Cross-language Software Development with isCOBOL Evolve

The Java Programming Language

Table of Contents

Introduction **5**
Why isCOBOL? **6**
What Makes isCOBOL an Ideal Choice for Use with the Java Language? **7**
True Java Class Output 7
Natural Support of Object-Oriented COBOL 7
isCOBOL Threads Are True Threads 7
RDBMS access via JDBC 8
Integrated ESQL Compiler 8
Veryant Database Bridge ..... **Error! Bookmark not defined.**
isCOBOL Compiler Architecture **10**
Two-step Process 10
Outputs POJOs (Plain Old Java Objects) 10
No Native Dependencies Required 11
Procedural Programs (i.e. PROGRAM-ID) 11
Example of Java Class Produced from a COBOL Subprogram 11
Object-oriented Programs (i.e. CLASS-ID) 13
Example of Java Class Produced from a COBOL Class 13
Defining COBOL Classes in Java Packages Using Only Java Types 15
Example of a Java Class Using Only Java Types 15
Calling COBOL from Java **17**
isCOBOL generates a main() method in the Java class 17
Calling Existing Programs with com.iscobol.java.IsCobol 17
com.iscobol.types.* classes 17
Samples Included with isCOBOL 18
Java Code Calling an Existing COBOL Program 18
Java Code Using a COBOL Object 18
Access COBOL Files Natively From Java Code with COBFILEIO 19
Developing and Debugging COBOL and Java Code Together **20**

Example Steps for Setting Up Eclipse for Cross-language Development 20
Using Other Java Debuggers Such as JSwat and jdb 21
Veryant JISAM 22
JISAM Limits 22
JUTIL Options 22
COBFILEIO 23
Usage: 23
Example Steps 24
How do you take your Java? - An Article 33
A little dash or a splash of COBOL can make for the perfect mix 33
Common Business Oriented Language – 50 Years Young 33
Originally called Green, then changed to Oak, finally known as Java – 15 years mature 33
COBOL with A Dash of Java 34
Example of COBOL Using Java Packages 35
Java with a Splash of COBOL 36
Example of COBOL Calling Java Classes 36
Example of Java Calling a COBOL Class 37
Conclusion 40
Appendix A 41
Quick FAQ 41

Introduction

COBOL is one of the oldest programming languages with hundreds of billions of lines of code still in use and hundreds of thousands of programmers worldwide.

The benefits of COBOL are numerous and can be found listed in many publications spanning over 50 years that COBOL has been in existence. Here are some facts that are especially relevant when considering whether to replace, rewrite or rejuvenate existing COBOL programs:

- COBOL's verbosity has made it the most readable, understandable and self-documenting programming language in use today. And this readability becomes more valuable the older a program becomes.
- The COBOL standard is not owned or controlled by any particular COBOL vendor. COBOL vendors do their best to comply with one and the same standard. This makes the COBOL source code itself very portable, and ensures that the COBOL language can be used on a wide variety of hardware platforms and operating systems.
- The COBOL standard stipulates that definitions of external references be made in the Environment Division. This simplifies platform changes.
- COBOL has evolved to support both procedural and object-oriented programming paradigms, and the changes have been incorporated into the COBOL standard. The COBOL language is constantly evolving to match the pace of hardware and software technology innovations.

It is critically important for COBOL development systems to keep up with the very latest technologies and for COBOL vendors to create multiple paths for COBOL to follow into the future.

This paper discusses isCOBOL Evolve software's internal and interface designs that make isCOBOL the ideal choice for rejuvenating existing COBOL programs by integrating and interoperating with applications and components written in the Java programming language and created for the Java platform.

Why isCOBOL?

isCOBOL software's purpose is to enable companies to retain current COBOL investments while simultaneously providing a path forward to take full advantage of the openness, portability, and power of Java technology without retraining or rewriting.

With isCOBOL, you no longer need to worry about having to replace COBOL 'someday,' instead isCOBOL provides the flexibility to choose the best programming language for the particular job at hand. If it makes business sense, developers can transition from COBOL programming to Java programming over time.

isCOBOL gives you the ability to keep your COBOL assets even if you choose Java as your programming language now or at some time in the future. This protects existing COBOL investments while opening the door to new opportunities available with the Java platform.

If you choose to further extend COBOL applications with Java language and technologies the path is made easier with isCOBOL. Since isCOBOL preserves current COBOL programs, the transition to Java can be done at your desired pace.

At Veryant we believe that it is the COBOL programmers who have the best understanding of the application structure, flow and logic. After transitioning to isCOBOL it remains true that COBOL is the best language to use in order to fix bugs and make small enhancements to the COBOL application because these types of changes require inline modifications to the application source code.

Developers using isCOBOL leverage current COBOL skills and can move to the Java platform without ever needing to know how to program in the Java language. With isCOBOL all development and debug processes can be performed using the COBOL language.

What Makes isCOBOL an Ideal Choice for Use with the Java Language?

True Java Class Output

The isCOBOL Compiler produces true Java classes. This means that there is no interoperability layer or additional mechanism needed between isCOBOL and Java components. Thus:

- An isCOBOL program can use Java classes as if they were written in COBOL.
- A Java program can use isCOBOL classes as if they were written in the Java programming language.
- Object-Oriented COBOL can be used to extend (subclass, derive from) a Java class.
- The Java programming language can be used to extend an isCOBOL class.

Natural Support of Object-Oriented COBOL

The isCOBOL Compiler naturally supports Object-Oriented COBOL. Object-Oriented COBOL provides a superior way to consume or produce objects such as those used with Web Services, Java Packages, .NET, ActiveX, OLE, and COM.

- Through object-oriented syntax, COBOL programs can use objects written in the Java language, and also create objects that can be used by Java programmers as if those objects were written using the Java language.

isCOBOL Threads Are True Threads

isCOBOL delivers true multithreading, as opposed to emulated threads, when running a COBOL application.

- During runtime, COBOL applications take full advantage of multiple and multi-core CPUs by letting the computer hardware divide up the work and execute it in parallel processing.
- This allows sorting and all other operations to take place on multiple threads.

- Thread switching is smooth and there is no need to configure, program, or be concerned about thread priorities or the thread switcher. Long operations do not block thread switching.

