

Beans and COBOL: Do They Mix?

Wayne Rippin

WHITE PAPER

 **MERANT™**

Abstract

COBOL and Java – a likely pairing? You wouldn't initially think that a programming language like COBOL, that's been around for over 30 years, could interoperate with a new language like Java. That's where you're wrong. COBOL and Java can coexist. Does that mean that COBOL programmers have to learn and master Java in order to mix the two languages? No.

This paper provides an overview of Enterprise JavaBeans and what is possible with Enterprise JavaBeans and COBOL. A real world example is the main focus of this paper, walking you through an application that uses existing COBOL code — no rewriting — while using Java to develop the application front-end and an application server to handle the processing on the server. MERANT Micro Focus COBOL and Net Express® are highlighted.

Table of Contents

Introduction	1
How Micro Focus COBOL Does It.....	2
What are Enterprise JavaBeans?	
An Example	3
Creating the Classes.....	4-10
Adding Functionality	10-11
Conclusion.....	12
References	
About the Author	

Introduction

When companies first decide to create an Internet presence or start rolling out applications on an internal intranet, they often start out using static HTML pages. Then they want more dynamic Web sites and start creating CGI (Common Gateway Interface) programs to serve up dynamic Web content. However, as the number of users or the application's complexity increases, the CGI model becomes a bottleneck and companies look for alternatives.

On Windows NT or Windows 2000, many people take the route of using Microsoft's Active Server Pages and packaging their business logic as COM objects. The alternative to the Microsoft solution is one of the many application servers on the market — vendors include BEA, IBM, and Iona, among others — which are especially attractive if you want to deploy your application on a Unix platform since these companies provide application servers on UNIX as well as Windows NT. Until recently, these servers' interfaces and capabilities differed considerably; however, since Sun Microsystems' release of the Enterprise JavaBean (EJB) specification, nearly all the vendors support EJB.

So how do COBOL programmers deal with these environments? When Microsoft was developing the specification for COM, they made it fairly language neutral. While most people think of languages such as Microsoft Visual Basic or Microsoft Visual C++ when discussing COM object development, they can be created in many other languages, including COBOL. In a previous article, *Developing Mixed Visual Basic/COBOL Applications*, I demonstrated how you can use Micro Focus Net Express to create COM objects in COBOL.

However, the situation with Enterprise JavaBeans is not so straightforward. As the name suggests, the EJB specification is for the Java programming language. That may be fine if you are developing a new application from scratch, but what if you already have a lot of existing code that implements your business processes — and that code is written in COBOL? What options do you have?

Consider the issues involved in calling other languages from Java. To enable Java applications to call *native* modules written in other languages, Sun provided the Java Native Interface (JNI). The JNI defines the structures used in a call from Java to a native-code module written in another language, as well as a set of programming interfaces that enabled the module to access the Java environment. Unfortunately for COBOL programmers, the designers of the JNI didn't consider languages such as COBOL. The JNI makes extensive use of pointers to procedures and other programming elements that are not part of the standard COBOL language, so the COBOL compiler vendors have had to look at various mechanisms for overcoming this problem. Each vendor is taking a different approach. This paper focuses on the direction taken by MERANT with Micro Focus COBOL.

How Micro Focus COBOL Does It

Programmers who use Micro Focus COBOL have been able to use the JNI for some time, since Micro Focus COBOL includes extensions such as procedure pointers. In fact, MERANT provided examples demonstrating how to use the JNI from COBOL. However, the interface has not been easy to use; for example, it takes three calls via the JNI to retrieve the value of a string that has been passed into the COBOL program. So MERANT decided to make it easier for Java and COBOL to interoperate. You can find an overview of the support provided by MERANT in the sponsored white paper, *From COBOL to Enterprise JavaBeans with Net Express*, on www.cobolreport.com. This article will examine the support for Enterprise JavaBeans in more detail.

What are Enterprise JavaBeans ?

Essentially, the Enterprise JavaBeans (EJB) specification defines:

- How to write multitiered transaction-processing applications built from components,
- What services an EJB application server must provide, and
- How the components should be written.

