Supercharge IMS Business Applications with Java

June 2021 Edition

Table of Contents

Figures

Preface

Java™ was introduced to the IBM® Z® and to IMS™ more than ten years ago. The idea of taking advantage of the Java eco-system to modernize, extend, and maximize the values of existing IMS business applications and assets has been tested, measured, implemented, and advocated. Much has been written about mainframe modernization, with new tools and new capabilities coming out on the market that facilitate the adoption of Java and integration of existing IBM Z assets with new business applications. This book is written with one focus-- to share the insights in the entire integration and adoption process based upon the authors' experience working on various enterprise application integration and modernization projects. From architecture decision making, infrastructure setup, application development, Java interoperability with COBOL, to problem determination, the information will help IT architects and application developers determine their integration approach and strategy, prepare for Java adoption, and supercharge their IMS business applications with Java!

Special thanks go to Fiducia & GAD IT AG, for sharing their experience with us!

Change history

Edition	Major changes
May, 2021	• A case study chapter based on a recent modernization project in Fiducia & GAD IT AG is added. From functional requirements, infrastructure landscape, environment settings, application development, to security considerations, this chapter provides great insight into the thought process, with information that you will find helpful with your own modernization project. • Content related to ESAF connection pooling and JCC statement caching support for Db2 access is added to Chapter 6. Related APARs to support these functions are added to the list of known problems and solutions in Chapter 7.

C H A P T E R   1

1. Introduction

Extending the reach of existing IMS™ applications through interoperation with Java™ technology allows you to take advantage of the Java eco-system within traditional business language containers. New applications can be created with faster time to market, and organizations can expand and maximize the values of existing IMS applications and business assets with the help of various tools and frameworks that come with Java. This book will focus on the essential tasks that help bridge between business languages and Java.

In this introduction chapter, a common understanding for Java on IBM® Z® is established through the discussions of the motivation and the business values that Java™ brings to your mainframe environment, especially to your IMS Transaction Manager.

The following topics are discussed:

- The motivations and business values to bring in Java to your environment
- Challenges and prerequisites
- Java overview and reasoning
- Options and associated cost

1.1. Aligning IT strategies for the future

Many mainframe shops experimented with Java™ on IBM Z. In most cases, the results were very promising, leading to a strategic adoption of the Java runtime on IBM Z Systems. Others left Java workloads to distributed systems, often because Java was seen as being too slow, too resource consuming, or too 'distributed.' Even though Java is over twenty years old, mature, and well proven, for some of us, it's still new and requires a major change in how we think.

Some years ago, experts had been very skeptical when enterprises started transforming from assembler to higher level languages such as COBOL or PL/I. No one questions today if this move was the right thing to do. We cannot imagine an assembler-only landscape today, and if we did, it would seem like a nightmare. Are we at the same paradigm shift today when we discuss enabling Java on z/OS®, for batch, online, or within classic transaction monitors like IMS™ or CICS®?

Technological progress triggered the shift to higher level languages in the 80's–more powerful servers, mature run times, and optimizing compilers.

Infrastructure enhancements have opened the door. However, no one migrated to a high-level language with the expectation of a better throughput (or less resource consumption). The move was driven by a need for higher application programmer productivity. In other words, these businesses asked for a faster time to market for their ideas and applications.

Today, response time and resource consumption are still key elements we need to focus on when we align IT strategy for the future. Some non-functional aspects we will touch upon in this document are:

- Programmer efficiency, time to market, and code reuse
- Resource consumption, performance, scalability, and response times
- Transactions and compensation
- Security
- Best fit for applications

1.2. Java on IBM Z primer

To understand the z/OS implications of Java, we must touch on at least three relevant aspects:

- The programming language
- The compilation process
- The Java runtime itself

1.2.1. IBM SDK for z/OS, Java Technology edition

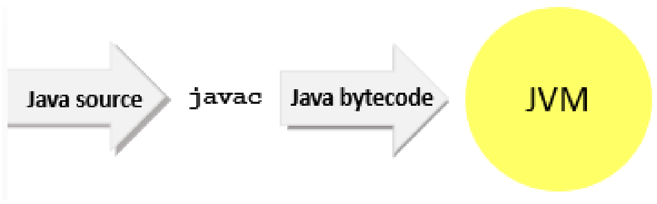
Java is a general purpose programming language that offers a simple class-based, object-oriented and concurrent paradigm. Its syntax is similar to C++, and it offers many developer-friendly features such as auto-boxing, generics, and full stack traces on exceptions as part of its language specification.

The **Java Software Development Kit (SDK)** is the deliverable (a UNIX™ `.pax.Z` file or SMP/E package) that includes all components required to execute Java code on the platform.

Major components of the SDK are as follows:

- **javac**: the Java compiler that converts source code into Java bytecodes.

Figure 1. Java compile



- **Java class libraries**: The libraries that provide the implementations of the extensive Java APIs that are part of the language specification.
- **Java Virtual Machine (JVM) and Java Runtime Environment (JRE)**: JVMs, Garbage Collector (GC), and Just in Time Compiler (JIT) technologies to execute the Java bytecodes on a target platform.
  On z/OS, JVMs are available natively in UNIX System Services. They can run as started tasks, in batch jobs, or embedded in subsystems, and therefore the application code is very portable between this most common and certified standard. Execution actually takes place in JRE. IBM’s JVM implementation is based on IBM’s J9 technology, and its SDK for Java is fully compliant to Java Certification Kit (JCK).
- Tools that support development, debugging, security monitoring, and more.

For z/OS, the IBM SDK for Java is the only Java SDK available. Starting from Java 8, the IBM SDK's compiler (javac) and Java class libraries are based on OpenJDK, the open source reference implementation of the Java platform.

Java source code needs to be compiled once to object code (bytecode compilation to .class files via ‘javac’) on any development platform. These .class files are typically packaged as part of .jar, .ear and .war modules. One can then distribute these object codes and modules to any target platform, and it would run there without further bytecode compilation in the JVM. Think of this for now as if a common instruction set would be available on any platform, or a virtual computer.

The internally used code page within the JVM is Unicode. As a result, Java programs are platform neutral and highly portable. Externals, like configuration files, are usually ASCII. One can configure the JVM to interact with the hosting platform with a code page of your choice (e.g. EBCDIC).

Important features responsible for the language's success are:

- IP communication capabilities are an integral part of the implementation and specification.
- Multithreading is used by the Java infrastructure itself (to parallelize housekeeping, for example) and is also supported for applications that execute in Java application servers, such as WebSphere® Application Server for z/OS.
- Support for graphical interfaces is an integral part of the Java language. Typically, the graphical interface that is generated for a server application would be a browser-based frontend, such as in Web 2.0 style.

For z/OS, there are some very useful functions built in to the IBM delivery (the SDK within the IBM Java for z/OS package):

- MVS CONSOLE interface
- SAF interface
- Workload management (WLM) APIs to classify and create WLM work units (for example, used by WebSphere Application Server to implement a state-of-the-art WLM support for Java transactions and batches)
- Many other z/OS-related functions that allow you to use Java for accessing a wide range of z/OS-specific functions and resources
- Access method generator for COBOL copybooks (available from IBM alphaWorks in the IBM Experimental Version of the JZOS Batch Toolkit community)
- Data access (VSAM, BSAM, QSAM, and HFS)

  Data Access Accelerators (DAA) is a set of accelerated Data Access APIs for native data manipulation. Because the Java specification does not support direct operations on native structures, costly conversion into Java objects is needed to achieve that goal. The IBM DAA API (com.ibm.dataaccess) automatically optimizes the data manipulation on DirectByteBuffers for the following conversions:
  - Integer to packed decimal (PD)
  - PD to integer
  - Long to PD
  - PD to long

  The DAA API also supports ByteArrays with the following features:
  - Many packed decimal operations
  - BigDecimal and BigInteger conversions from and to external decimal and Unicode decimal types
  - Marshalling operations: marshalling and unmarshalling primitive types (short, int, long, float, and double) to and from byte arrays
  - The com.ibm.jzos.fields package of the IBM JZOS Batch Toolkit for z/OS uses DAA by default for data marshalling and conversion

In addition, many z/OS subsystems (CICS and IMS, for example) deliver Java APIs to access subsystem-specific functions comparable to what is available in COBOL and PL/I.

1.2.2. Java object and native code compilation process

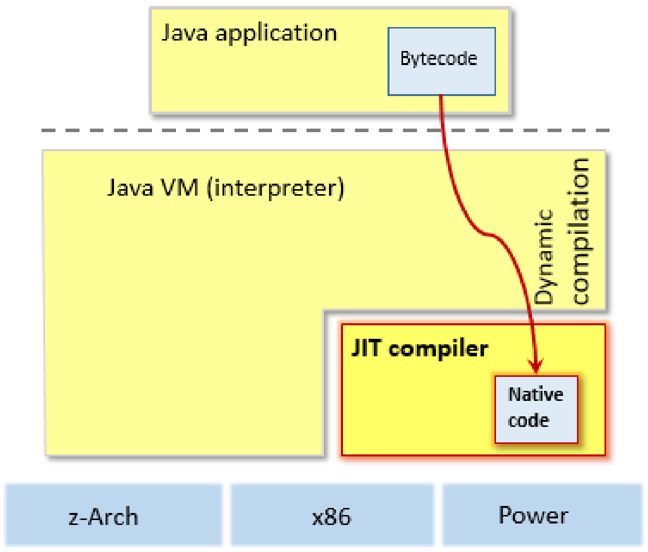
The Java compile process is a staged process and not really comparable to traditional compile-and-link-edit process:

1. First, the source code gets compiled to bytecode. If you adhere to standards, this code can be executed on any JVM on any platform. This is

what makes Java so special, as recompilation of the Java application is generally not required when deploying across platforms, and/or upgrades to newer levels of the SDKs.

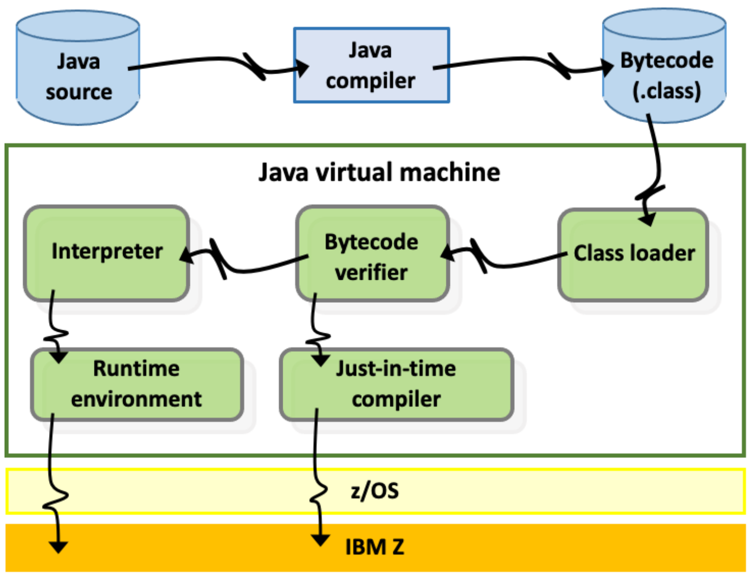
2. The bytecodes of a Java method are interpreted by the JVM at execution time, and is pretty slow compared to traditional compiled-and-link-edited code. To speed up the code execution, several features have been built into the z/OS JVM to optimize this code, manually or automagically, to bring performance up to bare metal speed.
3. While the bytecode is interpreted and executing, the JVM monitors the code execution. JVM's backend (the just-in-time compiler or JIT) optimizes and compiles bytecode of hot methods on the fly, down to optimized native instructions, exploiting any hardware and platform features available. The compilation target is adjusted to the application's access patterns and to the underlying hardware architecture. All this process is transparent and very efficient.

Figure 2. From interpretation to compilation



4. In addition, the JVM has the ability to cache away meta-data about classes and JIT compiled code. Subsequent JVM instances can benefit with significantly better startup performance by simply loading the cached code, instead of re-incurring the overhead of interpretation and compilation. AOT has to be done only once after initial program load (IPL). The compiled code is persistent for the LPAR lifetime until next IPL, and can be shared between multiple JVMs (Java class sharing). Optionally, in Java 8 SR1 or later, the contents of the cache can be persisted onto disk and reloaded after IPL. AOT can optimize applications and JVM code.

Figure 3. Java compilation, runtime, and JVM: the full picture



If you enable AOT, every JVM takes care not to AOT something that already has been AOT'd in the LPAR.

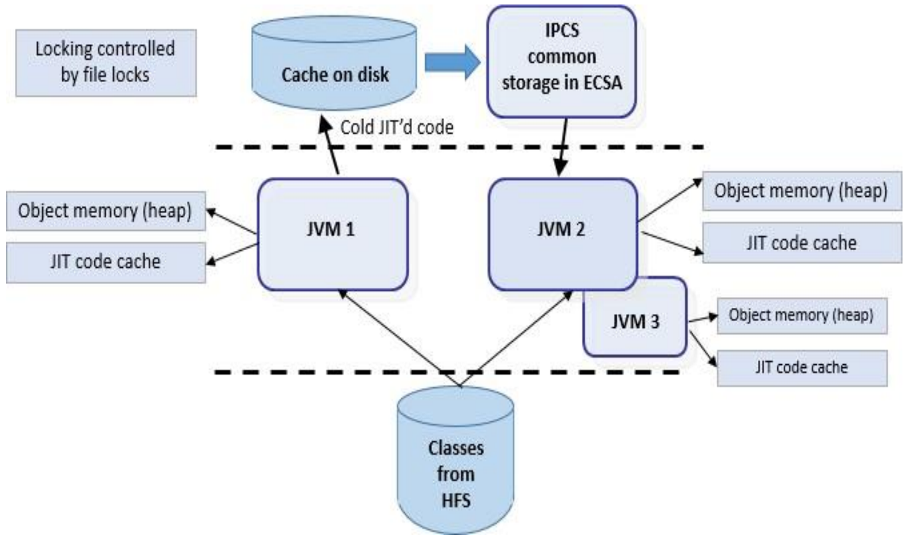
AOT can optimize applications and system classes and would need to be run separately for different applications or JVM versions. As a result, several versions of this code would have to be shared and cached.

<u>**Note**</u>: With shared classes technology, Java classes metadata can be shared across multiple JVM instances within an operating system image (LPAR/Guest). Given IMS Java applications are single-threaded, when incoming arrival rate exceeds the throughput that a single thread can manage, it is common to spin up additional JVM instances. Each of these additional JVMs within your LPAR can then benefit from the startup and footprint savings of these shared classes.

Compiled AOT code is stored within the z/OS UNIX System Services Interactive Problem Control System (IPCS) structures. An HFS file is used by the 'consuming' JVMs to find the location of the appropriate classes.

Figure 4 shows the relationship between those code caches.

Figure 4. Relationship and location of AOT and JIT code caches



Note: For class sharing, the JVM uses z/OS UNIX System Services (USS) resources defined in the parameters for the PARMLIB member BPXPRM*xx*: MAXSHAREPAGES, IPCSHMSPAGES, IPCSHMMPAGES and IPCSHMNSEGS.

The use of AOT speeds up startup and native (JIT) compilations, improves overall throughput, and minimizes CPU consumption and memory footprint.

To summarize, from interpreted to pre-compiled and shared up to highly optimized machine code, you can have various levels of automated code optimization. This hierarchy of compilation strategies is applicable to all IBM JVM implementations of the same release level (platform agnostic).

Note: JIT compilation is enabled by default. The use of AOT and class sharing is enabled by default when you run in a WebSphere Application Server. If you run your JVM native or in IMS, you can enable it using –Xshareclasses JVM options. AOT is complementary to JIT.

1.2.3. Java on IBM Z strength

To get the most out of the underlying platform for your Java applications, you would need to run your code in a JVM implementation that explicitly exploits your hardware architecture. On IBM Z, since there is only the IBM SDK to choose from, you just need to ensure that the SDK you deploy supports the hardware you use.

On IBM Z, especially on z/OS, we can enjoy some special benefits:

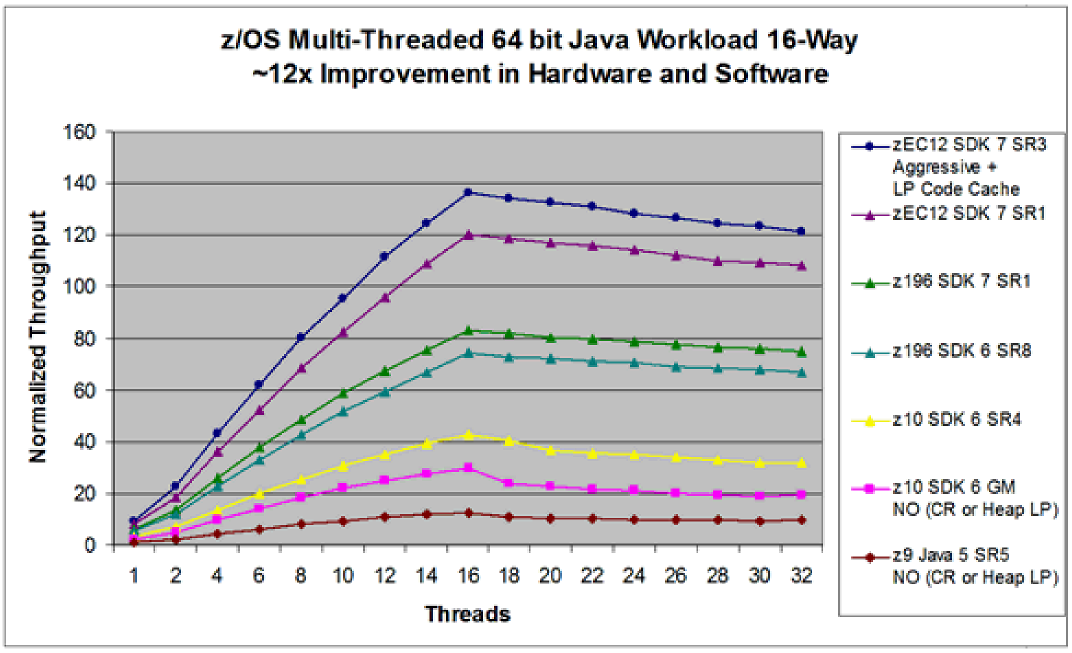
1. Pure Java application code and the JVM runtime itself are eligible to run on the IBM z Application Assist Processor (zAAP) or IBM z Integrated Information Processor (zIIP) engines. These engines offer an attractively priced execution environment for Java applications that is generally more cost effective than workloads running on general CPs.
2. On distributed systems, one would typically deploy a Java application in a Java Runtime Environment (JRE), such as a web application server, and interact with it using network protocols. On z/OS you could do the same. However, you could further integrate Java artifacts in almost any subsystem (IMS, CICS, Db2 for z/OS, or batch) and integrate those into your existing assets. You could decide on the integration depth (ranging from services- oriented component interaction to cross-memory integration). This architectural freedom lets you minimize risk, protect investments, and keep the existing service qualities delivered by the subsystems to your applications.
3. Automatic JIT compilation means that your program is always compiled to use all the available hardware features and instructions without the need to change and test the application code. This behavior is in sharp contrast to traditional languages such as COBOL, which must be manually recompiled to use new hardware facilities. At most sites it is very unlikely that all COBOL or PL/1 programs will be recompiled whenever a new machine is installed or upgraded. Compared to the third-generation languages, this paradigm shift allows much faster exploitation of the latest hardware and software, meaning that you automatically get the benefits of the throughput enhancements delivered by new hardware.

