working-storage section.
  01 full-name   string.
  01 dob         type DateTime.
```

2. Update the code to allow access to the attributes.
   Class variables (also known as attributes) include a property that indicates variable accessibility outside the class. The property is set to **private** by default. This indicates no external access. To allow access to the variables outside the class, set the property to **public** and name it for referencing as shown in the following.

```cobol
working-storage section.
  01 full-name   string public property as "FullName".
  01 dob         type DateTime public property as "DateOfBirth".
```

3. Create a constructor to allow the class fields to be used.
   A constructor is a method that instantiates and initializes the fields defined in the class. To ensure that the *Person* class always contains a name and date of birth, supply these values as parameters in the constructor as shown in the following. Without the parameters, the method throws an error which can be captured by the **Try** statement.

```cobol
method-id New public.
procedure division using by value fullname as string
  birthday as type DateTime.
    set full-name to fullname
    set dob to birthday
  goback.
end method.
```

4. Remove the default method.
   Creating a class also creates a method template. The method template is not required, so remove the following code.

```cobol
method-id InstanceMethod.
local-storage section.
procedure division.
  goback.
end method.
```
The *Person* class that contains the full name and date of birth of a person, which can only be successfully created if both the full name and date of birth are provided.

## Adding Methods to the Class

Now that the *Person* class is available, create a method for the functionality that relates to a person. The functionality that relates to a person is as follows:

1. **Number of days until next birthday functionality.**
   
   The application currently takes the date of birth and adds the birth year to ensure the stored date is the next birthday. Then the current date is incremented until the two dates match, incrementing a counter in parallel. Create a method that provides the next birthday, then use that date to determine the number of days until the next birthday.

   The following shows a method that provides the next birthday. It defines a local variable called *today* which provides today's date. It also defines a temporary variable called *return-value*. Because this is part of the method definition and includes the variable type, the run-time automatically creates this variable. Change the *set* statement from the original code to use *return-value*:

   ```
   method-id NextBirthday.
   01 today     type DateTime value type DateTime::Today().
   procedure division returning return-value as type DateTime.
      set return-value to new DateTime(today::Year, dob::Month, dob::Day)
   end method.
   ```

   This method provides the date of the next birthday. Create a second method used compare it with the current date and return the number of days.

   To do this create a variable to provide today's date. A temporary variable is created to populate the number of days. To use the method *NextBirthday()* defined in the current class, prefix it with the keyword *self*. This provides a *DateTime* object containing the date of the next birthday. *DateTime* has the method *Subtract* that returns the difference between it's contents and a provided *DateTime* object as a *TimeSpan* object. The *TimeSpan* object has a method to return the number of days. As a result, the method is coded as follows.

   ```
   method-id DaysUntilBirthday.
   01 today     type DateTime value type DateTime::Today().
   procedure division returning return-value as binary-long.
      set return-value to self::NextBirthday()::Subtract(today)::Days
   end method.
   ```

   Since there are two methods created that containing the *today* local variable, it might be tempting to make this a field on the class so it can be re-used. As it stands, the variable value is the actual date and time when the method is used. If moved, the value must be re-initialized before it can be used with confidence. Another alternative could be to define *DateTime* as part of the method call, which results in the following code.

   ```
   method-id DaysUntilBirthday.
   procedure division returning return-value as binary-long.
      set return-value to self::NextBirthday():Subtract(type DateTime::Today()):Days
   end method.
   ```

2. **Surname functionality.**
   
   The application extracts the entered surname and displays it as part of a phrase. As the surname is always part of the a person's name, Create a method to return it as a String. The offset needs to be determined, so define a temporary variable to hold the offset. Then use it to evaluate the data to return. To do this, use existing lines of code from the application and update it to populate the temporary variable:

   ```
   method-id FamilyName.
   01 temp1   PIC 99.
   procedure division returning return-value as type String.
      set temp1 TO full-name::Trim():LastIndexOf(' ')
   ```
set return-value to full-name(temp1 + 1:):Trim()
end method.

3. First name functionality.
   As with the surname, the sample application extracts entered first name and displays it as part of a
   phrase. Create a method to return the first name as a String
   Use existing code from the sample
   application as a basis:

   method-id FirstName.
   01 temp1             PIC 99.
   procedure division returning return-value as type String.
   set temp1 TO full-name::IndexOf(' ')
   set return-value to full-name(1:temp1)
end method.

   You can combine the two lines above for efficiency as shown in the following, keep the code as two
   lines for readability:

   method-id FirstName.
   procedure division returning return-value as type String.
   set return-value to full-name(1:full-name::IndexOf(' '))
end method.

Using the Person Class

With the Person class created, update the application to access it.

1. Create a variable to store the instance of the class.
   01 person1           type Person.

2. Change the variable test-date to accept the birth year and associated prompt.
   Now that we are going to store a persons’ date of birth we need to accept it from the user.

3. Instantiate the variable in the code.
   This is done in the same way as the dob variable and similarly can be wrapped up in error handling
   code. The Person class constructor requires a DateTime object as the second argument, so we pass
   in an instance of DateTime created using the information supplied by the user as shown:

   try
     set person1 to new type Person(full-name
                          new DateTime(test-year test-month test-day))
   catch argException
     display "Invalid arguments provided"
     display argException::Message
     stop run
   end-try

4. Replace application code with class methods.
   All code that determines the first name, family name and number of days until the next birthday can be
   replaced with the appropriate methods from the Person class and the now redundant section add-to-
   total can also be removed.

5. Tidy up defined variables.
   With some of the functionality moved into the Person class and the resultant code simplified, variables
   defined in the demonstration application can now be reviewed and redundant definitions removed.

Further Application Enhancements

Now that the code for the sample application has been simplified and functionality moved into a class,
enhancing is much easier to do. Redundant code and variables have been removed.
By introducing classes to provide standard methods, new applications are easier and quicker to write. You can update your existing applications to use the new class as the opportunity presents itself, steadily moving code towards more efficiency in development.

With the updated sample application, the text constants are contained in three variables. These variables could be removed, replacing them with the quoted string directly in the code, or the strings themselves could be provided via a second class. The console application itself could be completely replaced with a Windows application now that the functionality has been moved to a class.

The greeting text "Hello Mr " indicates that the application has been written assuming that the user will always be a man; however the application could ask the user to enter their gender. The Person class could then have a new method written to provide the formal greeting, allowing the text to simply read "Hello".

You can extend the demonstration application further with the suggestions above, or with additional functionality of your own design. Below is the final console application code:

```cobol
PROGRAM-ID. CblDemo5.
  01 person1           type Person.
  01 test-date.
    03 test-year     PIC 9999.
    03 test-month    PIC 99.
    03 test-day      PIC 99.
  01 full-name         PIC X(30).
  01 txt1              String value "Hello Mr ".
  01 txt2              String value " you have ".
  01 txt3              String value " days until next birthday".
  01 argException      type ArgumentException.

PROCEDURE DIVISION.
  display "Enter your full name"
  accept full-name
  display "Enter your date of birth in yyyymmdd format"
  accept test-date
  try
    set person1 to new type Person(full-name
       new DateTime(test-year test-month test-day))
  catch argException
    display "Invalid arguments provided"
    display argException::Message
    stop run
  end-try
  display txt1 & person1::familyName() & "",
  display person1::firstName() & txt2 & person1::daysUntilBirthday() & txt3
  stop run.

END PROGRAM CblDemo5.
```

Here is the final code for the Person class:

```cobol
class-id Person.
  01 full-name              string   public property as "FullName".
  01 dob                     type DateTime public property as "DateOfBirth".

method-id New public.
procedure division using by value fullname as string
  set full-name to fullname
  set dob to birthday
  goback.
end method.

method-id NextBirthday.
  01 today               type DateTime value type DateTime::Today().
```

Visual COBOL and COBOL for .NET Language Introduction | 27
```plaintext
procedure division returning return-value as type DateTime.
    set return-value to new DateTime(today::Year, dob::Month, dob::Day)
end method.

method-id DaysUntilBirthday.
procedure division returning return-value as binary-long.
    set return-value to self::NextBirthday()::Subtract(type DateTime::Today())::Days
end method.

method-id FamilyName.
  01 temp1                   PIC 99.
procedure division returning return-value as type String.
    set temp1 TO full-name::Trim()::LastIndexOf(' ')
    set return-value to full-name(temp1 + 1:)::Trim()
end method.