Using EJBs, the structure of a typical three-tier application would be:

- A program on the client (for example, a Java applet or JavaBean) makes calls to remote EJBs.
- The EJB components live in the middle tier and are managed by the EJB server in EJB containers.
- The underlying database resides on the third tier. EJBs can access the database themselves or allow the EJB container to handle their data storage needs for them.

Note that a client never communicates directly with the EJB. Instead, it communicates with a component through its *home interface* and *remote interface*. The home interface provides the initial contact point. To look up the EJB's location, the client uses a naming service, which provides a reference to an object that implements the bean's home interface. The client then calls a method on the home interface to get a reference to the EJB's remote interface. The remote interface provides methods that interact with the actual implementation of the bean.

The EJB specification defines two types of EJB: the *session bean* and the *entity bean*. Instances of session beans generally have a relatively short lifetime, as they are tied to the lifetime of a given client session. They are typically used to represent business processes that perform work for a client. They are also nonpersistent, meaning that the data in the bean is not saved in permanent storage. The category of session beans is subdivided into *stateless* and *stateful* session beans. A stateless session bean does not maintain state across the method calls from the client, meaning it only holds information for one method call. This also means that any stateless bean can service a request from any client. Stateful session beans, on the

other hand, maintain the state across method calls from a client. Each instance of a stateful session bean is associated with a single client.

Entity beans have long lives: they exist across client sessions, are shared by multiple clients, and remain alive even after a server restart or other failure. Typically, an entity bean is used to represent a set of data in persistent storage.

An EJB server expects to find EJB interfaces implemented as Java classes, so Net Express generates the Java classes that the EJB server expects to see, but enables the business logic to be implemented in COBOL. The Java classes generated are simply skeletons that make calls to COBOL runtime routines, which then route the method calls into the COBOL code. Figure 1 shows the EJB architecture and how COBOL fits.

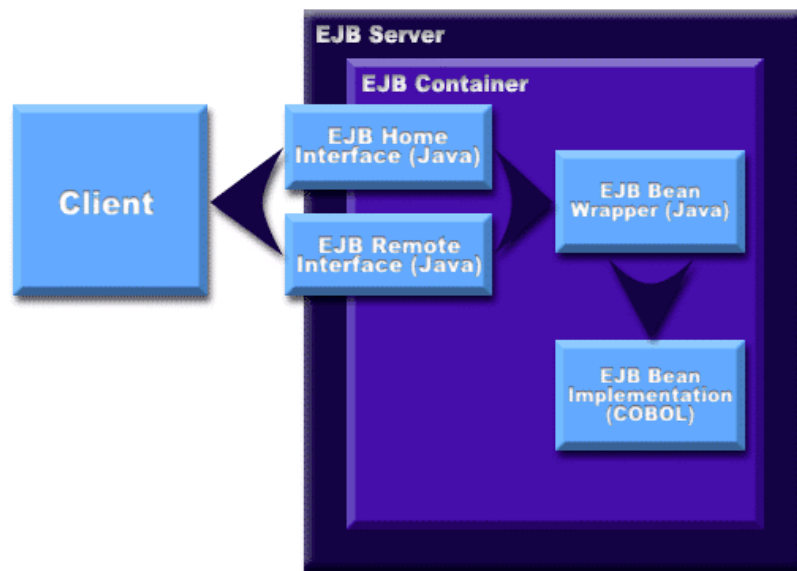


Figure 1. High-level view of the EJB architecture and how COBOL fits in.

An Example

Let's look at an example — the simplest form of EJB, the stateless session bean. I used this same example in my previous article, *Developing Mixed Visual Basic/COBOL Applications*. A Web-based intranet application enables a company's employees to select health and other benefits and see what the net impact on their paycheck will be. Now the company has decided to use Java to develop the application front-end and an application server to handle the processing on the server. However, they already have COBOL code that performs all the calculations and do not want to rewrite it. The solution is to use Net Express to create an EJB that incorporates the existing COBOL code. (You could manually write the code, but it is much easier to let Net Express generate it for you.)

Creating the Classes

The first thing you need is the basic Java and COBOL classes. You can create them using the Net Express Class Wizard; Figure 2 shows the initial dialog box.

