Introducing Micro Focus Net Express with .NET
Net Express with .NET is the latest version of Micro Focus Net Express, the market leading COBOL development environment for Microsoft Windows. As well as providing the extensive functionality provided with previous versions of Net Express, it also provides complete integration with Microsoft’s Visual Studio.NET, incorporating the compiler and other tools needed to develop COBOL applications that run under the Microsoft.NET Framework. This paper introduces Microsoft.NET and the .NET Framework and explains why it is important for many developers who target the Windows platform. The paper goes on to explain why COBOL is such a perfect fit with the other programming languages that support the .NET Framework and provides examples of how the features provided by Net Express with .NET and the .NET Framework can be put to use in your own applications.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction to Microsoft.NET</td>
<td>5</td>
</tr>
<tr>
<td>The .NET Framework</td>
<td>5</td>
</tr>
<tr>
<td>Introducing Net Express with .NET</td>
<td>6</td>
</tr>
<tr>
<td>COBOL's Role in Microsoft.NET</td>
<td>6</td>
</tr>
<tr>
<td>Why Microsoft.NET?</td>
<td>8</td>
</tr>
<tr>
<td>Component Versioning</td>
<td>8</td>
</tr>
<tr>
<td>Mixed-Language Development</td>
<td>9</td>
</tr>
<tr>
<td>Operating System Functionality</td>
<td>10</td>
</tr>
<tr>
<td>Difficult to Access</td>
<td></td>
</tr>
<tr>
<td>The Solution – The .NET Framework and the .NET Languages</td>
<td>11</td>
</tr>
<tr>
<td>The Common Language Runtime</td>
<td>11</td>
</tr>
<tr>
<td>The .NET Framework Class Library</td>
<td>11</td>
</tr>
<tr>
<td>How does the .NET Framework Help Application Developers?</td>
<td>12</td>
</tr>
<tr>
<td>Component Versioning</td>
<td>13</td>
</tr>
<tr>
<td>Multi-Language Support</td>
<td>13</td>
</tr>
<tr>
<td>Easy Access to Operating System Features</td>
<td>14</td>
</tr>
<tr>
<td>Visual Studio.NET</td>
<td>14</td>
</tr>
<tr>
<td>Should I do everything in COBOL?</td>
<td>15</td>
</tr>
<tr>
<td>What about the Existing Net Express IDE?</td>
<td>16</td>
</tr>
<tr>
<td>Should I move to Microsoft.NET now?</td>
<td>17</td>
</tr>
</tbody>
</table>
Introduction to Microsoft .NET

Since its inception, the exact definition of Microsoft .NET has undergone a number of changes. At the time of writing this paper, the Microsoft corporate web site defines Microsoft .NET as “Microsoft® .NET is a set of software technologies for connecting information, people, systems, and devices. This new generation of technology is based on Web services—small building-block applications that can connect to each other as well as to other, larger applications over the Internet”.¹

You might wonder why this is of relevance to you, especially if you are not currently involved in building Internet applications or Web Services. The reason is that Microsoft .NET will have a significant impact on the way all types of applications are developed for the Windows platform. Microsoft uses “.NET” as a broad term to cover a number of technologies that cover operating systems, server products and software development tools and it will have an impact on anyone who develops applications for the Microsoft Windows platform. For software developers, the term “.NET” is usually used to refer to the combination of the Microsoft .NET Framework which is a key piece of the overall .NET picture and Microsoft Visual Studio .NET, the integrated development environment used for creating applications that run under the .NET Framework.

The .NET Framework

The .NET Framework is the infrastructure for the overall Microsoft .NET initiative. It provides a new environment in which applications can be developed that can run on platforms running Microsoft operating systems, from hand-held computers running Windows all the way up to large multi-processor servers running Windows Server. The same techniques and technologies are used for creating graphical standalone applications or for distributed, web-based applications.

The .NET Framework introduces a new runtime system that is common across all supported languages (the Common Language Runtime or CLR), a new intermediate language produced by all language compilers that support the .NET Framework (Microsoft Intermediate Language or MSIL) and a comprehensive class library that provides a common set of classes for all languages that support the .NET Framework. If you are not already familiar with these terms, all of them will be described in more detail later in this paper.

References

¹ http://www.microsoft.com/net/basics/ 29th September 2003
Introducing Net Express with .NET

Net Express with .NET is the latest version of Micro Focus Net Express, the market leading COBOL development environment for Microsoft Windows. Net Express with .NET provides all of the functionality provided in Net Express 4.0. In addition, for customers who want to target Microsoft.NET, Net Express with .NET includes Microsoft Visual Studio Premier Partner edition, a complete version of the Microsoft Visual Studio.NET integrated development environment. Into this, Micro Focus has integrated the compiler and other tools needed to develop COBOL applications that run under the .NET Framework. The compiler compiles code to MSIL and you can run COBOL applications under the CLR. With Net Express with .NET, you make the decision about whether it is right for you to move to the .NET Framework. By providing all of the capabilities of earlier versions of Net Express, as well as the new support for Visual Studio.NET, Net Express with .NET enables you to migrate to the .NET Framework at your own pace, yet continue to support your existing applications on Windows using the tools you are already familiar with.

COBOL’s Role in Microsoft.NET

When people talk about languages for Microsoft.NET, most attention is focused on the languages provided by Microsoft, in particular, Visual Basic.NET and C#. However, the role of COBOL in the enterprise shows no sign of disappearing. According to Pete McBreen, “As Trygve Reenskaug once said, "As time passes, only COBOL lasts." The reality is that in the 1970s and 1980s lots of mission-critical applications were written in COBOL or similar vintage languages such as Assembler, FORTRAN, PL/1, and RPG. Even now, in the age of the Internet, Java, and web services, most companies are still dependent on applications written in these "legacy" languages. The Y2K fiasco didn't kill off all these mission-critical applications; they're just as important as they ever were”.

According to who you listen to, the estimates of how much COBOL code is still in active use in the world varies widely and it is doubtful that anyone can come up with an accurate figure. Ed Arranga quotes a Gartner Group study that between 150 and 175 billion lines of COBOL are currently in production worldwide. One quote that remains true is “Almost everyone criticizes it, laughs at it, kicks sand in its face. Few stand up to defend it. Any yet it remains the most used language in

---


3 Ed Arranga in a round table discussion, “In COBOL's Defense”, IEEE Software, 17(2), March/April 2000, pp. 70-72
software development and maintenance". There is still a huge amount of COBOL code deployed throughout the world, performing critical day-to-day operations. Rewriting that code is rarely a practical or cost-effective option and one figure puts the replacement costs for COBOL systems, estimated at $25 per line, as being in the hundreds of billions of dollars. The logical option is to integrate the COBOL code into the new applications being created using new technologies such as Microsoft.NET. This is where Micro Focus Net Express with .NET enters the picture. It integrates smoothly into the Visual Studio.NET development environment to provide all of the tools you need to bring your COBOL code into the .NET environment. Micro Focus has, for nearly 30 years, made it possible for COBOL programmers to perform tasks in COBOL that the language was never designed for. Micro Focus was the first company to support the development of Windows applications in COBOL, it provided the first 32-bit compiler for Windows and it was the first company to provide support for the development of COM components in COBOL. So, it is only natural that Micro Focus would produce a version of Net Express that provides full support for the latest Microsoft technologies for developing applications.

