Object-Oriented Programming for COBOL Developers

Introduction

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK
http://www.microfocus.com

2013-05-09

Contents

An Introduction to Object-Oriented Programming for COBOL Developers

Overview

This guide provides a basic introduction to Object-Oriented Programming (OOP) for COBOL developers who use Micro Focus Visual COBOL or Micro Focus Enterprise Developer. There are sections in the guide for each of the key concepts of object orientation.

Managed COBOL, which is the collective term for .NET COBOL and JVM COBOL, is regular procedural COBOL with extensions to take advantage of the features of the managed frameworks. This includes object-oriented syntax (OO) that allows access to large libraries of functionality you can use in your application and much more. To take full advantage of managed COBOL, you need to understand the object-oriented concepts.

Sample code

This guide includes a number of pieces of sample code to illustrate some of the concepts of object-orientated programming in COBOL. As you read the guide, you might want to type the code yourself, compile it and step through it in the debugger in Visual COBOL or in Enterprise Developer.

You need one of the following products installed:

- Micro Focus Visual COBOL for Visual Studio 2010 or for Visual Studio 2012
- Micro Focus Visual COBOL for Eclipse for Windows
- Micro Focus Enterprise Developer for Visual Studio 2010
- Micro Focus Enterprise Developer for Eclipse

Note: Visual COBOL and Enterprise Developer cannot co-exist on the same machine.

To run the examples, create a JVM COBOL project in Eclipse or a .NET managed COBOL console application in Visual Studio.

Classes and Methods

At the heart of the Object-Oriented Programming is the notion of a *class*. A class is said to encapsulate the information about a particular entity. A class contains data associated with the entity and operations, called *methods*, that allow access to and manipulation of the data. Aside from encapsulation of data, classes are also very useful for bridging your existing procedural programs with managed code technologies.

Here is a simple COBOL class:

```
       class-id MyClass.

       method-id SayHello static.

       linkage section.

       01 your-name pic x(10).

       procedure division using by value your-name.
               display "hello " & your-name

       end method.
```

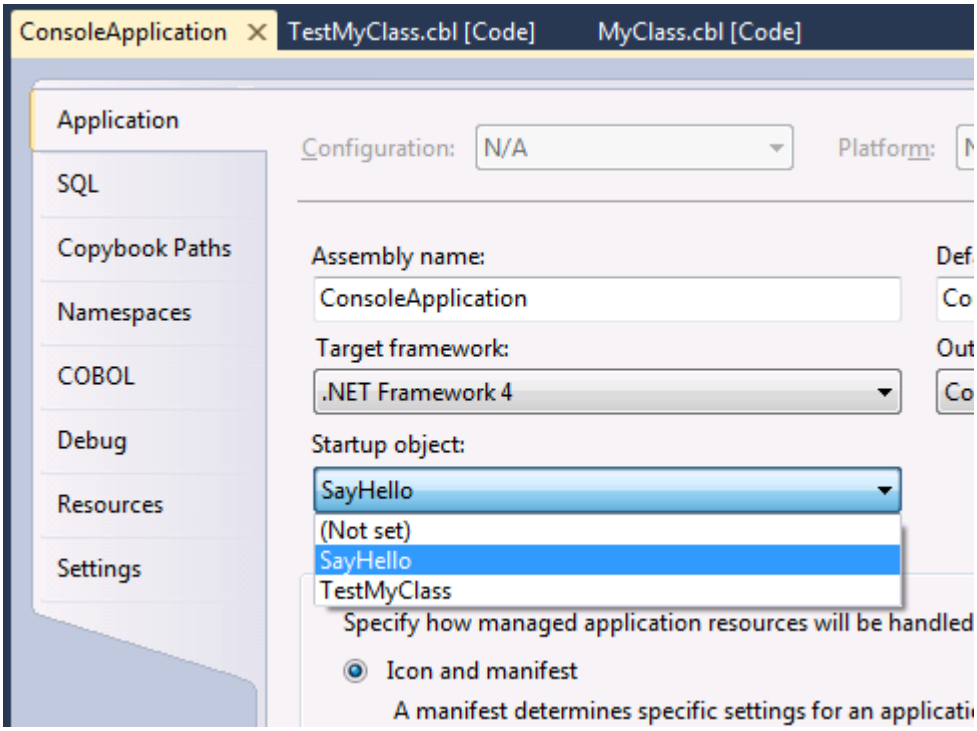
```
end class.
```

Before we look at the details of the class, let's see how you would invoke the single method contained in this class:

```
program-id. TestMyClass.

procedure division.

invoke type MyClass::SayHello(by value "Scot")

end program.
```

Note: To run this example in Visual Studio, you need to specify a **Startup object** for your managed console application. To do this:

1. In Visual Studio, right-click the solution in Solution Explorer.
2. Click **Add > New Item**, and click **COBOL class**.
3. Specify a name such as `MyClass.cbl`, and click **Add**.
4. In the same way, add a COBOL program with the name `TestMyClass.cbl`.
5. Add the two pieces of the example code above to the class and to the program, respectively.
6. Click **Project > *ProjectName* Properties** and click the **Application** tab.
7. Set **Startup object** to **TestMyClass**:



8. Click **Debug > Start Without Debugging** to execute the program.

As you would expect, the result of this program is:

```
Hello Scot
```

In this example, you can see how a procedural COBOL program can also use object-oriented semantics even though it is itself not a class.

Let's look at the details of the class, `class-id MyClass`.

