Modernizing an IMS COBOL application with 64-bit Java: a tutorial

Goal

In this tutorial, you will learn how to run a 31-bit COBOL application that talks to 64-bit Java™ in an IBM® IMS message processing region (MPR). This interoperability between 31-bit COBOL and 64-bit Java in an IMS dependent region is a new feature of IMS V15.

For a quick overview, you can check this blog post Good news! 31-bit COBOL can now talk to 64-bit Java in IMS.

Outline

Here's the outline of the sections in this guide:

- An overview of the simple application's logic
- The Java code
- The COBOL code
- A sample script to compile the COBOL code and the Java code
- Update the dependent region's JCL
- Define the STDENV member with configuration information
- Define IMS resources using Dynamic Resource Definition (DRD)
    - Program definition (CREATE PGM)
    - Transaction code definition (CREATE TRAN)
- Sample output from COBOL and Java

Overview of a simple application's logic

The sample IMS application used in this tutorial is simple to highlight the new IMS feature that allows 31-bit COBOL to interoperate with 64-bit Java.

The application performs the following high-level actions:

1. Prints informational messages to `SYSOUT` throughout its execution to indicate what is going on, see the section "Sample output from COBOL and Java".
2. Retrieves a message from the IMS message queue
   a. `Call 'CBLTDLI' using GU-FUNC, IOPCB, INPUT-MESSAGE`
3. Calls Java code
4. Inserts a reply using the IOPCB
   a. `Call 'CBLTDLI' using ISRT-FUNC, IOPCB, OUTPUT-MESSAGE`
5. Performs the previous steps until IMS returns a QC status code indicating the application needs to terminate. This is the typical behavior for an application that processes transaction messages for a transaction code defined as wait for input (WFI).

For simplicity, the name of the COBOL program and the transaction defined to IMS are `CBL2JAVA`. Note that they don't have to be the same. The full name of the Java class is `mpr.apps.HelloWorldJava64` and the method to be called is `sayHello(ByteBuffer inData)`. That is the source file `HelloWorldJava64.java` is in the package `mpr.apps`.

Looking at the method signature of `sayHello`, you can see that it expects to be called with a `ByteBuffer` as the only argument.

To keep things interesting, the COBOL code will pass a reference of the `IN-DATA` variable as the argument to the `sayHello` method using a direct byte buffer. That means that the application will do a GetUnique against the IMS message queue to pull a message and populate the contents of `IN-DATA` before calling the Java code.

The `sayHello` method will do the following:

1. Print some messages to `STDOUT` including one that says: `"Hello from Java!!"`
2. Make a copy of the contents of the `IN-DATA` variable and print that to `STDOUT`
3. Overwrite a portion of the contents of the `IN-DATA` variable with the string `"Java changed this!"`
4. Print some more messages to `STDOUT`
5. Return to COBOL

When COBOL receives control from the Java method call, it will copy the contents of `IN-DATA`(defined as part of `INPUT-MESSAGE`) into `OUT-DATA`(defined as part of `OUTPUT-MESSAGE`) and insert that as the reply message using the IOPCB.

For example, if the enqueued message contains the text `"Hello from Carlos #1.123456789*123456789*"`, then after pulling the message from the IMS message queue `IN-DATA` will contain that text:

```
IN-DATA  = "Hello from Carlos #1.123456789*123456789*"
```

After calling the `sayHello` method:
```
IN-DATA  = "Java changed this!#1.123456789*123456789*"
```

And the expected reply will be:
```
OUT-DATA = "Java changed this!#1.123456789*123456789*"
```

The Java code

The Java code is rather simple. It prints some messages to `STDOUT`, overwrites a portion of the contents in the `IN-DATA` variable defined in COBOL, prints some more messages, and returns to COBOL. Note that the class `HelloWorldJava64` is defined in the package `mpr.apps`. Thus, the full class name is `mpr.apps.HelloWorldJava64`. That is the value that will be used in the COBOL code to properly locate the class at run time.