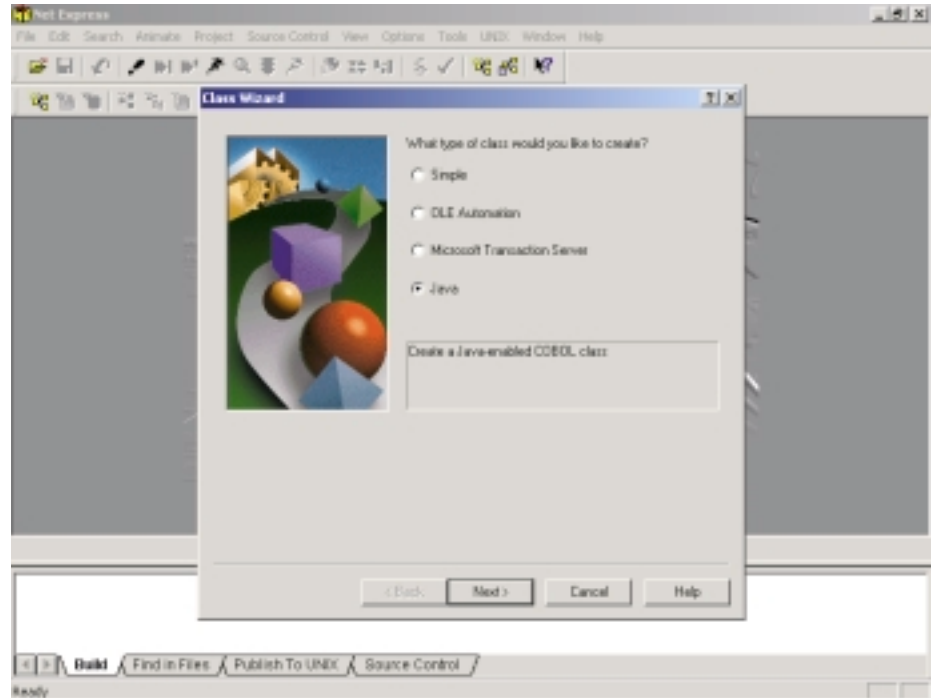


Figure 2. The Net Express Class Wizard.

The Class Wizard can generate COBOL classes for use in a variety of environments. In this case, we select the Java option. The wizard then prompts for the name and location of the Net Express project and the names of the files that will be created. Figure 3 shows the next dialog box where you specify that you want to create an Enterprise JavaBean. If you do not select this option, the wizard will generate one Java class that will act as a wrapper for your COBOL class, but it will not generate the infrastructure needed for an EJB. This dialog box also gives you the opportunity to change the names of the Java files that will contain the bean, home, and remote interfaces.