`MyClass` is the name of the class. When you reference a class, you do so by specifying its name much in the same way you would reference a COBOL program.

Our class contains no data but it does have a single method named `SayHello`:

```
       method-id SayHello static.
```

Notice that there is a `static` clause associated with this method. This keyword is important in that it allows us to call the method without creating an instance of the class. Instances of classes are called *objects* which we will come to later.

Static methods and static data can be useful at times but there is only ever one instance of the static data. Static methods can only operate on static data.

The remainder of the method declaration should be familiar as it is identical to a procedural program that takes a single parameter as an argument to the program:

```
       linkage section.

       01 your-name pic x(10).
       procedure division using by value your-name.
           display "hello " & your-name
       end method.
```

Let's look at the procedural program that invokes this method:

```
       invoke type MyClass::SayHello(by value "Scot")
```

Note the following key points about the code:

- The `invoke` keyword is synonymous with `CALL` but is used in the context of calling a method on a class.
- The `type` keyword allows us to specify the name of the class we are referring to.
- The `::` syntax allows us to refer to the specific method on the class we wish to invoke.

Before we go deeper, let's review some more aspects of the syntax:

```
       invoke type MyClass::SayHello(by value "Scot")
```

The `type` keyword is a new part of the COBOL language introduced with Visual COBOL and simplifies how you reference and invoke methods.

To illustrate this, here is the equivalent program conforming to ISO syntax:

```
       program-id. TestMyClass
       repository.
       class MyClass as "MyClass".

       procedure division.

       invoke MyClass "SayHello" using by value "Scot"

       end program.
```

ISO requires the use of the `Repository` section and quotes around method names. In this simple example, the additional syntax though verbose does not significantly degrade the readability of the program. However, in real world programs, this additional syntax along with other requirements of ISO very quickly becomes unwieldy and makes COBOL unfriendly for managed code applications.

Visual COBOL also simplifies other aspects of the COBOL language - let's look at a couple of cases in our example:

```
           invoke type MyClass::SayHello(by value "Scot")
```

Can become:

```
           invoke type MyClass::SayHello("Scot")
```

If the method contained further arguments, these might appear as:

```
           invoke type MyClass::SayHello("Scot", 37, "Bristol Street")
```

In fact, even the commas separating parameters are optional.

In future examples, we will use this abbreviated syntax.

The method can also be simplified as follows:

```
       method-id SayHello static.
       linkage section.
       01 your-name pic x(10).
       procedure division using by value your-name.
           display "hello " & your-name
       end method.
```

Can become:

```
       method-id SayHello static.
       procedure division using by value your-name as string.
           display "hello " & your-name
       end method.
```

Two important things have changed here:

- The explicit `linkage section` has been removed and the linkage argument been defined inline with the procedure division `using` statement.
- The `pic x(10)` argument has been replaced by a reference to string.

`String` is a predefined COBOL type which maps onto the JVM and .NET string class. Strings contain a variety of methods and are used to hold Unicode data of an arbitrary length. The Compiler can convert between many of the predefined types such as `string` into COBOL types such as `pic x` - we will look at this in more detail later on.

For future examples, we will adopt this convention of defining arguments inline. However, this is only possible when we use predefined managed types. COBOL records still need to be defined in the usual way.

Objects

Our simple example so far has helped demonstrate the basic concept of a class but the value of Object-Oriented Programming is not yet apparent. The power of Object-Oriented Programming really comes into play when we encapsulate data within a class, provide methods that perform actions on that data, and then create instances of the class for use at run time.

Creating an instance of a class results in the creation of an object. Each object maintains a separate set of data items that the methods act upon.

You can create many instances of a class so, therefore, you can have many objects, each with data distinct from other objects in the system. This data is called *instance data*.

For example, if we considered the kind of data we might need in a simple bank account class, we might think of such things as an account number, balance and some way in which we could store transactions. At run time, we could conceivably create a unique object for each customer we were dealing with where each object maintains distinct data from other customers at our bank.

Creating an Instance of a Class

Let's change our class a little and look at how we would create an object instance:

```
class-id MyClass.

working-storage section.
01 your-name pic x(10).
```

```
method-id SayHello.
procedure division.
    display "hello" & your-name
end method.

end class.
```

The variable `your-name` defined in the `working-storage section` is the *instance data* for this class:

```
working-storage section.
01 your-name pic x(10).
```

To invoke the `SayHello` method, we now do this using an object rather than the class. Here's how we create that instance:

```
program-id. TestMyClass
01 an-obj type MyClass.
procedure division.

set an-obj to new MyClass
invoke an-obj::SayHello

end program.
```

This is the declaration of the object, more formally known as an *object reference*:

```
       01 an-obj type MyClass.
```

If we were to try and invoke the `SayHello` method on this class at this point, we would get a run-time system error because the object has not yet been created.

This is the line that creates the object:

```
       set an-obj to new MyClass
```

The keyword `NEW` is responsible for creating our object. `NEW` requires we specify the type of the object we want to create. This may seem strange as we have already said what type our object is when we declared it, but later on we will see that an object can be declared as one type but, at run time, reference a different type.

The `SET` statement is frequently used in Object-Oriented Programming and is synonymous with `move` but applies to objects.

It is possible to declare another object reference and assign it the value of `an-obj` as follows:

```
       set another-obj to an-obj
```

In this case, `another-obj` now contains a reference to `an-obj`. It is important to note that while we have two object references, there is actually only one instance of `type MyClass` at this point, and both `another-obj` and `an-obj` refer to it. If we invoked the `SayHello` method on `an-obj` and `another-object`, they would operate against the same data in the working-storage section.