RDBMS Access via JDBC

The isCOBOL Compiler provides portable RDBMS access via JDBC. This simplifies deployment by eliminating the extra C API layer and the requirement to dynamically load or re-link the runtime with special libraries from the database vendor when deploying, upgrading or moving between database vendors.

- Embedded SQL is converted directly into calls to the JDBC driver.
- No knowledge of the C static or dynamic linker is required.
- Any supported RDBMS can be used without code changes or recompiling.
- JDBC driver interface to the database is provided by and supported by the database vendor.

Integrated ESQL Compiler

ESQL compiler capabilities are integrated in the isCOBOL Compiler so it is unnecessary to use a pre-compiler such as Pro*COBOL.

- There are no additional set up requirements, no need for complicated build scripts and only one set of source files to manage, thus increasing available disk space.
- The debugger shows the original source code, rather than code generated by a pre-compiler.
- Without a separate pre-compiler requirement, there is less software that has to be maintained.

Veryant Database Bridge

With the Veryant Database Bridge there is no need to change COBOL or to learn ESQL to access a RDBMS. Veryant Database Bridge technology takes the Extended File Description (EFD*) for a data file and generates a file handler with embedded SQL in COBOL. When you run an application, COBOL ISAM INDEXED file I/O operations on a data file are routed through the generated COBOL/ESQL interface which accesses the RDBMS via JDBC.

- This method often proves better than middleware interface layers, because it opens access to the COBOL/ESQL source code which becomes part of the application.
- There is no additional layer between the application and the JDBC driver. Support for the SQL and JDBC driver comes directly from the database vendor.
- Veryant Database Bridge automatically inherits all of the benefits of the integrated ESQL compiler capabilities in the isCOBOL Compiler.

** EFD - Extended File Description XML file produced by the compiler -efd option*

isCOBOL Compiler Architecture

At the core of Veryant's products is the isCOBOL Compiler. The isCOBOL Compiler converts COBOL source code into intermediate Java source code and then feeds that to the Java compiler to produce Java bytecode objects (i.e. .class files).

The isCOBOL Compiler was designed as a "pluggable" architecture so that code generators could be created for any native or virtual machine in the future. Because of the Java platform's portability Veryant chose to first create a code generator and runtime framework for the Java Virtual Machine (JVM).

Thus, Veryant's first compiler, the isCOBOL Compiler, translates COBOL source code into Java classes that can then be executed by the JVM. The isCOBOL compiler outputs Java bytecode instead of pseudo code (pcode) or native objects.

Two-step Process

To accomplish this task the isCOBOL Compiler follows a two-step process that is completely transparent to the developer:

1. The compiler translates the COBOL source code into intermediate Java source code.
2. This intermediate source code is then automatically fed into the Java compiler which produces Java bytecode objects (.class files).

Note: although the Java source code created in step 1 is temporary intermediate output and is not stored on disk, a COBOL program can be compiled with a special "-jj" option if an organization desires to retain this code.

Outputs POJOs (Plain Old Java Objects)

After compilation the COBOL programs are Java classes, POJOs (Plain Old Java Objects), in standard Java .class files and can be used like any other Java classes that were created using the Java programming language. The object classes are indistinguishable from classes programmed in the Java language because they are produced by compiling actual Java language source code.

The advantage of this technology approach is that the isCOBOL Compiler produces Java source code that functions exactly the same as the original COBOL source code, even though COBOL, being an early language, supports non-structured programming and has elements that do not have Java language equivalents such as GO TO, PERFORM THRU, and NEXT SENTENCE.

No Native Dependencies Required

At run-time the Java classes produced by the isCOBOL Compiler do not depend on machine-native object libraries or executables (i.e. other than the JVM) unless the COBOL program itself calls native objects such as those containing functions programmed using the C language.

isCOBOL applications are able to run on any device supporting the Java Virtual Machine (JVM) - from mainframes to mobile phones. This includes the application logic, data access and user interface.

Procedural Programs (i.e. PROGRAM-ID)

The isCOBOL Compiler transforms each procedural COBOL program (i.e. with PROGRAM-ID) into a Java class, each paragraph and entry-point into a Java class method, each LINKAGE SECTION item into a Java class method parameter.

isCOBOL also provides a COBOL types package so that Java programmers can access COBOL data types with set and get accessor methods that automatically convert between COBOL and Java types.

isCOBOL compiles a traditional COBOL program source file named myprog.cbl into a Java class named MYPROG with a main() method that takes the program's PROCEDURE DIVISION USING parameters (LINKAGE SECTION items) as Java type parameters defined in the isCOBOL types package.

Example of Java Class Produced from a COBOL Subprogram

Take, for example, a program that returns the sum of two numbers:

add2.cbl:

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. ADD2.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
```

```
       01 SUM PIC 9(5).
       LINKAGE SECTION.
       01 ADDEND-1 PIC 9(5).
       01 ADDEND-2 PIC 9(5).
       PROCEDURE DIVISION USING ADDEND-1 ADDEND-2 RETURNING SUM.
       MAIN.
              MOVE ADDEND-1 TO SUM.
              ADD ADDEND-2 TO SUM.
              GOBACK.
```

The Java class produced by the isCOBOL Compiler for this program has the following definition:

Note: Code has been removed for brevity in the following Java source examples

ADD2.java:

```
public class ADD2
implements com.iscobol.rts.IscobolClass, com.iscobol.rts.IscobolCall {
   public static void main(String args[]) {
              // ...
              com.iscobol.rts.IscobolCall theProgram = new ADD2();
              exitCode = ((CobolVar)
              theProgram.call(linkage)).toint();
              System.exit(exitCode);
     }
   }

   private com.iscobol.types.NumericVar SUM;
   public com.iscobol.types.NumericVar ADDEND_1;
   public com.iscobol.types.NumericVar ADDEND_2;

   public Object call(Object argv[]) {
         // ...
         try {
               perform(1, 1);
         } catch (GobackException ex$) {
               if (ex$.getReturnValue() != null)
                     return ex$.getReturnValue();
         } finally {
         }
         return SUM;
   }

   public void perform(int begin, int end) {
         boolean goOn = true;
         while (goOn) {
               try {
```