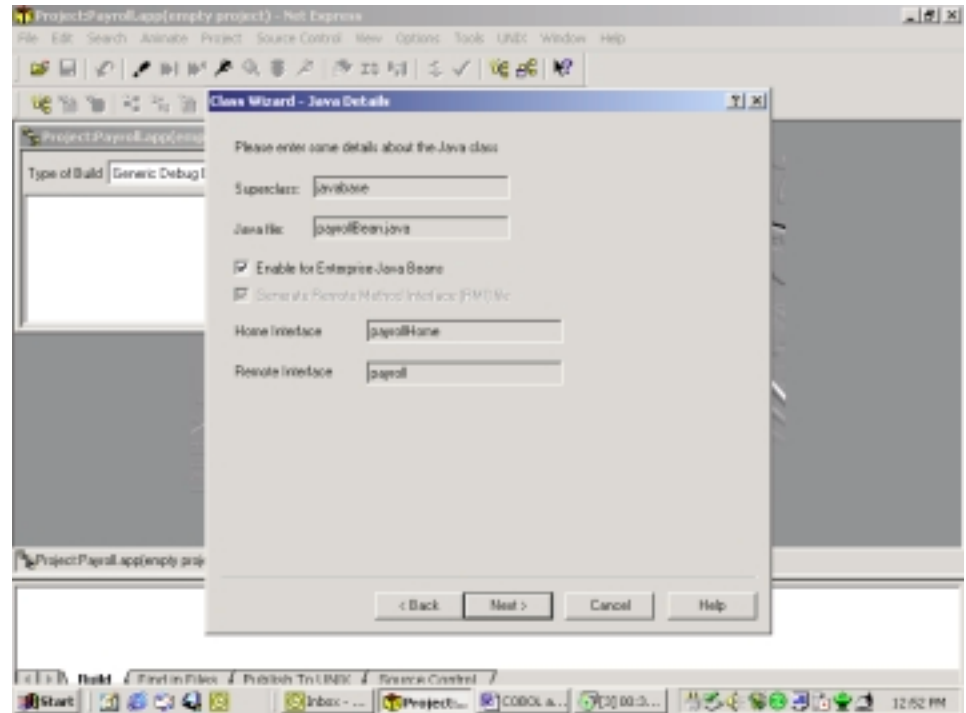


Figure 3. Creating an Enterprise JavaBean.

Figure 4 shows the deployment options dialog box. To deploy a bean, you must provide a deployment descriptor in the format expected by your EJB server. The wizard can currently generate deployment descriptors for two of the leading EJB servers, BEA Weblogic and IBM WebSphere. If you are using a different EJB server, the wizard can generate an XML file containing a descriptor that conforms to the EJB 1.1 specification; you can use that file as the basis for creating the deployment descriptor appropriate to your particular server. The deployment descriptor contains information such as the names of the EJB class, home interface, and remote interface and whether the bean is stateful or stateless. You also have the option of packaging all the Java classes into one JAR file. A JAR file is simply a mechanism for bundling and compressing all the classes and associated files into one file; it makes deploying the EJB easier.

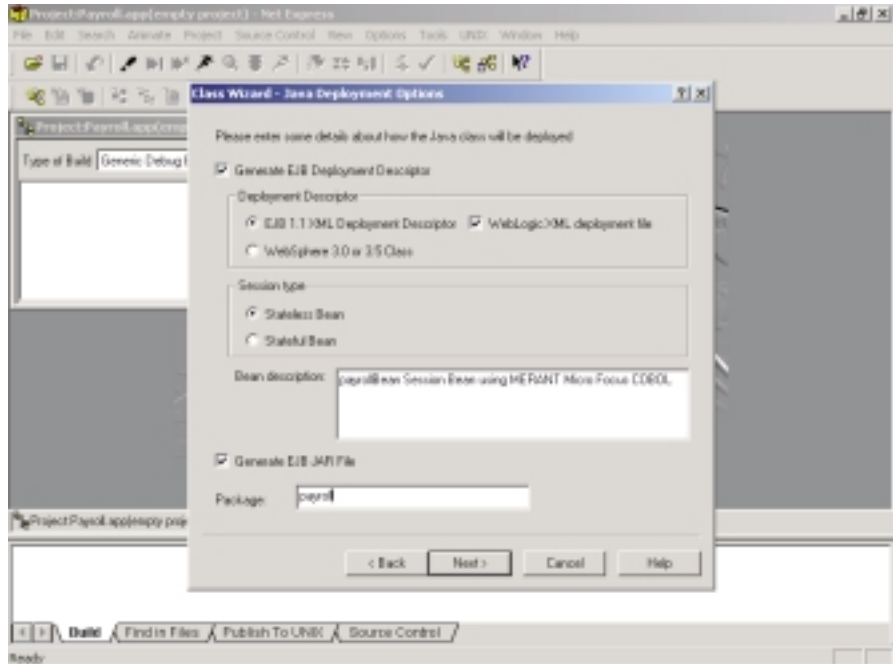


Figure 4. The deployment options dialog.

The wizard will now build the three Java class files, the skeleton COBOL class, and the deployment descriptor files and place them in a project file. Figure 5 shows the resulting project. Note that when you rebuild the project, if you have a Java compiler installed on your system, Net Express will also compile your Java files.