The only way to create an entirely new object is to use the `NEW` keyword:

```
       set another-obj to new MyClass
```

Our class has an issue at the moment. If we were to invoke the `SayHello` method, it would just print `Hello`, as the `your-name` data item has yet to be given a value.

There are several ways we can fix this. One way to do this is during the creation of the object which is otherwise known as *construction*. Right now, our class does not do anything during construction but we can do so if we create a method named `New`.

Constructors

```
       method-id New.

       procedure division using by value a-name as string.
       set your-name to a-name
       end method.
```

Whenever an object is created, the run-time system automatically invokes the `New` method on the class. If you did not code one, the Compiler automatically creates it for you.

In our method above, not only have we defined a constructor but we have also specified that it should take a parameter. Given this, we need to change our code that creates the object:

```
       set an-obj to new MyClass("Scot")
```

This code could also have been written as:

```
       set an-obj to  MyClass::New("Scot")
```

What we have done is that we provided a way for our object to be initialized and ensured that we get an argument passed to the constructor any time an object of type `MyClass` is created.

Method Overloading

However, it is possible to have multiple versions of the `New` method, each corresponding to different arguments that can be passed in when the object is created. This is called *method overloading* because the method name remains the same but different arguments are accepted by each method.

We can also use this ability of method overloading to reinstate the so-called *default constructor*, otherwise known as the *parameterless constructor*. To do so, we just code a new `New` method.

```
method-id New.

procedure division.
   move all 'x' to your-name
end method.
```

This has allowed us to create the object by either supplying a parameter or using the default constructor which takes no arguments but still allows us to initialize our working-storage section data.

Properties

Our class has some data associated with it, a string called `your-name`. This data is not accessible directly by the program using the class just as the `working-storage` of one program is not accessible to another program.

Properties allow you to expose your data items to the user of your class.

Currently, our single data item looks like this:

```
       01 your-name pic x(10).
```

We can turn this data item into a property as follows:

```
       01 your-name pic x(10) property.
```

As such, you can now access this property through an object reference:

```
       display an-obj::your-name
```

The `property` keyword allows us not only to get the value of a data item as well as set it:

```
       set an-obj::your-name to "Scot"
```

However, we can prevent anyone setting the value as follows:

```
       01 your-name pic x(10) property with no set.
```

The case of your types and properties is important in managed COBOL. The case of our property name is also taken from the declaration which is currently all lower case. We can change the name and case as follows:

```
       01 your-name pic x(10) property as "Name"
       ...
       display an-obj::Name
```

While we are looking at properties, let's return to the subject of the `static` clause which can also be applied to properties:

```
 01 dataitem pic x(10) property as "DataItem" static.
```

If you recall, there is only ever one instance of a static data item regardless of how many objects have been created. Static data items are referenced through the class itself; we do not need an instance to access them:

```
set MyClass:DataItem to "some text"
```

Method Visibility

The methods we have defined so far have all been public, which is the default for COBOL. A *public method* means that it can be invoked through the object reference. However, for most classes we need methods which we do not want to be visible outside of the class. Such methods are called *private methods*.

To declare a private method, use the keyword `private`:

```
       method-id ProcessData private.
```

This method cannot be invoked through the object reference and, if you tried, you would receive a Compiler error.

You can invoke this method from inside the class itself, say, from inside a public method:

```
       method-id DoSomething public.

       procedure division using by value a-name as string.

       invoke self::ProcessData

       end method.

       end class.
```

Notice the use of the special keyword `self`. In this case, that just means "invoke a method called `ProcessData` which is defined in this class".

Also note that we explicitly marked this method as `public` in its declaration. This is not required as it is the default visibility but it can be useful to do when first starting out.

Local Data

When writing procedural COBOL programs, we only have the choice of declaring all our data in the `working-storage` section. When working with classes, we still use the working-storage section for data that is associated with the class but we can also define data that is used only by a method, or so-called *local data*.

There are three ways to define local variables:

In the Local-Storage Section

In the following example, `mylocalvar` is a local variable for the method and it only exists for this method:

```
method-id ProcessData private.

local-storage section.
01 mylocalvar binary-short.

procedure division.
...
end method.
```

Using the DECLARE statement

In the following example, `mylocalvar` is defined using the DECLARE statement. The scope of the variable defined in this way is only within the method after the declaration:

```
method-id ProcessData private.

local-storage section.

procedure division.
declare mylocalvar as binary-short

end method.
```

Define as an inline variable

In the method, we can create a local variable called `counter` as part of the PERFORM statement. The lifetime and scope of this variable is associated with the execution and scope of the PERFORM statement. In other words, it is not possible to refer to `counter` after the END PERFORM statement.

```
method-id ProcessData private.

local-storage section.

procedure division.
perform varying counter as binary-long from 1 by 1 until
counter > 10
          display counter
end perform.
end method.
```

Data Types

So far, our classes have used COBOL data types such as `pic X`. All of the data types you use in procedural COBOL today are supported in managed COBOL.

Some data types such as `pic X` or group records are not understood by .NET and JVM. To help transition COBOL types to other languages, there is a set of predefined types which are natively understood by .NET and JVM, and map directly to the .NET and JVM types. These types are listed in the topic *Type Compatibility of Managed COBOL with Other Managed Languages* available in the *Visual COBOL documentation*.