Note: JIT is smart enough to identify the underlying platform and compiles accordingly. One can force JIT to compile for a specific hardware, for example, when the hardware support is new or experimental.

4. JIT code is not persistent, so you do not need to care about code versioning if the same application is going to run on systems with different hardware levels.
5. The fact that the IBM Z Systems architecture and the JVM backend compiler are created by the same company leads to combined engineering of both. In fact, significant hardware features have been co-designed with the JVM teams specifically to accelerate Java performance on the platform. Such innovations typically apply to both z/OS and Linux on IBM Z Systems.
6. Significant performance enhancements have been made in each generation

of System IBM Z processors. A recent benchmark testing shows that the hardware improvement, coupled with enhancements in multi-threaded 64-bit Java, resulted in 12x more throughput from Java V5 on IBM System z9, to Java V7 on IBM System z EC12. This dramatic performance increase is almost linear from single-threaded Java up to 16 threads.

Figure 5. Scalability of Java on IBM IBM Z Systems

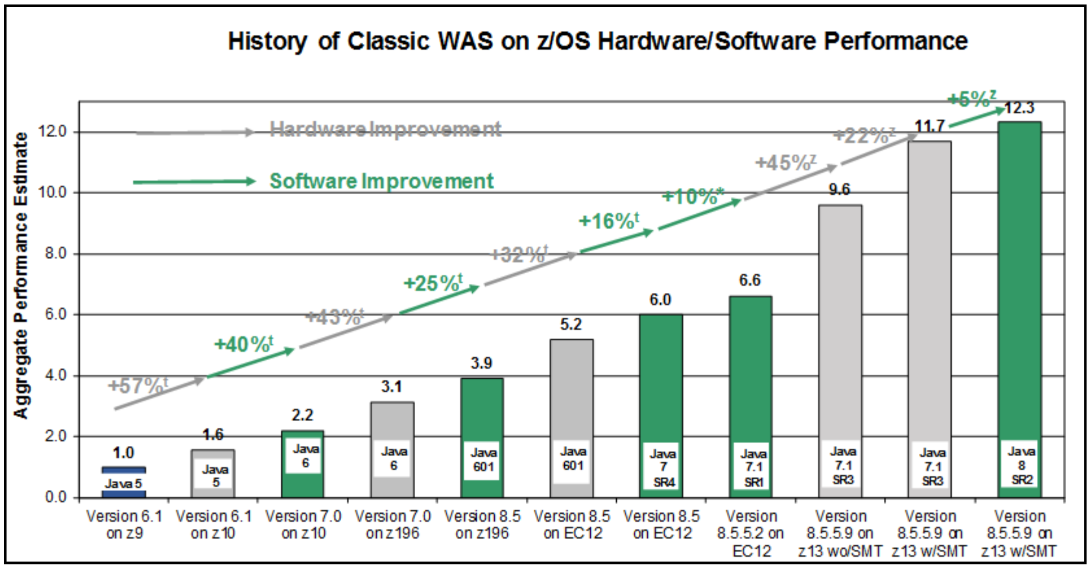


For example, zEC12 delivered superior throughput compared to the prior generation IBM System IBM Z, including:

- Up to 45 percent improvement for Java workloads,
- Up to 30 percent improvement for Db2 for z/OS operational analytics, and
- More than 25% improvement for CPU intensive applications

Another more recent WebSphere benchmark testing shows that IBM Z with Java 8 delivers more than 12 times aggregate software and hardware performance improvement over Java 5 on IBM z9.

Figure 6. Java performance improvements on IBM Z Systems



7. IBM Z Systems design principle features a combination of non-uniform memory access and large symmetrical multi-processor (NUMA/SMP) and large caches in a 4-level hierarchy, which are a great fit for memory intensive (Java) applications and mixed workloads.
8. On IBM Z Systems, IBM delivers special hardware support that is or can be leveraged to further improve the execution rate to enable enterprise class transaction rates and batch throughputs:

   a. Special hardware support in classical Z hardware, like z Systems Application AssistProcessor (zAAP), z Systems Integrated Information Processor (zIIP), instruction set, and class libraries to support special hardware functions (for example, symmetric cryptography). Note that on current z System hardware you would run zAAP workload on zIIP engines. That allows for better utilization of the hardware and lowers the cost of acquisition.
   b. Standard features such as large pages, hardware transactional memory, and hardware run-time instrumentation allow further optimized usage of the underlying hardware infrastructure.
      Runtime instrumentation is enabled automatically, and transactional memory can be exploited by middleware and applications.

Note: Large pages needs to be configured in the JVM (-Xlp) and also in z/OS SYS1.PARMLIB(IEASYSxx) LFAAREA and PAGESCM parameters. Large page requestor also needs access to RACF FACILITY profile IARRSM.LRGPAGES (or the SAF equivalent). Supporting hardware such as EC12 or higher and/or Flash Express) needs to be available for large pages. See the Configuring large page memory allocation topic in the IBM SDK, Java Technology Edition product information in IBM Knowledge Center.

Note: Your application might benefit from large page usage, but also important that large page usage takes stress from the systems Translation Lookaside Buffer (TLB), so the entire system will benefit from large page exploiters.

c. Additional accelerators for compression (zEDC), memory access (Flash Express), cryptography (Crypto Express), and IP communication over remote direct memory access (RDMA) over an Ethernet (RoCE) are supported by Java at delivery time!

d. Since IBM Z Systems z13 zIIP engines can run in symmetric multi-threading mode (SMT). The SMT mode will allow two threads to concurrently execute on the same core, thereby, delivering for more throughput at no additional cost. If anLPAR is running in SMT mode single thread performance might be degradedbut overall throughput enhanced. Given Java applications and the JVM are multi-threaded applications, they generally benefit from SMT enablement on zIIP engines. Care should be taken to assess impact on other zIIP eligible workloads that are sharing the zIIP engines.

e. SIMD, single instruction / multiple data HW support is kind of a vector engine (remember 3090 vector facility) supported by Java 8 and higher. The Java 8 JVM will transparently exploit SIMD instructions to accelerate various String APIs, crypto and loops where possible. If your workload can benefit from SIMD, it will experience a performance boost, again for no charge.

All these strengths make IBM Z the enterprise Java server with huge capacity, brilliant performance, and huge potential.

Java integration layer

The implementation of the crucial platform integration (such as compilers to native code) is left to the supplier of the SDK; in the case of IBM Z, it s IBM. On z/OS, embedded Java Runtime Environments (middleware like WebSphere, IMS, and CICS) allow very portable code to run in platform-optimized environments, delivering the highest service levels and integration, while remaining very portable. Integration and platform leverage takes place with adapter technology and application transparency enablement, all through the use of configuration options. Without this concept of platform independence, Java would be nowhere today!

Java allows you to move your code, without pain, from one platform to another. This puts pressure on the vendors to stay competitive. Also, this kind of code reusability allows one to move the business logic close to the data, or the data to the logic, to allow for higher service levels and easier management and by avoiding vendor lock-in.

The benefits that Java can provide should encourage rethinking of the classical multi-tier blueprints that evolved due to the limitations of both mainframe and distributed systems

1.3. Motivations and business values

Enabling Java within your existing mainframe application lifecycle componentry is a task that touches on all areas of your application lifecycle. It is an investment that needs some funding and sponsorship, even if many building blocks for Java enablement come for free.

What does your business get out of this? Java enablement in an existing mainframe business language (e.g. PL/I and COBOL) environment generates value in various areas of your application food chain:

Business

- **Reduce time to market.** Java technology enables many assets, tools and skills that drastically short your development cycles
- **Protect investment.** By adopting new technologies to existing assets, investments in business applications can be leveraged and continue to support your business as they did in the past. When needed, application components can be renewed and exchanged without any risk.
- **Facilitate integration of 3rd party products.** Ready-made Java software from 3rd parties could run instantly and be integrated into your existing business application, allowing you to select standard packages when needed.

Application architecture

- **Open the toolbox for many frameworks and protocols**.
- **Support services naturally.** Service enablement, including micro services and RESTful services are more natural with Java.
- **Eliminate additional compensation layers.** Use of infrastructure transaction services means there is no need to implement additional compensation layers into the application, allowing significant enhancements to surface throughout the entire application lifecycle.
- **Enable an architecture based on mainstream componentry and programming models.** Current and widely used programming models such as JDBC, Java persistence frameworks, SPARK, and Java adapter technologies can be utilized and taken advantage of.
- **Gain competitiveness.** Java enrichment in your environment enables you to leverage upcoming mainstream trends.

Application development

- **Enable code reuse**. Code from and for non-Z platforms can be reused.
- **Utilize non-mainframe skills.** It’s easier to encourage personnel with non-mainframe skills in development tasks.

- **Utilize publicly available development tools.** Many open source and commercial tools are available to speed up and ease programming tasks.

IT operations

- **Utilize publicly available operations management tools.** Publicly available tools can be used for managing, monitoring, analyzing and properly adjusting the Java run time.

General IT

- **Vitalize your mainframe landscape**. Java enablement vitalizes your mainframe landscape and people. In many IT shops, the mainframe is the IT backbone for the business, so it is crucial that it evolves and keeps pace with current market developments to guarantee its agility for the present and future.
- **Enable best delivery**. Java runs almost everywhere. If one platform (hardware, operating system, or software) does not deliver required quality, the workload might run on the next (best) platform. This challenge (or completion) keeps your vendor, your platform and your team fighting for best results.
- **Enable flexible placement option and best fit**. As you have a choice to run Java middleware and applications, you can place applications where they fit best, delivering the best price and performance
- **Utilize widely available skills**. Java skills are widely available. By opening up Java on the mainframe, you make the mainframe jobs more attractive. With a larger pool of potential talents, it is easier to build up critical skill areas. In addition, your Java experts can move around in your enterprise more easily because their skill set fits for many platforms. The more you shield platform specifics from application developers (e.g. with framework encapsulation), the easier it is to reuse Java skills.
- **Reduce risks**. Java enrichment instead of application porting avoids any porting project risk and allows for a seamless and soft forward migration or integration
- **Cut cost.** Java workloads are highly discounted on z/OS, eligible to be offloaded to specialty engines, tools even skills are cheaper to get than for traditional programming environments.
- **Open for more deployment options.** Java runs on many platforms. This gets you out of any vendor locking and enables you to take alternative deployment options.
- **Enhance agility**. Java is supported by a vast community. Enabling Java™ inherits all this technology momentum for your existing assets
- **Leverage hardware technologies.** With traditional programming

languages, it is necessary to explicitly recompile and redeploy code to get something out of technology advances. With Java, hardware exploitation comes automatically for most hardware features (Just-in-time code caches, for example). Specific hardware accelerators (for example, cryptography, compression, and networking) are very often supported by Java on the first day.

There is a lot an enterprise can get out of Java for their existing mainframe application landscape. Not-enabling this technology puts you into a position where your IT cannot competitively support your business. We will touch upon details of Java technology and its concrete benefits in the next section.

1.4. Decision guidance and checklist

Many enterprises have already adopted the Java technology in the distributed server landscape, but still lack the enablement of Java on IBM Z Systems. As we have seen above, there might be a lot of business value by embracing a Java strategy for existing core business systems.

To understand if your enterprise can leverage on these values here are some real and some rhetorical questions you should process in your mind to understand the potential of the technology. The checklist in Table 1 would help you with this mental exercise.

Table 1. Decision checklist

❑	Is Java already used in the enterprise, and reuse of existing Java code on z/OS is therefore a viableoption?
❑	This reuse and redeployment capability of code opens up for optimal placement of applications (forexample, run batches close to the data to be processed). Does this capability help you to run your ITmore efficiently?
❑	Smart application placement can save you from having to implement compensation logic into the application and allows that data integrity work to be done by a robust transaction monitor like IMS,CICS or WebSphere Application Server for z/OS. Does this capability allow you to write better applications with less lines of code?
❑	You can use Java to enhance existing core business functions without the need to rewrite well-workingapplications. Do you still maintain your existing assets written in classic business languages, and do you like to add new functionality to them?
❑	This code enhancement can minimize the well-known risk of rewriting applications in modernizationprojects. Is this risk avoidance a value for your enterprise?
❑	Java integration enables you to leverage all the feature-rich functions, frameworks, and solutions inyour traditional (for example, IMS COBOL) environments. Would this new set of functions and frameworks help your business to get new application features faster to market?
❑	Enhancing traditional applications over time and eventually resulting in zero COBOL or PL/I code allows you to move the applications to almost any Java container on almost any platform. Does this freedom of application placement help your enterprise in the long-term to make more flexible platformdecisions?
❑	Java on z/OS opens up for interesting z/OS charging models with IBM, models that could minimizetraditional mainframe software bill. Is your enterprise interested in entering this space of new workload rebates? Do you need to reduce software cost while delivering higher quality and more functions?
❑	Java runs for more than 90% on zIIP engines and therefore avoids monthly license charge (MLC). Do you have applications that contribute to the four-hour rolling average that could be offloaded to Java?
❑	The Java community is adopting new technologies faster than any other runtime environment. Does this innovation acceleration help you to deliver better applications and infrastructure services in a shorter time?
❑	Java is taught in schools and universities, thereby allowing organizations to hire standard skills fromthe market. Are you experiencing skills issues and would benefit from employing readily available IT skills right away from the large talent pool?
❑	Java is enabling programmers to crank out new applications for the mainframe much faster, because ofthe richness of tools and readiness of APIs. Is slow time to market an existing issue for your business?
❑	Common and modern data access patterns (for example, JDBC, Hibernate, and JPA) are readily available in Java. Would your mainframe application landscape benefit from the ability to build on those assets?
❑	Most of the open-source componentry and supporting tools for Java are available for z/OS. Open- source solutions often see broad contributions and evolve very fast. Does your enterprise support useof open-source solutions to leverage the communities and the progress?
❑	Java helps to open my mainframe environment for new readily-built solutions. Is your enterprise (interested in) buying ready-made software to be able to select from a broader set of vendor applications for a broader set of platforms?

If you checked some boxes and have yet to have a Java strategy for z Systems, you should consider to act now.

> Java on IBM Z Systems can help drive vitality, reduce complexity, and might help to prevent the forthcoming mainframe skill shortage problem. More importantly, by speeding up the application development lifecycle, it creates a competitive advantage for your business.

1.5. Summary

In this introduction chapter, we discussed the motivation and business values to bring in Java to your mainframe environment to expand and maximize the values of your existing z/OS applications. We also discussed the challenges, options, cost, and considerations involved when you engage in Java enablement.

1.6. Related resources

The following list provides useful resources that contain more detailed description of the topics that are covered in this section.

- IBM Redbook: *New Ways of Running Batch Applications on z/OS* at http://www.redbooks.ibm.com/abstracts/sg248119.html
- Configuring large page memory allocation topic in the IBM SDK, Java Technology Edition product information in IBM Documentation.

CHAPTER 2

2. Architectural considerations

Java™ on z/OS® can execute in various target environments. You can run it natively in the UNIX System Services (USS) environment, as a batch job within a batch launcher (such as JZOS), or with a Java application server (such as Apache Tomcat or IBM® WebSphere® Application Server for z/OS). You can run it in, for example, IMS™, CICS®, or Db2® for z/OS, or use it to interact with IBM MQ.

On z/OS, you typically do not run Java client applications with a graphical user interface; rather, you run their server applications. Systems of engagement (SoE) often embrace younger languages like Java, partly because they also have rich rendering capabilities. Systems of record (SoR) typically rely on COBOL, PL/I, and other matured languages for business logic. Many organizations find themselves in a situation where business logic implemented in Java is allowed to execute only in open systems, resulting in a spread of SOR logic over several platforms.

Enabling Java on all platforms means an organization can deploy Java code where it fits best, allowing for re-centralization and sensible grouping of componentry.

If you plan to walk down the integration path further, more thoughts are required to decide on the following aspects:

- The integration depth
- Programming model
- Data access pattern
- The target runtime container for Java

In this chapter we will discuss the major decision points from the application and infrastructure perspective.

2.1. Application migration versus application integration

When thinking about modernizing existing business applications, you would probably naturally think about replacing existing COBOL, PL/I, C or C++ code with Java, or deploying a new application written entirely in Java.

To protect your assets and minimize risks, you also have the option to integrate Java with different languages. If you need to add new functionality to existing code, you can rewrite the entire application in Java, or leave it as is and just add new Java functions to it.

Migrating an existing application to Java by re-implementing existing code might look like a clean solution. It enables new functionality (such as the Java frameworks), creates higher portability, and could leave the underlying data structures unchanged. However, if you already have a working business application (e.g. COBOL), rewriting the existing business functions might introduce unnecessary overhead, risk, and cost especially if you just want to enrich its business functionality.

