isCOBOL Evolve: Interoperability

Key Topics:

- Calling C Language Functions and Other External Routines
- Calling isCOBOL from C
- Calling isCOBOL from Java
- Converting Java Source Code to Object-oriented COBOL

Table of Contents

Chapter 1

isCOBOL and C

Calling C Language Functions and Other External Routines

Introduction

The ability of programs compiled with isCOBOL to interact with external routines, such as C language functions, is fully supported and described in this document.

Programs compiled with isCOBOL can easily call external language routines without the need for code changes, and it is unnecessary to know anything about Java Native Interface (JNI) technology.

Environment variables and isCOBOL runtime framework properties are adjusted to point the isCOBOL runtime framework to functions that are available in statically-linked or dynamically-linked libraries.

Directories containing the libraries must be specified in the operating system library search path variable, such as LD_LIBRARY_PATH, LIBPATH, SHLIB_PATH or PATH. See the man page for dlopen() on UNIX/Linux or MSDN "search path" for Windows.

NOTE - isCOBOL Runtime Framework Property, Framework Property and Property have the same meaning. They refer to runtime configuration variables, known on the Java platform as "properties", which can be set before or during program execution.

COBOL and C Data Types

As in other COBOL implementations, in order to successfully call C language functions from isCOBOL and exchange data, the C and COBOL data storage types must be matched.

For example:

In COBOL, an integer defined as	would be represented in C, as
`77 MY-DATA-INT SIGNED-INT EXTERNAL`	`int my_data_int;`

Calling C Language Functions from isCOBOL

isCOBOL supports two methods of external language interoperability: Dynamic and Static

Dynamic Method

These are the steps to use the Dynamic method of calling external language routines:

- Create a shared object library (or DLL on Windows) containing the routines; e.g. “myroutines.so” (or "myroutines.dll" on Windows)
- Make sure that the directory containing this library is listed in the operating system library search path variable setting; e.g. LD_LIBRARY_PATH, LIBPATH or SHLIB_PATH (or current directory or PATH on Windows).
- Add the $ISCOBOL/native/lib (or $ISCOBOL/bin on Windows) directory to the same environment variable. This directory contains the file “libdyncall.so” (or dyncall.dll on Windows) which is the isCOBOL runtime framework dynamic call interface.

  Note - for modularity, Veryant recommends using a different directory for user-created shared libraries.

Then Either

- Set the iscobol.shared_library_list property to specify the shared libraries to be used. e.g. “iscobol.shared_library_list=myroutines.so”

Note - multiple library names must be separated using the newline character, "\n", or the platform-specific separator; for UNIX platforms use the colon “:” character, for Windows use semi-colon “;”

-or-

- Load myroutines.so in the program logic using the CALL statement:

```
CALL "myroutines.so".
```

The isCOBOL runtime framework will then load “myroutines.so” and make the routines available for CALLing.

In case the function in the shared library is named the same way as the library itself, it can be called directly. Consider the following statement for example.

```
CALL "test" USING Param-1.
```

If the test function is included in libtest.so (or test.dll on Windows) shared library, there’s no need to call the shared library or to add it to iscobol.shared_library_list configuration property. The isCOBOL Framework will load the library automatically. Otherwise, if the function name does not match with the library name, it’s necessary to load the library through CALL or iscobol.shared_library_list before using its functions.

If the function name has the same name of the native library, you can call the function directly and the isCOBOL Framework will automatically load the library in memory.

For example, with the following statement

```
CALL "foo" USING Param-1.
```

the isCOBOL Framework tries to load the foo library (foo.dll on Windows, foo.so on Unix), unless the library is already loaded in memory, and then calls the foo function passing Param-1 to it.

isCOBOL allows to pass up to 255 parameters to a C function.

Static Method

Static C functions are searched in a library named *stacall* (e.g. stacall.dll on Windows, libstacall.so on Linux) for COBOL programs compiled without the -cp option and a library named *stacall_n* (e.g. stacall_n.dll on Windows, libstacall_n.so on Linux) for COBOL programs compiled with the -cp option.

The folder *native/src* in the isCOBOL installation directory includes the necessary items to build this library.

Add the code of your static functions to the file "usercall.c", then build the library following the instructions below.

Make sure that the directory containing the stacall library is available in the operating system library search path, e.g. LD_LIBRARY_PATH on Linux, current directory or PATH on Windows)

Building Stacall on Unix/Linux

1. Edit the *Makefile* and change the value of the JAVA_HOME variable according to the location of your JDK.
2. Run the following command:

```
make -f Makefile
```

The command generates both *libstacall.so* and *libstacall_n.so*.

Building Stacall on Windows

1. Open either the *stacall* or the *stacall_n* project depending on the library you wish to generate.
   Note - these projects were made using Microsoft Visual Studio 2008. An older version of Visual Studio will fail to open them, while a newer version will trigger a project conversion.
2. In *Project -> Properties -> C++ > General*, change the JDK path in the "Additional Include Directories" field in order to match your JDK installation.
3. Build the project.

Framework Properties

The properties used by isCOBOL when calling C programs, can be specified on the command line with the java -D option, or they can be specified in a properties file. See the Configuration section of the User Guide for more information on setting properties.

Property	Meaning
`iscobol.shared_library_list`	Specifies the names of UNIX/Linux shared object libraries or Windows DLLs

Program Examples

The following example programs show how a C language routine can be called from isCOBOL:

The C source file “calltest.c” has these lines:

```
#include <stdio.h>
calltestc(int *pitem1)
{
      printf("item1 = %x\n", *pitem1);
}
```

Compile this routine using the C compiler, then link it into a shared object library called “calltestc.so”.

The programmer can access this shared object library from COBOL in one of two ways:

- Use the CALL statement to load the library “calltestc.so”, making its routines available for subsequent CALL statements

-or-

- Set the isCOBOL property “iscobol.shared_library_list=calltestc.so”. The isCOBOL framework automatically loads “calltestc.so” making its routines available for subsequent CALL statements.

The isCOBOL program called, “calltest.cbl” has the following lines (Notice how the data variable item-1 is declared with storage to complement the C language routine it will be calling.):

```
       id division.
       program-id. calltest.
       data division.
       working-storage section.
       77 item-1 pic 9(8) comp-5.
       procedure division.
            move x#7fff to item-1.
            display item-1.
      * comment - delete the following line if using the
      *  iscobol.shared_library_list method:
       call "calltestc.so".

       call "calltestc" using item-1.
```

More examples can be found in the $ISCOBOL_HOME/sample/is-c folder installed with isCOBOL

Calling isCOBOL from C

INTRODUCTION

This chapter describes how a C program can call a COBOL program using isCOBOL.