Here are three examples of declaring the same type, a 16-bit signed integer:

```
       01 val1 binary-short.
       01 val2 pic s9(4) comp-5.
       01 val3 type System.Int16.
```

In C# and in Java, the equivalent declaration would use a type called `short`.

To specify a 16-bit unsigned integer, you would use:

```
       01 val1 binary-short unsigned.
       01 val3 type System.UInt16.
```

In C#, the equivalent declaration would use a type `ushort`. In Java, however, this does not have an equivalent as Java does not support unsigned types like this one.

The point to remember here is that, when working with classes, whatever data you expose to the caller of the class, whether it is as arguments to a method or as a property, it is generally a best practice to use *COBOL predefined types* as shown in the table in the topic *Type Compatibility of Managed COBOL with Other Managed Languages* in the documentation.

However, in one of our previous examples we did not do this. In fact, we exposed a `pic X` item as a property. When we do this, the Compiler is actually exposing the intrinsic `String` type, not the `pic X` field.

When a user of the property reads or sets it, the data is implicitly converted from native COBOL type to the .NET or JVM type, in this case a `string`.

Declaring a group item as a property actually exposes the whole group as a .NET or JVM `string` type.

Native numeric types such as `comp-5` are coerced to the nearest managed code equivalent.

Inheritance

Inheritance is an important part of Object-Oriented Programming. It allows us to create a new class by extending the functionality of an existing class. If we choose to, we can also change the behavior of the class we are extending.

Let's consider the example with the bank account we mentioned earlier. We might imagine that accounts of any type, checking, savings, etc., share common data such as an account number field and a balance, but the process of withdrawing money from an account might require different processing. A checking account may need to check whether an overdraft limit is in place and a savings account, which will not have an overdraft, will need to check other factors that affect interest earned, such as the amount of money that can be withdrawn within a given period.

An important consideration we will look at later is that whatever is using these objects, let's say the ATM machine, should not need to determine the type of account it's dealing with and then perform different processing. It simply wants to perform a withdrawal action against whatever account object it is using.

For now though, let's just look at how we can both extend an existing class and customize its behavior.

In order to test this in Visual Studio, create a managed Class Library project and add the classes as separate files to the project - right-click the project in Solution Explorer, and click **Add > New Item > COBOL Class**.

Here is a simplistic bank account class:

```
       class-id BankAccount.

       working-storage section.
       01 account-number pic 9(8) property as "AccountNumber".
       01 balance float-long property.
```

```
       method-id Withdraw.

       procedure division using amount as float-long
                   returning result as condition-value.
       *> Process for general withdrawal from a bank account
       ...
       end method.
       end class.
```

This type of class, named `BankAccount`, is often referred to as the *base class* as it forms the base of a hierarchy of classes that emanate from this one.

Let's create a new class to represent a specialization of the bank account, a savings account:

```
       class-id SavingsAccount inherits type BankAccount.

       method-id Withdraw override.

       procedure division using amount as float-long
                   returning result as condition-value.

       end method.
       *> Specialized process for Savings withdrawal.

       end class.
```

Besides defining a new class for savings accounts, we have used the `inherits` clause to denote we are extending an existing class in the system. All public members (methods, properties, fields defined as `public`) of the base class become part of the new class.

As such, an object that is of the type `SavingsAccount`, also has properties called `AccountNumber`, balance and a method named `Withdraw` which have been inherited from the base class `BankAccount`.

Our `SavingsAccount` class also has a method called `Withdraw` which will manage the different way in which money is withdrawn from a savings account. To indicate this is a change in behavior to the method in the base class, we use the `override` keyword. The significance of this keyword will become more apparent later on.

Let's create another specialization of the `BankAccount` class, a debit account:

```
       class-id DebitAccount inherits BankAccount.

       method-id Withdraw override.

       procedure division using amount as float-long
                   returning result as condition-value.

       end method.
       *> Specialized process for Debit withdrawal.

       end class.
```

This class for debit accounts is created in exactly the same way as the class for the savings account. It extends the bank account class and all public members of the bank account class become part of the new class.

The `DebitAccount` class also has a `Withdraw` method that overrides the `Withdraw` method of the `BankAccount` class, and it defines how money is withdrawn from the debit account. This method, however, is a completely different one from the `Withdraw` method in the `SavingsAccount` class.

We now have three classes in our class hierarchy - one base class (`BankAccount`) and two classes (`SavingsAccount` and `DebitAccount`) that are derived from it, each of which provide a different override for the `Withdraw` method.

Let's look at the effect of object instantiation and method invocation:

```
       program-id. TestBankAccounts.

       01 account1 type BankAccount.
       01 account2 type BankAccount.

       procedure division.

           set account1 to new SavingsAccount
           set account1::AccountNumber to 12345678
           set account1::balance to 500.00

           set account2 to new DebitAccount
           set account2::AccountNumber to 87654321
           set account2::balance to 100.00
           ...

       end program TestBankAccounts.
```

The key point to notice is the declaration of our object's type, `BankAccount`, and the creation of it once as a `SavingsAccount` and once as a `DebitAccount`.

We can do this because both `SavingsAccount` and `DebitAccount` inherit (or descend) from `BankAccount`. The value of doing this is not so apparent in this example but this next example might help:

```
       method-id PerformWithdrawal.

       procedure division using by value amount as float-long
                 account as type BankAccount.
           if not account::Withdraw(amount)
           *> perform error condition
              display "not true"
           else
              display "true"
           end-if

       end method.
```

In this case, a method receives an argument of type `BankAccount` from which it performs a withdrawal action. The method does not need to know about all the different types of accounts but, whichever object type is passed in, the correct `Withdraw` method associated with that type is executed, be that a savings or debit account.