```
                        switch (begin) {
                        case 1:
                              if ((begin = MAIN()) > 0)
                                    break;
                              if (end == 1) {
                                    goOn = false;
                                    break;
                              }
                        default:
                              goOn = false;
                              break;
                        }
                  } catch (GotoException e) {
                        begin = e.parNum;
                  } catch (ExitSectionException e) {
                        goOn = false;
                  }
            }
      }

      final private int MAIN() throws GotoException {
            ADDEND_1.moveTo(SUM);
            SUM.addToMe(ADDEND_2.tolong());
            throw GobackException.go;
            return 0;
      }
}
```

Object-Oriented Programs (i.e. CLASS-ID)

The isCOBOL Compiler transforms each object-oriented COBOL program (i.e. with CLASS-ID) into a Java class translating each COBOL class method (i.e. METHOD-ID) to a Java class method and each COBOL class variable into a Java class variable.

COBOL class members defined in the FACTORY paragraph are translated to Java static class members. COBOL class members defined in the OBJECT paragraph are translated to Java non-static class members.

Example of Java Class Produced from a COBOL Class

Take, for example, a COBOL object class that has a factory method to convert a string to uppercase.

ConvertCase.cbl:

```
      IDENTIFICATION DIVISION.
```

```
       CLASS-ID. CONVERTCASE as "ConvertCase".
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       IDENTIFICATION DIVISION.
       FACTORY.
       PROCEDURE DIVISION.
       IDENTIFICATION DIVISION.
       METHOD-ID. TOUPPER AS "toUpper".
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  RET-VAL PIC X ANY LENGTH.
       LINKAGE SECTION.
       01  IN-PARAM PIC X ANY LENGTH.
       PROCEDURE DIVISION USING IN-PARAM RETURNING RET-VAL.
       START-METHOD.
          MOVE IN-PARAM TO RET-VAL.
          CALL "C$TOUPPER" USING RET-VAL.
       END METHOD.
                END FACTORY.
```

The Java class produced by the isCOBOL Compiler for this program has the following definition:

ConvertCase.java:

```
import com.iscobol.rts.*;
import com.iscobol.types.*;

public class ConvertCase
implements com.iscobol.rts.IscobolClass {
   // ...

public static com.iscobol.types.PicX
      toUpper(com.iscobol.types.PicX IN_PARAM) {
         // ...
   }

}
```

The class above could be used from COBOL as follows:

TEST.cbl:

```
       ID DIVISION.
       PROGRAM-ID. TEST.
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       REPOSITORY.
```

```
      CLASS CONVERTCASE AS "ConvertCase".
DATA DIVISION.
WORKING-STORAGE SECTION.
01 ITEM-1 PIC X ANY LENGTH.
PROCEDURE DIVISION.
  MOVE CONVERTCASE::"toUpper"("abcdefghij") TO ITEM-1.
  DISPLAY ITEM-1.
  STOP RUN.
```

Defining COBOL Classes in Java Packages Using Only Java Types

Example of a Java Class Using Only Java Types

Often it is useful to define the COBOL class in a package and use only Java types in its public interface. For example,

```
IDENTIFICATION DIVISION.
CLASS-ID. JCONVERTCASE as "com.mycompany.JConvertCase".
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
    CLASS JSTRING AS "java.lang.String"
    .
IDENTIFICATION DIVISION.
FACTORY.
PROCEDURE DIVISION.
IDENTIFICATION DIVISION.
METHOD-ID. TOUPPER AS "toUpper".
DATA DIVISION.
WORKING-STORAGE SECTION.
01  RET-VAL OBJECT REFERENCE JSTRING.
01  TEMP-ITEM-1 PIC X ANY LENGTH.
LINKAGE SECTION.
01  IN-PARAM OBJECT REFERENCE JSTRING.
PROCEDURE DIVISION USING IN-PARAM RETURNING RET-VAL.
START-METHOD.
   SET TEMP-ITEM-1 TO IN-PARAM.
   CALL "C$TOUPPER" USING TEMP-ITEM-1.
   SET RET-VAL TO TEMP-ITEM-1.
END METHOD.
END FACTORY.
```

The Java class produced by the isCOBOL Compiler for this program has the following definition:

JConvertCase.java:

```
package com.mycompany;
```

```
public class JConvertCase
implements com.iscobol.rts.IscobolClass {
   // ...

   public static java.lang.String
      toUpper(java.lang.String IN_PARAM) {
         // ...
   }
}
```

The JConvertCase COBOL class could be used from Java as follows:

```
import com.mycompany.JConvertCase;

public class main {
   public static void main( String args[] )
   {
      System.out.println(JConvertCase.toUpper("abcdefghij"));
   }
}
```

Calling COBOL from Java

As mentioned above, with isCOBOL each procedural COBOL program (those with PROGRAM-ID) compiles to a Java class that has a call() method so it can be called as a subprogram.

isCOBOL generates a main() method in the Java class

After compile with isCOBOL initial programs also have a main() method so that they can be run with:

```
java PROGNAME cmd-line-arg1 cmd-line-arg2
```

Calling Existing Programs with com.iscobol.java.IsCobol

Existing programs can be invoked from Java using the isCOBOL framework class com.iscobol.java.IsCobol which is available in iscobol.jar (or isrun.jar for strictly runtime deployment). No COBOL code changes are required. For example,

```
import com.iscobol.java.IsCobol;

      int rc;
      rc = IsCobol.call("PROGNAME",
      new String[] { "cmd-line-arg1", "cmd-line-arg2"});
      IsCobol.cancel("PROGNAME");
```

com.iscobol.types.* classes

To invoke a COBOL subprogram with linkage items from Java you can use the com.iscobol.types.* classes to create the linkage items in Java and then pass them to the program in an Object[]. For example,

If your program has:

```
       LINKAGE SECTION.
       01 arg1.
          05 arg1-x pic x(5).
          05 arg1-n pic 9(5).
       01 arg2.
          05 arg2-x pic x(5).
          05 arg2-n pic 9(5).
```

Then you can call it from Java using something similar to:

```
import com.iscobol.types.PicX;
import com.iscobol.types.NumericVar;

      PicX ARG1;
      PicX ARG1_X;
      NumericVar ARG1_N;
      PicX ARG2;
      PicX ARG2_X;
      NumericVar ARG2_N;