Programs compiled with isCOBOL can easily be called by C programs through a specific bridge library.

This chapter describes the library usage and provides the reference of its functions.

A sample program for this feature is installed with isCOBOL in the folder $ISCOBOL_HOME/sample/is-c/c-call-iscobol.

The iscobolc Library

The ability to call a COBOL program from a C program is provided by the iscobolc library (identified by iscobolc.dll on Windows and libiscobolc.so on Unix).

There are two iscobolc libraries:

iscobolc	to call COBOL programs compiled without -cp option
iscobolc_n	to call COBOL programs compiled with -cp option

The iscobolc library has dependences to the Java jvm library (identified by jvm.dll on Windows and libjvm.so on Unix). Both of these libraries must be available to the C program.

Functions Reference

The iscobolc library provides the following functions:

isCobolInit

isCobolCall

isCobolCallNoStop

isCobolCancel

isCobolFunc

isCobolTidy

isCobolExit

isCobolError

isCobolGetJNIEnv

isCobolGoback

For each one of the above functions, an additional extended function is provided to work in multi-thread environments.

isCobolInitEx

isCobolCallEx

isCobolCallNoStopEx

isCobolCancelEx

isCobolFuncEx

isCobolErrorEx

isCobolGetJNIEnvEx

isCobolGobackEx

isCobolThreadTidy

These functions are defined in the iscobolc.h header file, that is installed with isCOBOL in the folder $ISCOBOL_HOME/native/include.

Note: COBOL programs called thru isCobolCall and isCobolFunc are not aware of the C environment. An environment variable set by the C program cannot be retrieved by the COBOL program using the ACCEPT FROM ENVIRONMENT statement. In order to set an environment variable for the COBOL program, create a separate COBOL program that sets the variable using the SET ENVIRONMENT statement and then call this program using isCobolFunc.

isCobolInit

The isCobolInit function initializes the JVM.

```
int isCobolInit (int optc, char *optv[], void *ejvm);
```

Parameters

optc	Number of options. If set to a value less than zero, it means that the jvm of the third parameter has already been created.
optv	Starting options. If there is an option in the form "-Djava.class.path=", the classpath is set accordingly, otherwise the classpath is got from the environment variable.
ejvm	Pointer to an existing JVM.

isCobolInitEx

The isCobolInitEx function is an extended version of the isCobolInit function that allows to pass a further argument in order to receive a pointer to the isCobol interface environment. This pointer shall be used in subsequent function call in multi-thread environments.

```
int isCobolInitEx (int optc, char *optv[], void *ejvm, void **pice);
```

Parameters

optc	Number of options. If set to a value less than zero, it means that the jvm of the third parameter has already been created. When a thread needs to be attached to an isCOBOL environment and the JVM is already initialized, this parameter must be set to -1 while the second and the third are ignored. In this case you may consider calling isCobolThreadInit instead.
optv	Starting options. If there is an option in the form "-Djava.class.path=", the classpath is set accordingly, otherwise the classpath is got from the environment variable.
ejvm	Pointer to an existing JVM.
pice	Pointer to the isCOBOL environment for the thread

isCobolThreadInit

The isCobolThreadInit function is a shortcut to create a new isCOBOL environment in an existing JVM. It has the same effect as isCobolInitEx.

```
int isCobolThreadInit (pice)
```

Parameters

pice	Pointer to the isCOBOL environment for the thread.

isCobolCall

The isCobolCall function enables C programs to call COBOL programs.

If a COBOL program issues a STOP RUN, the C program terminates.

Parameters are passed BY REFERENCE.

```
int isCobolCall (char *name, int argc, char *argv[], int argl[], long *crc);
```

Parameters

name	Name of the COBOL program to call.
argc	Arguments count. It should be set to the size of argv.
argv	Arguments values.
argl	Arguments length.
crc	Return code

isCobolCallNoStop

The isCobolCallNoStop function enables C programs to call COBOL programs.

If a COBOL program issues a STOP RUN, the C program continues.

Parameters are passed BY REFERENCE.

```
int isCobolCallNoStop (char *name, int argc, char *argv[], int argl[], long *crc);
```

Parameters

name	Name of the COBOL program to call.
argc	Arguments count. It should be set to the size of argv.
argv	Arguments values.
argl	Arguments length.
crc	Return code

isCobolCallEx

The isCobolCallEx function is an extended version of the isCobolCall function that allows you to pass a further argument specifying the isCOBOL environment when a call is performed in a multi-threaded environment..

```
int isCobolCallEx (char *name, int argc, char *argv[], int argl[], long *crc, void
*pice);
```

Parameters

name	Name of the COBOL program to call.
argc	Arguments count. It should be set to the size of argv.
argv	Arguments values.
argl	Arguments length.
crc	Return code.
pice	Pointer to an isCOBOL environment created by isCobolInitEx.

isCobolCallNoStopEx

The isCobolCallNoStopEx function is an extended version of the isCobolCallNoStop function that allows you to pass a further argument specifying the isCOBOL environment when a call is performed in a multi-threaded environment..

```
int isCobolCallNoStopEx (char *name, int argc, char *argv[], int argl[], long *crc,
void *pice);
```

Parameters

name	Name of the COBOL program to call.
argc	Arguments count. It should be set to the size of argv.
argv	Arguments values.
argl	Arguments length.
crc	Return code.
pice	Pointer to an isCOBOL environment created by isCobolInitEx.

isCobolCancel

The isCobolCancel function cancels a previously called COBOL program.

```
int isCobolCancel (char *name);
```

Parameters

name	Name of the COBOL program to cancel.

isCobolCancelEx

The isCobolCancelEx function is an extended version of the isCobolCancel function that allows to pass a further argument specifying the isCOBOL environment when a call is performed in a multi-threaded environment.

```
int isCobolCancelEx (char *name, void *pice);
```

Parameters

name	Name of the COBOL program to cancel.
pice	Pointer to an isCOBOL environment created by isCobolInitEx

isCobolFunc

The isCobolFunc function allows to call a COBOL program the same way as isCobolCall except that the program is automatically cancelled as soon as it terminates.

```
int ISCOBOLEXPORT isCobolFunc (char *name, int argc, char *argv[], int argl[], long
*crc);
```

Parameters

name	Name of the COBOL program to call.
argc	Arguments count. It should be set to the size of argv.
argv	Arguments values.
argl	Arguments length.
crc	Return code.

isCobolFuncEx

The isCobolFuncEx function is an extended version of the isCobolFunc function that allows to pass a further argument specifying the isCOBOL environment when a call is performed in a multi-threaded environments.

```
int isCobolFuncEx (char *name, int argc, char *argv[], int argl[], long *crc, void
*pice);
```