We can invoke that method with an argument of either `account1` or `account2`:

```
    invoke SomeClass::PerformWithdrawal(100,account1)
    invoke SomeClass::PerformWithdrawal(200,account2)
```

We've passed `account1` and `account2` and for each one of them, the appropriate `Withdraw` method associated with `SavingsAccount` and `DebitAccount` is executed.

This is a very useful feature of Object-Oriented Programming as it decouples implementation details from clients that use the classes. This, in turn, allows us to extend the system by adding new types of bank accounts but minimizing the impact on existing code.

Under both JVM and .NET, you can only inherit from one base class but, of course, the base class itself can inherit from a class and so on.

If a derived class needs to invoke the implementation of a method defined in the base class, it can do so using the `super` keyword. For example, we can call the BankAccount `WithDraw` method from within the `SavingsAccount` class as follows:

```
invoke super::Withdraw(100)
```

`super` can be used not only to invoke a method we have overridden in the derived class, but also to invoke any public method defined in the class hierarchy we have inherited.

Interfaces

Classes and inheritance allow us to decouple implementation details from the user of the class but there is another aspect of Object-Oriented Programming that can help further decouple implementation - the *interface*.

An interface, like a class, defines a series of methods and possibly data, too, but unlike a class, it does not provide any implementation within the methods. This is because the purpose of the interface is merely to define what behavior a class will have - behavior in this case being the methods and properties defined on the class.

Here is an example of an interface:

```
       interface-id ErrorHandler.

       method-id notifyError.
       procedure division using by value error-code as binary-short.
       end method.

       method-id notifyWarning.
       procedure division using by value warning-code as binary-short.
       end method.

       end interface.
```

This interface defines just two methods which we can probably deduce would be used for logging an error of some kind.

By defining a class that supports this interface, we are said to implement the interface:

```
       class-id MyErrorHandler implements type ErrorHandler.

       method-id notifyError.
       procedure division using by value error-code as binary-short.
       *> display message box to the user
       end method.
       method-id notifyWarning.
       procedure division using by value warning-code as binary-short.
       *> depending on the configuration, ignore this or print
       *> it to the console
       end method.
       end class.
```

The `implements` keyword defines the interface we intend to provide an implementation for in this class and the Compiler will check that all methods have been implemented correctly.

Unlike inheriting a class, which can only be done with a single class, you can implement as many interfaces as you like in a single class.

We can create an instance of our class and because we have implemented the `ErrorHandler` interface, we can pass an object reference of this class to any code that expects to be working with the `ErrorHandler` interface.

```
       class-id ProcessData.
       working-storage section.

       01 error-handler-list  List[type ErrorHandler] value null static.
```

```
       method-id RegisterErrorHandler static.
       procedure division using by value error-handler as type ErrorHandler.
           if error-handler-list = null
               create error-handler-list
           end-if.
           write error-handler-list from error-handler
       end method.

       method-id NotifyErrorHandlers static.
       local-storage section.
       01 error-handler type ErrorHandler.
       procedure division using by value error-code as binary-short.
           perform varying error-handler thru error-handler-list
               invoke error-handler::notifyError(error-code)
           end-perform
       end method.

       method-id DoProcessing.
       01 error-code binary-short value 1.
       procedure division.
       *> do something and, possibly, call NotifyErrorHandlers when something
       *> goes wrong
       *> ...
           invoke self::NotifyErrorHandlers(error-code)
       *> ...
       end method.
       end class.
```

```
       program-id. TestProgram.

       working-storage section.
       01 error-handler type MyErrorHandler.
       01 processData type ProcessData.

       procedure division.

       set error-handler to new MyErrorHandler
           invoke type ProcessData::RegisterErrorHandler(by value error-
handler)
           set processData to new ProcessData
           invoke processData::"DoProcessing"

       end program TestProgram.
```

Let's review this code as there are some new concepts as here.

First of all, we have a class, `ProcessData`. At some point during the execution of the method `DoProcessing`, `ProcessData` will inform any interested parties that an error has occurred. It does this by invoking methods on the `ErrorHandler` interface using the `NotifyErrorHandlers` method.

This class has the capability of notifying multiple parties as it allows clients to register their interface implementation using the `RegisterErrorHandler` method. Each interface is stored within a *list* object. We will not explore the list object now but let's assume such a class is provided to us by the .NET or JVM class frameworks.

When an error does occur and the `NotifyErrorHandlers` method is invoked, the code makes use of feature of the managed COBOL syntax that allows it to iterate through the collection of error handler interfaces contained in the list. Each iteration results in the `error-handler` local-storage object reference being set to the next item in the list. The code simply calls the `NotifyError` method and the implementation of this decides what to do about it.

The `TestProgram` constructs an instance of `MyErrorHandler` and passes this as an argument to the `RegisterErrorHandler` method. This call involves an implicit cast from the type `MyErrorHandler`, a class, to the type `ErrorHandler`, an interface.

Class Names

So far, our classes have had simple names but this can soon lead to clashes with classes created by other people. To resolve this, we simply create classes with longer names by employing the use of *namespaces* which is nothing more than a convention for naming classes.