In the `sayHello` method we are dealing with a `ByteBuffer`, `inData`, as the input. So, before interacting with the `ByteBuffer`, a call to `inData.position(0)` must be made to make sure the index is set to position zero. This index will advance from position zero to the end of the byte buffer as you retrieve data using the `inData` variable.

You will notice a second call to `inData.position(0)` to reset the index right before calling `inData.put((new String("Java changed this!"…` to overwrite a portion of the contents in the `IN-DATA` variable defined in COBOL.

```
package mpr.apps;
…
public class HelloWorldJava64 {
…
    public static void sayHello(ByteBuffer inData) {

System.out.println("*****************************************************");
        System.out.println("  HelloWorldJava64.sayHello(ByteBuffer inData):
Entry   ");

System.out.println("*****************************************************");
```

```
        try{
            // Say hello
            System.out.println("Hello from Java!!");
            System.out.println();

            // Make a copy of the contents of IN-DATA using the inData object
            inData.position(0);
            int inDataLenght = inData.capacity();
            byte[] userBytes = new byte[inDataLenght];
            inData.get(userBytes);

            String transactionData = new String(userBytes, "Cp1047");
            System.out.println("Transaction data: " + transactionData);

            // Modify the contents of IN-DATA using the inData object
            inData.position(0);
            inData.put((new String("Java changed
this!").getBytes("Cp1047")));
        }
        catch(IOException ioe){
          System.out.println(ioe);
        }

System.out.println("*******************************************************");
        System.out.println("  HelloWorldJava64.sayHello(ByteBuffer inData):
Exit    ");

System.out.println("*******************************************************");
        System.out.println(" ");
    }
…
```

The COBOL code

The COBOL code below is not a complete application and is intended to highlight the key elements for the interoperability between 31-bit COBOL and 64-bit Java.

1. In the Working-Storage Section define an area of memory (INPUT-MESSAGE) to hold the message that will be retrieved from the IMS message queue. The `IN-DATA` variable will be referenced later.

```
       01  INPUT-MESSAGE.
               03  IN-LL        PIC  S9(4) COMP.
               03  IN-ZZ        PIC  S9(4) COMP.
               03  IN-TRANCODE  PIC  X(8).
               03  IN-DATA.
                   04  IN-LINE1 PIC  X(70).
```

2. Also define an area of memory (OUTPUT-MESSAGE) to hold the output message that will be inserted as a reply using the IOPCB. The `OUT-DATA` variable will be referenced later.

```
       01  OUTPUT-MESSAGE.
           02  OUT-LL       PICTURE S9(3) COMP VALUE +70.
           02  OUT-ZZ       PICTURE S9(3) COMP VALUE +0.
           02  OUT-DATA     PICTURE X(70) VALUE SPACES.
```

3. Define a few more variables in the Local-Storage section:

First, define the 64-bit variables that will be used to store references to Java objects such as the Java class ID, the method ID, and the argument for calling the Java method. Also note that the Java class name, method name, and method signature need to be converted from EBCDIC to UTF-8. So, define variables to hold the converted text.

```
      * Define variables to store 64-bit Java object  references for
      * the class ID and the method ID
       01 classid  pic 9(18) comp-5.
       01 methodid pic 9(18) comp-5.

      * Define variables for calling NewDirectByteBuffer to share
      * COBOL's IN-DATA Working-Storage with Java.
      * NewDirectByteBuffer expects a pointer to a block of memory,
      * a 64-bit value representing the amount of memory to be
      * referenced, and returns a 64-bit object reference for the
      * allocated direct java.nio.ByteBuffer.
       01 in-data-ptr usage pointer.
       01 in-data-len pic s9(18) comp-5.
       01 input-data-buffer pic 9(18) comp-5.

      * Define variables to convert the Java class name, method name
      * and method signature from EBCDIC to UTF-8
       01 class-name-utf8  pic x(64).
       01 method-name-utf8 pic x(64).
       01 method-sig-utf8  pic x(64).
```