The overall positioning of Micro Focus COBOL in Microsoft.NET can be seen in figure 1. Not only is COBOL a “first class” language within the .NET Framework, but it can also continue to be used to produce applications using the same tools and techniques that you have been used to using.

---


Why Microsoft.NET?

As mentioned in the previous section, Micro Focus has been providing tools to enable companies to successfully migrate applications to Microsoft Windows platforms for many years. These companies have found the combination of Microsoft Windows, particularly Windows XP, and Micro Focus Net Express to be a stable, reliable platform on which to run their critical business applications. Once the applications have been migrated to Windows, many companies have then extended their applications to take advantage of the new capabilities provided to them by Windows, such as replacing the old character-based user interface with a graphical user interface.

Whether they have been extending existing applications or developing new applications for Windows, developers in all programming languages, not just COBOL, have encountered a common set of issues over the years. It was these issues that the .NET Framework was intended to resolve. Before looking at the .NET Framework in more detail, it is worthwhile looking at the issues that sometimes face developers on Windows.

Component Versioning

Until now, most applications for Windows have been written using development tools that compile the source code for the application into modules containing native machine code instructions. These modules are then linked together to form executable modules. Each application usually consists of one main executable module (an .EXE file) and one or more Dynamic Link Libraries (DLLs) that are dynamically loaded by the operating system as needed by the application. Many of these applications rely on features that are provided by the operating system or third-party companies. These features also reside in DLLs.

As applications, operating systems and third-party tools evolve, new features are added to a DLL and a new version is made available. In order to ensure that they have the version of a particular DLL they need, most application developers include the correct versions of the DLL required for their application with their code when they ship their application. While most application vendors are conscientious about ensuring that they update the DLLs installed on a user's system correctly, if an installation program for the application is not carefully written, this can cause problems:
The installation program for the application could replace an older version of the DLL that is already on the system with a newer version. In theory, new versions of a DLL should be backwards compatible with older versions and should not cause problems for applications that rely on the older version. Unfortunately, this is not always the case and differences in the code in the new DLL may cause the older applications to behave incorrectly or even crash.

Even worse, a badly written installation program for an application can replace a newer version of the DLL that is already on the system with an older version. This will almost certainly cause problems for the applications that are expecting to find the newer version of the DLL. Applications that have been running with no problems for some time suddenly start misbehaving or fail to even start.

A similar problem can occur with COM (Component Object Model) components. COM components are simply DLLs that contain defined interfaces that can be queried and used by applications. Each component and interface provided by a component is identified by a unique number. These identifying numbers need to be stored in the Windows registry of the machine they are installed on so that applications can locate and use the components. This registration of components is normally done by the installation program that installs the components. If the installation program does this incorrectly, then it can cause problems, not only for the application being installed, but also for other applications that have been using an earlier version of the COM component.

**Mixed-Language Development**

It is quite common today for applications to be developed using more than one programming language. For example, applications may be developed using Visual Basic to provide the user interface and COBOL to handle the business logic. Unfortunately, as you may have found if you have previously developed mixed-language applications, the following are just a few of the many pitfalls that you can face when trying to call a module written in one language from a module written in another:

- The way data types are represented in different languages can vary quite considerably. For example, a string in COBOL is a fixed length, space terminated item. In C, a string is a variable length, null terminated item. In Visual Basic, a string is represented as a structure that contains the length and then the actual string. Programmers have had to be aware of these differences and write code to perform the necessary conversions as strings are passed between modules written in different languages.
• The order in which parameters to a routine are processed can vary between languages.

• Some languages pass parameters between programs by passing a pointer to the data, rather than the actual data itself. This is the default for COBOL. In C, the default is to pass a copy of the data to the called program.

• Debugging mixed-language applications has always been difficult. Powerful debugging tools exist for each programming language, but debugging mixed-language applications involves using the different debuggers for each language which involves the programmer having to switch between different user interfaces during a debugging session.

Another issue with using multiple programming languages is that each language has its own runtime system modules that need to be included with the application. The runtime system loads and runs programs, as well as providing features such as memory management and interfacing with the operating system. Each language has had its own runtime system that provides the specific features required for that language. For example, if you develop an application using Visual Basic and COBOL, you need to include both the Visual Basic runtime system and the COBOL runtime system with your application. This can increase the size of your distributable application quite significantly.

Operating System Functionality Difficult to Access

To access features in the operating system, a program needs to either make calls to the Windows API (Application Programming Interface) or, in the case of features added more recently, make use of COM components. The Windows API was originally designed for C and C++ programmers. This means that the conventions for calling the API functions, the data-types used and the way parameters are passed are those used by C and C++. Other programming languages such as COBOL have had to be enhanced to provide capabilities that enable the API functions to be called from programs written using that language. To complicate matters further, the mechanisms used have been quite low-level, utilizing features such as pointers to data and functions, parameters passed by value and other language features that have not been part of the standard COBOL language in the past and therefore unfamiliar to COBOL programmers. Over the years, Micro Focus and third-parties have provided many excellent mechanisms by which the low-level functionality of the operating system can be accessed from COBOL. However, it is difficult for them to keep up with the constant changes and additions made to the operating systems.
The Solution – The .NET Framework and the .NET Languages

The designers of the Microsoft .NET Framework set out to solve the issues previously described, as well as provide new capabilities for software developers. Before looking at how the problems are solved, it is worthwhile taking an initial look at the two main pieces of the .NET Framework – The Common Language Runtime and the .NET Framework class library.

The Common Language Runtime

At the center of the .NET Framework is the Common Language Runtime (CLR). This loads and executes code that has been compiled by one of the language compilers that supports Microsoft .NET. These compilers do not produce modules that contain native code. Instead, they compile the source code to an intermediate format called Microsoft Intermediate Language (MSIL). This is often referred to simply as IL. MSIL is portable across all platforms that support the .NET Framework. At the moment, this includes all 32-bit Windows operating systems and the .NET Compact Framework that runs on the Windows platform used by many handheld computers. In the future, other platforms will support the .NET Framework, including 64-bit versions of Windows.

Modules containing MSIL that are produced by a compiler for Microsoft .NET are known as managed code. The modules produced by compilers that do not generate MSIL are known as unmanaged code.

Although MSIL is an intermediate language, the code is not interpreted at runtime. When the code is required to be executed, it is automatically compiled using a just-in-time compiler to the native machine instructions of the machine the code is to be executed on. The just-in-time compilers are provided as part of the .NET Framework.

This is a major change from how applications have been executed in the past. Previously, it was the responsibility of the language tools vendors to provide the compilers that compiled the source code to native code and the runtime systems to execute that code. Now, all the language vendor needs to supply is the compiler that compiles source code to MSIL. Regardless of which language the application was written in, the application is executed by the CLR.