Here is a fully-qualified class name:

```
com.acme.MyClass
```

`MyClass` is a different class from the following one:

```
com.yourcompany.MyClass
```

Everything leading up to the class name is considered a *namespace*, if working in .NET, or a *package name*, if working in JVM. In this case, the namespaces are `com.acme` and `com.yourcompany`.

This convention allows us to create classes that do not conflict with other classes of the same name.

While this is a naming convention, Compilers provide directives and syntax to make working with namespaces easier and, in fact, there can be certain rules about the accessibility of classes within namespaces.

When you reference a class that has a namespace, you need to use its fully qualified name. For example:

```
01 an-obj type com.acme.MyClass.
01 another-obj type com.yourcompany.MyClass.
```

The COBOL Compiler provides the `ILUSING` directive that allows you to use the abbreviated name of a class:

```
 $set ILUSING(com.acme)
```

When you use this directive in a source file, you import the namespace into your project and then reference the class by its shortened name in the code:

```
 01 an-obj type MyClass.
```

While this is generally accepted practice, as class names can otherwise become quite long, you should avoid needlessly importing lots of namespaces as it defeats the whole purpose of including classes in namespaces and packages. Besides, you may find you encounter a clash of class names, in which case you need to disambiguate the class name by specifying the full class name.

Intrinsic Types

The COBOL Compiler is aware of several classes within the .NET and the JVM frameworks and does not require you to specify their fully qualified names. The two classes we will look at are `Object` (`System.Object` in .NET and `java.lang.Object` in JVM) and `String` (`System.String` in .NET and `java.lang.String` in JVM).

`Object` is important because all classes ultimately inherit from this type, whether you specify it or not. Therefore, any object can be cast to this type.

`String` is used commonly for storing Unicode data. In both JVM and .NET, the string, once created, is considered immutable. Whichever method you invoke on the `String` class, the result is a new string object:

```
01 str1 type System.String.
01 str2 String.
01 str string.
```

All of the above declarations in .NET are equivalent.

Notice there is no need to call the `New` method when creating a string:

```
set str1 to "Hello World"
```

You can combine strings with regular `pic X` fields as follows:

```
01 a-pic-x pic X(10) value "something".
display a-pic-x & str1 & "blah"
```

Here is an example of using one of the many string methods in .NET COBOL:

```
 set str1 to str1::Replace("foo", "bar")
```

Notice how we assigned the result of this method to the original object. If we did not, `str1` would have remained unchanged.

The same example for JVM COBOL looks like this:

```
 set str1 to str1::replace("foo", "bar")
```

In .NET and JVM, the only difference between these methods is the case - `String.Replace` for .NET and `String.replace` for JVM.

The .NET and JVM Frameworks

.NET and JVM are huge frameworks of classes that provide a massive range of functionality. Learning all the classes in these frameworks can take a long time but there are many classes that you should get to know quickly, particularly the collection classes.

To help illustrate the usefulness of these frameworks, let's look at just one area - date and time arithmetic made easy with the following example in .NET COBOL:

```
       working-storage section.

       01 dt1 type System.DateTime.
       01 dt2 type System.DateTime.
       01 ts type System.TimeSpan.
       ...
       set dt1 to type System.DateTime::Now
       invoke System.Threading.Thread::Sleep(1000)

       set dt2 to type System.DateTime::Now
       set ts to dt2 - dt1

       display ts
```

Let's review what we have done.

First of all, we have declared three object references, two `DateTime` objects and one `TimeSpan` object.

The `DateTime` class provides an extensive set of routines for manipulating dates and times. To get an idea of its capabilities, see the description of the class on *MIcrosoft's MSDN*.

The `TimeSpan` class is used when calculating the difference between two `DateTime` objects.

In the first line of code, we initialize the `dt1` object reference using a static method on the `System.DateTime` class, `Now`. There are many other ways to initialize a `DateTime` object but this is a convenient way of getting the current date and time:

```
       set dt1 to type System.DateTime::Now
```

In thw next line, we again make use of a static method, this time to cause the current thread to sleep for a specified period. You could invoke the Micro Focus CBL_THREAD_SLEEP routine to achieve the same result as follows:

```
       invoke System.Threading.Thread::Sleep(1000)
```

The next line initializes our second `DateTime` object following the sleep:

```
       set dt2 to type System.DateTime::Now
```

The next line demonstrates a feature of the managed code COBOL Compiler called *operator overloading*:

```
       set ts to dt2 - dt1
```

Operator overloading is an advanced feature of Object-Oriented Programming and worth taking a quick look at. When defining a class, it is also possible to provide an implementation of some arithmetic operators such as `add` and `subtract`. The `DateTime` class defines several operators for date and time arithmetic and comparison.

While you can perform arithmetic on objects by using the operator overloads, classes usually provide equivalent methods you can invoke directly, as is the case for `DateTime`. The following line would achieve the same result as the previous one:

```
       set ts to dt2::Subtract(dt1)
```

Either approach results in a `TimeSpan` object. This object contains the result of the arithmetic expression.

Finally, we display the result.