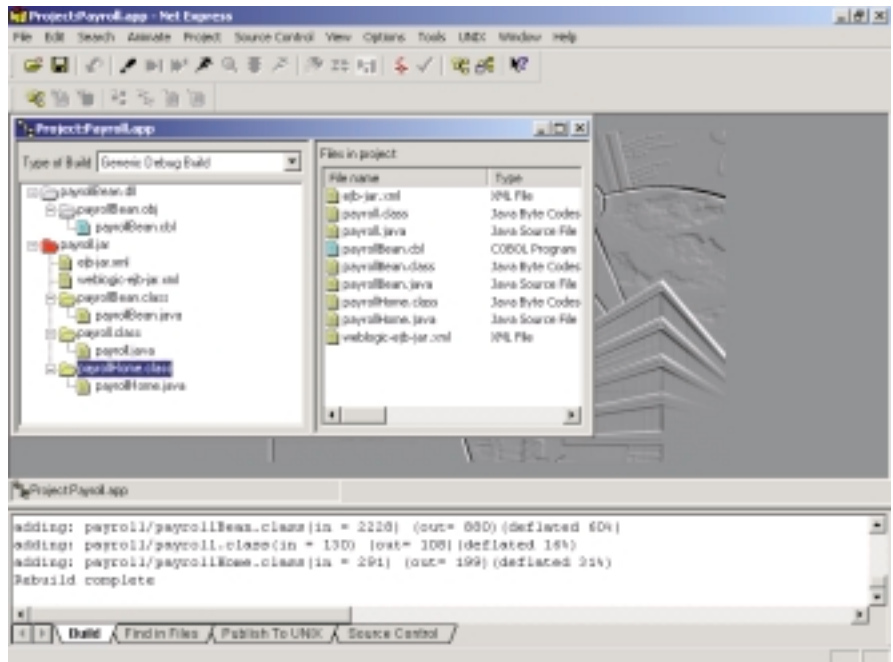


Figure 5. The Net Express project.

We now have a bean that will start up and shut down, but it does not yet do anything useful. Before we take care of that, let's look at the Java wrapper, which we'll call payrollBean.java, and the COBOL code, called payrollBean.cbl. (Examining the home and remote interface Java files that have been generated is beyond the scope of this article.)

payrollBean.java:

```
package payroll;

import mfcOBOL.*;
// OCWIZARD start imports
import java.util.*;
import java.rmi.RemoteException;
import javax.ejb.EJBException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
// OCWIZARD end imports

public class payrollBean extends mfcOBOL.runtime
                           implements SessionBean
{
    /* static methods - the COBOL runtime will register
these */
    public native static boolean cobinvokestatic_boolean
        (String methodName, Object[] params)
        throws COBOLException, Exception;
    public native static byte cobinvokestatic_byte
        (String methodName, Object[] params)
        throws COBOLException, Exception;
    public native static char cobinvokestatic_char
        (String methodName, Object[] params)
        throws COBOLException, Exception;
    public native static short cobinvokestatic_short
        (String methodName, Object[] params)
        throws COBOLException, Exception;
    public native static int cobinvokestatic_int
        (String methodName, Object[] params)
        throws COBOLException, Exception;
    public native static long cobinvokestatic_long
        (String methodName, Object[] params)
        throws COBOLException, Exception;
    public native static float cobinvokestatic_float
        (String methodName, Object[] params)
        throws COBOLException, Exception;
    public native static double cobinvokestatic_double
        (String methodName, Object[] params)
        throws COBOLException, Exception;
    public native static void cobinvokestatic_void
        (String methodName, Object[] params)
        throws COBOLException, Exception;
    public native static Object cobinvokestatic_object
        (String methodName, Object[] params)
        throws COBOLException, Exception;
```

```

        static private String javaClassName =
"payroll.payrollBean";
        static
        {
            /* Load the class */
            /* cobloadclass (libraryname, COBOLname,
fulljavaname); */
            cobloadclass ("payrollBean", "payrollBean",
                javaClassName);
        }

// OCWIZARD - start java methods
private javax.ejb.SessionContext _context;

public void ejbCreate() throws EJBException {
    try {
        cobinvoke_void("ejbCreate", null);
    }
    catch (Exception e) { throw new EJBException
(e.toString());}
}

public void ejbRemove() throws RemoteException {
    try {
        cobinvoke_void("ejbRemove", null);
    }
    catch (Exception e)
        { throw new RemoteException e.toString();}
}

public void ejbActivate() {
    try {
        cobinvoke_void("ejbActivate", null);
    }
    catch (Exception e) {}
}

public void ejbPassivate() {
    try {
        cobinvoke_void("ejbPassivate", null);
    }
    catch (Exception e) {}
}

public void setSessionContext(SessionContext sc) {
    try {
        _context = sc;
        Object params[] = { sc };
        cobinvoke_void("setSessionContext", params);
    }
    catch (Exception e) {}
}

// OCWIZARD - end java methods
}