Note: If a business needs new functionality, most of the time the corresponding applications are extended rather than re-developed. The effort to extend is so much smaller than to re-implement. For that reason, usually an application would only be re-implemented if is in a non-maintainable" state, which causes an extension to be more expensive than a re-implementation. By non-maintainable," we mean it is not only difficult and costly to keep the code running, but also difficult and costly to enhance or extend.

Adding Java componentry to existing applications truly opens up new options; it allows you to pull in different programmers, enables code reuse, and speeds up development time. You might even realize that re-implementation can be completely avoided, because it might be easiest to implement certain functionality with Java than in COBOL.

Another viable option to enrich an existing application with new functionality is to bring in standards-based, off the shelf software written in Java. In this case you will be dealing with a different set of migration issues, such as data and data model migration, application interfaces, education for users and IT staff, and how to embed the new software and its run time into the existing IT processes.

Integrating Java code into existing application code would perfectly minimize risks and efforts by leveraging existing service quality of the container, leaving existing code, interfaces, and data mostly untouched. The tighter you integrate, the more you gain in performance and subsystem qualities. When you link or integrate languages tightly together, you do not have 100% pure Java code, and your Java code would also have an integration and transformation layer, likely written in C (using JNI). To keep everything nicely usable and portable, that layer could be implemented by a code-generating framework.

2.2. Application integration decisions

Different techniques exist to couple your assets, each with its benefits and drawbacks. It's up to the enterprise and application architect to select the integration technology for an application or even for the enterprise.

Avoiding the risk and protecting existing asset consequently requires a lot of thinking and planning. That might be one of the reasons why people often choose to rewrite and redeploy. But again, be aware that larger rewriting projects have a tendency to exceed their budgets or fall behind their schedule, resulting in major delays or project incompletion.

Note: The enterprise architects must identify the most suitable integration technology pattern that fits best to existing applications and service level requirements. This integration pattern (and framework) would guide application programmers on how to add Java componentry to their existing business applications. The right integration or extension has to be selected per application.

2.2.1. Java containers on z/OS

IBM not only provides a complete Java SDK for z/OS and consequently full-blown Java application servers, but also enables relevant middleware subsystems (IMS TM, CICS, Db2 for z/OS) with Java technology. This opens up a variety of choices for your application deployment. Some applications are just a logical match to certain containers. For others, you would have a variety of choices. Selecting the right container is in most cases quite easy and natural. In Figure 7, the runtime selection is the last piece in the simplified flow because the deployment ends here. In reality all those decisions influence each other.

Figure 7. Major decisions to make on the integration path

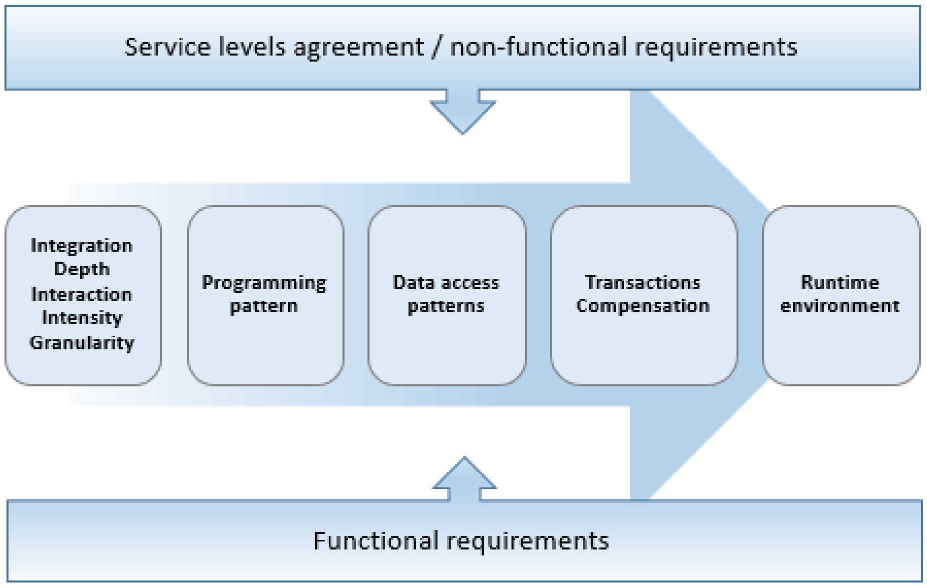
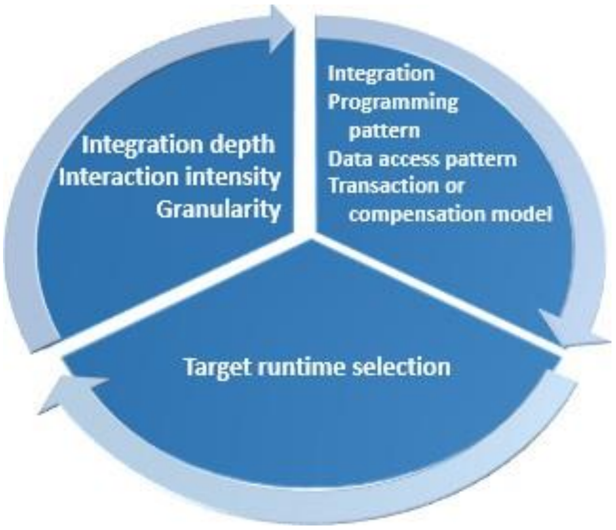

Service levels agreement / non-functional requirements
Integration Depth Interaction Intensity Granularity
Programming pattern
Data access patterns
Transactions Compensation
Runtime environment
Functional requirements

Figure 8. Integration decision points that influence each other

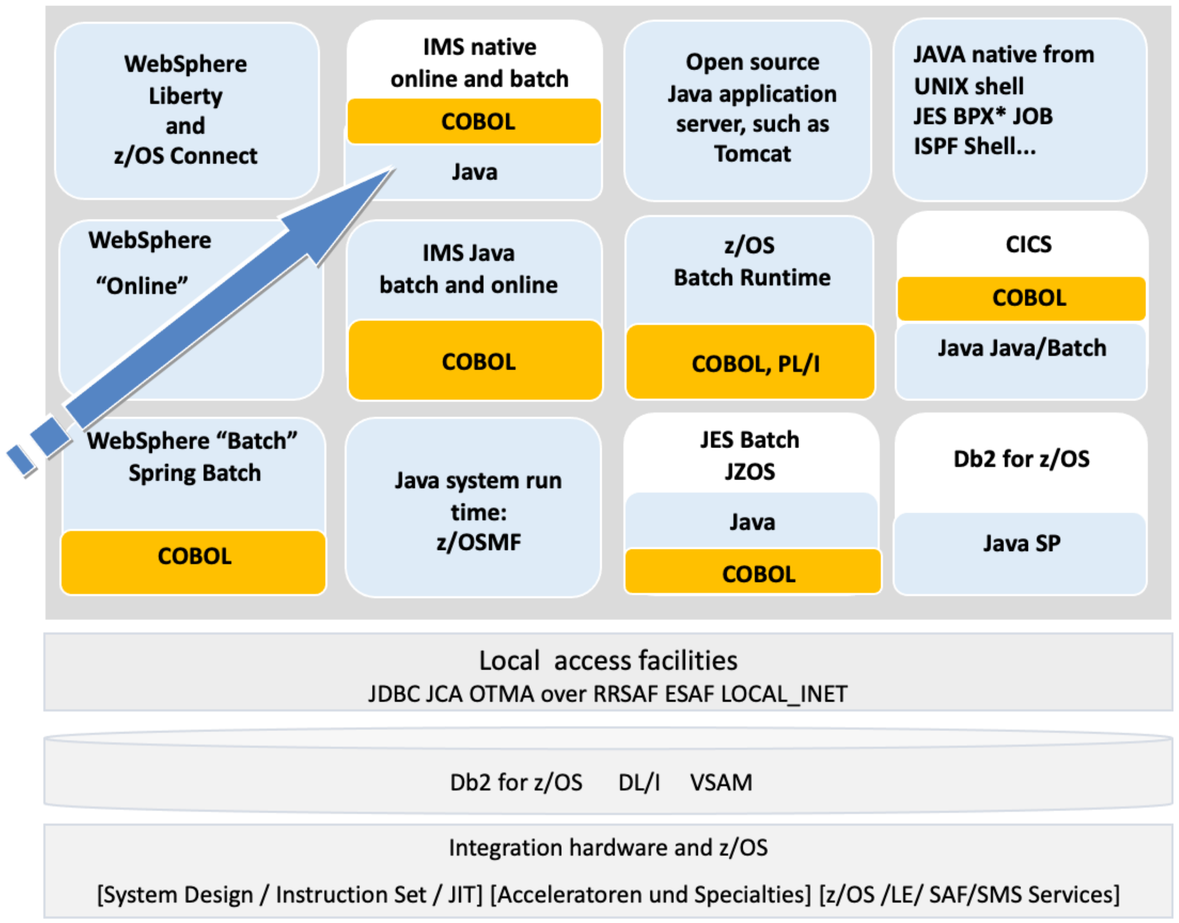


For example, if you set on a specific runtime, you might not have all options regarding integration depth. If you decide on a specific integration depth (and programming model), not all runtimes might be appropriate.

Figure 9 depicts the possible deployment zones where you can deploy Java or Java-COBOL/PL1 constructs. IMS native online and batch environment with COBOL and Java interoperability is the runtime environment that we will focus most here.

Note: If you have the option to run Linux on IBM Z Systems you will add other deployment and integration possibilities to the picture. You will be limited to IP based connectivity between Java on Linux and COBOL/PL1 on z/OS. The IP connectivity can be direct from LPAR to LPAR and therefore very fast, reliable and secure.

Figure 9. Java deployment zones on z/OS


WebSphere Liberty and z/OS Connect
IMS native online and batch
COBOL
Java
Open source Java application server, such as Tomcat
JAVA native from UNIX shell JES BPX* JOB ISPF Shell...
WebSphere "Online"
IMS Java batch and online
COBOL
z/OS Batch Runtime
COBOL, PL/I
CICS
COBOL
Java Java/Batch
WebSphere "Batch" Spring Batch
COBOL
Java system run time: z/OSMF
JES Batch JZOS
Java
COBOL
Db2 for z/OS
Java SP
Local access facilities
JDBC JCA OTMA over RRSAF ESAF LOCAL_INET
Db2 for z/OS DL/I VSAM
Integration hardware and z/OS
[System Design / Instruction Set / JIT] [Acceleratoren und Specialties] [z/OS /LE/ SAF/SMS Services]

2.2.2. Integration depth, interaction intensity, and component granularity

On the integration path you would need to decide on integration depth and density. This is pretty much influenced by the size and interaction intensity of the components. For example, if you integrate a high-speed Java "subroutine" into an existing COBOL application, you would very likely not select a full-blown web service approach for obvious reasons. Your decision on the required integration depth, interaction and granularity requirements will influence, again, the decisions on programming models and container selection.

Figure 10 shows the levels of granularity and the responding process coupling architecture.

Figure 10. Integration, interaction, and granularity

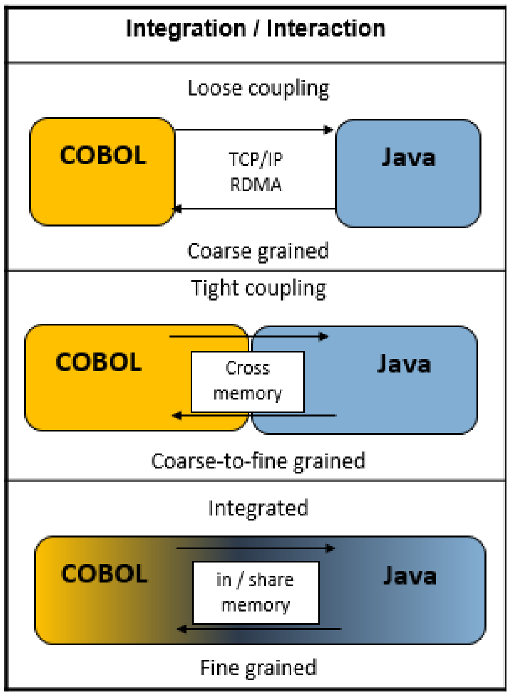


Table 2 provides some more details about the options and resulting qualities and requirements for Java to core business components integration.

Table 2. Options to adopt Java

Scope	Technology	Granularity	Benefits	Downside
Migration and rework	• Porting or new development • Off-the-shelf product	• Monolithic;no coupling	• 100% Java • Vendor solution possible	• High risk and effort
Loose coupling	• IP based-web services • RESTful services	• Coarse • Inter-systemto inter-process	• Very portable	• Complex • Performance not as good
Tight coupling	• Cross-memory adapters-- WOLA	• Coarse- to fine-grained • Inter- to intra-process	• Portable • Good performance • Asset protection	• Skills issue • 2nd run time for Java required
Full integration	• Inter-language communicationvia JNI	• Very fine • Intra-process	• Quality of service • Best performance • Asset protection • Low risk	• JNI layer shielding requires highly skilledstaff • JNI-based

To timely deliver a robust solution with minimum cost, a good solution design is achieved through the following strategies:

- Choose the right technology for the given challenge.
- Place the component wisely for affinity to resources and interaction efficiency.
- Consider component redeployment at a different location if appropriate.
- Hide coupling complexity.
- Protect existing assets.
- Allow for efficient and speedy application delivery.

2.2.3. Integration programming pattern

As seen above, there are many ways to couple application components written in different languages. The integration programming pattern that you identify will describe the how to glue the components together and how to pass data back and forth between the application components. The pattern you choose is heavily dependent on the following decisions and existing facts:

- Couple Java with COBOL or PL/I
- Middleware of choice to be deployed to
- Interaction intensity and required granularity
- Complexity of data structures (copybook)
- Requirement for transactional support

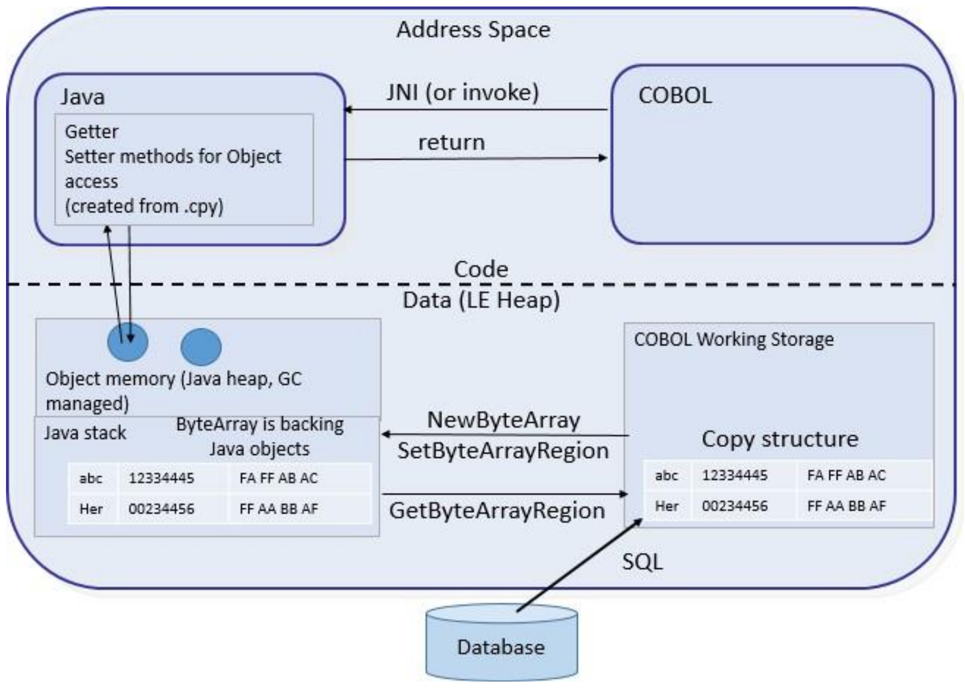
We will focus mostly on the JNI pattern here (such as tightest coupling and intra-process communication), therefore this pattern is described in detail.

2.2.3.1. A JNI-based language integration pattern

Below you can see the componentry and the flow of one possible integration pattern. There, of course, can be variations. Use the flow described in the JNI-integration approach.

Use figure 11 to familiarize yourself with a JNI-based integration approach.

Figure 11. A JNI-based integration pattern

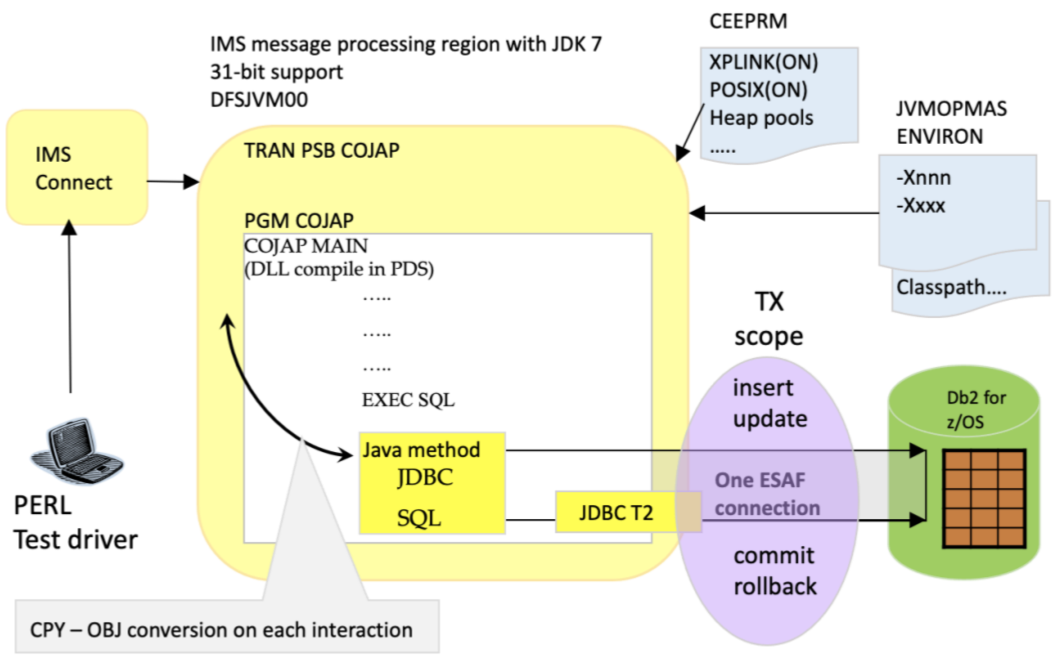


Data flow for this pattern is as follows:

1. COBOL main program initializes; COBOL code executes. In a native IMS environment, the JVM would be fully initialized before any user program gets loaded.
2. COBOL code does SQL queries and fetches data into the work area.
3. COBOL code manipulates that data using `.cpy` structures.
4. JVM initialization in a non-IMS native environment would be done by COBOL under the cover.
5. The JNI communication context JNIENV and JNIPTR would be initialized under the cover in a COBOL environment. All other languages would need to issue JNI calls `GetCreatedJavaVMs` and `AttachCurrentThread.`
6. COBOL uses JNI to allocate ByteArray in Java heap and pushes data from work area into ByteArray within Java heap.
7. COBOL uses JNI to invoke Java methods to operate on that data.
8. Java uses getter/setter methods to operate on that data objects backed by the ByteArray.
9. Many data manipulation types can be implemented (or generated) with Data Access Acceleration (DAA) APIs and are accelerated on z/OS.
10. When data manipulation is finished and Java returns, COBOL uses JNI to `getByteArray` back into work area.
11. COBOL uses SQL to persist manipulated data back into Db2 for z/OS table.
12. COBOL exits.