Parameters

name	Name of the COBOL program to call.
argc	Arguments count. It should be set to the size of argv.
argv	Arguments values.
argl	Arguments length.
crc	Return code.
pice	Pointer to an isCOBOL environment created by isCobolInitEx.

isCobolTidy

The isCobolTidy function shuts down the JVM.

```
int isCobolTidy (void);
```

isCobolThreadTidy

The isCobolThreadTidy terminates a single thread in a multi-thread environment.

```
int isCobolThreadTidy (void *pice);
```

Parameters

pice	Pointer to an isCOBOL environment created by isCobolInitEx.

isCobolExit

The isCobolExit function shuts down the JVM and exits.

```
int isCobolExit (int exitCode);
```

Parameters

exitCode	Exit code of the application.

isCobolError

The isCobolError function returns the last error message, if any.

```
int isCobolError (char *msg, int len);
```

Parameters

msg	Buffer to store the error message.
len	Length of the buffer.

isCobolErrorEx

The isCobolErrorEx function is an extended version of the isCobolError function that allows to pass a further argument specifying the isCOBOL environment when a call is performed in a multi-threaded environments.

```
int isCobolErrorEx (char *msg, int len, void *pice);
```

Parameters

msg	Buffer to store the error message.
len	Length of the buffer.
pice	Pointer to an isCOBOL environment created by isCobolInitEx.

isCobolGetJNIEnv

The isCobolGetJNIEnv function the JNIEnv pointer to handle the JNI API directly.

```
void * isCobolGetJNIEnv ();
```

isCobolGetJNIEnvEx

The isCobolGetJNIEnvEx is an extended version of the isCobolGetJNIEnv function that allows to pass a further argument specifying the isCOBOL environment when a call is performed in a multi-threaded environments.

```
void * isCobolGetJNIEnvEx (void *pice);
```

Parameters

pice	Pointer to an isCOBOL environment created by isCobolInitEx.

isCobolGoback

The isCobolGoback function makes the COBOL program behave like a GOBACK statement was executed by issuing a com.iscobol.so.GobackException exception.

```
int isCobolGoback (void);
```

isCobolGobackEx

The isCobolGobackEx function is an extended version of the isCobolGoback function that allows to pass a further argument specifying the isCOBOL environment when a call is performed in a multi-threaded environments.

```
int isCobolGobackEx (void *pice);
```

Parameters

pice	Pointer to an isCOBOL environment created by isCobolInitEx.

C++ Sample

This chapter contains two snippets that work as a sample for calling a COBOL program from C++ on Windows.

To build a COBOL program into a DLL requires Microsoft Visual Studio or another suitable C compiler. To call COBOL from C++ on Windows you use the isCOBOL C API which is available in the following files in the isCOBOL Evolve installation directory:

- bin\iscobolc.dll
- native\lib\iscobolc.lib
- include\iscobolc.h

For example, here is a C++ class that can be built into a DLL to call COBOL:

```
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include "TestIsCobolDoc.h"

extern "C" {
#include "iscobolc.h"
}

CTestIsCobolDoc::CTestIsCobolDoc(void)
{
}

CTestIsCobolDoc::~CTestIsCobolDoc(void)
{
}

#define MAX_MSG_LEN 1024
```

```
void
CTestIsCobolDoc::CallCobolProgram(void)
{
  long crc;
  char errmsg[MAX_MSG_LEN + 1];
  char **cobargv;
  int *cobargl;

  int retval;
  int jvmoptnum = 2;
  static char *jvmopts[2] = { "-
Djava.class.path=C:\\Veryant\\isCOBOL_SDK2021R1\\lib\\iscobol.jar;C:\\Users\\dlubin\\D
ocuments\\Visual Studio 2008\\Projects\\CallCOBOL\\debug",
      "-Discobol.display_message=3" };

  retval = isCobolInit(jvmoptnum, jvmopts, 0);
  if(retval != ISCOBOLC_SUCCESS)
  {
    fprintf(stderr,"Error calling isCobolInit\n");
    exit(1);
  }

  errmsg[MAX_MSG_LEN] = '\0';

  int cobargc = 4;

  cobargv = (char **)malloc(sizeof(char*) * cobargc);
  cobargl = (int *)malloc (sizeof(int) * cobargc);

  int error_val = 12345678;

  cobargv[0] = (char *)&error_val;
  cobargl[0] = sizeof(int);

  cobargv[1] = _strdup("DB Service   ");
  cobargl[1] = strlen(cobargv[1]);

  cobargv[2] = _strdup("Username    ");
  cobargl[2] = strlen(cobargv[2]);

  cobargv[3] = _strdup("Password    ");
  cobargl[3] = strlen(cobargv[2]);

 if (isCobolCall("TESTPROG", cobargc, cobargv, cobargl, &crc) !=
              ISCOBOLC_SUCCESS)
  {
    fprintf(stderr, "Error calling isCobolCall\n");
    if ( isCobolError(errmsg, MAX_MSG_LEN) != ISCOBOLC_ERROR )
    {
      fprintf(stderr,"errmsg = %s\n", errmsg);
    }
}
```

```
  free(cobargv[1]);
  free(cobargv[2]);
  free(cobargv[3]);

  fprintf(stderr, "COBOL returned return-code = %ld, error-val = %d\n", crc,
error_val);
  fprintf(stderr, "Exit = %ld\n", isCobolTidy());
}
```

In this example, the called COBOL program is defined as follows:

```
id division.
program-id. testprog.
data division.
linkage section.
01 error-val        pic s9(8) comp-5.
01 db-service       pic x(16).
01 username        pic x(16).
01 passwd         pic x(16).
procedure division using error-val, db-service, username, passwd.
  display "Hello from COBOL".
  display "Received ERROR-VAL : " error-val.
  display "Received DB-SERVICE: " db-service.
  display "Received USERNAME : " username.
  display "Received PASSWD   : " passwd.
  move 87654321 to error-val.
  display "MOVE " error-val " TO ERROR-VAL".
  goback.
```

Chapter 2

isCOBOL and Java

Calling Java programs or methods

This chapter describes how a Java program can call a COBOL program using isCOBOL.

A COBOL program can call a Java program using the CALL statement as long as the Java program implemenents the *com.iscobol.rts.IscobolCall* interface and exposes the following public method:

```
public abstract java.lang.Object call(java.lang.Object[])
```

Sample programs for this task are installed with isCOBOL in the subfolder *sample/is-java/iscobol-call-java*.

A COBOL program can invoke any Java method using object oriented syntax or the INVOKE statement.

Sample programs for this task are installed with isCOBOL in the subfolder *sample/is-java/iscobol-call-java-object*.