4. Include a copy of the COBOL provided JNI copybook in the Linkage Section.

```
       Linkage Section.
       COPY JNI.
```

5. Pull a message from the message queues using the CBLTDLI interface

```
Call 'CBLTDLI' using GU-FUNC, IOPCB, INPUT-MESSAGE
```

6. Use the COBOL provided special register, JNIEnvPtr, as usual to be able to issue Java Native Interface (JNI) calls:

```
           Set address of JNIEnv to JNIEnvPtr
           Set address of JNINativeInterface to JNIEnv
```

7. The next step is to convert the Java class name, method name, and method signature to UTF-8. There are different ways to do that. The sample below uses `string function display-of()` and specifies the Coded Character Set Identifier (CCSID) number 1208.

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7..
           String function Display-of(n'mpr/apps/HelloWorldJava64',
                                      1208) x'00'
                  delimited by size into class-name-utf8

           String function Display-of(n'sayHello',
                                      1208) x'00'
                  delimited by size into method-name-utf8

           String function Display-of(n'(Ljava/nio/ByteBuffer;)V',
                                      1208) x'00'
                  delimited by size into method-sig-utf8
```

Here's a **quick tip** about EBCDIC and method signatures that contain arrays, denoted by square brackets. Suppose we have the following method that expects a byte array as input:

```
public static void sayHello(byte[] input)
```

This method's signature is: `'(B[)V'`

Where:
`'B'` represents the data type of byte
`'['` represents the array type
`'V'` represents the return type of void
In the codepage I'm using, the character `'Ý'`, `xBA`, represents the left square bracket `'['`. To convert the method signature from EBCDIC to UTF-8, I would need use `String function Display-of(n'(BÝ)V'`… and that will generate the proper UTF-8 method signature.

8. After doing the conversion to UTF-8, you can now use JNI calls to get the references to the Java class object and the to the Java method.

Issue a JNI call to FindClass to retrieve the reference to the Java class object. Notice that we are receiving the reference in the 64-bit variable `classid`.

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7..
           Call FindClass using
                          by value JNIEnvPtr
                          by value address of class-name-utf8
                          returning classid
```

Issue a JNI call to `GetStaticMethodId` to get the reference to the Java method. Notice that we are passing in the 64-bit variable `classid` and receiving the method reference in the 64-bit variable `methodid`.

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7..
           Call GetStaticMethodId using
                                  by value JNIEnvPtr
                                  by value classid
                                  by value address of method-name-utf8
                                  by value address of method-sig-utf8
                                  returning methodid
```

9. Create a direct byte buffer to share the memory of the `IN-DATA` COBOL variable with Java code.

To do that, issue a JNI call to `NewDirectByteBuffer` and pass a pointer to `IN-DATA(in-data-ptr)`, `IN-DATA`'s length (`in-data-len`), and receive the allocated direct byte buffer in the 64-bit variable `input-data-buffer`.

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7..
           Compute in-data-len = length of IN-DATA
           Set in-data-ptr to address of IN-DATA
           call NewDirectByteBuffer using
                                     by value JNIEnvPtr
                                     by value in-data-ptr
                                     by value in-data-len
                                     returning input-data-buffer
```

Don't forget to check for and take care of Java exceptions after issuing the JNI calls. You can check for Java exceptions with a JNI call to `ExceptionCheck` and then determine what the application should do next if an exception is detected. For example, if an exception is detected just issue a GOBACK:

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7..
      * PROCEDURE CHECK-JAVA-ERROR
      * Simple error handling.
       CHECK-JAVA-ERROR.
           Call ExceptionCheck using by value JNIEnvPtr
                                returning err-flag
           if err-flag = x'01' then
             Display 'Unhandled Java exception encountered: terminating'
             Display ' '
             goback
           end-if
           Move x'00' to err-flag
           exit.
```

10. Let's call the Java method!

All the 64-bit Java object references have been obtained and it is time to call the Java method. It is as simple as issuing a JNI call to `CallStaticVoidMethod` and passing the `classid`, the `methodid`, and the `input-data-buffer` obtainer earlier.

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7..
               Call CallStaticVoidMethod using by value JNIEnvPtr
                                              by value classid
                                              by value methodid
                                              by value input-data-buffer
```

Don't forget to check for and take care of Java exceptions after issuing the JNI call.