method-id FirstName.
procedure division returning return-value as type String.
    set return-value to full-name(1:full-name::IndexOf(' '))
end method.
end class.
```

## Appendix

### Changing the Startup Object

The startup object of a project is the entry point when the application begins. When you create a project this is initially set to the program template that is created at the same time. You might want to change this because, for example, you are replacing the user interface from console to Windows. By switching the startup object between the two programs you will be able to check that functionality is maintained between the existing and new programs. The following steps show you how to change the startup object:

1. Right-click on the project within the Solution Explorer.
   The project properties are displayed in the Properties Page.

2. Select **Application** from the left tab list.
   The properties in the application tab are brought to the foreground in the Properties Page.

3. Select the object from the list box labeled **Startup object**.
   The next time the application executes it will begin with the selected object.
The table below shows the equivalent data types between legacy and current COBOL. In addition the data types recognized within the .NET environment have been included. We strongly recommend you use the current COBOL usage as this is recognized in both the Java Virtual Machine (JVM) and .NET environments.

<table>
<thead>
<tr>
<th>Current COBOL</th>
<th>Legacy COBOL</th>
<th>.NET Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary-char</td>
<td>pic s9(2) comp-5</td>
<td>System.SByte</td>
</tr>
<tr>
<td>binary-char unsigned</td>
<td>pic 9(2) comp-5</td>
<td>System.Byte</td>
</tr>
<tr>
<td>binary-short</td>
<td>pic s9(4) comp-5</td>
<td>System.Int16</td>
</tr>
<tr>
<td>binary-short unsigned</td>
<td>pic 9(4) comp-5</td>
<td>System.UInt16</td>
</tr>
<tr>
<td>binary-long</td>
<td>pic s9(9) comp-5</td>
<td>System.Int32</td>
</tr>
<tr>
<td>binary-long unsigned</td>
<td>pic 9(9) comp-5</td>
<td>System.UInt32</td>
</tr>
<tr>
<td>binary-double</td>
<td>pic s9(18) comp-5</td>
<td>System.Int64</td>
</tr>
<tr>
<td>binary-double unsigned</td>
<td>pic 9(18) comp-5</td>
<td>System.UInt64</td>
</tr>
<tr>
<td>character</td>
<td></td>
<td>System.Char</td>
</tr>
<tr>
<td>float-short</td>
<td>comp-1</td>
<td>System.Single</td>
</tr>
<tr>
<td>float-long</td>
<td>comp-2</td>
<td>System.Double</td>
</tr>
<tr>
<td>condition-value</td>
<td></td>
<td>System.Boolean</td>
</tr>
<tr>
<td>decimal</td>
<td></td>
<td>System.Decimal</td>
</tr>
<tr>
<td>object</td>
<td></td>
<td>System.Object</td>
</tr>
</tbody>
</table>
## Working with IntelliSense

IntelliSense® is the general term for the prompts that appear when the cursor hovers code, and for the visual indicators that assist to speed code development. Visual COBOL makes full use of IntelliSense, providing instant feedback and increased productivity. The main features of IntelliSense include the following.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variable information</strong></td>
<td>Placing the cursor over a variable displays a tooltip with information about the variable, including number of times reference, type of variable and length.</td>
</tr>
<tr>
<td><strong>Error highlighting</strong></td>
<td>Code is constantly recompiled in the background, which detects anything that would cause a build error. Spellings are checked and suggested corrections are given. Variable validation confirms that the variable has been defined.</td>
</tr>
<tr>
<td><strong>Colorization</strong></td>
<td>As you type the code it is validated and color coded. The statements, variables, constants and section names all have different assigned colors that can be customized.</td>
</tr>
<tr>
<td><strong>Word completion</strong></td>
<td>As you type statements, lists of variables display to assist in sentence completion. To access the lists, right-click in the code editor to select the option <strong>Suggest Word</strong>.</td>
</tr>
<tr>
<td><strong>Code snippets</strong></td>
<td>These are templates of common sections of code that are generated and pasted directly into the code. To inset a code snippet, right-click at the point where the snippet is to be inserted and select <strong>Insert Snippet</strong>. Double-click to select the snippet required from the provided list.</td>
</tr>
</tbody>
</table>