Running this pattern in IMS would require the configuration depicted below.

Figure 12. IMS environment that supports the JNI programming pattern



Note: JNI needs to be implemented very carefully, because it could cause issues such as abends, storage leaks, or uncaught exceptions. For broader usage, you should shield all the JNI work from the Java and COBOL programmers by implementing a framework.

2.2.3.2. WOLA-based language integration pattern

WebSphere Optimized Local Adapter (WOLA) is implemented as an external subsystem to and from IMS, with the following characteristics:

- Allows bi-directional byte array component communication to and from IMS.
- Usage is supported for Message Processing Programs (MPP), Batch Message Processing programs (BMP), IMS Fast Path (IFP) and Batch DL/I applications.
- Supports PL/I, COBOL, and Assembler.
- Delivers a clean API abstraction on both sides.
- WOLA is high speed, inter-process based.
- Supports IMS, CICS, Db2 for z/OS and JES batch containers.
- Requires WebSphere Application Server for z/OS (WASz) as a run time for the Java components.
- WOLA over BBOxxx API is a direct and local interface. Cross-platform functions (routing) might be handled by WASz.
- WOLA over OTMA can benefit from TCP/IP-based routing.

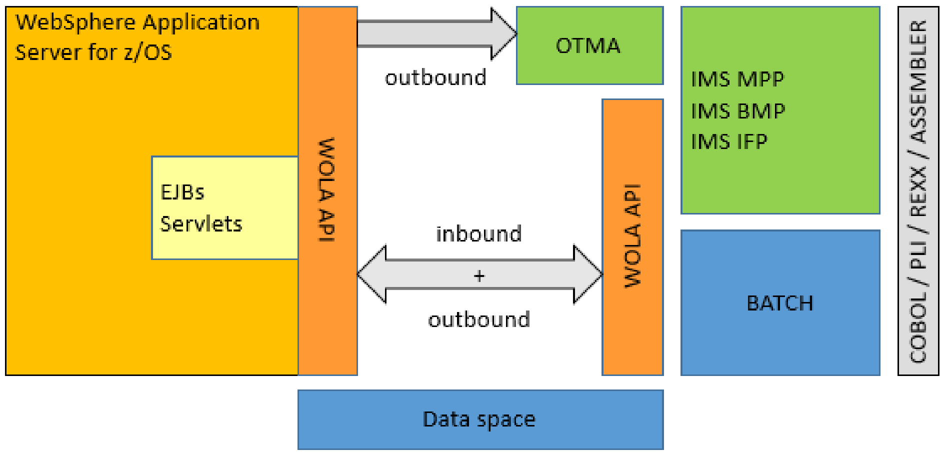
- Allows inter-process integration of 31-bit legacy with 64-bit Java code.
- Provides user delegation and transaction support.
- The external address space (not WASz), such as IMS, is responsible for the WOLA registration and native WOLA API (BBOA1REG) calls through IMS configuration, such as Extended Attach Facility (ESAF) or OTMA.
- WASz must be started before IMS does the registration (for example, via system automation).

Note: WOLA also delivers a 'remote mode' support. This provides proxying for client remote calls in both directions (batch/CICS/IMS to a remote EJB -over the WOLA proxy EJB installed in a locally running WebSphere Application Server for z/OS). This same proxy implementation supports a client in WAS on a distributed server calling remotely (over an IIOP-based interaction) through the WOLA proxy running on WAS z/OS and then on to CICS/IMS/batch.
For more information on this please refer to the following documentation:

http://www.ibm.com/support/knowledgecenter/SSAW57_liberty/com.ibm.websphere.nd.multiplatform.doc/ae/cdat_devmode_overview.html
http://www.ibm.com/support/knowledgecenter/SSAW57_liberty/com.ibm.websphere.nd.multiplatform.doc/ae/cdat_ola_remotequest.html

Figure 13 gives a high-level overview of IMS and WOLA integration options for both inbound and output communications. See Figure 8 for a more detailed view on a specific integration pattern.

Figure 13. Overview of WebSphere Optimized Local Adapter and IMS integration



2.2.3.2.1. Calls from IMS into WebSphere Application Server for z/OS

- Use the ESAF and WOLA native APIs.

- Senders are IMS programs that use the WOLA APIs to invoke WebSphere Application Server for z/OS.
- Receivers are stateless Enterprise JavaBeans (EJBs) that use the supplied WOLA class libraries.

2.2.3.2.2. Calls from WebSphere Application Server for z/OS into IMS

- Use OTMA or direct calls.
  - Direct calls mean a program in IMS uses native WOLA APIs to "host a service."
  - OTMA provides WOLA transparency to the IMS applications, but at a cost of some overhead. The advantage is that no application changes are required in IMS.
- Senders are Java programs (servlets or EJBs) that use the Common Client Interface (CCI) of the WOLA JCA Resource Adapter.
- Receivers are IMS programs that use the WOLA APIs to register for and receive data (direct calls approach).

Using the WOLA APIs in IMS applications reduces the overhead, resulting in better performance and greater throughput.

Tip: Using native WOLA APIs allows you to have the same application code in Batch and Online so there is no need to manage duplicate source code. Two-phase commit (2PC) transactions from a Batch application (native Batch or BMP) with RRS attach facility are also allowed.

2.2.3.2.3. User delegation:

- WebSphere Application Server for z/OS security must be set up to use Local OS user registry at the server level.
- The WebSphere Application Server for z/OS server must be configured to run with the SyncToThread option enabled.
- The IMS OTMA parameter OTMASE=FULL must be set.
- The RACF administrator must define a FACILITY class profile and optional SURROGAT class profile to ensure that the Synch To OS Thread Allowed option is utilized.

2.2.3.2.4. Two-phase commit:

- IMS or batch outbound:
  The WebSphere variable ola_rrs_context_propagate is to be configured to support two-phase commit (2PC) from external address spaces calling into WebSphere Application Server for z/OS where they already have an RRS context.
  On the IMS or batch side, using BBOA1REG WOLA API register flag reg_flag_trans, bit 30 needs to be set to 1 or true. When this flag is set, the

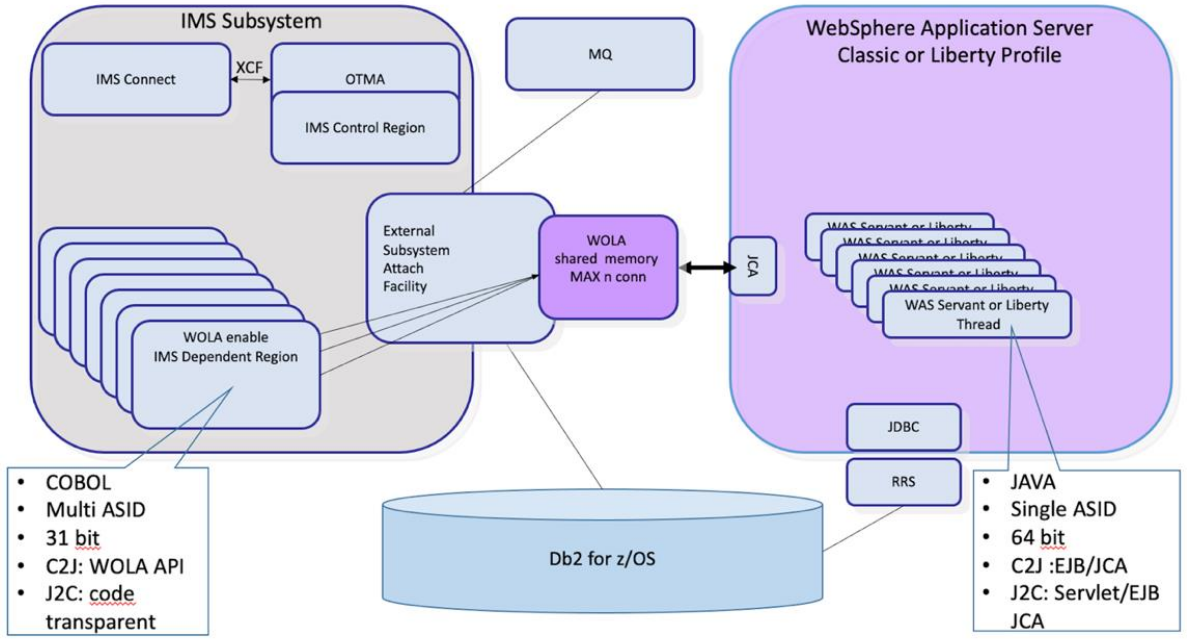
adapter API attempts to register with the z/OS Resource Recovery Services (RRS) and a global transaction supporting 2PC is created between WebSphere Application Server and the API exploiter's environment.

Note: Starting with WebSphere Application Server Version 8.5.0.2, if you are running in an IMS dependent region and set this flag to 1, make sure that you are running your IMS environment with RRS=YES, and the server, or node-level environment variable ola_rrs_context_propagate has been added to your configuration and is set to true (or 1). Under IMS, the RRS context is in effect on the thread when the Invoke or Send Request WOLA API call is propagated and asserted in the WebSphere Application Server for z/OS EJB container

- IMS inbound via OTMA:
  The application to use a connection that is configured to use OTMA, and the RRSTransactional managed connection factory attribute is set to True (or 1). Setting the RRSTransactional attribute to True (or 1) enables RRS transactions for connections. The WebSphere Application Server for z/OS server is running with the **ola_rrs_context_propagate_otma** environment variable installed and set to true (or 1).

The following figure depicts one recommended operational model for a WOLA-based IMS COBOL/PCI to Java in WebSphere Application Server for z/OS integration pattern. The IMS in- and outbound data flows and data operations are also described to provide a better understanding of component and code interaction.

Figure 14. Overview of WebSphere Optimized Local Adapter and IMS integration



2.2.3.2.5. <u>IMS outbound: COBOL to Java data flow with WOLA</u>

The following use case demonstrates the data flow when an IMS COBOL transaction works on the same relational data as the Java program does.

1. WebSphere Application Server for z/OS server starts up. IMS Control Region MPPs start up.
2. IMS MPP1 registers WOLA (by infrastructure, such as a step in region startup job).
   In this process, WOLA checks for the availability of the WebSphere Application Server for z/OS server. It could address another WebSphere Application Server for z/OS server if the original target is not available. Shared Memory address space under WebSphere control is established. WOLA provides connections (via connection factories) and shared memory to operate on. The shared memory is 'assigned' to MPP1 (can deregister and destroy) but managed by WebSphere Application Server for z/OS. If Daemon stops, shared memory address space shuts down.
   There is one register per MPP.
3. IMS TRAN1 does COBOL and SQL processing and reads Db2 data over copybook structures into working storage.
4. IMS begins transaction.
5. IMS TRAN1 COBOL updates the table via static SQL bound plan.
6. IMS TRAN1 puts parameter for subsequent Java method into WOLA request area (shared storage) with the getter method generated by Rational

Application Developer.
WOLA INVOKE (BBOA1INV parameter) is called to operate on the data based on request area information.

7. COBOL/IMS waits.
8. Java reads the table via JDBC into heap.
9. Java processes the data.
10. Java updates the table.
11. Java updates WOLA response area with results using J2C-generated setter method.
12. COBOL gets control and processes WOLA response. COBOL can work directly with the data in the response area.
    As a best practice, COBOL would copy response back to COBOL working storage and then works with the copybook structures on that data.
13. IMS ends the transaction.
14. IMS schedules next transaction.

Note the following:

- If the WebSphere Application Server for z/OS server restarts, any registrations in use at the time with that server are re-activated automatically. Only when WebSphere Daemon restarts, IMS Java dependent region might need to be restarted too.
- Each IMS region is registered with a WebSphere Application Server for z/OS daemon group and server with minimum and maximum connections. Because IMS is single threaded it does not make sense to have more than 1 connection per registration for IMS outbound calls.
- ESAF and RRS support WOLA as a subsystem. Only RRS manages two-phase commit between WOLA/WebSphere Application Server for z/OS and Db2.
- Java can be 64 bit and multi-threaded. It can work on large amounts of data. Therefore, Java is very good for batch.
- Security of the IMS transaction represented by an ACEE is carried forward into WebSphere Application Server for z/OS as a Java subject and can also be synced to a native Java thread identity by setting the EJB Sync To Thread option. Sync to thread needs to be set also in the deployment artifact. Registry must be Local OS. As a result, no security or auditing trail is broken.

2.2.3.2.6. IMS inbound: Java to COBOL data flow with WOLA

The following use case demonstrates the data flow when a Java method integrates with an IMS COBOL transaction to work on the same relational data.

1. WebSphere Application Server for z/OS application (servlet or EJB) uses CCI of the WOLA JCA Resource Adapter.
2. WOLA calls IMS TRAN2.
3. IMS TRAN2 executes and registers with a WASz daemon group and server with a BBOA1REG call.
4. IMS TRAN2 receives data.
5. IMS TRAN2 moves the response to the response area.
6. WOLA sends the data in the response area to the servlet or EJB.

Again, the servlet or EJB can also call IMS via OTMA by specifying the IMS OTMA server name, XCF group ID and transaction level as attributes on the managed connection factory, or use the corresponding `ConnectionSpecImpl` setter methods to provide this information.

2.2.3.3. Network integration patterns

For coarse-grained interaction between core business componentry and Java artifacts, a TCP/IP-based coupling architecture can be the choice, and you also have many options here. Typically, one would want to expose existing assets as a service rather than consuming Java functionality from an existing asset. Nevertheless, it is valid and possible to call, for example, RESTful services from an existing application.

For consummation of RESTful services, you can use the z/OS Client Web Enablement Toolkit.

If you want to REST-enable existing business components it is advised to evaluate the z/OS Connect run time. z/OS Connect run time can technically use the WOLA to IMS OTMA inbound pattern, but is very optimized for the REST/JSON use case and comes with additional functions for that purpose.

2.2.4. Data access pattern

Your existing applications most likely access data from various sources. If you are going to extend your core business applications with Java components, you need to decide if the data access layer will stay in the existing applications, or you will allow the Java components to act directly on that data. Db2 for z/OS, IMS DB, VSAM, or MVS datasets might be the sources for some of that data.

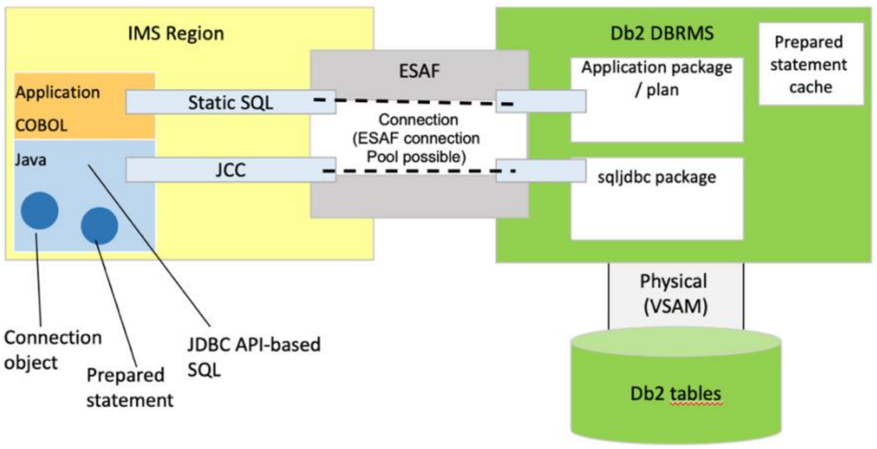
2.2.4.1. Db2 access from Java

If you allow your Java components to interact with data stored in Db2 for z/OS, you basically have two options: JDBC for dynamic queries or SQLJ for static queries. Using the IBM PureQuery product, you might convert dynamic queries to static SQLs on the fly.

In the classic IMS environment (MPP/BMP), COBOL and Db2 for z/OS Universal JDBC Driver (JCC) share the same connection under IMS External Subsystem Attach Facility (ESAF) control. It is perfectly fine to mix dynamic with static queries; you could even get transactional context managed by IMS.

Figure 15 provides a graphical view of the underlying components for this scenario.