      ARG1_X.set ("abcde");
      ARG1_N.set (12345);
      ARG2_X.set ("vwxyz");
      ARG2_N.set (56789);
      rc = IsCobol.call("SUBPROGNAME", new Object[] { ARG1, ARG2 });
```

Samples Included with isCOBOL

Java Code Calling an Existing COBOL Program

After installing isCOBOL you will find a working sample showing Java code calling an existing COBOL program in the following folder:

```
C:\Program Files\Veryant\isCOBOL2010\sample\is-java\java-call-iscobol
```

Java Code Using a COBOL Object

It is often cleaner to define your own class and methods using OOCOBOL that calls your legacy COBOL subprograms. This approach is highly recommended because it will give you precise control over the interface that your Java programmers will use. For an example, see Example of Java Calling a COBOL Class below. This example is included in the following folder:

```
isCOBOL2010\sample\is-java\java-uses-cobol-object
```

From a program such as this, you could call any other COBOL subprogram you have. The Java code that invokes this program does not need to reference isCOBOL framework classes.

Access COBOL Files Natively From Java Code with COBFILEIO

Another Veryant tool to consider is the COBFILEIO utility. The COBFILEIO utility works together with the isCOBOL Compiler to read COBOL source code and generate Java classes that can be used to access COBOL files and records. The Java programmer does not need any knowledge of COBOL data types or their underlying storage format, and COBFILEIO automatically generates Javadocs for the file and record classes. A data record is managed as an object with set and get accessor methods for each elementary field. The individual record fields are accessed using Java data types. The set methods automatically perform data validation and throw customizable exceptions.

COBFILEIO generates Object-Oriented COBOL source code for the classes. This source code can be customized and maintained in either COBOL or Java language. For more information, see the COBFILEIO section later in this document.

Developing and Debugging COBOL and Java Code Together

Using an Eclipse-based IDE it is possible to develop and debug COBOL and Java code together. It is possible to install the isCOBOL IDE plug-ins into an existing Eclipse-based IDE and to install the Java Development Toolkit into the isCOBOL IDE.

Example Steps for Setting Up Eclipse for Cross-language Development

Here are example steps for setting up an Eclipse-based IDE for Java Developers to enable COBOL development and cross-language debugging:

1. Use Eclipse for Java Developers or other IDE based on Eclipse 3.5 or later. You can download "Eclipse IDE for Java Developers (92 MB)" from http://www.eclipse.org/downloads/
2. To add the isCOBOL IDE Plugins, select "Install New Software" from the Help menu and click on "Add…". Enter "isCOBOL IDE" in the Name field and http://support.veryant.it/eclipse in the Location field, and press OK. Click the checkbox next to the isCOBOL release and all of the checkboxes in the items that begin with "isCOBOL". Click on "Next >" and follow the prompts
3. Once you have created or imported your COBOL project, create a new Debug Launch Configuration for your COBOL program
4. Allow the Java debugger to attach to the COBOL program by adding remote debug options to the System Properties. For example, copy/paste the following (all on one line):
```
-Xdebug
-Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=n
```
5. Create a new Debug Configuration for a Remote Java Application setting Connection Type to Standard (Socket Attach), and the Connection Properties to Host: localhost and Port: 8000
6. Start the COBOL program using the Debug Launch Configuration from steps 3 and 4
7. Start the Java Debugger using the Debug Configuration from step 5

8. To "step into" or "return from" Java code called from COBOL or COBOL code called from Java, first set breakpoints in the Java debugger or COBOL debugger at the appropriate entry or exit points.

Using Other Java Debuggers Such as JSwat and jdb

Java debuggers such as JSwat and jdb can be used for cross-language debugging with COBOL. Start debugging in the isCOBOL Debugger, and then use the remote debug feature in the Java debugger attach to the running Java process (i.e. COBOL program).

Veryant JISAM

Veryant JISAM is a Java-based ISAM file system for INDEXED files. Written completely in Java, Veryant JISAM works on a range of Java-compatible platforms from mainframes to handheld devices and includes a JUTIL utility for basic file management.

JISAM Limits

Maximum file size	9+ exabytes (over 9M terabytes)
Minimum record size	1 byte
Maximum record length	2 GB
Maximum number of keys	No limit
Maximum key length	256 bytes
Max number of segments per key	16
Maximum number of records	No limit

JUTIL Options

```
java JUTIL -info       <filename>
           -load       <filename> <binary sequential file>
           -unload     <filename> <binary sequential file>
           -loadtext   <filename> <line sequential file>
           -unloadtext <filename> <line sequential file>
           -loadr2     <filename> <binary sequential file>
           -shrink     <filename>
           -check      <filename>
           -rebuild    <filename>
```

COBFILEIO

The COBFILEIO utility works together with the isCOBOL Compiler to read COBOL source code and generate Java classes that can be used to access COBOL files and records.

COBFILEIO reads External File Description (EFD) XML files that are produced by the isCOBOL Compiler when it has been executed with the -efd compiler option. An EFD is a data dictionary that contains the mapping to use when COBOL files and records are accessed externally. COBFILEIO reads two files, an EFD file and an FD file containing the standard COBOL file descriptor, and generates a Java class for the COBOL file and a Java class for the COBOL record.

The Java programmer does not need any knowledge of COBOL data types or their underlying storage format, and COBFILEIO automatically generates Javadocs for the file and record classes.

The resulting classes can be brought into any Java development environment and Java developers can take advantage of rapid development features such as Eclipse's "code assist" to pop-up documentation and provide single key or click code completion.

A data record is managed as an object with set and get accessor methods for each elementary field. The individual record fields are accessed using Java data types. The set methods automatically perform data validation and throw customizable exceptions.

COBFILEIO generates Object-Oriented COBOL source code for the classes. This source code can be customized and maintained in either COBOL or Java language.

Usage:

```
cobfileio fileName [-e]
```

Where:

fileName is the external file name in all lowercase letters. For example, if in the file's SELECT statement there is ASSIGN TO DISK "/mydir/MYFILE", then the COBFILEIO command line would be "java COBFILEIO myfile".

The –e option causes COBFILEIO to generate the exception classes. This needs to be done only for the first file because the same exception classes are reused for every file.

java COBFILEIO with no additional arguments reports a usage message.

Example Steps

1. Create the EFD file by compiling the COBOL program with the -efd compile option. If desired, add the -efo=DirName compiler option to specify the directory where the EFD file will be output.
2. Execute the COBFILEIO utility with java COBFILEIO fileName -e. Include the -e option only the first time you run COBFILEIO.
3. Compile the exceptions classes. For example, javac *.java.
4. Compile the generated COBOL object classes. First compile the record class, FileNameRec.cbl. Then compile the file class, FileNameFile.cbl. If desired, add the -jj and -jc compiler options to generate Java source code.
5. If desired, use the javadoc utility that comes with the JDK to create Javadocs.

NOTE - By default, COBFILEIO attempts to read an EFD named fileName.xml from the current working directory. If your EFD file is in another directory then specify this directory as the value of the cobfileio.efd_path property. For example,

```
java COBFILEIO -J-Discobol.cobfileio.efd_path=../efd fileName
```

COBFILEIO Sample

CorpFile.cbl and CorpRec.cbl are OO COBOL programs generated by COBFILEIO. The first one implement a class to allow Java programs perform File operations on the Corp COBOL data file, like Open, Close, Read, Write, etc. The second one implements a class to allow Java programs perform sets and gets from record data items.

DataTesterCorp.java represents a native Java program that uses CorpFile and CorpRec classes in order to access the Corp COBOL data file. In this sample it opens the Corp file for Input, reads the record with main Key = “01”, closes the file and then retrieves a few data items from the read record.

A few representative methods have been shown on the CorpFile and CorpRec classes instead of the full program.

CorpFile.cbl

```
>> SOURCE FORMAT FREE
IDENTIFICATION DIVISION.
CLASS-ID.    CorpFile as "com.sample.cobfileio.CorpFile".

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
     CLASS JString as "java.lang.String"
     CLASS JShort as "short"
     CLASS CorpRec as "com.sample.cobfileio.CorpRec"
     CLASS FileException as "com.sample.cobfileio.FileException"
     .

IDENTIFICATION DIVISION.
OBJECT.

INPUT-OUTPUT         SECTION.
FILE-CONTROL.
   SELECT CORP-FILE ASSIGN TO "CORP"
      ORGANIZATION IS INDEXED
      ACCESS MODE IS DYNAMIC
      RECORD KEY IS CORP_NO
      FILE STATUS IS FILE-STATUS.

FILE SECTION.
>> SOURCE FORMAT FIXED
COPY "corp.fd".
>> SOURCE FORMAT PREVIOUS

WORKING-STORAGE SECTION.
PRIVATE.
77  FILE-STATUS PIC X(2).
01 ERROR-STATUS.
   05 PRIMARY-ERROR   PIC X(2).
   05 SECONDARY-ERROR PIC X(10).
77 STATUS-MESSAGE PIC X ANY LENGTH.
77 J-FILE-STATUS  object reference JShort.
77 J-EXTENDED-STATUS object reference JShort.
77 J-STATUS-MESSAGE object reference JString.
77 FILE-EXC object reference FileException.
77 TRACE-ON PIC 9(1).

PUBLIC.
77 KEY_CORP_NO   object reference JString.

PROCEDURE DIVISION.
```

```
*>*********
       IDENTIFICATION DIVISION.
       METHOD-ID. RaiseException as "raiseException".
       PROCEDURE DIVISION raising FileException.
       MAIN.
           call "C$RERR" using error-status status-message.
           string "FS [" delimited by size
                  file-status delimited by size
                  "], EXFS [" delimited by size
                  secondary-error delimited by space
                  "], MSG [" delimited by size
                  status-message delimited by size
                 "]" delimited by size
             into status-message.

           set j-file-status to file-status.
           set j-extended-status to error-status.
           set j-status-message to status-message.
           set file-exc to FileException:>new(j-status-message,
                           j-file-status, j-extended-status).
           raise file-exc.
       END METHOD.
*>*********
.
.
.
*>*********
       IDENTIFICATION DIVISION.
       METHOD-ID. FileNew as "new".
       PROCEDURE DIVISION.
       MAIN.
           self:>traceOff().
           set KEY_CORP_NO to JString:>new("CORP_NO").
           goback.
       END METHOD.

*>*********
       IDENTIFICATION DIVISION.
       METHOD-ID. DoOpen as "open".
       WORKING-STORAGE SECTION.
       77  open-type pic xx.
       LINKAGE SECTION.
       77  par-type object reference JString.
       PROCEDURE DIVISION using par-type raising FileException.
       DECLARATIVES.
       RAISE-EXCEPTION SECTION.
           use after standard error procedure on CORP-FILE.
```

```
            self:>raiseException.
            goback.
        END DECLARATIVES.
        MAIN.
            set open-type to par-type.
            evaluate open-type
            when "O" open output CORP-FILE
            when "I" open input  CORP-FILE
            when "IO" open i-o   CORP-FILE
            when "E" open extend CORP-FILE
            end-evaluate.
            goback.
        END METHOD.
*>*********
        IDENTIFICATION DIVISION.
        METHOD-ID. DoClose as "close".
        WORKING-STORAGE SECTION.
        LINKAGE SECTION.
        PROCEDURE DIVISION raising FileException.
        DECLARATIVES.
        RAISE-EXCEPTION SECTION.
            use after standard error procedure on CORP-FILE.
            self:>raiseException.
            goback.
        END DECLARATIVES.
        MAIN.
            close  CORP-FILE
            goback.
        END METHOD.
*>*********
.
.
.
*>*********
        IDENTIFICATION DIVISION.
        METHOD-ID. DoRead as "read".
        WORKING-STORAGE SECTION.
        77  key-name       PIC X(32).
        LINKAGE SECTION.
        77  par-rec object reference CorpRec.
        77  par-key object reference JString.
        PROCEDURE DIVISION using par-rec, par-key raising FileException.
        DECLARATIVES.
        RAISE-EXCEPTION SECTION.
            use after standard error procedure on CORP-FILE.
            self:>raiseException.
            goback.
```

```
        END DECLARATIVES.
        MAIN.
            invoke par-rec "getCorpRec" using CORP_REC
            set key-name to par-key
            evaluate key-name
            when "CORP_NO"
              read CORP-FILE key is CORP_NO
            when other
              read CORP-FILE key is CORP_NO
            end-evaluate
            invoke par-rec "setCorpRec" using CORP_REC
            if TRACE_ON = 1
               display "FS=[" FILE-STATUS "] RV=[" CORP_REC "]"
               display "CORP NO = " corp-no " Corp Top Price Xlg = " Corp-
Top-Price-Xlg
            end-if
            goback.
        END METHOD.
*>*********


        END OBJECT.
```

CorpRec.cbl

```
        >>SOURCE FORMAT FREE
        IDENTIFICATION DIVISION.
        CLASS-ID.    CorpRec as "com.sample.cobfileio.CorpRec".