The .NET Framework Class Library

The .NET Framework class library gives your application access to all of the services provided by the operating system. One way to view this is as a ‘wrapper’ for the

- 11 -
Windows API and the other system-level APIs. All languages for Microsoft.NET use the same class library – there is not a different version for each language.

If you are a COBOL programmer, you may have noticed one issue arising from the above paragraph. The use of terms such as class implies the use of an object-oriented language. This is indeed the case and all languages, including COBOL, that support Microsoft.NET have to provide support for object-orientation. Luckily, the recently published COBOL standard includes object orientation and support for object-orientation is included in Micro Focus Net Express with .NET. This will be covered later in this paper.

The .NET Framework class library contains thousands of classes and interfaces, providing access to functionality such as:

- User interface support, including traditional Windows user interfaces (WinForm), web-based user interfaces (WebForm) and console interfaces
- Data access and manipulation of relational and non-relational data sources (ADO.NET)
- Sending and receiving data using different network protocols
- Creating and working with Windows system-level services
- Application configuration
- Creation and management of threads for multi-threaded applications
- Encrypting and decrypting files
- Services for application security

How does the .NET Framework Help Application Developers?

Later sections will examine how the new features offered by the .NET Framework can be exploited from COBOL. However, given the descriptions of the issues faced by developers earlier, it is worthwhile examining how the .NET Framework solves those issues.
Component Versioning

Firstly, the complexities of deploying applications has been made much simpler, making life for the creator of the installation programs much easier and the process less prone to errors. Executable modules produced under the .NET Framework are known as Assemblies. They are self-describing, which means that, as well as containing the MSIL code, they also contain Meta Data. Meta Data contains all of the information the CLR needs to know about the module, including details of all of the interfaces exposed by the module and the security requirements for the module. This information no longer needs to be placed in the Windows registry. A module can be copied to a new computer and used immediately without needing to go through the process of registering the module.

Just as importantly, the CLR can load multiple versions of a module into memory at the same time. The Meta Data for each assembly identifies the particular version of the module that it requires and the CLR loads the correct version for that module. This means that the problems of incompatible versions are significantly reduced by the .NET Framework and, for new applications; the problems of having the wrong components installed are a thing of the past.

Multi-Language Support

The CLR only executes code compiled to MSIL. The original source language used is irrelevant as long as the compiler can produce MSIL. All languages for Microsoft.NET use the same data types and the same conventions for interoperability. This makes the task of calling a module written in a different programming language significantly easier.

What this means for COBOL programmers is that COBOL becomes a ‘First Class’ language under Microsoft.NET. Anything you can do in C# or Visual Basic.NET, you can also do in COBOL. Therefore, you can choose the language based on its suitability for the programming task, rather than whether it supports the operating system. For complex business processing, you might choose Net Express with .NET, but for creating a user interface, you might prefer to use Visual Basic.NET.

Since all languages use the CLR, you no longer need to provide a different runtime system for each language. Currently, the .NET Framework is a separately installable package, but over time, the .NET Framework will become a standard feature of Windows operating systems. This will lead to smaller distributable packages for applications since they will no longer need to include a runtime system – it will be included with the operating system.
Easy Access to Operating System Features

All languages for Microsoft.NET support the .NET Framework class library. Therefore, regardless of the programming language being used, all application developers have access to the full functionality provided by the operating system. In addition, developers do not need to learn a different mechanism for different programming languages – the .NET Framework class library is the same for each language.

Visual Studio.NET

Until now, virtually every programming language came with a different integrated development environment (IDE) that included the tools needed to build and debug applications in that language. Although the basic features of the development environments were the same, the menu structures and window formats were different, requiring programmers to learn different tools for each language. Microsoft provided different development environments for its own programming languages such as Visual C++ and Visual Basic. For COBOL developers, Micro Focus provided the Net Express IDE.

Alongside the introduction of the .NET Framework, Microsoft has consolidated all of its development environments into one – Visual Studio.NET. All of the Microsoft languages use the same IDE and you can switch from using one language to another inside the same IDE.

Microsoft has also taken the major step of making the Visual Studio.NET IDE available to other programming language vendors. This means that languages supplied by vendors other than Microsoft can be developed inside the Visual Studio.NET IDE and is quickly leading to Visual Studio.NET being the standard IDE used for most languages for Microsoft.NET. This has many benefits for software developers:

- You can use the same editor, debugger and other tools for all languages. This means that you only need to learn one set of menus, keystrokes, etc.
- You can step from one language to another in the same debugger when debugging mixed-language applications.
- As Microsoft add new capabilities to Visual Studio.NET, they become available to all of the languages, not just the languages provided by Microsoft.

Because of these advantages, programming language vendors have been quick to adopt Visual Studio.NET for their own products. This includes Micro Focus. Net Express with NET is integrated into Visual Studio.NET. Now, when programming
for .NET, programmers can use the same Visual Studio.NET for developing COBOL programs that they use for developing programs in other languages.

**Should I do everything in COBOL?**

Of course, if you want to, you can develop entire applications for Microsoft.NET using Net Express with .NET. Net Express with .NET provides COBOL programmers with full access to all of the classes and methods in the .NET Framework. However, you do not need to and, in some cases, you might find it beneficial to use some of the other available languages for Microsoft.NET to implement parts of your application. If you install Net Express with .NET on to a machine that already has Visual Studio.NET 2003 installed, including one or more of the Microsoft.NET languages, Net Express with .NET will integrate into your existing copy of Visual Studio.NET. You will be able to develop COBOL modules alongside Visual Basic.NET, C#, and so on, all in the same development environment. Given that the .NET Framework and Visual Studio.NET provides such an excellent environment for mixed-language development, rather than developing the entire application in COBOL, it might be more appropriate to use the best language for the particular task to be performed.

COBOL is a perfect language for solving business problems. As Glass states: “Business programming needs to be boiled down to four fairly straightforward characteristics:

- The capability for heterogeneous, “record-structure” data
- The capability for decimal arithmetic
- The capability for convenient report generation
- The capability for accessing and manipulating masses of data (typically made up of heterogeneous data structure records).

COBOL is good or adequate in all four, whereas more recent COBOL “replacement” languages like Visual Basic and Java are good at few if any of them”. Micro Focus COBOL excels in all of these four characteristics. As well as all of the features of standard COBOL, Micro Focus provides enhanced access to data via the support for relational databases provided by OpenESQL. This support has been further enhanced in Net Express with .NET by the addition of EXEC ADO which provides powerful support for the manipulation of ADO.NET datasets and provides full access to the functionality of ADO.NET in a COBOL-centric manner without having to deal with the underlying classes and methods. EXEC ADO will be covered in detail in a future paper.
However, when it comes to tasks such as developing graphical user interfaces, although it is entirely possible to do this in COBOL, most of the code involved in such tasks consists of a series of invokes of methods or the setting or retrieving of properties of objects. Because there are so many web sites that provide many examples of using Visual Basic.NET and C# that can be easily reused and because of the additional level of integration that Visual Basic.NET and C# have with the design tools in Visual Studio.NET, an option that might appeal to you is to use Visual Basic.NET or C# for developing the front-end of your application and use COBOL where it really excels, to perform the critical, back-end, business processing.