Whenever you use the `DISPLAY` verb with an object reference, the Compiler automatically invokes the `ToString` method that is defined on the base class, `Object`. If you remember, all classes ultimately descend from `Object`. Ordinarily, the `ToString` method simply returns the name of the `Type`, but the `TimeSpan` class overrides the `ToString` method and returns a meaningful string about the `TimeSpan` object:

```
       display ts
```

Now let's look at how the same example translates to JVM COBOL:

```
      $set ilusing"java.util.concurrent"
       program-id. Program1 as "Program1".

       data division.
       working-storage section.
       01 start-time binary-double.
       01 end-time binary-double.
       01 interval binary-double.

       01 hr binary-double.
       01 min binary-double.
       01 sec binary-double.
       01 ms binary-double.


       procedure division.

       *> get the start time
       set start-time to type System::currentTimeMillis()
       *> wait a second
       invoke type Thread::sleep(1000)
       *> get the end time
```

```
       set end-time to type System::currentTimeMillis()


       *> calculate the difference between start and end time
       set interval to end-time - start-time
       set hr to type TimeUnit::MILLISECONDS::toHours(interval)
       set min to type TimeUnit::MILLISECONDS::toMinutes(interval - type
TimeUnit::HOURS::toMillis(hr))
       set sec to type TimeUnit::MILLISECONDS::toSeconds(interval - type
TimeUnit::HOURS::toMillis(hr) - type TimeUnit::MINUTES::toMillis(min))

       display String::format("%02d:%02d:%02d", hr, min, sec)


           goback.

       end program Program1.
```

Here we do a few things differently.

The equivalent of the `System.DateTime` class in the .NET framework is `Date` in JVM and it does not have a `subtract` method which can be overloaded. That's why we need to use `set`.

The .NET `System.TimeSpan` class does not have an alternative in the JVM out-of-the-box, which is why we have to use `TimeUnit` to calculate the time span between two set dates.

Reflection

When you compile a COBOL program for .NET or for JVM, the Compiler creates an executable that conforms to the specification of these platforms. This specification must be adhered to by any language supporting .NET or JVM. Because all languages conform to a common underlying protocol, classes written in any language can be easily integrated.

Another advantage of this commonality is something called *reflection*. Reflection is the ability to examine the underlying details of a given type. With this ability, it is possible to inspect the methods that a given type provides, the arguments to each method, and even the code in the body of the method. This is a very powerful feature of the framework and opens up many possibilities for application development. Although it may not seem immediately obvious as to how reflection can be valuable, understanding that it is possible can help when considering how various technologies in managed code can do what they do without having prior knowledge of how classes are defined.

One example of the use of reflection is something called . Intellisense is a feature of the Visual Studio and the Eclipse IDEs that assists you by showing a list of methods and properties available on a given object.

Calling COBOL From Other Languages

Let’s take a look at our very first example of a simple class:

```
       class-id MyClass.

       method-id SayHello static.

       linkage section.
       01 your-name pic x(10).
       procedure division using by value your-name.
```

```
           display "hello " & your-name
       end method.

       end class.
```

Because managed COBOL follows the rules of any other managed language, you can invoke its methods from other managed languages.

`your-name` is a PIC X item and this maps onto a `string` item in other managed languages.

You can invoke the SayHello method from a C# project provided that you have set a project reference in your C# project to the COBOL project. To do this:

1. In your solution, create a C# project and add a reference to it to the COBOL project:
   a. In Solution Explorer, right-click the C# project and click **Add Reference**.
   b. On the Project tab, select your COBOL project, and click **OK**.
2. In the same way, add a reference in the C# project to the `Micro Focus Runtime` assembly - in the **Add Reference** dialog, click the **.NET** tab, select `Micro Focus Runtime`, and click **OK**.
3. In the C# program, type the following to invoke the COBOL class:

```
...
    class MyCSharpClass
    {
        static void Main(string[] args)
        {
             MyClass.SayHello("MyName");
        }
    }
```

To do the same from Java:

1. In Visual COBOL for Eclipse, create a COBOL JVM project, `MyCOBOLProject`, and a Java project, `MyJavaProject`.
2. Create a new COBOL JVM class in your COBOL project, `MyClass`, and assign it to the default package.
3. Paste the COBOL code of the simple class to the class file.
4. Create a new Java class in the Java project, `MyJavaClass`, and assign it to the default package.
5. In the properties of the Java project, on the **Java Build Path** page, click the **Projects** tab.
6. Click **Add**, enable the check box for your COBOL JVM project, and click **OK**.
7. Click the **Libraries** tab, click **Add Library**, select **COBOL JVM Runtime System**.
8. Click **Next**, **Finish**, and then click **OK**.
9. Edit the Java class and add the code below to it:

```
public class MyJavaClass {
       public static void main(String[] args) {
             MyClass.SayHello("Mike");
       }
}
```

   You do not need to import a package in your Java class because both classes are defined in the default package for the project,

That's covered the basics but now let's look at a more complicated case.

Even though .NET and JVM COBOL make it much easier to call other code modules regardless of the language they're written in, it can still be difficult to call a procedural COBOL program from Java or C#. That's because COBOL programmers have access to a rich set of data types and are able to create complex hierarchical structures in COBOL that are not easily reproduced by other languages.

There are several approaches you can take to make a COBOL program and the linkage area easily callable by another language. One method is to create an OO COBOL wrapper class that sits around the

procedural COBOL program and presents a natural interface to Java or C# programmers. You can find out more about how to write wrapper classes *here*.

Alternatively, you can use the Visual COBOL Compiler to create the interfaces for you using a Compiler directive, ILSMARTLINKAGE, which helps expose the linkage section items and entry points to other managed languages.

Compiling your code with the ILSMARTLINKAGE directive generates a class for each group item with the lower level data items being the members of the class which, in turn, you can access from other managed languages.