Figure 15. Accessing Db2 via ESAF



You must be aware that:

- Dynamic queries are not as fast as statically bound packages.
- The security model between static and dynamic queries is different. You would need to decide how you grant access to the JDBC-driven SQLs (DYNAMICRULES bind option for JDBC packages).
- IMS tries to reuse connections and responding threads in Db2 as much as possible to keep the JDBC connection creation and destruction overhead as low as possible. Make sure that you comply with the rules for ESAF connection pool reuse.
- Do not break infrastructure resource reuse mechanism with inappropriate programming models. Define a standard JDBC access pattern and ensure that you can measure and see what is happening. The following is a working example:

```
<begin app>
   start-message-GU-loop
    IMS message GU call
    open JCC connection
      DB work
    end-message-GU-loop
   close JCC connection
<end app>
```

Note: Using JDBC from IMS classic regions requires the Db2 Universal JDBC Driver (JCC) that delegates ACID (*atomicity, consistency, isolation* und *durability*) transaction handling to ESAF. To enable two-phase commit transactional support, the driver needs to use JDBC TYPE 2 connectivity (cross memory). Theoretically one could also use JDBC TYPE4 (network) based connectivity from an IMS Java program. Be aware that you then would lose the ability for IMS to roll back your JDBC Type 4-initiated Db2 updates. If you need to use JDBC Type 4-based SQL, the following recommendations apply:

* Any Db2 commit should happen at the very end of an IMS transaction.
* You would need to handle the Db2 commits around checkpoint/restarts as well.

In general, it is recommended to use JDBC Type 4 API set, as only the JCC V4 driver would have the latest and greatest support embedded.

2.2.4.2. IMS DB access

To access an IMS DB resource, you would need to enable the IMS Open Database JDBC interface and connect to the IMS Database by using the JDBC Data source or the IMS JDBC DriverManager Interface.

2.2.4.3. VSAM access

To access VSAM key-sequenced data sets (KSDS), use the APIs and samples from the JZOS package, which is part of the z/OS Java SDK.

2.2.4.4. MVS data set access

To access classic MVS data sets, use the APIs and provided samples from the JZOS package which is part of the z/OS JAVA SDK

2.2.5. Transaction or compensation model

To ensure data integrity and recoverability when data processing is distributed, logic can be implemented by the application (compensation) or delegated to the infrastructure (ACID transactions). Having the ability to do the latter could help you avoid a lot of complexity, reduce investment, and achieve better maintainability and faster time to market. Executing Java and existing core business components in or next to a transaction monitor container (IMS, CICS, or WebSphere) together with the appropriate programming and data access pattern would delegate the transactional integrity to the infrastructure. Furthermore, enabling this co-location or integration technology might allow for redeployment of remote application with embedded compensation logic into a transactional run time. This approach alleviates you from the burden of having to maintain compensation logic and opens up the ability to delete the code segments from your application.

2.3. Summary

In this architecture considerations chapter, we discussed the various aspects for considerations to arrive at an integration decision. We examined the target runtime container for Java, integration depth, interaction intensity, component granularity, programming patterns, data access pattern, and programming models.

In the next chapter, we will discuss how to examine and manage requirements and cost by considering both functional and non-functional requirements to help you develop your integration architecture blueprint.

CHAPTER 3

3. Managing requirements and cost

As discussed in the previous chapter, application integration architecture should be a blueprint for enriching and extending mainframe-based business applications for new business needs. Additional aspects must also be taken into considerations to ensure the new pattern would be able to fulfill those requirements. The aspects include functional and non-functional requirements and the cost.

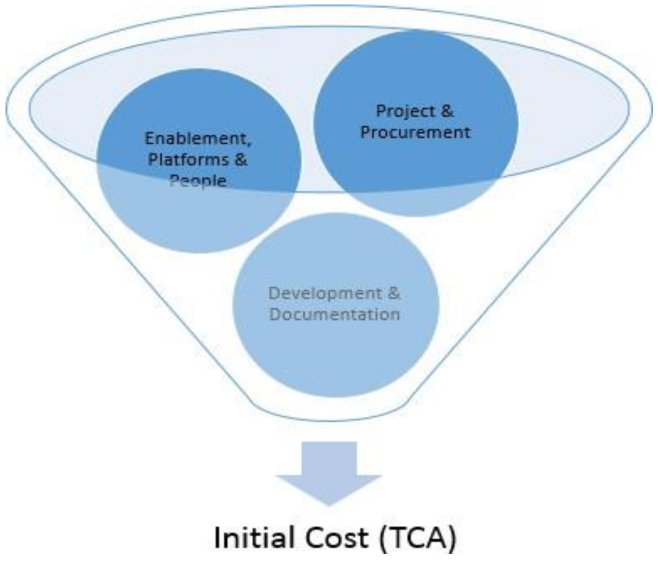
3.1. Cost and savings

The cost of the solution architecture (implementing architecture plus developing and deploying hybrid application) is one of the most important drivers of your decisions. A perfect start would be a business requirement, or an IT strategy to weight into the math.

One needs to implement most of the required functions and qualities (non-functional requirements) for the least or given cost. Typically, compromises are done, and everybody is interested in saving cost for a given project.

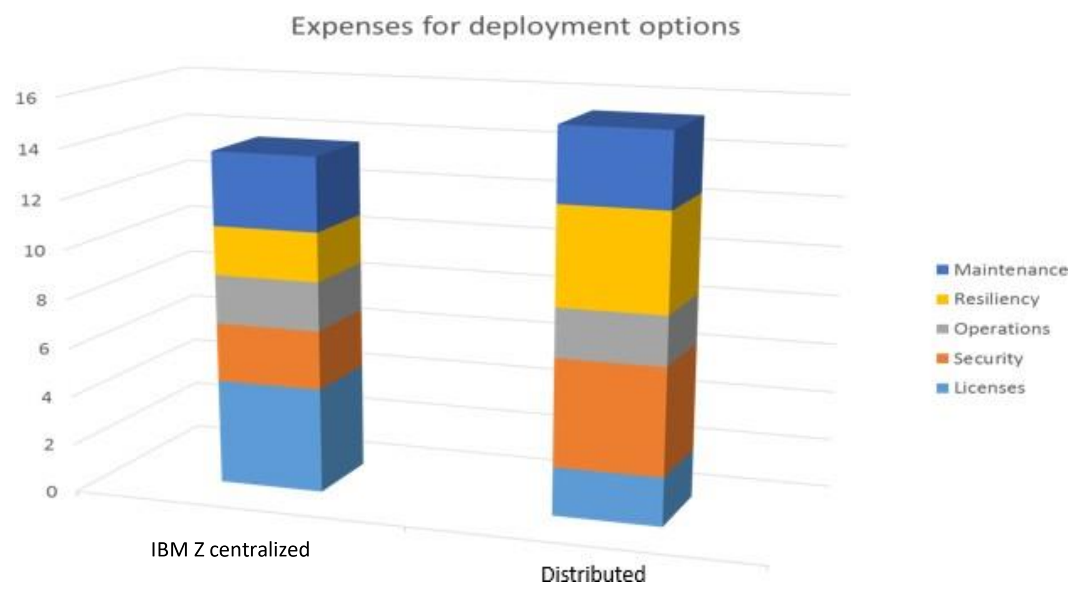
There will be initial cost and running expenses, altogether those will differ according to the enterprise architecture you chose and depending on where you deploy and run the application artifacts (if you have a choice).

Figure 16. Some major initial one-time cost



Comparing cost: Estimating one-time cost and running expenses is one task. Weighting those estimations against (maybe non-existing) alternative solutions adds another level of complexity. You do have alternatives at various levels, coarse- or fine-grained. For example, you can run a needed business function within Java on the mainframe or somewhere else. Every coarse-grained decision is followed by many fine-grained decisions (for example, integration depth).

Figure 17. Expenses based on deployment model (Example)



You will have a wide range of influencing factors. You need to understand how all your decisions influence cost and be able to articulate a major cost vs. benefits view to the decision board, with alternative solutions factored in. This high-level view might be one of the most crucial starting points to get buy-in and support by your decision makers.

Cheap is sometimes expensive. At least in IT, money is not spent for fun. Total cost of acquisition (TCA) and total cost of ownership (TCO) must be weighed against the benefits you gain out of the solution. To help make the best decision from a cost perspective, factor in the "global" cost depending on where you run your application artifacts. For example, if you run a JDBC-driving Java application on a distributed platform, you would be consuming zero CPU instruction on the mainframe for the driving application. If the accessed database is on Db2 for z/OS, you would run that query for a large percentage on zIIP engines and that would save some cost. Be careful, though, of the so-called pretzel logic. In the end you might end up consuming more cycles on general processors on IBM Z Systems plus additional cycles on zIIPs because the path length of a remote request can be easily 10 times higher than that of a local one. In addition, because you have a more complex solution, investments and re-occurring cost for operations, security (such as SSL), change management and such all add up. The message here is very simple: you need to understand and articulate the whole picture, the alternative solutions, and all associated cost aspects when you do your cost comparisons. So yes, local JDBC calls on z/OS might drive additional CPU cycles, but at the same time they also avoid additional cycles on the same system and related cost on and off the platform. And we are not even considering performance-related issues here.

Let’s touch on some of the aspects that have a direct influence on the cost factor. The items listed here are only eye catchers. To seriously calculate cost, including considerations of benefits and savings, you would need to execute a formal TCO study.

- Development cost for implementing a new architecture and framework Initial cost and any maintenance cost for the framework should be considered.
- Developing against the new programming model
  - Consider the initial cost for education and tooling and repeated cost per development project.
  - Running Java components together with existing applications in an transaction monitoring run time avoids development time and cost for alternative solutions (e.g. compensation logic).
  - The ability to re-use applications or code artifacts for different platforms can reduce development cost significantly.
- Cost of running the new infrastructure
  - Complexity equals cost! Less complexity means less operational cost (people, tools, procedures, security, auditing, and more).
  - Offload to specialty engines lowers software bill from many vendors. Be careful and do not blindly head for high specialty engine utilization, which might become more expensive in the end
- Recovering applications and infrastructure
  - Disaster recovery and normal recovery procedures might already be in place for existing application landscape. You might inherit those qualities for no charge when you just add Java to the mix.
  - Less complexity leads to faster problem determination and error handling and therefore can lower the cost and damage of outages.
- Availability of existing and enriched applications. If your underlying infrastructure supports high availability for your existing applications, you might inherit this high availability at no charge into the Java layer, too. If you have more moving parts, you also need more spare parts readily available.
- Managing cost by application placement

    - By writing artifacts in Java, you enable your application topology to the "best fit" platform. For non-business critical applications, that could be the lowest cost platform.
    - Being able to deploy anywhere eliminates vendor stickiness.
    - Running applications or artifacts where they fit best would allow you to make the most out of your investments in infrastructure and middleware.
- People cost. Java developers are likely easier and less expensive to get than COBOL or PL/1 experts. However, a good and experienced Java programmer might not be as cheap as you expect, especially if the developer can write enterprise class applications.
- Benefits. This is the most crucial item on the list. By adding Java components to existing business applications, you can significantly speed up application development and time to market:
    - Use the skills that are available
    - Embed ready-made solutions and frameworks
    - Write code on the platform of choice
    - Enable agile programming paradigms
    - Reuse what you have and avoid re-inventions
    - Delegate as much as possible to the underlying infrastructure (such as security, auditing, and transaction handling)

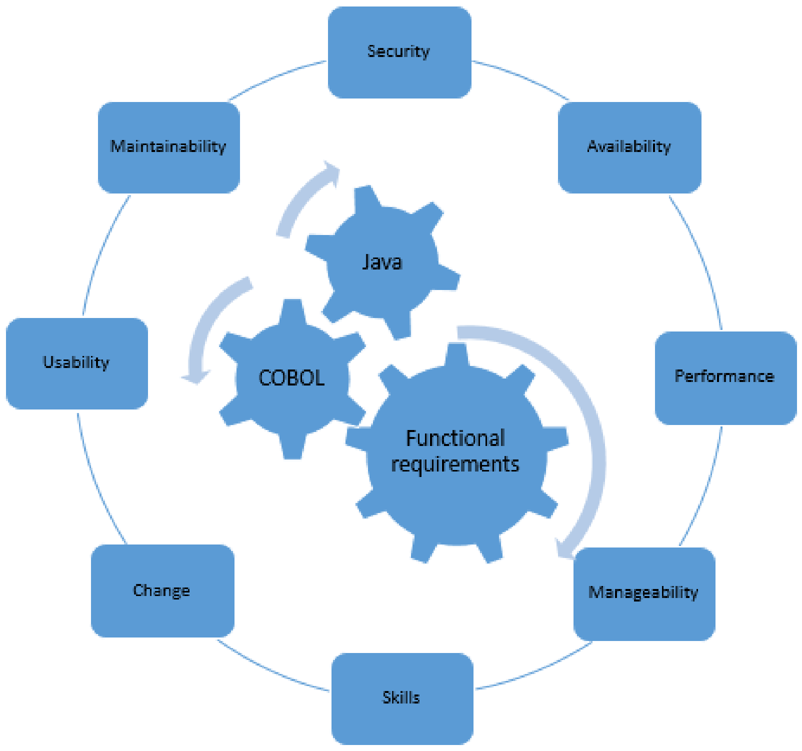
Long term view

To make the cost picture complete, think about how your solution might rate in the long term. Thinking through topics such as competitive advantage, vendor lock-in, portability and reuse, time to market, strategic match, skills and flexibility might add another layer to your blueprint.

3.1.1. Requirements

The boundary between functional and non-functional requirements is not always clear. Non-functional requirements (NFRs) describe more the qualitative aspects of a solution. Service Level Agreements, for example, might be the base for NFRs. Functional requirements describe what a system should do. In the following we well touch on those requirements where we feel are most relevant for this kind of integration projects.

Figure 18. Some important requirements



3.1.2. Non-functional requirements

NFRs influence more than just your integration technology decisions; other qualities also need to be considered. And again, you might want to satisfy requirements for a full range of applications, so look at the more challenging goals in the SLA to blaze a trail for an integration programming model and a range of functions that will be implemented based on that model.

3.1.2.1. Availability, robustness, reliability, recovery and resiliency

The applications following a local integration scheme needs to support the availability SLAs. Make sure that none of your decisions would break any of those. In general, if you add Java to your IMS COBOL run time, most of the availability measures in place are inherited, and disaster recovery would not change. You might want to consider, though, if changes at the lower level would cause changes to your availability and disaster recovery mechanics that are in place:

- You might need more CPUs, different CPU, or more memory to run your application. For disaster recovery you need to plan for sufficient resources to cover workload on fallback machines.
- If you need to execute an emergency change, you might need to adjust procedures. A “deployment unit” is no longer a compiled and linked COBOL module, but very likely HFS objects that go along with it, such

as .jar or .so files.

- You cannot load and destroy a JVM for every IMS transaction that is scheduled. If you enable Java, the Language Environment (LE) and the JVM stay up. That might cause issues if you are used to an automated cleanup of your environment. You will need to consider the IMS CANCEL_PGM=Y setting. With this setting, IMS will cancel candidate programs at the end of each transaction message commit scope (that is, the message GU processing or UOR boundary), which causes negative performance impact on transaction throughput. You also might need to initialize LE storage on each CANCEL to avoid unwanted behavior, especially if your programmers have been a bit sloppy.

3.1.2.2. Change and deployment

Managing change that affects multiple platforms and multiple languages and code versions is more challenging than managing a change on a single platform with a "deployment unit" that integrates multiple application artifacts for one run time. Still, the physical artifacts for deployment will change, and you might want to touch up you code management systems and the attached deployments procedures.

IBM Rational Team Concert and IBM UrbanCode Deploy are solutions that support this kind of environments.

3.1.2.3. Maintainability

Keeping everything tidy and on current levels is much easier if you have less moving parts. To apply fixes in a tightly integrated z/OS-based environment, you have one-stop shopping with SMP/E.

If you have more moving parts that are not tightly integrated, you need to ensure that all the pieces have compatible software levels at the same time. This requires coordination via change processes.

3.1.2.4. Manageability

The solution needs to be run and managed. Tools and processes need to be available to support it. Some areas to look into are:

- Monitoring and reporting
  Java components need to be monitored in context with the corresponding existing business applications. Some available solutions for Java on z/OS are:
  - OMEGAMON Monitoring for JVM on z/OS
  - IBM Application Performance Analyzer for z/OS
  - SMF Java Records for IMS (SMF 29), WebSphere (SMF 120) and JZOS (SMF 121)

    - IBM Support Assistant tools, such as IBM Health Center for Java
    - Many 3rd party and open source tools that plug into the JVM for that purpose
- Debugging, fault analysis, and problem determination
    - IBM Debug Tool for z/OS support COBOL, Java, and COBOL/Java
    - IBM Fault Analyzer supports COBOL, Java, and COBOL/Java
    - Java problem determination data (such as garbage collection, thread dump, and heap dump) are all available on z/OS and also for an IMS environment. You can use Java tools (in IBM Support Assistant) to analyze those data. Java problem determination data is also available in the typical dumps in z/OS. The data can be analyzed with z/OS tools, such as the interactive problem control system (IPCS) or IBM Fault Analyzer or extracted for distributed or command-line tools.
- Automation integration
  An integrated solution does not require many changes to the automation. The more you disintegrate, the more coordination of automated tasks is required.

3.1.2.5. People and skills

If you add Java to the classic infrastructure, make sure that people understand Java on the infrastructure and development side. Provide best practices and training.

If you do it right, you can reuse available Java skills for mainframe-oriented tasks. Java is Java, and good people are willing to learn the enterprise classic environment. In the long run, you might consider having Java folks support all Java platforms in your enterprise.

3.1.2.6. Performance and scalability

An integrated solution will perform best. Considerations to take into account are listed here. More details on those are discussed in the Infrastructure, setup, and scenarios chapter.

- Java is more expensive than COBOL. Remote Java is more expensive than local Java.
- Dynamic SQL is more expensive than static SQL.
- Data marshalling and data transformation are expensive.
- For best performance, stay scheduled in IMS.
    - Set the PROCLIM parameter higher than 1
    - Use ESAF connection pooling and understand the options and requirements (Keep Dynamic, Plan/Userid, Db2 bind RELEASE on deallocate)
    - Use the appropriate OTMA security model and the ACEE enhancements (ESAF_SIGNON_ACEE) delivered in APAR PI64496.