Calling isCOBOL from Java

Introduction

This chapter describes how a Java program can call a COBOL program using isCOBOL.

Programs compiled with isCOBOL can easily be called by Java programs through specific bridge classes.

Sample programs for this task are installed with isCOBOL in the subfolder *sample/is-java/java-call-iscobol*.

There are basically two approaches for calling a COBOL program from Java:

- rely on bridge classes generated by the The EasyLinkage feature

or

- invoke The com.iscobol.java.IsCobol Class directly.

The first approach is preferable because you don't have to take care of defining parameters for the COBOL program.

The EasyLinkage feature

The isCOBOL Compiler can generate bridge classes that a Java program can use to easily call your COBOL programs. These bridge classes include an object for each Linkage Section data item; such object allows to set and inquire the data item value. The bridge class provides a method named *run()* that allows to call the COBOL program.

Note that only standard COBOL programs identified by a PROGRAM-ID can be called through EasyLinkage. It's not possible to call COBOL objects that are identified by a CLASS-ID.

In order to make the Compiler generate bridge classes, the EasyLinkage feature must be activated through the following configuration setting:

```
iscobol.compiler.easylinkage=1
```

See Properties for the EasyLinkage feature for all the configuration settings that affect the EasyLinkage feature.

EasyLinkage and its settings can also be set directly in the source code through the SET Directive.

The generated bridge class is produced according to the current compiler options. For example, if the *-jj* option is used on the command line, then the Java source of the bridge program is left on disc. The only option ignored is *-od*; the bridge class is always generated in the "easylinkage" subfolder in the directory indicated by the setting of iscobol.compiler.generate.root_dir whose default is the same directory as the COBOL source file.

Let's analyze a small practical example.

Consider the following COBOL program:

```
       program-id. prog1.
       linkage section.
       77 p1 pic 9.
       procedure division using p1.
       main.
           add 1 to p1.
           goback.
```

The EasyLinkage feature, run with default settings, generates a bridge class named linkPROG1 that includes an object named p1.

The following Java program calls PROG1 through the bridge class by passing the p1 parameter set to 1 and expecting it set to 2 when PROG1 returns.

```
public class test {
   public static void main (String[] args) throws Exception {
      //create an instance of linkPROG1
      linkPROG1 prog1 = new linkPROG1();
      //set the p1 parameter to 1
      prog1.p1.set(1);
      //do the call
      prog1.run();
      //check if p1 was incremented by 1
      if (prog1.p1.toint() == 2) System.out.print("OK");
   }
}
```

Note that the COBOL program uses the GOBACK statement in order to return to the calling Java program. If STOP RUN is used instead the whole JVM terminates.

If the COBOL program includes entry points, the Java program can call each entry point as a separate function. However, it's important to call the main program first, like you would do with a caller COBOL program, otherwise the entry points are not available.

Additional usage (Java replacement for called functions)

The EasyLinkage feature is able to generate a stub class for every CALL statement found in the compiled program. This alternative usage is useful if you plan to provide a Java replacement or a Java implementation for some called functions. Two scenarios where this feature may be useful are:

- replacement of C functions with equivalent Java functions in order to have a pure Java application,
- implementation of a library routine that is currently missing in isCOBOL.

In order to make the Compiler generate stub classes, the EasyLinkage feature must be activated through the following configuration setting:

```
iscobol.compiler.easylinkage=2
```

See Properties for the EasyLinkage feature for all the configuration settings that affect the EasyLinkage feature.

EasyLinkage and its settings can also be set directly in the source code through the SET Directive.

The *-od* compiler option is ignored; the bridge class is always generated in the "easylinkage" subfolder in the directory pointed by iscobol.compiler.generate.root_dir whose default is the same directory as the COBOL source file.

Let's analyze a small practical example.

Consider the following COBOL program:

```
       program-id. showtempdir.

       working-storage section.
       77 buflen   pic s9(4) comp-5.
       77 pathname pic x(128).

       procedure division.
       main.
           set buflen to size of pathname.
           call "GetTempPathA" using by value     buflen
                                     by reference pathname.
           display pathname.
           goback.
```

The program calls the GetTempPathA Windows API in order to retrieve the path of the system's temporary directory. This program will work only on Windows systems and requires the isCOBOL interface to C (the dyncall library) in order to work.

By compiling the program with EasyLinkage enabled, you obtain the following Java source named GETTEMPPATHA.java, with the following content:

```
import com.iscobol.java.IsCobol;
import com.iscobol.java.StopRunAsException;
import com.iscobol.rts.IscobolRuntimeException;
import com.iscobol.rts.Factory;
import com.iscobol.rts.IscobolCall;
import com.iscobol.types.*;

public class GETTEMPPATHA implements IscobolCall {

   // Linkage variable declarations

   public byte returnCode$0[];
   public    com.iscobol.types.NumericVar returnCode;
   public    com.iscobol.types.NumericVar returnUnsigned;
   public byte transactionStatus$0[];
   public    com.iscobol.types.PicX transactionStatus;
   public java.lang.Throwable exceptionObject;

   //  variable declaration BUFLEN
   public byte buflen$0[];
   public    com.iscobol.types.NumericVar buflen;
   //  variable declaration PATHNAME
   public byte pathname$0[];
   public    com.iscobol.types.PicX pathname;

    {
      returnCode$0=Factory.getMem(8);
      returnCode=Factory.getVarBinary(returnCode$0,0,8,false,null, null,null,"RETURN-
CODE",false,18,0,true,false,false);
    }
    {
            returnUnsigned=Factory.getVarBinary(returnCode,0,8,false,null, null,null,"
RETURN-UNSIGNED",false,18,0,false,false,false);
    }
    {
      transactionStatus$0=Factory.getMem(2);
      transactionStatus=Factory.getVarAlphanumPrv(transactionStatus$0,0,2,false,null,
null,null,"TRANSACTION-STATUS",false,false);
    }
    {
          }
    {
      buflen$0=Factory.getMem(2);
      buflen=Factory.getVarNativeBinary(buflen$0,0,2,false,null, null,null,"BUFLEN",fa
lse,4,0,true,false,false,false);
    }
    {
      pathname$0=Factory.getMem(128);
      pathname=Factory.getVarAlphanum(pathname$0,0,128,false,null, null,null,"PATHNAME
",false,false);
    }

  @Override
   public void perform(int arg0, int arg1) {
   }
```

```
   @Override

   public Object call(Object[] argv) {
      final int argl=(argv==null)?0:argv.length;

      switch (argl) {
      default:
      case 2: pathname.link((CobolVar)argv[1]);
      case 1: buflen.link((CobolVar)argv[0]);
      case 0: break;
      }

/*  Write here the routine logic
 *        This is the call prototype:
*
 *   try {
 *      returnCode.set(Factory.call("_new_GETTEMPPATHA", null,
 *                                  new CobolVar[] {pathname.byRef(),
 *                                                  buflen.byVal()}));
 *   }
 *   catch (CallOverflowException ex) {
 *      throw new WrapperException(ex);
 *   }
 *
/
      return returnCode;
   }

   @Override
   @SuppressWarnings("deprecation")
   public void finalize() {
   }

}
```