```

payrollBean.cbl:

```

$set ooctrl(+p) ooctrl(-f)

*>-----
*> Class description
*>-----
class-id. Payroll
        inherits from javabase.

object section.
class-control.
    Payroll is class "payrollBean"
    *> OCWIZARD - start list of classes
    javabase is class "javabase"
    *> OCWIZARD - end list of classes
    *>---USER-CODE. Add any additional class names below.
    .
*>-----
working-storage section. *> Definition of global data
*>-----

*>-----
class-object. *> Definition of class data and methods
*>-----
object-storage section.

*> OCWIZARD - start standard class methods
*> OCWIZARD - end standard class methods

end class-object.

*>-----
object.      *> Definition of instance data and methods
*>-----
object-storage section.

*> OCWIZARD - start standard instance methods

method-id. "ejbCreate".
procedure division.
    exit method.
end method "ejbCreate".

method-id. "ejbRemove".
procedure division.
    exit method.
end method "ejbRemove".

method-id. "ejbActivate".
procedure division.
    exit method.
end method "ejbActivate".

method-id. "ejbPassivate".
procedure division.
    exit method.
end method "ejbPassivate".

```

```

method-id. "setSessionContext".
linkage section.
01 theContext object reference.
procedure division using by value theContext.
    exit method.
end method "setSessionContext".

*> OCWIZARD - end standard instance methods

end object.

end class Payroll.

```

As you can see, Net Express has generated quite a lot of code for an EJB that can only be loaded and shut down. However, this shows the structure common to every session bean. The payrollBean.java wrapper contains the interfaces that the EJB server calls. Each of these, in turn, simply calls its COBOL counterpart in payrollBean.cbl to perform the actual processing (for details on the cobinvoke functions used to call from Java to COBOL, refer to the white paper mentioned earlier).

All the methods in the code generated so far are mandated by the EJB specification for session beans. The EJB container:

- Calls `ejbCreate` when an instance of the bean is created.
- Calls `ejbPassivate` if the EJB server is going to swap the bean out of memory. This enables the bean to perform any necessary cleanup first. For example, if the bean has a file open and it receives a call to `ejbPassivate`, it has a chance to close the file.
- Calls `ejbActivate` (the opposite of `ejbPassivate`) when the EJB server swaps the bean back into memory.
- Calls `ejbRemove` as a result of a client calling the `Remove` method of the bean's home or remote interface. This gives the bean a chance to do any last-minute cleanup before the instance of the bean is destroyed.
- Calls `setSessionContext` at the beginning of a session bean's life. The bean is passed a reference to the object that implements the `SessionContext` interface. This provides access to services such as security and transaction management (see the EJB documentation for further information).

These methods must always exist in a session bean, even if they are empty. This will be the case in this example, since we are creating a bean that performs discrete functions that will not need any cleanup.

Adding Functionality

We must now add methods to the payroll class to enable it to perform useful functions. The easiest way to do this is to use the Net Express Method Wizard. For this example, we will create one method, `RetrieveTotalPaycheck`, which calculates and returns the net paycheck amount. This will take two input parameters, the employee ID and the benefit year, and return the total net paycheck amount.

The Method Wizard automatically detects that you are adding a method to a class that will be deployed as an EJB. It prompts you for the names of the COBOL method and the corresponding Java method (in most cases, these will be the same). The Method Wizard next prompts for the names and types of the input and output parameters. Because the bean will be called via Java, we must restrict our use of data types to those supported by Java. Both input parameters in this example are four-byte integers, so we enter the parameters' names (EmployeeID and BenefitYear) and select the type as PIC X(4) COMP-5. Finally, the Method Wizard lets us select a return value. To return the net paycheck amount (TotalPaycheck), we use an item defined as COMP-1 to indicate a floating-point value. The Method Wizard now builds the skeleton code for the method and inserts it into the class. It also updates any of the other files affected in the project.