    - Preload.
    - Avoid automatic LE STOR initialization.
    - Avoid RACF statistics on every Db2 for z/OS call.
- Make sure you understand the performance characteristics for the steps you make. Good measurement tools are mandatory.
    - IBM Application Performance Analyzer for z/OS does sampling for IMS and Db2 and profiling for Java.
    - A Java Profiler will help to understand Java behavior.
    - IBM Health Center gives a high-level view on Java performance and on the JVMs that are running in traditional IMS.

Set your performance goals and manage towards them.

3.1.2.7. Security, auditability, and compliance

An integrated solution does not need additional protocol protection between its components, because there is no networking protocol. You can get an auditing trail also for you Java pieces from SMF. The more you disintegrate components, the more security measures you need to take, and the more you need to invest in creating and correlating the individual audit trails.

3.1.2.8. Usability

An integrated solution has less moving parts and is less complex, more usable, and more consumable. Nevertheless, the integration layers might be complex and needed to be shielded from developers. This complexity is only seen in a tightly integrated solution.

Typical development tools for Java (Eclipse) exist and might be used. IBM Developer for z/OS Systems (formerly known as IBM Rational Developer for z Systems), also supports classical host programming languages and integrates nicely into the z/OS environment with an eclipse frontend.

3.1.3. Functional requirements

There might be additional requirements that the system needs to deliver. Your environment might have different or additional requirements, but here are some common ones:

- **Batch capabilities**. If your application needs to support running in a batch run time, you will need to select a container that delivers the required function. All IMS BMP capabilities are inherited to the Java run time. For example, checkpoint processing is readily available.
- **Online capability**. If your integrated application is used for transaction processing and minimal response time and transactional behavior is required, it needs to run in the appropriate container. Running Java in a classical IMS transaction monitor will inherit IMS transaction capabilities to the Java run time.

- **Messaging and asynchronous communication.** If you want to get called from a messaging infrastructure, nothing changes if Java is running inside IMS. If you want to initiate messaging from IMS, you can use JMS from the Java layer. For sending JMS messages, the same rules that apply for WebSphere MQ messages apply here.

3.2. Best-fit application placement

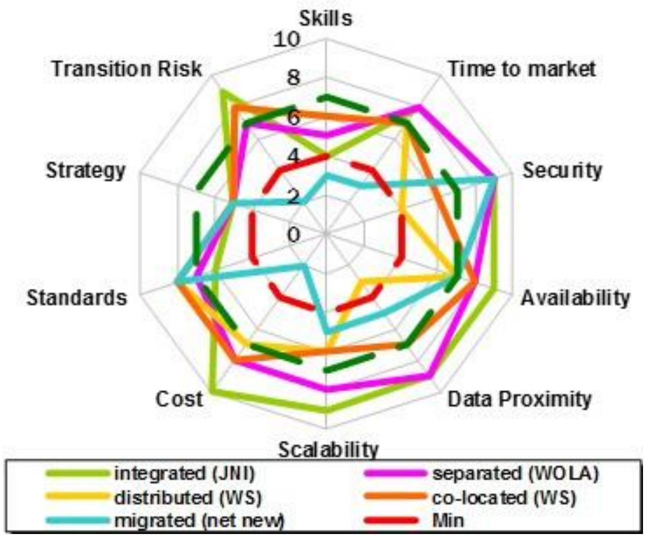
If you went through all that thinking and decided to enable Java on multiple platforms, then frequently the ultimate question comes up: Now that I have the choice, where should I place or deploy my Java application?

This is a challenging question to answer, and it gets more complex when you add the middleware decision on top of the platform decision.

If you look at an application, its interaction with other systems or services, its API set, its non-functional requirements etc., your guts will probably tell you where to run it. Very likely you would need a more scientific approach to make that decision process transparent. A selection approach, one that is sanctioned by all possible (and appropriate) infrastructure providers in your enterprise, seems to be the golden bullet here. You might end up describing the application requirements and map platform capabilities to those requirements and weigh them. Then route it through your steering or architecture boards to determine the best-fit platform.

If you need help in this process, call your IBM representative. IBM Z System Client Architects are armed with a methodology that is called ***Fit for Purpose***, an approach that would provide a good starting point for that exercise.

Figure 19. Example of a Fit for Purpose deliverable for a given application



Alternatively, you can make this decision by discussing placement benefits and tradeoffs within an architecture board and document their decision or create a decision guidance for the management.

As a rule of thumb, the following criteria would suggest that there is a very good fit for your Java application on z/OS:

- The application relies heavily on data that sits on the mainframe.
- The application topology could be drastically simplified when the application is moved to the mainframe (e.g. remove IP connections, encryption, protocol switches, and tiers).
- The redeployment on z would simplify application logic (e.g. remove compensation).
- The application has high security and audit requirements that can be satisfied by z/OS.

3.3. Summary

In this chapter, we discussed how to examine and manage requirements and cost by considering both functional and non-functional requirements to help you develop your integration architecture blueprint.

CHAPTER 4

4. Application development

In this chapter, we will discuss application development for mixed-language applications, as well as Java™-only applications. A high-level discussion on how a batch application can use COBOL and Java with interoperability will be provided. We will also discuss how IMS applications can access IMS™ DB, Db2® for z/OS®, or IBM® MQ, and how you can generate access methods from COBOL copybook data structures.

The target audience of this chapter is application developers as well as personnel responsible for setting up and supporting the development environments.

4.1. Development environments

In many cases, the development and versioning for Java applications is different from what is used in the mainframe environment, for example, for COBOL applications.

There are different approaches for developing Java applications on the mainframe or for Java interoperability:

- Use of Eclipse for mixed-language applications with Java.

  Some mainframe vendors have already enhanced their products for software versioning and life cycle management so that they can also be used from the Eclipse environment. This allows the coding of both Java and traditional languages within Eclipse.
- Use of existing infrastructure for traditional languages to coexist with existing Java development environment and the build and versioning infrastructure for Java.

  For example, IBM® Rational® Team Concert can be used from within IBM Developer for z/OS®. At client sites, Eclipse-based development environments with Maven builds have been used in addition to Concurrent Versions System (CVS) or subversion as versioning and team infrastructure.
- Standalone development of Java-only applications with existing tooling for Java on distributed platforms with no difference to deployment on z/OS.

There are software products from IBM, vendors, and open source tools that can suit the needs of development, teaming, versioning, and building the infrastructure for enterprise use. The approach, requirements, and features should be carefully evaluated for your specific environment.

4.2. Writing and testing Java applications outside IMS

Today, Java application programmers prefer developing and testing in an integrated environment. In this section, we will discuss some hints and tips on how to write IMS™ applications in an environment such as Eclipse, which allows for testing without the need to run on z/OS® in an IMS region and provides database access to IMS DB and Db2® for z/OS.

The simplest approach is to treat the future IMS Java application as a Java main program or a Plain Old Java Object (POJO). If the application can run outside a Java Platform, Enterprise Edition (Java EE, previously known as J2EE) application server and IMS, with no message queue access and no system calls, you can set up remote connections to IMS DB using IMS Open Database Manager. In addition, for Db2 access you can use the Java Database Connectivity (JDBC) drivers in Db2 on z/OS. For further details on how to use JDBC to access IMS data, refer to IBM Redbook publications IMS 11 Open Database (SG24-7856) [1] or *IMS Integration and Connectivity Across the Enterprise*[2]. Open Database was introduced in IMS 11, and the concepts and techniquesdiscussed in these books apply to later versions of IMS. There are also similar publications for Db2 on z/OS.

With this approach, programming and unit testing can be done in Eclipse, and the integration tests are usually done on the mainframe. When tests outside the mainframe are completed, the connection URL for IMS DB and Db2 needs to be changed to use the IMS Universal JDBC type 2 drivers for running the code inside an IMS region. In addition, for Java-only applications, the code to access the IMS message queue (a `getUnique` call before a `getNext` + `insert` loop) needs to be added. For mixed-language applications, this is usually done in the calling language part, e.g. COBOL or PL/I. This approach might not fit all scenarios, but data access and data manipulation services can be developed and tested without the need to write and to deploy on z/OS to run in an IMS region.

Integration tests, such as interaction with other batch workloads or with the IMS message queue (queuing messages or triggering transaction), still need to be done on the mainframe with z/OS and IMS.

4.3. Bridging from Java to COBOL (or PL/I or Assembler)

In this section we will provide a high-level discussion on how a batch application can use COBOL and Java with interoperability. A deep-dive into Java Native Interface (JNI) and Java interoperability with detailed examples and steps to compile and run them in UNIX System Services (USS) and MVS

are provided in the “Bridging from Java to business languages” chapter.
Most of the statements in this section also apply to other languages such as PL/I or Assembler, but the `INVOKE` syntax to call Java methods is unique toCOBOL. Java has the option to call and to be called from other languages byproviding APIs and the JNI interface. JNI was developed for the C/C++ languages, but on z/OS it is available for all Language Environment (LE) compliant languages.

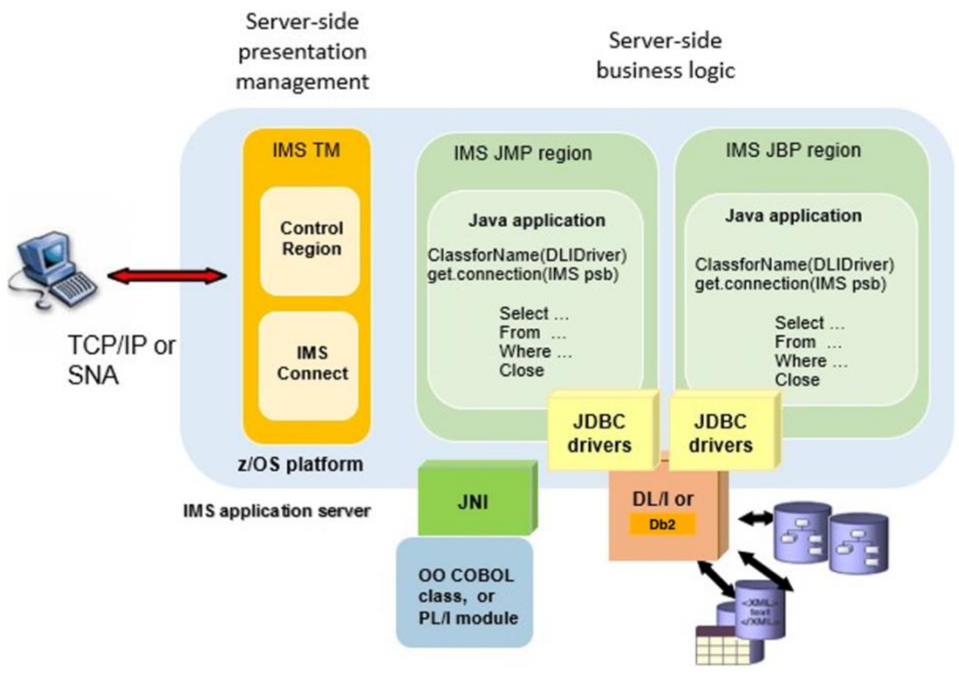
Java invoking COBOL and vice versa are the two possible execution environments.

4.3.1. Java calls COBOL

In this scenario, a Java `main` method starts the application and calls COBOL. Cascading calls from Java to COBOL to Java and to COBOL are possible. The runtime environment is an IMS Java dependent region, such as the Java Batch Processing (JBP) region.

- Due to LE requirements, the COBOL code (first COBOL module in the calling chain) that is invoked from Java is required to be object-oriented (OO) COBOL classes, but has the option of implementing `CALL` statements to procedural COBOL modules. It can do static or dynamic calls to procedural COBOL modules, although dynamic calls can only be made to DLL-compiled COBOL modules. A statically-linked Wrapper module for a NODLL module also meets this requirement.

Figure 20. Java calls to the LE languages through the JNI



4.3.2. COBOL calls (invokes) Java

In this scenario, a COBOL main application starts and calls one or more Java methods. Cascading calls from COBOL to Java to COBOL and back to Java are possible. The runtime environment can be a message processing program (MPP) or batch message processing (BMP) program.

Due to LE requirements, if the COBOL INVOKE syntax is used, the COBOL code that calls Java is required to be mixed-case, with support for long field names as well as DLL-compiled COBOL code. Actually, only the program names need to be mixed-case. PGMNAME(LONGMIXED) is the required compiler option and all other items can be short and upper case, but in most cases that might not be practical, especially with mixed case constants for Java class and method names.
If the caller is NODLL-compiled, a statically linked wrapper module for a DLL module is required. DLL-compiled COBOL code is not required if JNI calls only are coded and no INVOKE is used.

These options also apply to other languages, for example, for the LE-conforming assembler (DLLs) and Enterprise PL/I applications. The difference with COBOL is that the JNI calls and its function pointers in PL/I are more difficult to implement due to the fact that there is no built-in support for interfacing with Java. For example, a DLL can be created with LE Assembler Macros and with PL/I, but the Java wrapper stub source is only generated by the COBOL compiler and has to be manually created for the PL/I and Assembler-based DLLs.

The CEEENTRY assembler macro allows the creation of LE-compliant DLLs written in assembler. Please refer to the documentation for more information about how to create an LE-compliant DLL with CEEENTRY and its required options in https://www.ibm.com/docs/en/zos/2.1.0?topic=wdc-writing-your-language-environment-conforming-assembler-dll-code [3]. By manually creating a Java class as the JNI wrapper, it is possible to call an Assembler DLL from Java.

4.3.3. COBOL INVOKE v.s. JNI

The IBM Enterprise COBOL compilers introduced a new syntax pattern in Version 3 that allows Java methods to be invoked with a COBOL statement called INVOKE. This allows a simple way to invoke Java without coding JNI calls, yet with some limitations.

The syntax for the COBOL INVOKE statement and examples can be found in the Enterprise COBOL for z/OS product documentation [4].

4.3.4. Getting started with JNI?

To get your hands on JNI, see the "Bridging from Java to business languages" chapter for Java and COBOL samples with detailed steps to compile and run them in UNIX System Services (USS) and MVS.

In general, JNI programming is not simple and requires quite a bit of knowledge. There are not many COBOL-based samples out there, but it is possible to look up C-based samples on the web to get an idea about how the JNI calls should be used.

4.3.5. Options to pass data between languages

Since not all data can be passed as a String or simple types, it is best to pass byte arrays that represent byte-compatible data in the COBOL application between COBOL and Java. In addition, because Java methods can return only one data item, it is not possible to return more than one String, byte array, or simple type.

For parameters that are input to the Java method, they can be overwritten if the caller (COBOL) has allocated them in a way that makes them accessible. For example, the following Java method cannot return a string:

```
javaMethod(new String(“Hello”));
```

But the following code can return a string because you can access mySTring after the Java call.

```
String myString = new String("Hello");
javaMethod(myString);
```

Use the JNI functions such as `GetByteArrayRegion` and `SetByteArrayRegion` to transform a COBOL structure into a byte array and vice versa. The called Java method is defined as accepting a byte array and returning a byte array. The marshalling of transformation from COBOL to Java type and back is then done when the data is accessed using a getter or setter method from within Java. This approach has been proven successful in several client cases and performance tests as the most efficient way of passing data back and forth between Java and traditional LE languages.

4.3.5.1. Direct byte buffers

Using the directBytebuffer object, it is possible for Java and traditional languages such as COBOL and PL/I to share the same storage areas for byte arrays.

There are basically two ways to achieve that:

- Call the NewByteBuffer JNI function with a pointer address and length of the COBOL copybook structure (or storage area) that is to be shared. It returns a jobject containing the java.nio.ByteBuffer reference.
- Call the Java methods with the ByteBuffer as an input/output parameter. When Java uses put or get methods to read or write to ByteBuffer, it also changes the data that COBOL has access to.

In this scenario the ByteBuffer is allocated outside the Java heap, so the Bytebuffer is not backed by a Java byte array. This means that if the byte array should be used (for example with a J2C Wizard generated Record structure and its getter and setter methods), a copy from the ByteBuffer to a byte array is needed and the byte array needs to be copied back to the ByteBuffer, which has no advantage over using `SetByteArrayRegion` and `GetByteArrayRegion` for a data exchange with ByteArrays from COBOL or PL/I.

The advantage is there only if not all data of the copybook structure is needed to be accessed and the put and get methods of the ByteBuffer can be used, then it saves CPU.

4.4. Accessing Db2 from mixed-language applications

IMS applications can access IMS DB and Db2 for z/OS data. For mixed-language applications, each of the languages (e.g. Java and COBOL) can access Db2, and IMS ensures that the updates from all languages are within the same unit of work or transaction boundary. Furthermore, static and dynamic SQL canbe mixed as required. It is also possible to call Db2 stored procedures in addition to SQL queries.

Modernization is usually started with existing COBOL or PL/I applications that use static SQL and the traditional EXEC SQL calls. Moving to dynamic SQL requires user/caller authorization against the database object, whereas for static SQL, the authorization is against the package.

For Java to use the Db2 Universal JDBC Driver, the following three .jar files are required in the IMS region's classpath:

- `db2jcc.jar`
- `db2jcc_javax.jar`
- `db2jcc_license_cisuz.jar`

You also need to use a compatible connection URL, such as:

```
String url = "jdbc:db2os390sqlj:";
```

The connection to the Db2 subsystem for the Db2 JDBC Driver is configured as Resource Recovery Services (RRS) connectivity type for IMS Java dependent regions (JMP or JBP) and as External Subsystem Attach Facility (ESAF) connectivity type for all other Java-capable IMS regions (message processing program, or MPP, and batch message program, or BMP).

Note: Due to restrictions in the usage of the Db2 stub linkage, the connection type needs to be different for Java and non-Java IMS regions.

Currently modules that have the Db2 stub statically linked can be used in either ESAF or RRSAF environments, but not in both. Modules serving the same purpose or implementing the same functionality must have different names, one for the ESAF and the other for the Resource Recovery Services attachment facility (RRSAF) environment.

The connection definition for an IMS application is done by using subsystem member (SSM) definitions with two entries, as shown in the following example, one for ESAF and the other for RRS.