Every programming language has its fans who will argue that their particular language can be used for everything and that no other language is needed. COBOL is no different. When mixed-language programming was difficult, then there was a much stronger case for only using one particular language for a project. The introduction of Visual Studio.NET and the .NET Framework has changed that. Now, it is easy to use the best language for the task at hand. If you want to create a graphical user interface, you can use Visual Basic.NET or C#. However, for the business processes of your application, code them in COBOL and simply call the COBOL code from the other languages.

What about the Existing Net Express IDE?

Given the statements in earlier sections about Visual Studio.NET, you might have questions about the future of the existing integrated development environment in Net Express. Micro Focus recognises that, although some customers might make the move to the .NET Framework and Visual Studio.NET immediately, many others will not be in a position to migrate to the Microsoft.NET platform yet. Many users of Net Express use it for cross-platform development where Windows is used as the development platform, but the applications are architected to run on many different operating systems. The decision about which development environment is best for you can be summarised as follows.

- If you are developing applications to be deployed in the .NET Framework, use Visual Studio.NET
- If you are developing applications to be deployed on Windows, but are not making use of the features of the .NET Framework or are developing cross-platform applications that will be deployed on one or more of the varieties of Unix or Linux, use the existing Net Express development environment.

Both development environments are fully supported for the roles described above. Over time, you can expect that Micro Focus will enhance the support for Visual Studio.NET to cover other development scenarios.
Should I move to Microsoft.NET now?

Having seen the benefits of the.NET Framework, you might be wondering if it makes sense for you to make the move to Microsoft.NET immediately. There is no easy answer to that question since it depends on your particular environment and the applications you are developing.

The key point to make is that the .NET Framework introduces a new way of developing applications for Windows. It does not introduce a new version of Windows and it does not make your existing Windows applications obsolete. If you have been developing applications for Windows already, then your existing tools will continue to work and you can continue to use them.

There are four main categories of COBOL application which can benefit from the .NET Framework today:

1. Applications that need to be updated with a graphical user interface or web-based user interface, but do not have such an interface today. In this case, if the COBOL business logic can be separated out and made callable, then using Visual Studio.NET to produce the front-end code is an option.

2. Existing componentized applications that make use of COBOL components that are being migrated to the .NET Framework.

3. Batch style applications that perform specific tasks, but which could benefit from certain features provided by the .NET Framework.

4. If you have a COBOL application that you want to publish as a web service under the .NET Framework using a similar mechanism to other languages for Microsoft.NET. Web services are software components that can be accessed using standard internet protocols. If you are not familiar with web services and how they apply to COBOL, more information can be found at http://www.cobolportal.com/webservices/.

The remaining sections in this paper examine examples of scenarios 1 and 3 in detail.
Extending an Existing Application with .NET Framework Functionality

This section will look at an example of how a COBOL program can be enhanced by the use of functionality provided by the .NET Framework class library.

The example is a simple report generating program. This is a very common type of task that COBOL excels at. The report is a simple sales summary. The program has been enhanced by providing the capability of emailing the generated report as an attachment to someone.

The following shows the source code for this example. The code to generate the actual report has been removed for brevity. The complete program is supplied in the example code available with this paper:

```cobol
$set sourceformat(variable)

environment division.  
* The repository is used to declare which .NET classes  
* and enumerations we are going to use in this  
* application.  
repository.  
class MailAttachment as  
"System.Web.Mail.MailAttachment"  
class SmtpMail as "System.Web.Mail.SmtpMail"  
class IList as "System.Collections.IList"  
enum MailFormat as "System.Web.Mail.MailFormat"  
class ExceptionClass as "System.Exception"
.

input-output section.  
file-control.  
select sales-file  
assign to disk "sales.di"  
organization is indexed  
access is sequential  
record key is sr-record-id  
alternate record key is sr-sales-rep  
with duplicates.  
select sales-report  
assign to printer  
"c:\ReportLocation\sales-rpt.txt".
```
data division.

file section.

fd sales-file.
01 sr-sales-record.
  05 sr-record-id pic x(5).
  05 sr-record-code pic x(2).
  05 sr-state pic x(2).
  05 sr-branch pic x(3).
  05 sr-sales-rep pic x(5).
  05 sr-date-of-sale pic x(6).
  05 sr-product-code pic x(7).
  05 sr-product-description pic x(16).
  05 sr-quantity-sold pic s9(3).
  05 sr-unit-price pic 9(2)v99.

fd sales-report.
01 sales-report-line.
  05 pic x(132).

working-storage section.

* Rest of working storage section removed for brevity

01 Object-References.
  * These are the variables used to hold references to objects. All of the object types referenced were declared in the Repository previously.
  03 MyMailMessage object reference MailMessage.
  03 MyMailAttachment object reference MailAttachment.
  03 MyMailIList object reference IList.
  03 MailException object reference ExceptionClass.

01 Mail-Details.
  * These variables specify the information needed for the email, such as the 'To' address, the name of the SMTP server and so on.
  03 Mail-From pic x(26) value "fromperson@fromaddress.com".
  03 Mail-To pic x(22) value "toperson@toaddress.com".
  03 Mail-Subject pic x(12) value "Sales Report".
  03 Mail-Body pic x(66) value "Here is the sales report from today" & x"0D0A0D0D0A" & "Regards" & x"0D0A0D0D0A" & "Report Generator".
  03 SmtpServer pic x(20) value
procedure division.

000-print-sales-report.
  * Initialization sequence
  open input sales-file
  output sales-report
  *
  * Code to generate report removed
  *
  close sales-file
  sales-report
  perform 300-Email-Report
  stop run.

300-Email-Report.
  * The first thing we need to do is create a new
  * instance of the MailMessage object.
  invoke MailMessage "NEW" returning MyMailMessage
  * Now set the various properties of the mail message
  set MyMailMessage::"From" to Mail-From
  set MyMailMessage::"To" to Mail-To
  set MyMailMessage::"Subject" to Mail-Subject
  set MyMailMessage::"Body" to Mail-Body
  * The format is set using an enumeration specified in
  * the .NET Framework
  set MyMailMessage::"BodyFormat" to MailFormat::"Text"
  * Create the attachment
  invoke MailAttachment "NEW" using
    "C:\ReportLocation\sales-rpt.txt"
    returning MyMailAttachment
  * and add it to the message
  set MyMailIList to MyMailMessage::"Attachments"
  invoke MyMailIList "Add" using MyMailAttachment
  * Now set the name of the SMTP server we are going to
  * use
  set SmtpMail::"SmtpServer" to SmtpServer
  * Send the email. This code is put inside a
  * try...catch block in order to trap and handle
  * exceptions that may occur (such as being unable to
  * contact the SMTP server)
  try
    invoke SmtpMail "Send" using MyMailMessage
    display "Email sent"
  catch MailException
    * An exception occurred while trying to send the
    * email. Work down the exception list,
displaying all of the messages returned at each level
display "Exception occurred:"
perform with test after until MailException = null
    set ExceptionText to MailException::"Message"
* Remove trailing spaces
move 256 to ExceptionTextLen
perform until
    ExceptionText(ExceptionTextLen:1) not = space
    or ExceptionTextLen = 0
    subtract 1 from ExceptionTextLen
end-perform
display "  ".
    ExceptionText(1:ExceptionTextLen)
set MailException to MailException::"InnerException"
end-perform
end-try.