You can add the necessary Java code to retrieve the temporary directory, e.g. replace the comment

```
/*  Write here the routine logic
 *        This is the call prototype:
*
 *   try {
 *      returnCode.set(Factory.call("_new_GETTEMPPATHA", null,
 *                                  new CobolVar[] {pathname.byRef(),
 *                                                  buflen.byVal()}));
 *   }
 *   catch (CallOverflowException ex) {
 *      throw new WrapperException(ex);
 *   }
 *
/
```

with

```
pathname.set(System.getProperty("java.io.tmpdir"));
```

After compiling GETTEMPPATHA.java, you have a replacement for the GetTempPathA function.

From now on, the COBOL program will use the Java replacement instead of the original Windows API. In this way the program can work on every operating system and doesn't require C libraries anymore.

EasyLikage uses the following rules to generate stub classes:

1. The name of the generated class will be the upper-case conversion of the name of the called function.
2. If the name is a data item rather than a constant string, then the data item name is used.
3. If the same function is called multiple times with different parameters, a separate stub for each different call is generated, along with a main stub that takes care of invoking the correct class depending on the parameters.
4. If the stub class already exists, it's not generated again, in order to avoid accidental overwriting of code added by the user.

Generating both kinds of class

In order to generate both bridge classes and stub classes described above, use the following configuration setting

```
iscobol.compiler.easylinkage=3
```

The com.iscobol.java.IsCobol Class

The ability to call a COBOL program from a Java program is provided by the com.iscobol.java.IsCobol class that is stored in the iscobol.jar library. Along with the com.iscobol.java.IsCobol class Veryant provides the com.iscobol.java.CobolVarHelper class, that allows to easily define COBOL data items to be passed to the called program

Let's analyze a small practical example.

Consider the following COBOL program:

```
       program-id. prog1.
       linkage section.
       77 p1 pic 9.
       procedure division using p1.
       main.
           add 1 to p1.
           goback.
```

The following Java program can call the above COBOL program:

```
import com.iscobol.java.*;
import com.iscobol.rts.ICobolVar;
public class test {
   public static void main (String[] args) {
      //
create an instance of CobolVarHelper and use it to generate the variable: 77 P1 PIC 9(
1)
      CobolVarHelper cbh = new CobolVarHelper("FOO", CobolVarHelper._DCA).pic9("P1", 1
, 0);
      ICobolVar p1 = cbh.get("P1");
      //set the P1 to 1
      p1.set(1);
      //to the call
      IsCobol.call("PROG1", new Object[] { p1 });
      //check if the variable was incremented by 1
      if (p1.toint() == 2) System.out.print("OK");
   }
}
```

Consult the javadoc installed with isCOBOL in the folder $ISCOBOL_HOME/javadoc for the full reference of the IsCobol and CobolVarHelper classes.

The com.iscobol.rts.IscobolRuntimeException and com.iscobol.java.StopRunAsException Classes

The IscobolRuntimeException and com.iscobol.java.StopRunAsException classes are part of the com.iscobol.rts package, that is included in the Framework libraries. It provides the COBOL error description.

Consult the javadoc installed with isCOBOL in the *javadoc* subfolder for a reference of the available methods.

Mixing Java dialogs and COBOL windows in the same application

In the previous chapters we described two possible scenarios:

- COBOL programs calling Java programs, and
- Java programs calling COBOL programs

In both scenarios it can happen that both the COBOL program and the Java program display a dialog in order to interact with the user.

The coexistence of Java dialogs and COBOL windows is possible as long as these simple rules are respected:

When calling Java from COBOL:

- Before opening a modal JDialog, the isCOBOL key buffering should be disabled. It can be done by the static method *com.iscobol.gui.client.KeyboardBuffer.setBufferOff()*. After the JDialog is closed, enable again the key buffering with the static method *com.iscobol.gui.client.KeyboardBuffer.setBufferOn()*. These methods could be invoked either by the Java program or by the COBOL program.
- Before opening any JDialog, the COBOL program should retrieve the current active window (a *java.awt.Window* instance) to pass it as 'owner' of the JDialog. In this way, with the Alt+Tab key combination you will see only a window as the Java dialog is child of the COBOL window. If this optional step is omitted, then the Alt+Tab key combination will show the Java dialog and the COBOL window as two separate windows.
- In Thin Client environment, the COBOL program should call another COBOL program via CALL CLIENT. The called program resides client-side along with the Java program and it does the necessary operations to invoke the Java program that shows its dialog as described above.

When calling COBOL from Java:

- The call must not be executed in the AWT-Event-Thread.
- The COBOL program should not display a INITIAL/STANDARD window, it should display only INDEPENDENT or FLOATING windows.

Sample programs for this task are installed with isCOBOL in the subfolder *sample/is-java/mixed-gui*.

Converting Java Source Code to Object-oriented COBOL

Introduction

The Java world offers many objects that can be integrated into your application enhancing it with additional features.

Sample Java code snippets and/or programs are provided as samples for most of these objects.

There are two ways to take advantage of these objects in a COBOL application:

1. Implement the functionality using the Java programming language and expose it to COBOL as one or more callable subprograms.
2. Implement the functionality directly in COBOL using OOP (Object Oriented Programming) syntax.

This article describes the second option providing suggestions for converting a Java source code example into the equivalent OOP COBOL code.

The COBOL code is usually easier to maintain because:

1. It is easier to read and maintain for a COBOL programmer
2. It can be all debugged using the isCOBOL Debugger (i.e. no Java source debugging required)

To convert a Java code example to the equivalent COBOL code, we'll demonstrate how to read the content of a web page.