11. Copy `IN-DATA` to `OUT-DATA`

When the call to the `sayHello` Java method returns control back to COBOL, the contents of `IN-DATA` have been modified and you can set that as the reply message to be inserted using the IOPCB.

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7..
      * Set the reply message in OUT-DATA
           Move spaces to OUT-DATA
           Move IN-DATA to OUT-DATA.
```

12. Insert a reply message using the CBLTDLI interface

```
Call 'CBLTDLI' using ISRT-FUNC, IOPCB, OUTPUT-MESSAGE
```

And those are all the relevant pieces of the COBOL application to call 64-bit Java code.

A sample script to compile the COBOL code and the Java code

A couple of things to note before looking at the sample script to compile the COBOL code and the Java code.

The COBOL code must be compiled using the DLL side-deck files libjvm31.x (provided by the Java SDK) and igzxjni2.x (provided by COBOL). These new DLL side-deck files are required so the appropriate code is generated and executed at run time.

It should be noted that Object Oriented (OO) features are not supported in this mixed addressing mode environment (31-bit and 64-bit). So, do not use the existing side-deck file igzcjava.x because it is intended for OO COBOL features only.

The commands shown below are part of a shell script to make it easier to compile and link the COBOL code and the Java code. In the commands that follow, the uppercase strings that start with a dollar sign and are surrounded by curly braces, like `${COB2}`, are exported variables declared in the script and have been assigned appropriate values.

For example, COB2 is the name of a variable that will hold a string constructed with the contents of the already defined variable `COBOL_HOME` plus the string `/bin/cob2`. Here is an example of the variables I defined for my script:

```
#!/bin/sh

# Set WORK_DIR to the directory that hosts:
# the build.sh script
# and the application code
export WORK_DIR=/u/omvsadm/java3164/sampleApplications

# Set COBOL_HOME to the COBOL installation directory
export COBOL_HOME=/usr/lpp/cobol/cob630/igyv6r3

# Set COB2 to the cob2 COBOL compiler command
export COB2=$COBOL_HOME/bin/cob2

# Set JAVA_HOME to point to the installation direction for
# Java with mixed 31/64-bit support (FixPack 36).
export JAVA_HOME=/usr/lpp/java/java180/J8.0_64/

# Set STEPLIB to include any COBOL compiler data sets
export STEPLIB=IGYV6R30.SIGYCOMP

# Set the IMS SDFSRESL data set
export SDFSRESL=IMSBLD.I15RTSMM.SDFSRESL

# Set MPRPGMLIB to the program library data set to store COBOL
binaries
export MPRPGMLIB=IMSCONT.CBL2JAVA.PGMLIB
```

A slash `'\'` at the end of a line is the line continuation marker.

The sample below shows how to compile and link the COBOL code using the cob2 command.

It is your typical cob2 command, but it includes the new side-deck file `igzxjni2.x` and `libjvm31.x`.

```
"${COBOL_HOME}"/lib/igzxjni2.x
"${JAVA_HOME}"/lib/s390x/j9vm/libjvm31.x
```

And because this is an IMS application, a reference to the IMS SDFSRESL data set is needed to be able to generate the code for the various IMS DL/I calls:

```
-l"//'${SDFSRESL}'"
```

The COBOL application source is located at:

```
${WORK_DIR}/cobol31-to-java64/cobol_src/CBL2JAVA.cbl
```

The executable COBOL application will be stored as `CBL2JAVA` in the program library for the dependent region:

```
-o "//'${MPRPGMLIB}(CBL2JAVA)'"
```

The JNI copybook included in the Linkage Section of the COBOL application is in the `include` directory of the COBOL compiler:
`-I "${COBOL_HOME}"/include`

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7.....

# Compile and link the COBOL program
# Place the executable in the data set specified in MPRPGMLIB
#
echo ""
echo ""
echo " Compile and link CBL2JAVA"
${COB2} -comprc_ok=4 ${WORK_DIR}/cobol31-to-java64/cobol_src/CBL2JAVA.cbl \
      -e DLITCBL \
      -o "//'${MPRPGMLIB}(CBL2JAVA)'" \
      -qLIST,NODYNAM,MAP,APOST,XREF,THREAD,NOEXP \
      -bLET,CASE=MIXED \
      -I "${COBOL_HOME}"/include \
      "${COBOL_HOME}"/lib/igzxjni2.x \
      "${JAVA_HOME}"/lib/s390x/j9vm/libjvm31.x \
      -l"//'${SDFSRESL}'"
```

And the `javac` command shown below compiles the Java code.
The Java code is located at:
`${WORK_DIR}/cobol31-to-java64/java_src/HelloWorldJava64.java`

And the resulting byte code will be placed in the directory `/tmp`.

```
#
# Build the Java class HelloWorldJava64
# Put the resulting HelloWorldJava64.class file in the /tmp directory
#
echo ""
echo ""
echo "Build HelloWorldJava64.java"
${JAVA_HOME}/bin/javac -d /tmp -cp . \
      ${WORK_DIR}/cobol31-to-java64/java_src/HelloWorldJava64.java
```

Update the dependent region's JCL

Configuring the dependent region for 31-bit COBOL and 64-bit Java support is straight forward. First include the PROC parameter JVM= and assign it the value 3164. Then, in the PARM= section of the EXEC statement, add a reference to the JVM= parameter using &JVM. Remember that the parameters in the PARM= section are positional, so make sure to put &JVM in the proper position.

Here is the sample proc I used to start my dependent region.

I have specified `JVM=3164` and `&JVM` in the correct position.
In the STEPLIB concatenation I have included:

- The data set `IMSCONT.CBL2JAVA.PGMLIB`. This is the data set I specified as the output of the `cob2` command to compile and link the COBOL application.
- The `SDFSJLIB`, which contains new IMS code needed for this new IMS feature. Failure to include this data set will result in dependent region termination with U0101 ABEND and reason code 'E'x (14 decimal).

Finally, the `STDENV` DD card specifies the `STDENV00` member, which contains the rest of the configuration options. See the next section for a sample of the `STDENV00` member.

```
//MPR00001 PROC AGN=,ALTID=,APARM=,APPLFE=,CL1=001,CL2=002,
//    CL3=003,CL4=004,DBLDL=,ENVIRON=,IMSID=IMS1,
//    JVMOPMAS=,LOCKMAX=,NBA=5,OBA=5,OPT=W,OVLA=0,PARDLI=,
//    PCB=040,PREINIT=,PRLD=,PWFI=N,SOD=,SPIE=0,SSM=,STIMER=0,TLIM=01,
//    VALCK=0,VFREE=,VSFX=,JVM=3164
//REGION   EXEC PGM=DFSRRC00,
//              PARM=(MSG,&CL1&CL2&CL3&CL4,
//              &OPT&OVLA&SPIE&VALCK&TLIM&PCB,&PRLD,&STIMER,&SOD,
//              &DBLDL,&NBA,&OBA,&IMSID,&AGN,&VSFX,&VFREE,&SSM,
//              &PREINIT,&ALTID,&PWFI,&APARM,&LOCKMAX,&APPLFE,
//              &ENVIRON,&JVMOPMAS,&PARDLI,&JVM)
//STEPLIB  DD DISP=SHR,
//       DSN=IMSCONT.CBL2JAVA.PGMLIB
//          DD DISP=SHR,
//       DSN=IMSTESTU.IMS1501.MARKER
//          DD DISP=SHR,
//       DSN=IMSBLD.I15RTSMM.SDFSJLIB
//          DD DISP=SHR,
//       DSN=IMSBLD.I15RTSMM.CRESLIB
//PROCLIB  DD DISP=SHR,
//       DSN=USER.PRIVATE.AUTOSRVR.PROCLIB
//STDENV   DD DISP=SHR,
//       DSN=IMSCONT.CBL2JAVA.STDENV.CONFIGS(STDENV00)
//PRINTDD  DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//STDOUT   DD SYSOUT=*
//STDERR   DD SYSOUT=*
//*
//          PEND
```