Many of the techniques used here may be new to you, so the following sections, introduce the key aspects of the program. The syntax used to access the classes and methods is the new syntax to support object orientation introduced in the most recent COBOL standard.

Variable Source Format

The first thing to notice in this program is the compiler directive at the start of the code:

$set sourceformat(variable)

Micro Focus has offered the directive sourceformat(free) for some time. This directive enables COBOL source code to be placed anywhere on the line, without worrying about whether comment markers are in column 7, the line must be broken before column 72 and so on. The new directive, sourceformat(variable), works well with existing code which uses the COBOL column structure already (for example, comment indicators in column 7), but where you want more flexibility in the future to have longer lines. This directive retains the standard COBOL usage of columns 1 through 7, but it allows lines of code to be longer than column 71. This can often help increase readability when you have long statements that use long variable names or long literals. You no longer need to break the lines at column 72.

Note. To enable the source code to fit inside the margins of this paper, the lines of code have been split at appropriate points in the example above. If you examine
the source code provided with this paper, you will see that many of the lines extend beyond column 71.

The Repository

In order to use any of the classes or enumerations in the .NET Framework class library, you need to declare it in the repository section. This associates a name with each class or enumeration that will be used later in the program. For example:

```csharp
repository.
    class MailAttachment as "System.Web.Mail.MailAttachment"
    class SmtpMail as "System.Web.Mail.SmtpMail"
    class IList as "System.Collections.IList"
    enum MailFormat as "System.Web.Mail.MailFormat"
    class Exception as "System.Exception"
```

A discussion of the different classes that form the .NET Framework class library is beyond the scope of this paper. The best source of information on the classes that make up the .NET Framework class library and their constituent methods and properties is the Microsoft Developer Network Library. This can be found on the web at [http://msdn.microsoft.com/library/default.asp](http://msdn.microsoft.com/library/default.asp). Descriptions of all of the classes used in this example and in the many examples that Micro Focus provide with Net Express with .NET can be found at this location.

If you are familiar with the new COBOL standard, you might be surprised that the properties of objects used in this program are not also declared in the repository as defined in the standard. For example, you could have a declaration of the form:

```csharp
property ToAddress as “To”
```

If you wish, you can do this. However, if your program sets or retrieves a large number of properties, having to give names to all of the properties can result in a very large repository and can obscure readability. A very useful Micro Focus extension enables you to reference a property directly without needing to declare it in the repository. Depending on your point of view, this improves readability of the code, so this extension has been used in the example programs provided with this paper. However, this is a matter of personal preference and there is no performance penalty for using either method. You will see examples of setting properties later.
Object References

Most of the classes used by this program require that an instance of an object of that class is created before using it. In order to do this, you need to declare variables of type object reference. For example:

03 MyMailMessage   object reference MailMessage.

This indicates that the variable MyMailMessage is used to hold a reference to an instance of the class System.Web.Mail.MailMessage (this was declared in the repository earlier).

However, you will find methods and properties in some classes that are declared as static. In this case, you do not need to create an instance of the object before using the method. For example, the methods and properties used in the class System.Web.Mail.SmtpMail are static methods and properties. In this case, the class reference is used instead of an object reference, e.g.:

invoke SmtpMail "Send" using MyMailMessage

Creating a New Instance of an Object

As mentioned previously, before you can use most of the classes in the .NET Framework class library, you need to create a new instance of an object of that class. One way to create a new instance is to use the "NEW" method. For example:

invoke MailMessage "NEW" returning MyMailMessage

This creates a new instance of an object of the class System.Web.Mail.MailMessage. We can now call methods on this object or set or retrieve properties of the object.

Another mechanism for obtaining a reference to an object for use in your program is as a returned value from a method or property of another object. For example:

set MyMailIList to MyMailMessage::"Attachments"

This accesses the Attachments property of the mail message which returns a reference to an object of class System.Collections.IList which can subsequently be used in your program.

Methods

Methods are actions that an object can perform. They are called using the invoke syntax. For instance:

invoke MyMailIList "Add" using MyMailAttachment

This code adds the specified attachment to the list of attachments for the mail message.
Properties

Properties define the state of the object. For example:

```
set MyMailMessage::"To" to Mail-To
```

sets the target of the email, that is, who it is going to be sent to, to the value contained in the variable Mail-To.

The following line sets the variable MailException to the value of the property “InnerException” on the specified exception object.

```
set MailException to MailException::"InnerException"
```

As mentioned earlier, this program utilises a Micro Focus extension that makes using properties much easier. If you wish, you can use the standard COBOL syntax. In the first example, you would declare the property in the repository section using:

```
Property ToAddress as "To"
```

You could then set this property using the syntax:

```
set ToAddress of MyMailMessage to Mail-To
```

Which syntax you use is down to your own personal preference.

Enumerations

An enumeration is a range of values that are identified by name. For example, the enumeration System.Web.Mail.MailFormat has two values – *Html* which specifies that the format of the email will be HTML and *Text* which specifies that the format of the email will be in text format. To use an enumeration in a COBOL program, declare it in the repository section. For example:

```
enum MailFormat as "System.Web.Mail.MailFormat"
```

The values of an enumeration can be specified using the same syntax used for properties. For example, the following line sets the format of the body of the mail message to send to be a text message:

```
set MyMailMessage::"BodyFormat" to MailFormat::"Text"
```

Exception Handling

New syntax has been added to Net Express with .NET to provide powerful support for exception handling. You can see this in the use of the new syntax *try...catch* around the code that attempts to send the email message.

```
try
  invoke SmtpMail "Send" using MyMailMessage
display "Email sent"
```

catch MailException
*        An exception occurred while trying to send the
*        email.
...
end-try.

This enables you to trap any exception that occurs during the execution of the code between the try and catch statements.

Exceptions are an alternative mechanism for handling errors that occur in a program. Traditionally, a programmer would test for errors after statements that could potentially cause an error by testing return codes or other variables that are used to hold error conditions. There are a few problems inherent with this approach:

- The error handling code is intertwined with the business logic. This can obscure the readability of the code and make it hard to understand.

- It is very easy to ignore a return value or forget to test a particular case.

To make error handling easier, most errors in the .NET Framework are raised as exceptions. The main idea in exception handling is to separate the code that implements the business logic of the program from the error handling code.

The business logic is placed inside a try block. If none of the statements inside the try block generates an exception, they all run, one after the other, to completion. However, if an exception occurs, control is passed to the code in the catch block. This code can then perform the appropriate actions needed to correct the error if possible or inform the user if this is not possible. In this example, the code simply loops through the list of exceptions raised and displays them to the user.