Here is a Java example that does the job:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.URL;
import java.net.URLConnection;

public class webcontent {
   public static void main(String[] args) {
   try {
       URL url = new URL("https://www.veryant.com");
       URLConnection uc = url.openConnection();
       BufferedReader bf =
       new BufferedReader(
       new InputStreamReader(uc.getInputStream()));
       String inputLine;
       while ((inputLine = bf.readLine()) != null) {
          System.out.println(inputLine);
       }
       bf.close();
     } catch (Exception e) {
       e.printStackTrace();
     }
   }
}
```

Classes

As a first step in converting the example Java program to COBOL, gather the classes used.

Java source usually begins with a list of "import" statements that specify the classes used by the program.

The syntax in Java is:

```
import [packagename.]classname;
```

When writing COBOL, necessary classes must be listed in the REPOSITORY paragraph, in the CONFIGURATION SECTION.

The syntax for COBOL is:

```
class logical-class-name as "[packagename.]classname"
```

Where:

- logical-class-name is any valid COBOL name of your choice. It is used to reference the class in the COBOL code.
- [packagename.]classname is the full name of the class, the same name used in the "import" statement in the Java code.

For example, to convert the following Java statement to COBOL:

```
import java.io.BufferedReader;
```

Specify the class in the REPOSITORY paragraph as follows:

```
CONFIGURATION SECTION.
REPOSITORY.
  class jBufferedReader  as "java.io.BufferedReader"
  .
```

In Java programs, classes of the package “java.lang” are always available, and therefore do not need to be imported; but COBOL programs MUST explicitly import these packages . For example,

```
class jSystem            as "java.lang.System"
```

Java offers a shortcut to import all the classes of a package with a single import statement.

The Java syntax is:

```
import packagename.*
```

COBOL does not have an equivalent syntax, requiring instead that each class be declared in the REPOSITORY paragraph.

The Java program “import” statements below:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.URL;
import java.net.URLConnection;
```

Are translated to the following COBOL declarations:

```
CONFIGURATION SECTION.
REPOSITORY.
   class jBufferedReader  as "java.io.BufferedReader"
   class jInputStreamReader as "java.io.InputStreamReader"
   class jURL     as "java.net.URL"
   class jURLConnection   as "java.net.URLConnection"
   class jString    as "java.lang.String"
   class jSystem    as "java.lang.System"
   .
```

NOTE: Multiple class declarations in the REPOSITORY paragraph are separated by newline characters, not by periods. Ending the paragraph with a period on a line by itself allows additional class declarations to be inserted.

Once the classes are declared, the COBOL procedure division code must be written following the Java source example.

Java OOP statements

Java OOP statements can be divided into two groups:

- Object Creation statements
- Method Invocation statements

Object Creation

The syntax of a standard Java object creation statement is:

```
[ObjectType] ObjectName = new classname [(parameters)];
```

The equivalent COBOL syntax is:

```
SET ObjectName TO logical-class-name:>new [(parameters)]
```

Where:

- ObjectName is the name of the instance created by the “new” method. In Java programs, the ObjectName is implicitly defined, while COBOL programs must define it in the WORKING-STORAGE SECTION as follows:

```
77 ObjectName OBJECT REFERENCE logical-class-name.
```

- logical-class-name is the logical name specified in the REPOSITORY paragraph.

For example, the statement in Java:

```
BufferedReader bf = new BufferedReader(new InputStreamReader(uc.getInputStream()));
```

is translated to COBOL as:

```
CONFIGURATION SECTION.
REPOSITORY.
   class jBufferedReader  as "java.io.BufferedReader".

WORKING-STORAGE SECTION.
77  bf   object reference jBufferedReader.

PROCEDURE DIVISION.
set bf to
    jBufferedReader:>new(jInputStreamReader:>new(uc:>getInputStream())).
```

Method Invocation

The syntax of a standard Java object method invocation is:

```
[result] = ObjectName.method [(parameters)];
```

The equivalent COBOL syntax is:

```
[SET result TO] ObjectName:>method [(parameters)]
```

Where:

- ObjectName can be:

    a. the name of a data-item that contains the object reference, if we’ve created a new instance of the object (e.g. as shown above with java.io.BufferedReader)

b. the logical-class-name as defined in the REPOSITORY, if we're invoking a static method (e.g. methods in the class "java.lang.String")

- result can be any COBOL data-item that is of suitable type to receive the method's return value.

NOTE: While ObjectName is not case sensitive because it is COBOL data-item, the name of the method is case sensitive because Java language is case sensitive.

The following Java syntax:

```
[ObjectType] resultingObjectName = ObjectName.method [(parameters)];
```

is translated to COBOL as:

```
SET resultingObjectName TO ObjectName:>method [(parameters)] [AS ObjectType]
```

- [AS ObjectType] is not always necessary. In most cases the isCOBOL Compiler automatically performs the proper type cast while compiling the OOP statement.

Note: isCOBOL doesn't support *autoboxing*, you can't pass native data types where Object are required. For example, consider the following constructor:

```
myMethod (Object...)
```

With Java (starting from version 1.5) you're allowed to call the method in this way:

```
int i = 0;
long l = 2;
myMethod (i, l);
```

With isCOBOL you need to use java.lang objects instead:

```
[...]
configuration section.
repository.
 class JInt as "java.lang.Integer"
 class JLong as "java.lang.Long"
[...]
procedure division
[...]
myMethod (JInt:>new(0), JLong:>new(2));;
[...]
```

Error Handling

Java statements are usually bracketed in try/catch blocks, so the program can intercept and handle exception conditions. Some methods require a try/catch block and the compiler will return an error if one is not included. In any case, using try/catch is the preferred and recommended practice.

COBOL and Java have similar try/catch syntax.

The Java syntax is:

```
try {
        java statements
    } catch (Exception e) {
        e.printStackTrace();
    }
```

The COBOL syntax is:

```
try
   cobol statements
catch exception
   exception-object:>printStackTrace()
end-try
```

Java Program Converted to isCOBOL

Here is the example Java program used in this document after being translated to Object Oriented COBOL:

```
       PROGRAM-ID. webcontent.
       CONFIGURATION SECTION.
       REPOSITORY.
           class jBufferedReader    as "java.io.BufferedReader"
           class jInputStreamReader as "java.io.InputStreamReader"
           class jURL               as "java.net.URL"
           class jURLConnection     as "java.net.URLConnection"
           class jString            as "java.lang.String"
           class jSystem            as "java.lang.System"
           .
       WORKING-STORAGE SECTION.
       77  url       object reference jURL.
       77  uc        object reference jURLConnection.
       77  bf        object reference jBufferedReader.
       77  inputLine object reference jString.
       PROCEDURE DIVISION.
       main.
           try
             set url to jURL:>new ("https://www.veryant.com")
             set uc  to url:>openConnection()
             set bf  to jBufferedReader:>new
                        (jInputStreamReader:>new(uc:>getInputStream()))
             perform until exit
                set inputLine to bf:>readLine()
                if  inputLine = null
                    exit perform
                end-if
                jSystem:>out:>println (inputLine)
             end-perform
           catch exception
             exception-object:>printStackTrace()
           end-try.
           GOBACK.
```

Known cases of Java syntax that is not directly translatable to COBOL

In this chapter are listed few cases of Java syntax that can't be directly translated to object oriented COBOL along with a equivalent syntax that can be used, if applicable.