Define the STDENV member with configuration information

This is a sample of the `STDENV00` member I created for my dependent region.
The relevant info is the location of the 64-bit Java SDK, denoted by `JAVA_HOME`, and the `LIBPATH` that points to the j9vm directory that contains the libjvm.so file needed to start the 64-bit JVM. Also note that `CLASSPATH` includes the "`/tmp`" directory which contains the executable Java bytecode for the `HelloWorldJava64` class.

```
export JAVA_HOME=/usr/lpp/java/java180/J8.0_64
LIBPATH="$JAVA_HOME"/lib/s390x/j9vm/
LIBPATH="$LIBPATH":"$JAVA_HOME"/lib/
LIBPATH="$LIBPATH":/tmp/
export LIBPATH="$LIBPATH":
APP_HOME=$JAVA_HOME
CLASSPATH=$APP_HOME:"${JAVA_HOME}"/lib:"${JAVA_HOME}"/lib/ext
```

```
CLASSPATH="$CLASSPATH":/tmp/
export CLASSPATH="$CLASSPATH":
IJO="-Xms256m"
IJO="$IJO -Xmx1024m"
export IBM_JAVA_OPTIONS="$IJO "
```

Define IMS resource using Dynamic Resource Definition (DRD)

IMS needs to know key information before it can run the COBOL application. First, we need to tell IMS the name of the program we want to run, along with some other attributes shown below. Then, we need to define the transaction code associated with the program so IMS knows what program to run when messages are enqueued for that transaction code. You can use your preferred method for defining the program and transaction to IMS. In this example I'm using a DRD enabled IMS and below are the type-2 commands for creating the program and the transaction definitions.

Program definition (CREATE PGM)

```
CREATE PGM NAME(CBL2JAVA) SET(FP(N) BMPTYPE(N) GPSB(Y) LANG(COBOL))
```

This type-2 command creates a program definition for the `CBL2JAVA` COBOL application in IMS. The command sets various attributes with key information needed for proper scheduling:

`FP(N)` - indicates the program is not a Fast Path-exclusive program
`BMPTYPE(N)` - indicates the program is not a BMP
`GPSB(Y)` - tells IMS to generate the program specification block (PSB) and application control block (ACB) that are associated with the program at run time
`LANG(COBOL)` - tells IMS that this is a COBOL program

For more information, see the CREATE PGM command in the IMS documentation.

Transaction code definition (CREATE TRAN)

```
CREATE TRAN NAME(CBL2JAVA) SET(FP(N) PGM(CBL2JAVA) MSGTYPE(SNGLSEG) RESP(Y)
WFI(Y))
```

This command creates a transaction with name `CBL2JAVA` in IMS and associates the transaction with the program `CBL2JAVA` defined with the previous command.

The command sets various attributes with key information needed for proper scheduling:

`FP(N)` – indicates that transaction is not a candidate for Fast Path processing
`PGM(CBL2JAVA)` - specifies the name of the program previously defined with the CREATE PGM command
`MSGTYPE(SNGLSEG)` – indicates that the incoming message is one segment in length
`RESP(Y)` - indicates this is a response mode transaction.
`WFI(Y)` - indicates that this transaction is wait for input enabled and the application processing the messages for this transaction can remain scheduled after it has processed any available input messages.

For more information, see the CREATE TRAN command in the IMS documentation.

Sample output from COBOL and Java

Before looking at the sample output produced by the COBOL code and the Java code, let's look at the messages printed by the dependent region in the job log. The messages below start with the string "`DFSJVM36:`" indicating that this dependent region has been started with the option JVM=3164. A pure 31-bit dependent region would print messages that start with DFSJVM00, and a 64-bit JMP or JBP would display messages that start with DFSJVM64.