        ENVIRONMENT DIVISION.
        CONFIGURATION SECTION.
        REPOSITORY.
             CLASS JString as "java.lang.String"
             CLASS JBigDecimal as "java.math.BigDecimal"
             CLASS JLong as "long"
             CLASS JBigInt as "java.math.BigInteger"
             CLASS JStringTooLongException as
"com.sample.cobfileio.StringTooLongException"
             CLASS JWrongScaleException as
"com.sample.cobfileio.WrongScaleException"
             CLASS JNumberTooLongException as
"com.sample.cobfileio.NumberTooLongException"
             .

        IDENTIFICATION DIVISION.
```

```
       OBJECT.

       WORKING-STORAGE SECTION.
       PRIVATE.
       >> SOURCE FORMAT FIXED
       COPY "corp.rec".
       >> SOURCE FORMAT PREVIOUS
       77 StrExc object reference JStringTooLongException.
       77 ScaleExc object reference JWrongScaleException.
       77 NumberExc object reference JNumberTooLongException.
       77 exception-message PIC X ANY LENGTH.
       77 j-exc-message object reference JString.
       77 TRACE_ON  PIC 9(1).

       PROCEDURE DIVISION.
.
.
.
*>*********
       IDENTIFICATION DIVISION.
       METHOD-ID. setCorpNo as "setCorpNo".
       WORKING-STORAGE SECTION.
       77 par-len      pic 9(5).
       LINKAGE SECTION.
       77  par-val object reference JString.
       PROCEDURE DIVISION using par-val
                         raising JStringTooLongException.
       MAIN.
           set par-len to par-val::"length"().
           if par-len > 2
              string "String Too Long. Field " delimited by size
                     "[CORP_NO] " delimited by size
                     "Max Length [2] " delimited by size
                     "Current Length [" delimited by size
                     par-len delimited by size
                     "]" delimited by size
                     into exception-message
              end-string
              self:>raiseStrException
              goback
           end-if.
           set CORP_NO to par-val.
           if TRACE_ON = 1
              DISPLAY "setCorpNo: VALUE=[" CORP_NO "]"
           end-if
           goback.
       END METHOD.
```

```
*>*********
       IDENTIFICATION DIVISION.
       METHOD-ID. getCorpNo as "getCorpNo".
       WORKING-STORAGE SECTION.
       LINKAGE SECTION.
       77  ret-val object reference JString.
       PROCEDURE DIVISION returning ret-val.
       MAIN.
           set ret-val  to CORP_NO.
           if TRACE_ON = 1
              DISPLAY "getCorpNo: VALUE=[" CORP_NO "]"
           end-if
           goback.
       END METHOD.
*>*********
       IDENTIFICATION DIVISION.
       METHOD-ID. setCorpCentury as "setCorpCentury".
       WORKING-STORAGE SECTION.
       77 par-len      pic 9(5).
       77 big-number   pic 9(30).
       LINKAGE SECTION.
       77  par-val object reference JLong.
       PROCEDURE DIVISION using par-val
                         raising JNumberTooLongException.
       MAIN.
           set big-number to par-val.
           if big-number > 99
              string "Number Too Large. Field " delimited by size
                     "[CORP_CENTURY] " delimited by size
                     "Max Number [99] " delimited by size
                     "Current Number [" delimited by size
                     big-number delimited by size
                     "]" delimited by size
                     into exception-message
              end-string
              self:>raiseNumberException
              goback
           end-if.
           set CORP_CENTURY to par-val.
           if TRACE_ON = 1
              DISPLAY "setCorpCentury: VALUE=[" CORP_CENTURY "]"
           end-if
           goback.
       END METHOD.
*>*********
       IDENTIFICATION DIVISION.
       METHOD-ID. getCorpCentury as "getCorpCentury".
```

```
       WORKING-STORAGE SECTION.
       LINKAGE SECTION.
       77  ret-val object reference JLong.
       PROCEDURE DIVISION returning ret-val.
       MAIN.
           set ret-val  to CORP_CENTURY.
           if TRACE_ON = 1
              DISPLAY "getCorpCentury: VALUE=[" CORP_CENTURY "]"
           end-if
           goback.
       END METHOD.
*>*********
.
.
.
       END OBJECT.
```

DataTesterCorp.java

```
import java.math.BigDecimal;
import com.sample.cobfileio.*;

public class DataTesterCorp {

   private static String CORP_KEY_NAME = "CORP_NO";

   public static void main(String[] args) throws Exception {
               Boolean errorFound;

         CorpFile corpFile = new CorpFile();
         CorpRec corpRec = new CorpRec();
         corpFile.traceOn();
         errorFound=false;
         try {
               corpFile.open("I");
               corpRec.setCorpNo("01");
               corpFile.read(corpRec,CORP_KEY_NAME);
               corpFile.close();
         } catch (com.sample.cobfileio.FileException e) {
               errorFound = true;
               e.printStackTrace();
         } catch (com.sample.cobfileio.StringTooLongException e) {
               errorFound = true;
               e.printStackTrace();
         }
```

```
                corpRec.traceOn();
                System.out.println(corpRec.getCorpCentury());
          System.out.println( corpRec.getCorpCookMin());

   System.out.println( corpRec.getCorpTopPriceXlg().doubleValue() );