The .class Syntax

In Java, if the type is available but there is no instance then it is possible to obtain a Class by appending ".class" to the name of the type. An equivalent short syntax is not available in isCOBOL. In order to obtain the Class of an object, you can rely on the forName() static method exposed by java.lang.Class.

The following Java example:

```
import java.io.File;
public class FilePackage {
    public static void main (String[] args){
        System.out.println(File.class.getPackage());
    }
}
```

is translated to isCOBOL as follows:

```
       program-id. FilePackage.
       configuration section.
       repository.
           class JClass  as "java.lang.Class"
           class JFile   as "java.io.File"
           class JSystem as "java.lang.System"
           .

       procedure division.
       main.
           try
              JSystem:>out:>println
                    (JClass:>forName("java.io.File"):>getPackage())
           catch exception
              exception-object:>printStackTrace()
           end-try.
           goback.
```

Because the syntax *JFile:>class* would not be compiled.

Note that, due to the use of the forName() method, a exception block is required.

Array of arrays

Java doesn't provide multidimensional arrays. The concept of a multidimensional array is implicitly translated to an array of arrays.

On the contrary COBOL has multidimensional arrays and does not support implicit arrays of arrays instead.

For the above reason, a Java code like this

```
public class arrays {
    public static void main (String[] args) {
        String[][] strarr = new String[3][2];
        String[] subarr = new String[2];
        subarr[0] = "OK";
        subarr[1] = "OK";
        strarr[0] = subarr;
        System.out.println (strarr[0][0]);
        System.out.println (strarr[0][1]);
    }
}
```

cannot be translated to COBOL as follows:

```
       program-id. arrays.
       configuration section.
       repository.
           class StringArr1 as "java.lang.String[]"
           class StringArr2 as "java.lang.String[][]"
           class JSystem   as "java.lang.System"
           .
       working-storage section.
       77 subarr object reference StringArr1.
       77 strarr object reference StringArr2.

       procedure division.
       main.
           set strarr to StringArr2:>new(3, 2).
           set subarr to StringArr1:>new(2).
           set subarr(0) to "OK".
           set subarr(1) to "OK".
           set strarr(0) to subarr.
           JSystem:>out:>println(strarr(0, 0)).
           JSystem:>out:>println(strarr(0, 1)).
           goback.
```

The correct translation is the following:

```
       program-id. arrays.
       configuration section.
       repository.
           class StringArr1 as "java.lang.String[]"
           class StringArr2 as "java.lang.String[][]"
           class JSystem   as "java.lang.System"
           .
       working-storage section.
       77 subarr object reference StringArr1.
       77 strarr object reference StringArr2.
       77 i      pic 9(3).

       procedure division.
       main.
           set strarr to StringArr2:>new(3, 2).
           set subarr to StringArr1:>new(2).
           set subarr(0) to "OK".
           set subarr(1) to "OK".
           perform varying i from 0 by 1 until i = subarr:>length()
              set strarr(0, i) to subarr(i)
           end-perform.
           JSystem:>out:>println(strarr(0, 0)).
           JSystem:>out:>println(strarr(0, 1)).
           goback.
```

Generic classes and type-variables

A generic class is defined with the following format:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the type parameters (also called type variables) T1, T2, ..., and Tn.

This kind of syntax has no equivalent in COBOL, so you don't have other choice but to discard it.

The following example

```
import java.util.ArrayList;

public class ALtest {
    public static void main (String[] args) {
        ArrayList<String> al = new ArrayList<String>();
        al.add(new String("xxx"));
    }
}
```

is translated to isCOBOL as follows:

```
       program-id. ALtest.

       configuration section.
       repository.
           class ArrayList as "java.util.ArrayList"
           class JString   as "java.lang.String"
           .

       working-storage section.
       77 al object reference ArrayList.

       procedure division.
       main.
           set al to ArrayList:>new().
           al:>add(JString:>new("xxx")).
           goback.
```

Note that it will generate the following warnings at compile time:

```
Note: ALTEST.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

The instanceof operator

The instanceof keyword is a binary operator used to test if an object (instance) is a subtype of a given type.

There is no equivalent statement in COBOL, but you can rely on the isAssignableFrom() method exposed by java.lang.Class.

The following Java source

```
import java.io.OutputStream;
import java.io.FileOutputStream;

public class InstOfTest {
    public static void main (String[] args) {
        try {
            FileOutputStream f = new FileOutputStream("/tmp/foo");
            if (f instanceof OutputStream) {
                System.out.println("The f object is an OutputStream");
            }
        } catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

can be translated to

```
       program-id. InstOfTest.

       configuration section.
       repository.
           class OutputStream      as "java.io.OutputStream"
           class FileOutputStream  as "java.io.FileOutputStream"
           class JClass            as "java.lang.Class"
           class JSystem           as "java.lang.System"
           .

       working-storage section.
       77 f object reference FileOutputStream.

       procedure division.
       main.
           try
              set f to FileOutputStream:>new("/tmp/foo")
              if JClass:>forName("java.io.OutputStream")
                       :>isAssignableFrom (f:>getClass())
                 JSystem:>out:>println
                     ("The f object is an OutputStream")
              end-if
           catch exception
              exception-object:>printStackTrace()
           end-try.
           goback.
```