The output correlates to the options coded in the `STDENV00` member specified in the `STDENV` DD card.

```
...
DFSJVM36: Enter STDENV script: xxx xxx nn nn:nn:nn.nnn nnnn
DFSJVM36: Option 0 = -Xms256m
...
DFSJVM36: Option 4 = -
Djava.class.path=/usr/lpp/java/java180/J8.0_64:/usr/lpp/java/java180/J8.0_64/lib
:/usr/lpp/java/java180/J8.0_64
 /lib/ext:/tmp/:
DFSJVM36: Option 5 =
LIBPATH=/usr/lpp/java/java180/J8.0_64/lib/s390x/j9vm/:/usr/lpp/java/java180/J8.0
_64/lib/:/tmp/:
DFSJVM36: Option 6 = JAVA_HOME=/usr/lpp/java/java180/J8.0_64
...
DFSJVM36: Exit STDENV script: xxx xxx nn nn:nn:nn.nnn nnnn
...
DFSJVM36: Option 11 = -XX:+Enable3164Interoperability
DFSJVM36: JVM initialization started:  xxx xxx nn nn:nn:nn.nnn nnnn
DFSJVM36: JVM initialization complete: xxx xxx nn nn:nn:nn.nnn nnnn
```

Now let's look at how we can trigger the application.

For example, let's say that the current outstanding reply number for IMS is 16. I can then use that number to submit a transaction to IMS using the transaction code `CBL2JAVA` followed by the data "`Hello from Carlos #1.123456789*123456789*`". I could also use an IMS Connect client and submit the transaction that way.

```
*16 DFS996I *IMS READY*  IMS1

 R 16,CBL2JAVA HELLO FROM CARLOS #1.123456789*123456789*
 IEE600I REPLY TO 16 IS;CBL2JAVA HELLO FROM CARLOS #1.123456789*123
*17 DFS996I *IMS READY*  IMS1
```

I can then use to the new outstanding reply number, 17, to show the reply from the COBOL program, `CBL2JAVA`:

```
*17 DFS996I *IMS READY*  IMS1
 R 17,.
 IEE600I REPLY TO 17 IS;.
 DFS000I Java changed this! #1.123456789*123456789*
...
```

As expected, the message inserted as a reply by the COBOL program, `CBL2JAVA`, is the message retrieved from the message queue and modified by the Java code to contain the string "`Java changed this!`".

Here is sample output sent to `SYSOUT` by COBOL code and to `SDTOUT` by Java code.

This is sample output produced by the COBOL application and sent to `SYSOUT` **before** calling the Java method sayHello(ByteBuffer inData). Note the value assigned to `IN-DATA`.

```
 ************************************************
             CBL2JAVA: Execution begin
 ************************************************
...
 Input Message has:
 IN-LL: 0054
 IN-ZZ: 0000
 IN-TRANCODE: CBL2JAVA
```

```
 IN-DATA:  HELLO FROM CARLOS #1.123456789*123456789*

 COBOL getting Java objects
 COBOL got Java objects
 Calling the StaticVoidMethod
```

This is sample output produced by the Java method sayHello(ByteBuffer inData) and sent to `STDOUT`. Note that the data pointed to by the `inData` parameter are the contents of the `IN-DATA` variable defined in COBOL.

```
**************************************************
   HelloWorldJava64.sayHello(ByteBuffer inData): Entry
 **************************************************
 Hello from Java!!
 Transaction data:  HELLO FROM CARLOS #1.123456789*123456789*
 **************************************************
   HelloWorldJava64.sayHello(ByteBuffer inData): Exit
 **************************************************
```

This is sample output produced by the COBOL application and sent to `SYSOUT` **after** calling the Java method sayHello(ByteBuffer inData). Note that the data in `IN-DATA` has been modified by Java.

```
IN-DATA now has: Java changed this! #1.123456789*123456789*
Insert reply
...
 CBL2JAVA attempting to read another message.!!
```