Example 1. IMS subsystem definition for Db2 with both ESAF and RRS

```
SST=DB2,SSN=DSNA,LIT=SYS1,ESMT=DSNMIN10,REO=R,CRC=-
SST=DB2,SSN=DSNA,COORD=RRS
```

If there is a requirement to allow Java programmers to use dynamic SQL with JDBC but to use the authorization schema of static SQL, then using IBM Optim™ pureQuery to query Db2 can be considered as a solution. Dynamic JDBC calls can be recorded and turned into a static package, and the pureQuery runtime executes those calls as if the source code had static SQL.

4.5. Using pureQuery for SQL-like access to Db2

In addition to allowing the capture of dynamic JDBC calls that can be turned into a package with static SQL, pureQuery also works when you are running Java in any IMS region. From the IMS side, the `.jar` files for pureQuery need to be in the classpath of the JVM running in the IMS region.

pureQuery allows dynamic JDBC applications to use statically bound packages. This capability allows Db2 administrators to use a single authorization environment, while static SQL is authorized for the package and dynamic SQL is authorized for the object. Requiring only one authorization for a single application is a benefit when you have mixed-language applications that interoperate with Java.

It is possible to capture the dynamic JDBC calls, most parts of the application, and run it static. Then to define the additional dynamic JDBC statements which have not been recorded during the capture phase and should be executed, rejected, or recorded and later bound to the static package, the incremental recording.

pureQuery also provides the following functions:

- Fix dirty programming, for example, to not use parameter markers for WHERE clauses, which cause the optimizer to calculate the path for every SQL statement.
- Allow you to bind a WHERE *abc=?* SQL statement to the package and execute all WHERE=*value* statements that have been created in Java as the result of string concatenations as if it were prepared statements with parameter markers.
- Together with IBM Data Studio, it allows for some impact analysis. For example, you can analyze what happens when the name of a column in table *xyz* is changed or which SQLs runs against column *abc*.

In order to exploit pureQuery in an IMS environment, install either of the following:

- IBM Data Studio, which provides an integrated, modular environment for database development. It can be downloaded for free at: http://www.ibm.com/products/ibm-data-studio
- The Eclipse based products can be installed standalone or in an existing Eclipse IDE, including Rational Developer for System z (RDz) and Rational Application Developer (RAD).

Details on pureQuery, its configuration, and how to make it work are discussed in *Using Integrated Data Management To Meet Service Level Objectives,*SG24-7769 [5].

4.6. Accessing IBM MQ from mixed-language applications

When accessing IBM MQ in traditional languages, with either ESAF for MPP and BMP or with RRS for JMP and JBP, it is a matter of the definition in the IMS SSM member. The use of the WebSphere MQ Java classes are not currently are notsupported. The IBM MQ Java classes lack the support for ESAF because it is not implemented. But with the use of RRS in IMS Java dependent regions, the IBM MQ Java classes can be configured to work and be part of the IMS unit of work.

See the example below for a sample code that works in a JMP. It is required to use the option MQPMO_SYNCPOINT for the IBM MQ calls to belong to the IMS unit of work.

The following example is just a code excerpt and not a complete Java class to be executed.

Example 2. MQ sample

```
...
private static final String qManager = "QM01";
private static final String qName = "TEST.QL1";
private static MQQueueManager qMgr = null;
```

```
int openOptions = MQConstants.MQOO_INPUT_AS_Q_DEF | MQConstants.MQOO_OUTPUT;

MQQueue queue = qMgr.accessQueue(qName, openOptions);

MQMessage msg = new MQMessage();
msg.writeUTF("Hello, World!");

MQPutMessageOptions pmo = new MQPutMessageOptions();
pmo.options = MQConstants.MQPMO_SYNCPOINT;

queue.put(msg, pmo);

MQMessage rcvMessage = new MQMessage();

MQGetMessageOptions gmo = new MQGetMessageOptions();
gmo.options = MQConstants.MQGMO_ACCEPT_TRUNCATED_MSG + MQConstants.MQGMO_SYNCPOINT;
gmo.matchOptions = MQConstants.MQMO_NONE;

queue.get(rcvMessage, gmo);

String msgText = rcvMessage.readLine();
System.out.println("The message is: " + msgText);

queue.close();

qMgr.disconnect();
}
catch (MQException ex) {
   System.out.println("A WebSphere MQ Error occured : Completion Code " +
ex.completionCode + " Reason Code " + ex.reasonCode);
   ex.printStackTrace();
   for (Throwable t = ex.getCause(); t != null; t = t.getCause()) {
      System.out.println("... Caused by ");
      t.printStackTrace();
   }
}
catch (java.io.IOException ex) {
   System.out.println("An IOException occurred while writing to the message buffer: " +
ex);
   ex.printStackTrace();
}
return;
```

Subsystem member definition is required to use the IBM MQ Java classes with RRS (See Example 3). The RRS environment is currently only usable with IMS JBP and JMP regions. All other environments use ESAF as attach facility to IBM MQ.

Example 3. Subsystem member definition to use IBM MQ Java for RRS

```
SST=DB2,SSN=WMQA,COORD=RRS
```

In addition to the classes required for IMS Java-based applications and for access to Db2, the following `.jar` files need to be in the IMS region's classpath in order for the above example to work.

- `com.ibm.mq.headers.jar`
- `com.ibm.mq.pcf.jar`
- `jta.jar`
- `connector.jar`
- `com.ibm.mq.commonservices.jar`
- `com.ibm.mq.jar`
- `com.ibm.mq.jmqi.jar`
- `com.ibm.ffdc.jar`

IBM MQ V9 for z/OS allows ESAF environments for transactional access by Java applications that run in IMS based on Java Message Service (JMS).

4.7. COBOL Version 4, 5, and 6 considerations

In COBOL V4 and earlier, you need to have the LIB option in effect to handle copybooks. Starting with COBOL Version 5, the LIB compiler is always on, and does not need to be specified.

4.7.1. Generating methods from Java to copybook structures

This section discusses the different options to pass data between COBOL and Java.

4.7.1.1. Strings

Some customers prefer to simply pass variable-length strings such as key-value pairs between the languages. This generally can be done in two ways:

- Use APIs that create string objects in the JVM based on EBCDIC data (`NewStringPlatform`) to pass the strings to the JVM
- Create null terminated EBCDIC strings based on references to string objects from the JVM (`GetStringPlatform`).

4.7.1.2. JZOS Record Generator

JZOS Record Generator is the most common tool for passing data between COBOL and Java, where copybooks are treated as byte arrays and passed to Java. The un-marshalling into Java fields and data types is done in Java. The JZOS solution can be used from IMS to exploit built-in native JVM functions, which are faster than pure Java data conversions. Although other solutions use the new z/OS batch container, they still do marshalling and un-marshalling when passing data between COBOL and Java.

The JZOS Record Generator uses the ADATA output from the COBOL compiler.

A step-by-step guide for using both JZOS and the J2C approaches is included in the *JZOS for z/OS SDKs Cookbook* [6].

The JZOS Record Generator also allows conversion of assembler CSECTs into Java records, and as such, can be quite useful for processing data that is defined in z/OS as assembler source only. It uses the ADATA output from the assembler.

Unfortunately at the time of writing this book, the generated records only accept Java Byte Arrays as input, so there is no or limited CPU benefit in using Direct Byte Buffers to pass data between Java and traditional languages, because a Java Byte Array copy needs to be created from the Direct Byte Buffer to use the generated records and copied back to the Direct Byte Buffer after a possible change of the data.

4.7.1.3. Rational CICS/IMS Data Binding Wizard (J2C)

The J2C wizards for creating a CICS/IMS Data Binding are part of either IBM Rational Application Developer for Java or IBM Rational Developer for System z with Java (RDz). The standalone RDz does not have the wizards packaged.

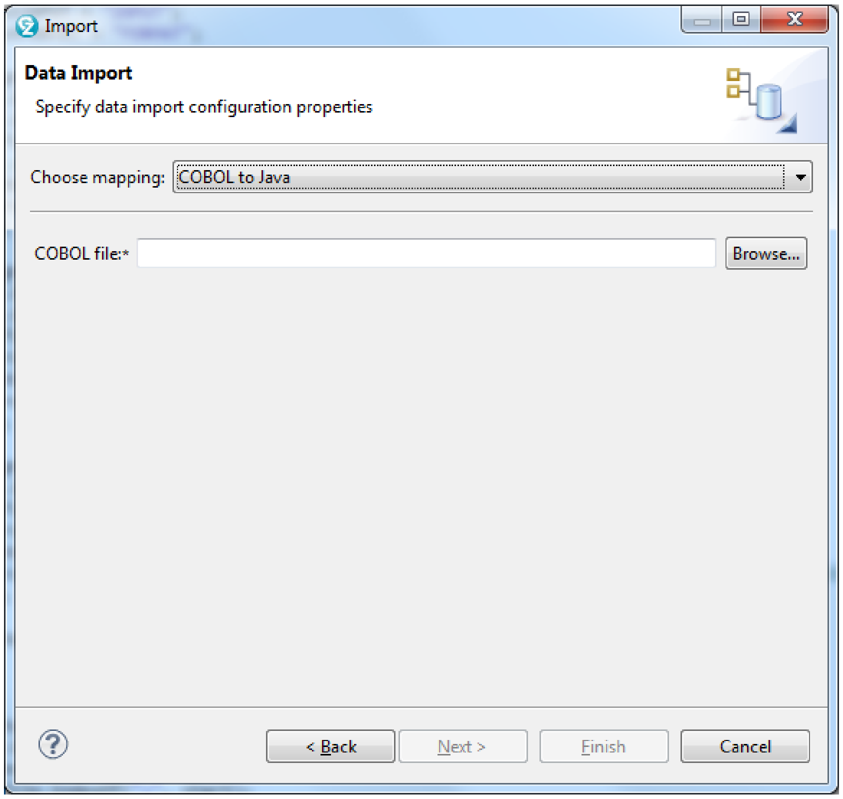
Note that the trial version of the latest Rational Developer for System z that you can download for an evaluation period of 90 days from http://www.ibm.com/developerworks/downloads/r/rdz/ do not have these wizards packaged.

The following steps outlines how to start the J2C wizard and generate the Java classes:

1. The J2C wizard is invoked with the menu option: **File -> New -> Other** and then select **J2C -> CICS/IMS Java Data Binding**.

   A screen shot of the CICS/IMS Java Data Binding Wizard starting with the source data import screen is shown in the following figure.

Figure 21. CICS/IMS Java Data Binding Wizard in IBM Developer for z/OS



Specify data import configuration properties

Choose mapping: COBOL to Java

COBOL file:*

2. Go through the wizard to create:
    - CICS/IMS data binding
    - Import, for example, the COBOL 01 record type
    - Generate the Java class
3. Then the Java class needs to be populated with the byte array that was passed to the Java method from COBOL. For other languages such as, C or PL/I, the process is identical.

Unfortunately, at the time of writing this book, the generated records only accept Java Byte Arrays as input, so there is no or limited CPU benefit in using Direct Byte Buffers to pass data between Java and traditional languages, because a Java Byte Array copy needs to be created from the Direct Byte Buffer to use the generated records and copied back to the Direct Byte Buffer after a possible change of the data.

4.8. Special application requirements

Depending on application scenarios or functional equivalents that the LE runtime, z/OS, or its subsystems (such as IMS and CICS) provide, there are special application requirements.

4.8.1. Preload or initialization for Java objects on JVM startup

Unlike for LE modules, there is no special functionality to allow functions like preload or pre-initialization on the Java side. However, it is possible to run some Java code on IMS region initialization with the `-javaagent` parameter that points to a `.jar` file. This parameter is there to allow for instrumentation of byte code, but when its `premain` method is called, it can run any Java code it wants.

This `.jar` file requires a manifest that points to a class with a `premain` method (`public static void premain(String args, Instrumentation inst)`). As the name indicates, this method is invoked before the first call to execute a main or other method in the JVM.

Note that in the JDK for z/OS, the **`Premain-Class`** parameter needs to be specified with the exact case as printed here. Otherwise, the JVM ends with the message that it cannot find the **`Premain-Class`** parameter in the manifest.

4.9. Summary

In this chapter we discussed application development approaches for mixed-language applications, as well as Java™-only applications. A high-level discussion on how a batch application can use COBOL and Java with interoperability was provided, including options to pass data between languages and how to access Db2 data or WebSphere MQ. Tools for generating access methods from COBOL copybook data structures were also discussed.

4.10. Related resources

The following list provides useful resources that provide more detailed description of the topics that are covered in this section.

1. *IMS 11 Open Database* (SG24-7856-00)
   http://www.redbooks.ibm.com/abstracts/sg247856.html?Open
2. IBM Redbook: *IMS Integration and Connectivity Across the Enterprise*
   http://www.redbooks.ibm.com/redbooks/pdfs/sg248174.pdf
3. Writing your Language Environment-conforming assembler DLL code:
   https://www.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.ceea200/leasdll.htm
4. The syntax for the COBOL INVOKE statement and examples can be found in the Enterprise COBOL for z/OS product documentation.

Enterprise COBOL for z/OS product documentation

5. Using Integrated Data Management To Meet Service Level Objectives (SG24-7769)
   http://www.redbooks.ibm.com/abstracts/sg247769.html

6. *JZOS for z/OS SDKs Cookbook* at:
   https://www.ibm.com/services/forms/preLogin.do?source=zossdkcookbook

CHAPTER 5

5. Bridging from Java to business languages

In this chapter we will discuss in greater details how bridging between Java™ and business languages is done via Java Native Interface (JNI) with concrete examples.

The target audience of this chapter is application developers writing Java applications that call COBOL applications or vice versa.

5.1. What is JNI?

JNI is a native programming interface that is part of the Java Software Development Kit (SDK). This interface defines the infrastructure between Java and other languages, such as C and C++, COBOL, PL/I and assembly. While JNI was developed for C/C++ languages, it is available on z/OS® for all Language Environment-compliant languages.

JNI provides the solution for two-way communications. It defines a method for Java to invoke native methods and a method, via a defined API, for native code to invoke Java code. With the defined API, the native code can manipulate Java objects, call Java methods, and share data with Java.

Enterprise COBOL introduced the concept of Object-Oriented (OO) COBOL that simplifies the interactions between Java and COBOL. Using OO COBOL is now the easiest and most convenient way to create OO applications, in particular when crossing the boundary between Java and COBOL is required.

We will next explain the basics of JNI in the context of Java-COBOL interoperability in z/OS. The same concepts apply to other native and business languages and other platforms. Then, we will describe the main considerations for COBOL and Java interoperability with emphasis on Enterprise COBOL (OO COBOL).

5.2. How JNI works

JNI provides a means for Java code to call native code. This interaction is useful in cases when native capabilities are more efficient than Java or when the specific code is already available in native and you want to use it. JNI provides JNI services (also referred as functions) via the JNI API to act on the Java data and use Java features.

The JNI also provides the native code with means to embed a JVM in the native code and to access Java features, without having to link with the JVM.

5.2.1. JNI services and API

JNI provides many services for the native code to access Java features, such as class operations, calling methods, string and array operations, etc. It also provides services via the Invocation API to embed a JVM in the native application.

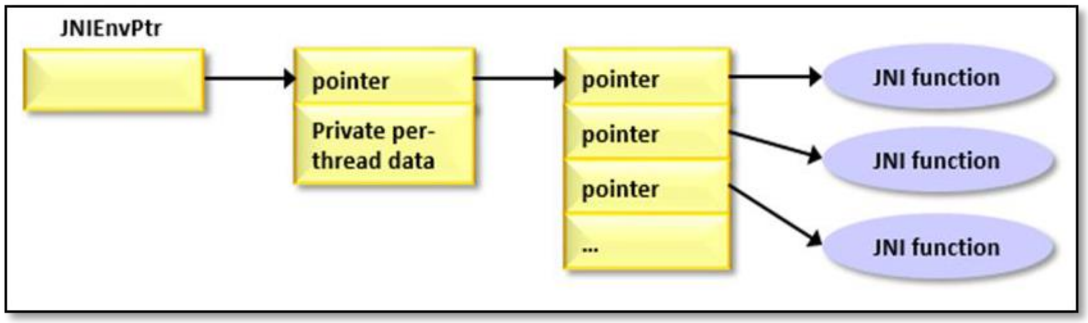
The JNI APIs are defined under a language-specific file that holds all API functions and the type mapping for the Java JNI types. For COBOL, the APIsare defined in the JNI copybook called `JNI.cpy`, which is typically located under the `/usr/lpp/cobol/include` directory. To use the JNI services, the JNIcopybook needs to be included.

There are two main JNI structures defined in the JNI APIs that are key to theJNI execution:

1. The "JNI environment" structure-- the JNI service API.

   Accessing the JNI environment structure is done via two levels of indirection using an interface pointer (in COBOL, it is the special register, JNIEnvPtr). This pointer points to a per-thread JNI environment pointer (JNIEnv, in COBOL it is declared in the JNI copybook) that points to an array of JNI function pointers (a function table named JNINativeInterface) for accessing the callable service functions. The JNI environment pointer is thread-specific and cannot be passed from one thread to another. See the following figure:

Figure 22. JNI environment pointer



2. The "JavaVM" structure-- the JNI Java VM service API that allows to create and destroy the JVM.
   The JavaVM structure is accessed via an interface pointer, JavaVMPtr. This pointer points to a pointer that points to an array of JNI function pointers (a function table named JNIInvokeInterface) for accessing callable "invocation interface" functions.

Java for the z/OS platform provides an extended API to deal with strings due to the encoding differences. It defines the following extensions: NewStringPlatform, GetStringPlatformLength, GetStringPlatform.

These EBCDIC services are packaged as a dynamic link library (DLL) file that is part of the IBM® Java SDK. They are called directly and not dereferenced from the JNIEnv pointer (the JNIEnvPtr). This extended API is resolved through the `libjvm.x` DLL side file, provided with your IBM Java SDK, which needs to be included in the link-edit step.

The section provides the programming guidance for accessing the JNI services from COBOL code.