Here is the Java and COBOL code added to payrollBean.java and payrollBean.cbl:

Java code:

```
public float RetrieveTotalPaycheck (int employeeID, int
BenefitYear)
    throws Exception, COBOLException, RemoteException
{
    // Parameters are passed to COBOL in an array
    Integer param1 = new Integer(employeeID);
    Integer param2 = new Integer(BenefitYear);
    Object[] params = {param1, param2};
    return (cobinvoke_float ("RetrieveTotalPaycheck",
                             params));
}
```

COBOL code:

```
method-id. "RetrieveTotalPaycheck".
    local-storage Section.
    *>---USER-CODE. Add any local storage items needed
below.
    linkage Section.
    01 employeeID                pic x(4) comp-5.
    01 BenefitYear               pic x(4) comp-5.
    01 TotalPaycheck             comp-1.

    procedure division using by value employeeID
                          by value BenefitYear
                          returning TotalPaycheck.

    *>---USER-CODE. Add method implementation below.

    exit method.
    end method "RetrieveTotalPaycheck".
```

The final step is to add your business logic to the method RetrieveTotalPaycheck using either the Net Express editor or any other text editor. This code will be the same as the code added to the Automation object in my previous article. You can add additional methods to the class using the Method Wizard in the same way. You should never need to make any changes to the Java code generated by Net Express.

The mechanism for deploying the EJB bean differs for each EJB server, so you will need to refer to that server's documentation for this information.

Conclusion

This paper has only touched the surface of what is possible with Enterprise JavaBeans and COBOL. Clearly, COBOL can coexist with the new application servers on the market, and COBOL programmers need minimal knowledge of Java. All the runtime and compiler support that has been implemented in Net Express to support Java has been implemented in MERANT Micro Focus Server Express™ Version 2.0.10, so you can deploy Enterprise JavaBeans written in COBOL on the leading Unix platforms as well as on Windows NT and Windows 2000.

References

For more information on Enterprise JavaBeans: <http://java.sun.com/products/ejb>

For more information on Net Express:

http://www.merant.com/products/microfocus/net_express

For the white paper, *From COBOL to Enterprise JavaBeans with Net Express*:

<http://www.COBOLreport.com/whitepaper/whitepaper.pdf> (requires Adobe Acrobat Reader)

About the Author

Wayne Rippin is a self-employed consultant. Previously, he worked for MERANT 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. Contact him at wayne@wmrconsulting.com.

FOR MORE INFORMATION

MERANT is the leading provider of software solutions for enterprise change management; development, transformation and integration of legacy applications; and data connectivity. More than 5 million professionals use MERANT technology at 60,000 customer sites, including the entire Fortune 100 and the majority of the Global 500. Founded in 1976, MERANT has 1,700 employees and more than 600 technology partners.

MERANT Worldwide Sales

Corporate headquarters 301-838-5000

Asia/Pacific(+65) 834 9880

Australia

Melbourne(+61) 3 9522 4466

Sydney(+61) 2 9904 6111

Belgium(+32) 15 30 77 00

Brazil(+55) 11 3048 8304

Canada(+1) 905-306-7280

Denmark(+45) 33 37 72 90

Finland(+35) 8 962 27 25 00

France(+33) 1 70 92 94 94

Germany

Dortmund(+49) 231 75 850

Munich(+49) 89 962 7 10

Italy

Milan(+39) 02 694 34 01

Rome(+39) 06 51 53 93 1

Japan(+81) 3 5401 9600

Netherlands(+31) 33 450 20 70

Norway(+47) 22 91 07 20

Portugal(+35) 11 384 5010

South Africa(+27) 11 237 7800

Spain

Barcelona(+34) 93 435 7001

Madrid(+34) 91 302 82 26

Sweden

Gothenburg(+46) 0 31 727 78 40

Stockholm(+46) 0 8 54 51 33 90

United Kingdom

Newbury(+44) 1635 32 646

St. Albans(+44) 1727 812 812

United States 800 547 4000

Also available at authorized resellers.

877-772-4450

merant.com

info@merant.com