**Garbage Collection**

Finally, if you have used Micro Focus COBOL to create COM (Component Object Model) components in the past, you might have noticed that new instances of objects created by this program are not explicitly destroyed. When using COM components, if you do not destroy or finalize the component when you have finished using it, the component will remain in memory, leading to potential problems later. One of the advantages of the .NET Framework is that all managed objects are automatically removed from memory when they are no longer needed by the system garbage collection process. Programmers no longer need to explicitly destroy instances of objects created by the program.
Calling COBOL from Other Languages under the .NET Framework

The previous example demonstrated how a COBOL application can be extended to utilize functionality provided by the .NET Framework class library. Another area where you will find a reason to migrate to the .NET Framework is if you want to use your COBOL skills to create the business logic for your applications, but you are considering using one of the other languages that support the .NET Framework to produce a compelling user interface. This could be either a traditional Windows graphical user interface or a web-based user interface using the ASP.NET technology.

This section will examine how COBOL code can be called from other languages. The example used is very simple, but the techniques can be applied to far more complex examples.

The Example

The COBOL program used here is a simple query example. It looks up patrons of a small library and returns information about them. The program is called Library-Subprogram. Its linkage section and procedure division using statement can be seen below:

Linkage Section.

01  ls-patron-number   pic X(03).
01  ls-patron-info.
   10  ls-patron-name   pic X(23).
   10  ls-street-address   pic X(20).
   10  ls-city-state-zip   pic X(28).
   10  ls-book-right-sw   pic X comp-5.
   10  ls-books-out   pic 9(02).
   10  ls-seniority-date.
      15  ls-seniority-month   pic 9(02).
      15  ls-seniority-day   pic 9(02).
      15  ls-seniority-year   pic 9(04).
   10  ls-total-fees   pic 999V99
      packed-decimal.

Procedure Division Using ls-patron-number
   ls-patron-info.
As you can see, the program has two parameters. The patron-number contains the reference number of the patron to be queried. The patron-info record returns the information about that patron of the library. The focus of this section will be on how COBOL programs that use data types specific to COBOL, such as those used in this example, can be used effectively from other languages.

Creating a Wrapper Class

The .NET Framework is fundamentally an object-oriented environment. The languages for the .NET Framework provided by Microsoft are inherently object-oriented languages and, as a result, for most actions, the way that languages inter-operate is via classes and methods. In addition, the data types used by the other languages for the .NET Framework vary quite significantly to those normally used in COBOL applications. Most languages do not have types such as the fixed-point decimal numeric types in COBOL – which is one of the reasons why COBOL is still such a powerful language for business processing. However, because other languages do not include these types, there invariably needs to be some data type conversion when COBOL is called from another language.

Therefore, if you wish to continue to perform your COBOL programming using standard procedural COBOL or you have existing procedural COBOL code that you want to reuse, the easiest approach to making this code accessible to other languages is to create a simple wrapper class in COBOL that acts as the interface between other languages for .NET and your procedural code. Creating a wrapper class for your COBOL program allows your program to be exposed to other languages using data types that they normally use, but at the same time passing your COBOL program data in the format it expects. Of course, if you are writing new code, you can write it all using object-oriented techniques and ensure that the data types used for parameters conform to those expected by other languages. In this case, you would not need to create a wrapper since your class could be used directly.

Data Types

Before looking at how to create a wrapper class, it is worthwhile taking a quick look at how different data types are handled.

All of the languages that support the .NET Framework have data types that are unique to that particular language. However, if you wish to create classes that can be used from any language that supports the .NET Framework, you should ensure that the data types used as parameters are compatible with the types used in other languages. One of the specifications in the .NET Framework is for the Common Language Specification (CLS). This defines a set of rules for interoperability of languages, including a set of common types. If you just use these types for
parameters, then you can be sure that any other language will be able to make use of your class.

To make things easier, many of the new COBOL data types defined in the ISO 2002 standard map directly on to the CLS common types. However, you can also use the traditional style of COBOL type definition. The table below shows the names of the basic common types, the COBOL ISO 2002 equivalent and the traditional COBOL equivalent.

<table>
<thead>
<tr>
<th>.NET Type</th>
<th>Description</th>
<th>COBOL ISO 2002 equivalent</th>
<th>‘Traditional’ COBOL equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>An 8-bit unsigned integer</td>
<td>Binary-char unsigned</td>
<td>Pic 9(2) comp-5 or pic x comp-5</td>
</tr>
<tr>
<td>Int16</td>
<td>A 16-bit signed integer</td>
<td>Binary-short</td>
<td>Pic 9(4) comp-5 or pic xx comp-5</td>
</tr>
<tr>
<td>Int32</td>
<td>A 32-bit signed integer</td>
<td>Binary-long</td>
<td>Pic 9(9) comp-5 or pic x(4) comp-5</td>
</tr>
<tr>
<td>Int64</td>
<td>A 64-bit signed integer</td>
<td>Binary-double</td>
<td>Pic 9(18) comp-5 or pic x(8) comp-5</td>
</tr>
<tr>
<td>Single</td>
<td>A single-precision (32-bit) floating point number</td>
<td>Float-short</td>
<td>Comp-1</td>
</tr>
<tr>
<td>Double</td>
<td>A double-precision (64-bit) floating point number</td>
<td>Float-long</td>
<td>Comp-2</td>
</tr>
<tr>
<td>Boolean</td>
<td>A Boolean value (true or false)</td>
<td>Condition-value</td>
<td>Not available</td>
</tr>
</tbody>
</table>

If your COBOL program contains types such as these in the linkage section, then no additional conversion is needed. However for other numeric data types, such as usage packed decimal or usage display, conversion will be needed between the CLS type and the COBOL data type.

You may have noticed that there is one key data type missing from the above table - the string. In the .NET Framework, a string is represented using the class System.String that is included in the .NET Framework class library. Therefore, if
your program is expecting to be passed a string, conversion will be needed to the pic x(n) format used in COBOL. Net Express with .NET makes this conversion process very easy as we will see later.

Finally, there is no data type that corresponds to the COBOL record structure. So, we need a mechanism by which COBOL records can be exposed to other languages and this will be seen in the next section.

**Representing a COBOL Record using a Class**

The best way to make a COBOL record structure available to other languages is to create a COBOL class that encapsulates the record inside an object and makes the fields of the record visible to other languages as properties of the object. The methods in the COBOL class that set and get the properties handle any data conversion needed. In this application, a class called PatronInfo has been provided to represent the patron-info record.

**Application Structure**

Before looking at the individual components of the application in detail, figure 2 shows the overall application structure and how the different pieces fit together.