   System.out.println( corpRec.getCorpDelvyChg().doubleValue() );
          System.out.println(corpRec.getCorpDineinMinAlpha().trim());
          System.out.println(corpRec.getCorpCity().trim());
      }
   }
```

How do you take your Java? - An Article

A little dash or a splash of COBOL can make for the perfect mix

isCOBOL allows programmers to take advantage of the best of both COBOL and Java by providing common ground between the two technologies. With isCOBOL, organizations can add the power and flexibility of the Java platform to existing COBOL programs and also make COBOL investments easily accessible to the Java world. This article briefly examines some of the synergies that exist between COBOL and Java technology and includes a few examples to illustrate how COBOL and Java programmers can incorporate each other's technologies with isCOBOL Evolve.

Common Business Oriented Language – 50 Years Young

One of the original design goals for COBOL was to offer a programming language that would be easy to read and understand by managers with no programming training. Although managers rarely took advantage of that -- either fifty years ago or today -- a direct outcome of this goal is that the COBOL language is comprised of familiar English building block structures such as verbs, clauses, sentences, sections, and divisions. The Environment Division, for instance, is a set area found in all COBOL programs that describes the particular environment for which the program is written and all external references, such as to devices, files, currency symbols, and object classes that a COBOL application requires. A COBOL developer looking to move a program to a new platform or region knows that this centralized area of code will have to be reviewed and potentially altered to accommodate the new target platform's requirements.

COBOL's straightforward programming approach, clear organizational structure, ability to deal with business logic, adoption of latest technological paradigms, and portability are some of the primary factors that contributed to its widespread adoption and persistence in the industry.

Originally called Green, then changed to Oak, finally known as Java – 15 years mature

The Java programming language leverages small, modular building blocks (known as object classes) that are tied together to form a complete application. Two important concepts in object-oriented programming are encapsulation and inheritance:

- *Encapsulation* means that a Java class can be viewed as a "black box" whose internal workings are hidden. To use a class it is only necessary to know what the class does, not the details of how the class does it. As long as the public interface stays the same, the internal mechanisms of a class can be improved or the class can be replaced with a different one without impact on other components.
- *Inheritance* means that new classes can be formed using classes that have already been defined, leveraging and extending their functionality.

Object classes can be easily assembled, disassembled, and reused in new applications as business requirements change. Since Java programs run in any environment that supports the proper Java Runtime Environment (JRE) they are extremely portable and can be moved between platforms without any program code changes.

It is this modularity, portability, and reusability that contribute to the growing adoption of Java programming in the industry today.

COBOL with A Dash of Java

With isCOBOL, developers can quickly add a wealth of new functionality to COBOL applications by integrating free, reusable, modular solutions found in the Java SE and Java EE Development Kits as well as in various open source communities such as The Apache Software Foundation and SourceForge. Solutions available to Java developers are now equally available to COBOL developers -- the challenge is only to identify the existing Java solution needed for a particular programming task and to write the COBOL code to interface with it.

It is important to emphasize that to integrate these Java solutions, a COBOL programmer does not need to be trained and experienced in the Java language. The programmer's only requirements are to be able to understand basic object-oriented concepts such as classes and methods, to know how to create an object and invoke a method in COBOL, and to understand how to read and use the Javadocs for the particular Java class to be used. These topics are covered in the isCOBOL Language Reference Manual and any beginner's book or online resource on Java programming.

As an example of the new functionality isCOBOL makes available to the COBOL developer, consider the full array of XML APIs and tools available to a Java programmer. To parse local XML files or XML from a URL, a COBOL developer could use the JDOM Java package. Here

is a simple program that retrieves an XML document from a URL and parses the XML using JDOM.

Example of COBOL Using Java Packages

```
program-id. XMLfromURL.

       configuration section.
       repository.
           class J_Iterator      as "java.util.Iterator"
           class J_Element       as "org.jdom.Element"
           class J_SAXBuilder    as "org.jdom.input.SAXBuilder"
           class J_Document      as "org.jdom.Document"
           class J_URL           as "java.net.URL"
           .

       working-storage section.
       77  W_SAXBuilder               object reference J_SAXBuilder.
       77  W_Document                 object reference J_Document.
       77  W_Element                  object reference J_Element.
       77  W_Iterator                 object reference J_Iterator.
       77  xml                        pic x any length.

       procedure division.
       main.
           try
           move "file:///C:/Program%20Files/Veryant/isCOBOL2008/sample
      -          "/issamples/files/Members.xml" to xml
         set W_SAXBuilder to J_SAXBuilder:: "new"
         set W_Document to W_SAXBuilder:: "build"(J_URL:: "new"(xml))
         set W_Element to W_Document:: "getRootElement"
         set W_Iterator to W_Element:: "getChildren":: "iterator"
             perform until not W_Iterator:: "hasNext"
                 set W_Element to W_Iterator:: "next" as J_Element
                 display W_Element:: "getChildText"( "first_name")
                             " " W_Element:: "getChildText"( "name")
                             " = " W_Element:: "getChildText"( "age")

               end-perform
           catch exception
               display exception-object
           end-try.

       stop run.
```

Developers can also take advantage of Java interfaces offered by third-party commercial software providers. For example, to add an electronic payment processing feature, a business could choose an electronic payment service provider and request the Java interface. The COBOL developer could use that Java interface directly from COBOL.

Java with a Splash of COBOL

With isCOBOL, developers can now write programs in COBOL that can be called directly from Java as if they were written in the Java language. Java developers do not need to learn COBOL in order to make use of COBOL assets.

COBOL developers can present their subprograms as POJOs (Plain Old Java Objects). The Java developer receiving the objects needs only to know the class and method names, parameters, and return values to make use of them. The fact that the object class was written in COBOL or that it calls other COBOL subprograms is inconsequential.

The isCOBOL Compiler automatically converts existing COBOL programs into POJOs. No COBOL code changes are required. The COBOL program name becomes the Java class name. The resulting Java class has a "call" method which Java code can invoke to call the COBOL program. If required, the COBOL developer can precisely define the program's object interface using object-oriented COBOL syntax. This COBOL object can accept and return Java data types to make the Java developer's task even easier. For instance, if an organization is looking to integrate with a JEE server and needs to deploy a Servlet, Web Service, or Enterprise JavaBean (EJB), etc. those can all now be created natively in COBOL.

So what does this look like on the ground level to a developer? Let's look at a few examples.

Example of COBOL Calling Java Classes

Here is the source code for a COBOL program that uses object-oriented COBOL syntax to access a Java class "java.lang.System" and use a Java data type "java.lang.String".