Here it is how it works in .NET:

1. Create a new managed COBOL solution with a managed COBOL console application and a C# Windows console application.
2. In your C# project, add a project reference to the COBOL project and a reference to the `Microfocus.Runtime` assembly.
3. Modify the `Program1.cbl` program in your COBOL project and add the code below:

```
program-id. Program1.

linkage section.
01 your-name.
     03 first-name   pic x(10).
     03 middle-name  pic x(10).
     03 surname      pic x(10).
01 your-address.
     03 street-name  pic x(10).
     03 town-name    pic x(10).
     03 country-name pic x(10).

procedure division using by reference your-name
                         by reference your-address.

       display "hello "
       display your-name
       display "from"
       display your-address

end program Program1.
```

In order to expose these to managed code, you need to use the ILSMARTLINKAGE directive and compile your managed COBOL project with it:

1. In your COBOL project, navigate to the project properties, click the **COBOL** tab, and type `ILSMARTLINKAGE` in the **Additional Directives** field.
2. Build your COBOL project.

   Compiling with ILSMARTLINKAGE exposes the group items `your-name` and `your-address` as new classes, `YourName` and `YourAddress`, and the 03 items as properties of these classes. The ILSMARTLINKAGE directive removes the hyphens from the names and changes the case to camel case.
3. In your C# program you can insert the following code to access the group item entries as follows:

```
    class Program
    {
        static void Main(string[] args)
        {

          Program1 TestProgram = new Program1();
          YourName CustomerName = new YourName();
          YourAddress CustomerAddress = new YourAddress();
          CustomerName.FirstName = "Mike";
          CustomerAddress.TownName = "London";
```

```
          TestProgram.Program1(CustomerAddress CustomerName);
        }
    }
```

Here, we have defined two new C# variables, `CustomerName` and `CustomerAddress`, and have made them to be of type `YourName` and `YourAddress`. Then, we have accessed the `first-name` item in the linkage section of our COBOL class which is exposed as `FirstName`.

For more information on calling managed COBOL from .NET, see your product help (*Exposing COBOL Linkage Data as Managed Types*).

To access the group items in the same example from Java, you would need to do a few things differently:

1. In Visual COBOL for Eclipse, create a new COBOL JVM project, `MyCOBOLProject`, and a Java project, `MyJavaProject`.
2. Create a new COBOL program in your COBOL project, `Program1.cbl`, and assign it to the default package.
3. Add the COBOL code to the class:

```
program-id. Program1.

linkage section.
01 your-name.
     03 first-name   pic x(10).
     03 middle-name  pic x(10).
     03 surname      pic x(10).
01 your-address.
     03 street-name  pic x(10).
     03 town-name    pic x(10).
     03 country-name pic x(10).

procedure division using by reference your-name
                         by reference your-address.

       display "hello "
       display your-name
       display "from"
       display your-address

end program Program1.
```

4. Navigate to the properties of your COBOL project, expand **Micro Focus**, and click **Build Configuration**.
5. Type `ILSMARTLINKAGE` in the **Additional directives** field, click **Apply**, and then **OK**.
6. Create a new Java class in the Java project, `MyJavaClass`, and assign it to the default package.
7. In the properties of the Java project, on the **Java Build Path** page, click the **Projects** tab.
8. Click **Add**, enable the check box for your COBOL JVM project, and click **OK**.
9. Click the **Libraries** tab, click **Add Library**, select **COBOL JVM Runtime System**.
10. Click **Next**, and then **Finish**.
11. Click **Add Class Folder**, enable the check box for the `bin` folder in the COBOL project, and then click **OK** twice.
12. Edit the Java class and add the code below to it:

```
public class MyJavaClass {

    public static void main(String[] args) {

  Program1 testProgram = new Program1();
        YourName customerName = new YourName();
        YourAddress customerAddress = new YourAddress();
        customerName.setFirstName("Mike");
        customerAddress.setTownName("London");
  testProgram.Program1(customerAddress customerName);
```

```
    }
}
```

You do not need to import a package in your Java class because both classes are defined in the default package for the project. To learn how you can import classes that are not defined in the default package, read your product help (*Java Calling JVM COBOL*).

What Next?

This guide provides a basic introduction to Object-Oriented Programming and has covered many of the fundamental concepts. Object-Oriented Programming is, however, an extensive subject, and there are many other areas to cover including the many technologies provided by the .NET and JVM platforms. Now that you know the basics, you can continue with the advanced topics.

For further reading on Object-Oriented programming, we recommend you check the following Web sites:

- For programming for the .NET framework, see *.NET Framework 4* on the MSDN.
- For JVM programming, see *Learning the Java Language* on the Oracle Web site.

Various demonstrations of managed COBOL are available with Visual COBOL and Enterprise Developer. To access the demonstrations, open the Samples Browser from the **Start** menu group of the product - click **Samples** for Visual COBOL, or **Samples > Visual COBOL Samples** for Enterprise Developer.

The *Getting Started* section in your product help offers a number of tutorials on programming using managed COBOL.

Aside from additional self-study, you should also consider a dedicated training course in C#, Visual Basic, or Java. These courses will build upon your knowledge of Object-Oriented Programming and enable you to build applications in C#, Visual Basic, Java, or COBOL as the principles remain the same across all of these languages - the key difference being syntax.

A great way to accelerate your understanding of Object-Oriented Programming and managed code frameworks is to work directly with colleagues skilled in C#, Visual Basic, Java or COBOL.

Index