![Figure 2 – Application Structure](image-url)
The PatronInfo Class

The full source for PatronInfo.cbl can be found in the Appendix. As mentioned earlier, this class provides a mechanism by which a COBOL record can be accessed and updated by other programming languages in a way that is natural for those languages. Each field in the patron-info record is exposed as a property. For most of the fields, methods have been written to set or get the property since some type of data conversion is required. As you will see in the code for the PatronInfo class, the methods contain code that simply converts the data from one format to another. In most cases, this can be handled via a single COBOL move or set statement. For example, in the case of the field ws-total-fees, this is represented in the COBOL record as a packed-decimal field. The PatronInfo class exposes this field as a property called ‘Fees’ of type float-short, i.e. a single-precision floating point number. The methods to set and get this property are as follows (the variable ws-total-fees is the field in the ws-patron-info record structure):

```
method-id. get property fees as "Fees".
data division.
linkage section.
01 ls-total-fees  float-short.
procedure division returning ls-total-fees.
  move ws-total-fees to ls-total-fees
  exit method.
end method.

method-id. set property fees as "Fees".
data division.
linkage section.
01 ls-total-fees  float-short.
procedure division using ls-total-fees.
  move ls-total-fees to ws-total-fees
  exit method.
end method.
```

The method to set the property simply moves the floating point value to the packed-decimal field. The method to get the property moves the packed decimal value to the floating point value.

Most of the other fields in the record are set in a similar way. The only exception is the field ws-book-right-sw. This field is defined as pic x comp-5 which, as can be seen in the earlier table, corresponds directly to the CLS common type Byte. In this
case, no data conversion is needed, so we can just specify the PROPERTY clause on
the field, e.g.

10 ws-book-right-sw pic X comp-5 property as "BookRights".

The COBOL compiler automatically generates the methods to set and get the
property.
One point to note is that not only can the techniques covered here be used to convert between basic data types, but they can also be used to convert more abstract types such as dates. For example, some of the fields in the patron-info record structure define a date, i.e.

```plaintext
10  ws-seniority-date.
15  ws-seniority-month  pic 9(02).
15  ws-seniority-day    pic 9(02).
15  ws-seniority-year   pic 9(04).
```

In Visual Basic.NET or C#, a logical way to represent a date is to use the class System.DateTime. Therefore, property set and get methods have been defined that expose the date information as an object of type System.DateTime. The code for these methods can be seen below:

```plaintext
method-id. get property senioritydate as "SeniorityDate".
data division.
local-storage section.
   01 year-part   binary-long.
   01 month-part  binary-long.
   01 day-part    binary-long.

linkage section.
   01 ls-seniority-date  object reference DateTime.

procedure division returning ls-seniority-date.
   move ws-seniority-year to year-part
   move ws-seniority-month to month-part
   move ws-seniority-day  to day-part
   *    Create a new object of type System.DateTime to
   *    be passed back to the caller
   invoke DateTime "NEW" using year-part
                month-part
                day-part
       returning ls-seniority-date
   exit method.
end method.
```

```plaintext
method-id. set property senioritydate as "SeniorityDate".
data division.
local-storage section.
   01 year-part   binary-long.
   01 month-part  binary-long.
   01 day-part    binary-long.

linkage section.
```
01 ls-seniority-date object reference DateTime.

procedure division using ls-seniority-date.
  * Use properties of the System.DateTime class
  * to get the component parts of the date
set year-part to ls-seniority-date::"Year"
move year-part to ws-seniority-year
set month-part to ls-seniority-date::"Month"
move month-part to ws-seniority-month
set day-part to ls-seniority-date::"Day"
move day-part to ws-seniority-day
exit method.
end method.

Finally, two methods have been added to the class that are used by the wrapper class to retrieve the complete contents of the record (RetrieveRecord) before the call to the original COBOL program and to update the contents of the record (UpdateRecord) upon return from the COBOL program.

The Wrapper Class

All that is needed now is the wrapper class, inquiry.cbl. This can be seen below:

$set sourceformat(variable)
class-id. Inquiry as "Inquiry".
environment division.
configuration section.
repository.
  class StringClass as "System.String"
  class PatronInfo as "Library.PatronInfo".
static.
data division.
working-storage section.

  01  ws-patron-number pic X(03).

  01  ws-patron-info.
    10  ws-patron-name pic X(23).
    10  ws-street-address pic X(20).
    10  ws-city-state-zip pic X(28).
    10  ws-book-right-sw pic X comp-5.
    10  ws-books-out pic 9(02).
    10  ws-seniority-date.
    15  ws-seniority-month pic 9(02).
    15  ws-seniority-day pic 9(02).
    15  ws-seniority-year pic 9(04).
The class simply contains one static method that converts the parameters into the format expected by Library-Subprogram, calls the COBOL program and then converts the parameters back before exiting.

The patron-number parameter is supplied as an object reference to a data item defined as type System.String. To move the contents of this to a pic x(n) field, all that is needed is the COBOL ‘set’ statement, e.g.

```
set ws-patron-number to patron-number
```

Similarly, to move the contents of the pic x(n) field back to the System.String object, we can use:
set patron-number to ws-patron-number

Note that the wrapper class uses the RetrieveRecord method in the PatronInfo class to read the complete record before calling the COBOL program and the UpdateRecord Method to update the contents upon return from the COBOL program.

Using the Program from Other Languages

To use the program from other languages, the steps are simply:

- Create a new object of type PatronInfo
- Set the properties in PatronInfo and the value of the patron number
- Call the sub-program via the wrapper class
- Read the properties from PatronInfo

The following code shows the Visual Basic.NET code used to accomplish this. The fields that are not defined in the code (such as PatronNumberField) are input/output fields on a form.

```vbnet
Dim PatronInfo As New Library.PatronInfo
Dim PatronNumber As String

PatronNumber = PatronNumberField.Text
PatronInfo.Name = ""
PatronInfo.StreetAddress = ""
PatronInfo.CityStateZip = ""
PatronInfo.BooksOut = 0
PatronInfo.Fees = 0.0
PatronInfo.SeniorityDate = DateTime.Now
Library.Inquiry.Execute(PatronNumber, PatronInfo)

PatronName.Text = PatronInfo.Name
PatronStreetAddress.Text = PatronInfo.StreetAddress
PatronCityStateZip.Text = PatronInfo.CityStateZip
PatronFees.Text = PatronInfo.Fees.ToString
PatronSeniorityDate.Text = PatronInfo.SeniorityDate.ToLongDateString
```

The techniques described here can be used to make any callable COBOL program usable from other languages in a manner that is natural for those languages. By creating a separate COBOL class for each of the record structures that are passed as parameters and a wrapper class to handle any other data type conversion required, programmers using Visual Basic.NET or C# do not need to have any knowledge of
COBOL data types and the existing COBOL code can be used without any modification.

**Further Reading**

This paper can only touch on some of the aspects of language interoperability. For more information, the on-line documentation provided with Net Express with .NET provides an extensive section on interoperability.
Conclusion

This paper has introduced Microsoft.NET and the .NET Framework and explained the important role that Net Express with .NET has to play in this environment. By integrating COBOL into the powerful Visual Studio.NET development environment, Micro Focus provides a painless route to incorporating COBOL into applications that target the .NET Framework and making the vast .NET Framework class library available to your COBOL applications. Future papers will explore how Net Express with .NET opens up other aspects of the .NET Framework to COBOL programmers.