```
PROGRAM-ID. obj-system.
        CONFIGURATION SECTION.
        REPOSITORY.
             class Sys as "java.lang.System"
             class JString as "java.lang.String"
             .
```

```
 WORKING-STORAGE SECTION.
 77  pic-x-item  pic x(50).
* There are two styles for specifying classes in object references
 77  jstring1    object reference "java.lang.String".
 77  jstring2    object reference JString.
 PROCEDURE DIVISION.
 main.
* There are 3 styles of invoking an object method
* Use the INVOKE statement
     invoke Sys "getProperty" using "os.name" giving pic-x-item.
     display "Operating System: " pic-x-item.
* Use the SET statement with the double-colon operator
     set jstring1 to Sys::"getProperty" ( "os.arch" ).
     display "OS Architecture: " jstring1.
* Use the SET statement with the colon-greater-than operator
     set jstring2 to Sys:>getProperty ( "java.version" ).
     display "Java Version: " jstring2.
     goback.
```

This example demonstrates the use of the repository paragraph to define COBOL user-words that can be used to reference Java classes. It is also possible to specify the full Java class name when declaring a variable as an object reference. This example shows 3 styles of invoking methods. It shows the ability to use Java data types in COBOL statements taking advantage of the isCOBOL Framework to automatically convert the Java data type to the COBOL type that will work in that statement. This is just one example, any Java class and any Java data type can be used.

Example of Java Calling a COBOL Class

COBOL developers can write programs in COBOL that can be called directly from Java as if they were written in the Java language. Here is the source code for an object class "isobject" written in COBOL.

```
IDENTIFICATION DIVISION.
CLASS-ID. isobj as "isobject".

IDENTIFICATION DIVISION.
FACTORY.

CONFIGURATION SECTION.
REPOSITORY.
    class jint as "int".
```

```
PROCEDURE DIVISION.

IDENTIFICATION DIVISION.
METHOD-ID. method1 as "add1".
WORKING-STORAGE SECTION.
77  var1      pic 9(9).
77  var2      pic 9(9).
77  result    object reference jint.
LINKAGE SECTION.
77  buffer object reference "java.lang.String".
77  num     object reference jint.
procedure division using buffer num returning result.
MAIN.
    display "1st parameter: " buffer.
    display "2nd parameter: " num.
    display "> isobject.add1: result = num + 1".
    set var1 to num.
    compute var2 = 1 + var1.
    set result to var2 as int.
    goback.

END METHOD.
END FACTORY.
```

This class has one method "add1" which takes a Java string and a Java integer, displays them to the standard output stream, adds 1 to the integer, and returns the result. The isCOBOL Compiler will output a file named isobject.class which can be used by the Java developer.

And here is the source code for a Java program which uses the COBOL object "isobject" just as if it were written in Java language.

```
public class callCobolObject {
   public static void main( String args[] ) {
      String PAR1 = "BUFFER";
      int PAR2 = 5;
      System.out.println ("I'm invoking isobject.add1 with"+PAR1+
      ", "+PAR2);
int RESULT = isobject.add1(PAR1, PAR2);
System.out.println ("I'm back with: "+RESULT);
System.exit(0);
 }
}
```

Notice the call to isobject.add1(PAR1, PAR2) is simple Java code. The IDE used by the Java developer will assist with code completion. Even the IDE doesn't know that isobject.class was written in COBOL because behind-the-scenes the isCOBOL Compiler converted the COBOL to Java source code.

Conclusion

With isCOBOL, COBOL is your language, and Java is your platform. You get the benefits of the Java platform automatically but you also retain all of your understanding of your application and the ability to quickly fix bugs and enhance your application in COBOL, while being able to switch to Java programming at any time if you wish to, or integrate more and more Java programming at your own desired pace. The isCOBOL remote debugging capability is useful for debugging Web Services and other programs that are run by server software or that do not have a user interface.

To learn more about isCOBOL and related Veryant COBOL technologies, visit www.veryant.com.

Appendix A

Quick FAQ

What does the compilation create? What does the code run on?

The isCOBOL Compiler translates COBOL source code into Java classes that are then executed with the Java Virtual Machine (JVM).

How is the integration with new Java code?

Since the isCOBOL Compiler generates pure Java output, once a desired third-party Java solution has been identified, a developer need only write the COBOL code required to interface with it. isCOBOL allows COBOL user-words to reference Java classes and automatically converts Java data types to the COBOL type that will work in a particular statement, simplifying the integration process.

The isCOBOL Compiler automatically converts existing COBOL programs into POJOs (Plain Old Java Objects). No COBOL code changes required. The Java developer receiving the objects needs only to know the class and method names, parameters, and return values to make use of them. The fact that the object class was written in COBOL or that it calls other COBOL subprograms is inconsequential.

How does it help companies to move to Java in the future?

With isCOBOL, it is up to the user as to whether or not the best route forward is to continue programming and maintaining code in COBOL or the Java programming language. isCOBOL enables programmers to choose between continuing to use proven COBOL language code for as long as appropriate, or moving to Java technology when that becomes a better fit for your business.

By default, the Java intermediate source code is created in memory and is not written to disk by the isCOBOL Compiler. However, you can specify the "–jj" compiler option at any time if you want to save this source code to disk in Java source files.

If desired to switch from COBOL programming to Java programming down the road, the business would likely wrap its COBOL programs in objects using Object-Oriented COBOL or Java for future use. In this case, the Java programmers would write new code in the Java

programming language that would access the Java objects produced from the isCOBOL code. After compilation with isCOBOL Compiler, the COBOL programs are Java classes (i.e. .class files), and can be used by other Java classes without regard for the fact that they were originally written in COBOL. This way a Java developer would never need to look at the source because they would treat the COBOL at that point just like any other Java object.

How do you work in parallel with both COBOL and Java?

Because Veryant has adapted the Eclipse platform to build the isCOBOL IDE, it is possible to develop, maintain and debug COBOL and Java language programs together. Having a common development environment for both COBOL and Java helps overcome the divide between development teams, fosters cross training and facilitates the diffusion of valuable business and institutional knowledge.