Chapter 3

isCOBOL and Tuxedo

Working With Oracle Tuxedo

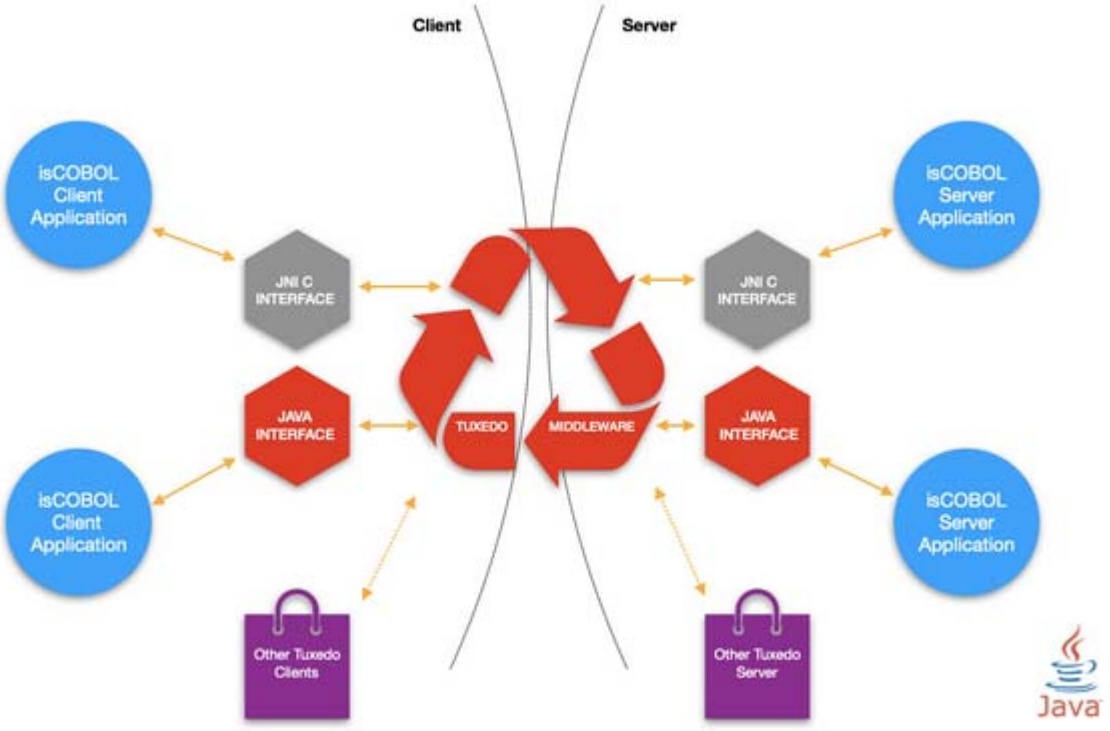
Tuxedo (Transactions for Unix, Extended for Distributed Operations) is a middleware platform used to manage distributed transaction processing in distributed computing environments. It distributes applications across multiple platforms, databases, and operating systems using message-based communications and distributed transaction processing.

isCOBOL developers using the Oracle Tuxedo platform for distributed transaction processing and message-based application development can create Tuxedo clients and Tuxedo services from COBOL applications. The isCOBOL 2020R1 SDK is certified for Oracle Tuxedo 12c on all supported platforms.

isCOBOL and Tuxedo can work together in a distributed processing (client/server) environment providing two flavors of execution:

- Legacy COBOL approach based on calling C routines provided from Tuxedo
- Java approach based on OOP java methods to be used to create clients and services.

In a distributed processing environment, the interaction occurs as depicted in the following diagram:



In this environment, isCOBOL interoperates with Tuxedo using the same routines used to call C programs (COBOL-calling-C routines on the client and C-calling-COBOL routines on the server) or using a pure Java approach based on Oracle Tuxedo Java Programming guide that can take advantage of a traditional Tuxedo multithread model.

In Tuxedo client/server distributed applications, the client requests the services of a server, which may have multiple services to provide, and the outcome of the request is returned to the client by the service. This architecture allows you to distribute application processing among multiple machines to optimize performance.

Tuxedo Server Program

A Tuxedo server is a process that provides one or more services to a client. To build server processes, applications combine their service subroutines with a controlling program provided by the Tuxedo system.

Using legacy COBOL approach

To create a Tuxedo server with the legacy COBOL approach, follow these steps:

1. create and compile a COBOL program that performs a specific task or service,
2. create a configuration file for the service to establish the identifiers,
3. link the isCOBOL runtime libraries into the controlling program provided by the Tuxedo system.

Contact Veryant for more details about this process.

Using Tuxedo Java server (TMJAVASVR)

To create a Tuxedo server under TMJAVASVR, you should write a CLASS-ID program, that must respect these conditions:

- It has to inherit the TuxedoJavaServer class and have a default constructor.
- Methods, which will be advertised as Java services, have to take the TPSVCINFO interface as the only input argument and should be declared to public.
- It has to implement tpsvrinit() method, which will be called when Tuxedo Java server starts up.
- It has to implement tpsvrdone() method, which will be called when Tuxedo Java server shuts down.
- The service could use Tuxedo Java ATMI (e.g, tpcall, tpbegin, etc).
- The service could return result to client by using tpreturn and exit by throwing exception.

In this environment it's possible to debug the CLASS-ID via remote debugging.

Contact Veryant for more details.

Tuxedo Client Program

A Tuxedo client is a software module that forwards a user request to a server that offers the requested service.

Using legacy COBOL approach

To create a Tuxedo client with the legacy COBOL approach, you must relink the isCOBOL native libraries and Java JNI libraries with the Tuxedo libraries.

1. Add the necessary Tuxedo calls to your COBOL program. These calls–used to open and close resources, begin and end transactions, and support communication between clients and servers–are collected in a Tuxedo API known as the Application to Transaction Monitor Interface (ATMI). ATMI functions are described in the Tuxedo documentation,
2. run *buildclient* to generate the client executable,
3. set client environment variables.

Contact Veryant for examples and more details about this process.

Chapter 4

isCOBOL and JasperReports

Interoperability with JasperReports

isCOBOL can communicate with the JasperReports runtime in order to populate, preview and print reports.

Required software

The following software (not provided by Veryant) is required:

- **Jaspersoft Studio:** editing software for JasperReports. It will help you design and run report templates; build report queries; write complex expressions; layout visual components like over 50 types of charts, maps, tables, crosstabs, and custom visualizations; and much more. Setups available at https://community.jaspersoft.com/project/jaspersoft-studio/releases.
- **JasperReports Library:** the world's most popular open source business intelligence and reporting engine. It is entirely written in Java and it is able to use data coming from any kind of data source and produce pixel-perfect documents that can be viewed, printed or exported in a variety of document formats including HTML, PDF, Excel, OpenOffice and Word.
  The isCOBOL examples discussed below are certified for version 6.2. You can download JasperReports Library 6.2 at https://sourceforge.net/projects/jasperreports/files/jasperreports/JasperReports%206.2.0. Look for a file whose name is like "jasperreports-6.2.0-project.zip".

Examples

Along with isCOBOL Evolve and isCOBOL SDK are installed some examples that print reports via the JasperReports library after retrieving data from different data sources.

Sample	Description
CSV	Prints a report with the data read from a CSV file.
c-treeSQL	Prints a report with the data read from a c-treeSQL database via JDBC. The same logic is applicable also with other JDBC-compliant databases.
JSON	Prints a report with the data read from a JSON file.
XML	Prints a report with the data read from a XML file.

In isCOBOL Evolve, these samples are installed under the folder ide/sample/jasper in the form of projects that you can import in your workspace.

In isCOBOL SDK, these samples are installed under the folder sample/jasper in the form or programs that you can compile and run from the command line.

Follow the instructions of the README file of each example in order to test it.