About the author: Wayne Rippin is a lecturer in computing at the University of Derby in England. Previously, he worked for Micro Focus for 16 years, first as a systems programmer and later as a product manager. His most recent role there was director of product management, leading a team of product managers responsible for Net Express, Mainframe Express and UNIX compiler products.

© 2003 Micro Focus. All Rights Reserved. Net Express is a registered trademark, and Server Express is a trademark of Micro Focus. Other trademarks are the property of their respective owners.
Appendix

The PatronInfo Class

$set sourceformat(variable)
* Class to represent the patron-info record.
* This exposes the fields of the patron-info record as
* properties that can be read/written by other languages.
* The properties are exposed as types familiar to those
* languages.
*
* In most cases, two methods are provided to set and get
* each property. These methods handle any data type
* conversion needed. In the case of the field ws-book-
* right-sw, the type of pic x comp-5 is already
* equivalent to one of the common types supported by the
* .NET Framework, so no type conversion is needed.
* In that case, we can just use the PROPERTY clause and
* the compiler automatically generates the methods needed
*
* In addition, two methods are provided, RetrieveRecord
* and UpdateRecord. These are used by the COBOL wrapper
* class to read the complete record so that it can be
* passed to the original COBOL program and to update the
* record upon return from the COBOL program.

class-id. PatronInfo as "PatronInfo".
environment division.
configuration section.
repository.
class StringClass as "System.String"
class DateTime as "System.DateTime".

object.
data division.
working-storage section.

01 ws-patron-info.
  10 ws-patron-name pic X(23).
  10 ws-street-address pic X(20).
  10 ws-city-state-zip pic X(28).
  10 ws-book-right-sw pic X comp-5 property as
     "BookRights".
  10 ws-books-out pic 9(02).
  10 ws-seniority-date.
     15 ws-seniority-month pic 9(02).
     15 ws-seniority-day pic 9(02).
     15 ws-seniority-year pic 9(04).
10 ws-total-fees pic 999V99
    packed-decimal.

procedure division.

* Methods to set and retrieve the patron-info record.
* These will only be called from COBOL programs.

method-id. RetrieveRecord as "RetrieveRecord".
local-storage section.
linkage section.

01 ls-patron-info.
  10 ls-patron-name pic X(23).
  10 ls-street-address pic X(20).
  10 ls-city-state-zip pic X(28).
  10 ls-book-right-sw pic X comp-5.
  10 ls-books-out pic 9(02).
  10 ls-seniority-date.
     15 ls-seniority-month pic 9(02).
     15 ls-seniority-day pic 9(02).
     15 ls-seniority-year pic 9(04).
  10 ls-total-fees pic 999V99
     packed-decimal.

procedure division returning ls-patron-info.
  move ws-patron-info to ls-patron-info
  exit method.
end method RetrieveRecord.

method-id. UpdateRecord as "UpdateRecord".
local-storage section.
linkage section.

01 ls-patron-info.
  10 ls-patron-name pic X(23).
  10 ls-street-address pic X(20).
  10 ls-city-state-zip pic X(28).
  10 ls-book-right-sw pic X comp-5.
  10 ls-books-out pic 9(02).
  10 ls-seniority-date.
     15 ls-seniority-month pic 9(02).
     15 ls-seniority-day pic 9(02).
     15 ls-seniority-year pic 9(04).
  10 ls-total-fees pic 999V99
     packed-decimal.

procedure division using ls-patron-info.
  move ls-patron-info to ws-patron-info
  exit method.
end method UpdateRecord.

* Methods to set and get the different fields of the
* record. These methods expose the fields as properties
* of the object that can be set or read. The separate
* methods handle any type conversion needed.

*---------------------------------------------------------

method-id. get property patron-name as "Name".
data division.
linkage section.
01 ls-name   object reference StringClass.
procedure division returning ls-name.
    set ls-name to ws-patron-name
    exit method.
end method.

method-id. set property patron-name as "Name".
data division.
linkage section.
01 ls-name   object reference StringClass.
procedure division using ls-name.
    set ws-patron-name to ls-name
    exit method.
end method.

*---------------------------------------------------------

method-id. get property patron-street-address
             as "StreetAddress".
data division.
linkage section.
01 ls-street-address object reference StringClass.
procedure division returning ls-street-address.
    set ls-street-address to ws-street-address
    exit method.
end method.

method-id. set property patron-street-address
as "StreetAddress".

data division.

linkage section.

01 ls-street-address object reference StringClass.

procedure division using ls-street-address.
    set ws-street-address to ls-street-address
    exit method.
end method.

*---------------------------------------------------------

method-id. get property patron-city-state-zip
as "CityStateZip".

data division.

linkage section.

01 ls-city-state-zip object reference StringClass.

procedure division returning ls-city-state-zip.
    set ls-city-state-zip to ws-city-state-zip
    exit method.
end method.

method-id. set property patron-city-state-zip
as "CityStateZip".

data division.

linkage section.

01 ls-city-state-zip object reference StringClass.

procedure division using ls-city-state-zip.
    set ws-city-state-zip to ls-city-state-zip
    exit method.
end method.

*---------------------------------------------------------

method-id. get property booksout as "BooksOut".

data division.

linkage section.
01 ls-books-out binary-short.

procedure division returning ls-books-out.
  move ws-books-out to ls-books-out
  exit method.
end method.

method-id. set property booksout as "BooksOut".
data division.

linkage section.
01 ls-books-out binary-short.

procedure division using ls-books-out.
  move ls-books-out to ws-books-out
  exit method.
end method.

*---------------------------------------------------------

method-id. get property senioritydate as "SeniorityDate".
data division.
local-storage section.

01 year-part binary-long.
01 month-part binary-long.
01 day-part binary-long.

linkage section.
01 ls-seniority-date object reference DateTime.

procedure division returning ls-seniority-date.
  move ws-seniority-year to year-part
  move ws-seniority-month to month-part
  move ws-seniority-day to day-part
  invoke DateTime "NEW" using year-part
    month-part
    day-part
    returning ls-seniority-date
  exit method.
end method.

method-id. set property senioritydate as "SeniorityDate".
data division.
local-storage section.

01 year-part binary-long.
01 month-part binary-long.
01 day-part binary-long.

linkage section.

01 ls-seniority-date object reference DateTime.

procedure division using ls-seniority-date.
   set year-part to ls-seniority-date::"Year"
   move year-part to ws-seniority-year
   set month-part to ls-seniority-date::"Month"
   move month-part to ws-seniority-month
   set day-part to ls-seniority-date::"Day"
   move day-part to ws-seniority-day
   exit method.
end method.

*---------------------------------------------------------

method-id. get property fees as "Fees".
data division.

linkage section.

01 ls-total-fees float-short.

procedure division returning ls-total-fees.
   move ws-total-fees to ls-total-fees
   exit method.
end method.

method-id. set property fees as "Fees".
data division.

linkage section.

01 ls-total-fees float-short.

procedure division using ls-total-fees.
   move ls-total-fees to ws-total-fees
   exit method.
end method.

end object.
end class PatronInfo.