5.2.2. Making the connection

When an application starts with native code and requires to call Java features via JNI APIs, it first needs to load a JVM using the JNI API, unless it is using OO COBOL. With OO COBOL invoking Java, COBOL will start the JVM for you.

When an application starts with Java, it calls the native language via native methods.

The native methods called from Java are declared in the Java code and implemented in the native language. The name of the native method follows the JNI naming rules. In the simple case, the method would be in the format of `Java_`*`classname`*`_`*`methodname`*. See the JNI Specification [1] for more detailed information regarding native method names and arguments.

As part of the JNI calling convention, the native method has two hidden arguments that are not seen on the Java declaration. Those arguments are predefined by the **JNI:JNIEnv** interface pointer and jobject. In COBOL, the JNIEnv interface pointer is the JNIEnvPtr and it is used to dereference the JNI service. The jobject is a pointer to the class in case the class is static, or to the instance of the class if it is not static.

You can use the *javah* tool to generate .h header files that can be inspected for the signature of the native method generated. These header files are used by C and C++ compilers, not by the COBOL compiler, but you might find them useful for checking the naming of native methods. Although generating the .h files is optional, it has great value in providing the expected signature.

Once the native code is compiled and linked, the native methods are accessed via an executable DLL module that is loaded on the Java side. There are two ways for Java to load the executable DLL module:

- Using `System.loadLibrary(“`*`Name`*`”)`
  In this case the DLL name needs to be in the format of lib*Name*.so.
- Using `System.load("<absolutePath>/`*`Name`*`")`
  In this case there is no restriction on the name of the library.

For proper loading and binding, the executable DLL needs to reside on the z/OS UNIX side, have the right attributes (executable) and follow naming rules.

Example 1: “Hello world” from COBOL” (in the section at the end) provides a sample of Java code invoking a native COBOL method that prints “Hello from COBOL”. The example has no parameters passed to nor no return data from the native method. This example's objective is to demonstrate fundamental building blocks of using JNI to create a healthy Java-native communication. Here are the steps for running this application:

1. Declare the native method in the Java code.
2. Compile the Java code using *javac* to generate a `.class` file.
3. Generate the native code.

   There are two main parts that need special attention: the program-id and the procedure division's parameters and return value. The program-id should follow the JNI rules and match the native method name as declaredon the Java side. In this case, following the `Java_`*`classname`*`_`*`methodname`* convention, the program-id is `Java_HelloTest_printHello`. The native method declared has no arguments but the COBOL procedure division declares the two 'hidden' arguments required for JNI, even though they arenot used in the procedure (as we do not facilitate the JNI services). Note that there is no use of JNI services. Therefore the JNI copybook is not included.
4. Compile the COBOL code with the required options given in the example. These compilation options are described in the “” section.
5. Link-edit step to generate the executable DLL module (following naming rules to allow proper loading). The required options are described in the “” section.
6. Run the Java program.

   If the `System.loadLibrary(“`*`Name`*`”)` method is used, the Java will search afile of the type `lib`*`Name`*`.so` in the library path provided via the `-Djava.library.path` option. If you use the `System.load(...)` method, the path points to the library file.

5.3. Connecting Java and COBOL

This section provides specific information for connecting Java and COBOL. It assumes basic knowledge of the COBOL language including object-oriented (OO) COBOL, and compiling and linking jobs under z/OS UNIX and MVS.

We will first discuss the basics of JNI and how to use it to make the Java-COBOL connection. This methodology does not require the OO COBOL capabilities and thus it also works with legacy COBOL. We will focus on Enterprise COBOL as it provides an extended way to deal with interoperability between Java and COBOL via the OO COBOL.

Enterprise COBOL provides Java-oriented capabilities in addition to the basic OO capabilities available directly in the COBOL language. For example, it allows creation of Java and COBOL classes and invocation of methods on Java and COBOL objects using the INVOKE statement. For basic Java-object capabilities, using the OO capabilities is sufficient for the Java-COBOL interoperability. For additional Java-object capabilities you need the JNI services.

There are several ways to structure the OO application. You can either have COBOL invoking Java or Java invoking COBOL. When starting with COBOL, use the INVOKE statement to invoke a Java method. It is possible to start with COBOL program from both the z/OS UNIX side and the MVS. See the "Communicating with Java methods" topic in the *Enterprise COBOL for z/OS programming guide* [2].

5.3.1. The main method

To start with Java, there needs to be a 'main' method written. It can be written in Java or as a compiled COBOL class definition that contains a factory method called 'main' with the appropriate arguments.

Typically, when using OO COBOL, the easiest way for Java to invoke COBOL is via an OO COBOL 'wrapper' class, which calls the procedural COBOL logic and thus "wraps" it.

Since Java resides on the z/OS UNIX side, if you start with Java, all the application components (classes and executable DLL modules) must reside on the HFS. To launch Java, you can use the shell command prompt (running the *java* command) or with the BPXBATCH utility from JCL or TSO/E.

5.3.2. Working with 'wrapper' OO COBOL classes

A COBOL source file that contains a class definition is often the gateway between Java and COBOL. For details on OO COBOL and class definition see Writing object-oriented programs in the Enterprise COBOL for z/OS Programming Guide [3].

Once this COBOL source file is compiled, a Java file is generated on the USS side. This Java file contains the native method declaration based on the JNI rules. If, for example, the class had a method **foo**, then the generated Java code will have a declaration of a native method named `Java_`*`classname`*`_`**`foo`**. The generated Java code (.java file) needs to be compiled on the USS side by using the *javac* compiler to produce the .class file.

When using OO COBOL with class definition, there are these naming rules:

- The name of the resulting DLL module needs to match the expected class name. If *ClassName* is the external class name, then the name of the DLL module must be `lib`*`ClassName`*`.so`, as this is the name expected in the generated Java file (as it uses `System.loadLibrary` method to load the DLL).

- If the class is part of a package and thus there are periods in the external class name, the periods should be changed to underscores in the DLL module name (based on JNI naming rules).

5.3.3. Accessing JNI services

In COBOL the JNI API is defined in the JNI copybook called `JNI.cpy`. As mentioned earlier in the “” section, the JNI copybook contains the definition of JNINativeInterface, the COBOL group structure that maps the JNI environment structure, which contains an array of function pointers for the JNI callable services. To facilitate access to JNI services, the `JNI.cpy` file should be defined in COBOL program under the LINKAGE SECTION. For example, if the file is named *JNI* :

```
Linkage section.
COPY “JNI”
```

Before you reference the contents of the JNI environment structure, you must code the following statements to establish its addressability:

```
Procedure division.
Set address of JNIEnv to JNIEnvPtr
Set address of JNINativeInterface to JNIEnv
```

The code sets the address of JNIEnv, a pointer data item that `JNI.cpy` provides, and JNIEnvPtr, the COBOL special register that contains the environment pointer. The JNIEnvPtr is implicitly defined as USAGE POINTER and should not be used as a receiving data item. Use this special register JNIEnvPtr to reference the JNI environment pointer to obtain the address for the JNI environment structure.

After the pointers are set, the JNI callable services can be accessed with CALL statements that reference the function pointers. The JNIEnvPtr special register is the first argument to the services that require the environment pointer, as shown in the following example:

```
01 InputArrayObj usage object reference jlongArray.
01 ArrayLen pic S9(9) comp-5.
. . .
Call GetArrayLength using by value JNIEnvPtr
InputArrayObj
                  returning ArrayLen
```

Note: Pass all arguments to the JNI callable services by value.

5.3.4. Compiling and linking for COBOL

To compile COBOL source code that uses JNI services or contains OO syntax such as INVOKE statements or class definitions (that is, COBOL code that directly communicates with Java):

- Use the following compiler options: RENT, DLL, THREAD, and DBCS. The RENT and DBCS options are IBM-supplied defaults.
- Set PGMNAME(LONGMIXED) for long name support.
- Set the RECURSIVE attribute on COBOL classes and methods or on COBOL programs that invoke Java methods.

Compiling the COBOL with DLL affects the overall COBOL program structure. In general, DLL-linkage-built COBOL programs can only call out to other external DLL-linkage-built programs. Similarly, dynamic call-built COBOL programs can only call out to other external dynamic call built programs.

However, static linking of objects with two of these external program call mechanisms is allowed. This provides the bridging between the DLL linkage that Java requires and the traditional COBOL dynamic call. For more information regarding the DLL consideration see:

- "DLL considerations for COBOL and Java" in the *z/OS Batch Runtime Planning and User's Guide* [4].
- Section 2.9.1 in "How Java can call COBOL and vice versa" in "*New Ways of Running Batch Applications on z/OS: Vol 4 IBM IMS*" [5].

The link step creates an executable DLL module. It is required to link the object file with the following two DLL side files:

- `libjvm.x`, which is provided with your IBM Java SDK.
- `igzcjava.x`, which is provided in the `lib/` subdirectory of the COBOL directory in the HFS. The typical complete path is `/usr/lpp/cobol/lib/igzcjava.x`.
  This DLL side file is also available as the member IGZCJAVA in the SCEELIB PDS (part of Language Environment).

If the application starts with a Java program or the main factory method of a COBOL class, the XPLINK environment is automatically started by the *java* command that starts the JVM and runs the application.

If an application starts with a COBOL program that invokes methods on COBOL or Java classes, you must specify the XPLINK(ON) runtime option so that the XPLINK environment is initialized. XPLINK(ON) is not recommended as a default setting. Use XPLINK(ON) only for applications that specifically require it.

In older versions of COBOL, using the COPY statement required the compiler option 'lib'. This option is not required in Enterprise COBOL v5 and up, and it is always in effect.

For more information, see Compiling, linking, and running OO applications in the Enterprise COBOL for z/OS Programming Guide [6].

5.3.4.1. Java and COBOL under z/OS UNIX

When you compile, link, and run OO applications in a z/OS UNIX environment, application components reside in the HFS. The compilation and linking is done using z/OS UNIX shell commands, and application is launched via a shell command prompt or with the BPXBATCH utility from JCL or TSO/E.

Compiling and linking options are as mentioned in “”: RENT, DLL, THREAD, and DBCS (RENT and DBCS are IBM-supplied defaults), and long name support (longmixed).

For compilation use: `cob2 -c -qdll,thread`

For linking use: `cob2 -bdll`

The -bdll option specifies that the executable module is to be a DLL and also:

- The COBOL compiler uses the compiler options DLL, EXPORTALL, and RENT, which are required for DLLs.
- The link step produces a DLL definition side file that contains IMPORT control statements for each of the names exported by the DLL.

You need to specify the include subdirectory of the `JNI.cpy` by using the -I option of the **cob2** command or by setting the SYSLIB environment variable. The `JNI.cpy` resides under the include subdirectory of the COBOL install directory (typically, `/usr/lpp/cobol/include`).

See “Example 2: Java calling procedural COBOL with JNI service calls (USS)” on page 89 for Java calling procedural COBOL with JNI service calls.

5.3.4.1.1. Java invoking OO COBOL with class definition on USS

When you compile a COBOL class definition, there are two output files generated:

- The object file (`.o` file) for the class definition.
- A Java source program (.java) that contains a class definition that corresponds to the COBOL class definition.

Compile the generated Java source with the Java compiler to create a class file (`.class`).

The class file generated and the DLL module generated (after linking) are the executable components of the OO COBOL application, and are generated in the current working directory.

If you change the COBOL class definition, you must regenerate both the object file and the Java class definition by recompiling the updated COBOL class definition.

When Java classes are involved, set the CLASSPATH environment variable to contain the path to the necessary classes.

For OO applications that start with Java, you need the LIBPATH environment variable to contain the path to the executable DLL module.

For more details see "Compiling, linking, and running OO applications under z/OS UNIX" in *Enterprise COBOL for z/OS 6.1.0 Programming Guide* [7].

See "Example 3: Java calling OO COBOL with class definition (USS)" on page 92 for an application that starts with Java and has OO COBOL with class definition.

5.3.4.1.2. OO COBOL application calling Java

For COBOL applications calling Java, you need the LIBPATH to point to the path for the Java libraries.

```
LIBPATH=/usr/lpp/java/IBM/J7.1/bin/classic:$LIBPATH
```

Also, use the _CEE_RUNOPTS environment variable to set the XPLINK(ON)option :

```
_CEE_RUNOPTS="XPLINK(ON)"
```

Exporting _CEE_RUNOPTS="XPLINK(ON)" so that it is in effect for the entire z/OS UNIX shell session is not recommended, however. Suppose for example that an OO COBOL application starts with a COBOL program called App1.

One way to limit the effect of the XPLINK option to the execution of the App1 application is to set the _CEE_RUNOPTS variable on the command-line invocation of App1Driver as follows:

```
_CEE_RUNOPTS="XPLINK(ON)" App1
```

See "Example 4: OO COBOL application invoking Java using INVOKE (USS)" on page 95 for OO COBOL application calling Java using INVOKE.

5.3.4.2. Java and COBOL under MVS (JCL or TSO/E)

Many customers have their COBOL code on the MVS side. Here we provide the details of performing compilation/linking and running the application under MVS using JCL. Similarly one can use TSO/E for compiling/linking steps. For more information, see "Compiling, linking, and running OO applications in JCL or TSO/E" in *Enterprise COBOL for z/OS 6.1.0 ProgrammingGuide* [8].

Compiling and linking options are as mentioned in "": RENT, DLL, THREAD, and DBCS. Again, RENT and DBCS are defaults and specify long name support (longmixed).

To facilitate and exploit access to JNI services, copy the file `JNI.cpy` from the HFS to a PDS or PDSE member called JNI, identify that library with a SYSLIB DD statement, and use a COPY statement of the form COPY "JNI" in the COBOL source.

The generated object file is written, as usual, to the data set that has ddname SYSLIN or SYSPUNCH.

Link the load module for the COBOL program into a PDSE. COBOL programs that contain object-oriented syntax must be link-edited with AMODE 31.

To link with the DLL side files `libjvm.x` and `igzcjava.x` when binding the object deck for the main program specify INCLUDE control statements. For example:

```
INCLUDE '/usr/lpp/java/IBM/J7.1/bin/classic/libjvm.x'
INCLUDE '/usr/lpp/cobol/lib/igzcJava.x'
```

See "Example 5: Java calling procedural COBOL (MVS)" on page 96 for Java calling procedural COBOL on MVS.

When compiling a COBOL class definition, there are two outputs generated:

1. The object module for the COBOL class definition – on the MVS side
2. A Java source program (.java) that contains a class definition that corresponds to the COBOL class definition. Ensure that the .java file for OO COBOL reside on the HFS. Use the SYSJAVA ddname to write the generated Java source file to a file in the HFS. Make sure to name the Java file with *`Classname`*`.java` where *`Classname`* is the COBOL class defined (this is expected by the Java) and have the right file permissions. For example:

```
//SYSJAVA DD PATH='/u/userid/java/Classname.java',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=SIRWXU,
// FILEDATA=TEXT
```

Compile Java class definitions by using the *javac* command from a z/OS UNIX shell command prompt, or by using the BPXBATCH utility.

If you change the COBOL class definition, you must regenerate both the object file and the Java class definition by recompiling the updated COBOL class definition.

When Java classes are involved, set the CLASSPATH environment variable to contain the path to the necessary classes.

For OO COBOL with class definitions the executable DLL module needs to reside in the HFS. Use the SYSLMOD option with PATH parameter. Also, make sure to follow the naming rules for the module as discussed in the “” section, and have the right file attributes, and access privilege to the file. For example:

```
//SYSLMOD DD PATH='/u/userid /java/libTSTJNI.so',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=(SIRWXU,SIWUSR,SIRGRP)
```

See “Example 6: Java calling OO COBOL with class definition (MVS)” on page 98 for an application that starts with Java and has OO COBOL with class definition.

5.3.4.2.1. OO COBOL calling Java via INVOKE

When you run an OO application that starts with a COBOL program, use the _CEE_ENVFILE environment variable to indicate the location of a file that contains the environment variable settings required by Java. Set _CEE_ENVFILE by using the ENVAR runtime option. Also specify the POSIX(ON) and XPLINK(ON) runtime options. For example:

```
// PARM=' /ENVAR("_CEE_ENVFILE=/u/userid/ENV")
//              XPLINK(ON) POSIX(ON)'
```

Environment variable required for Java, such as PATH, LIBPATH and CLASSPATH are set in a file on the HFS side. To customize the initialization of the JVM that will be used by the application, you can set the COBJVMINITOPTIONS environment variable in the same file. For example, a `/u/userid/javaenv` file might look as follows:

```
PATH=/bin:/usr/lpp/java/IBM/J7.1/bin
LIBPATH=/lib:/usr/lib:/usr/lpp/java/IBM/J7.1/bin:/usr/lpp/java/IBM/J7.1
/bin/classic:/u/userid/applications
CLASSPATH=.:/u/userid/applications
COBJVMINITOPTIONS="-Xms10000000 -Xmx20000000 -verbose:gc"
```

Also, use DD statements to specify files in the HFS for the standard input, output, and error streams for Java:

- JavaIN DD for the input from statements such as `c=System.in.read();`
- JavaOUT DD for the output from statements such as `System.out.println(string);`
- JavaERR DD for the output from statements such as `System.err.println(string);`

Ensure that the SCEERUN2 and SCEERUN load libraries are available in the system library search order, for example, by using a STEPLIB DD statement.

See "Example 7: COBOL invoking Java via INVOKE (MVS)" on page 101 for an OO COBOL application calling Java using INVOKE.

5.4. Invoking Java from native COBOL

If you can't use OO COBOL or DLL in your environment because of restrictions in the LE (e.g. calling the same program from non-DLL and DLL) or simply because of the fact that you need to make a static call, all you need to do is replace the INVOKE statement. This is the only statement that requires the DLL option.

However, without the INVOKE statement, you need to call Java another way. That means you have to use JNI services such as `FindClass`, `GetStaticMethodID` and `GetMethodID`.

The downside of not using the INVOKE statement is, you will have to write a lot more code with JNI calls and handle the encoding from EBCDIC to UTF-8 by yourself.

On the other hand, you can now cache the references of your classes and methods so you don't need to make unnecessary JNI calls, which INVOKE does every time.

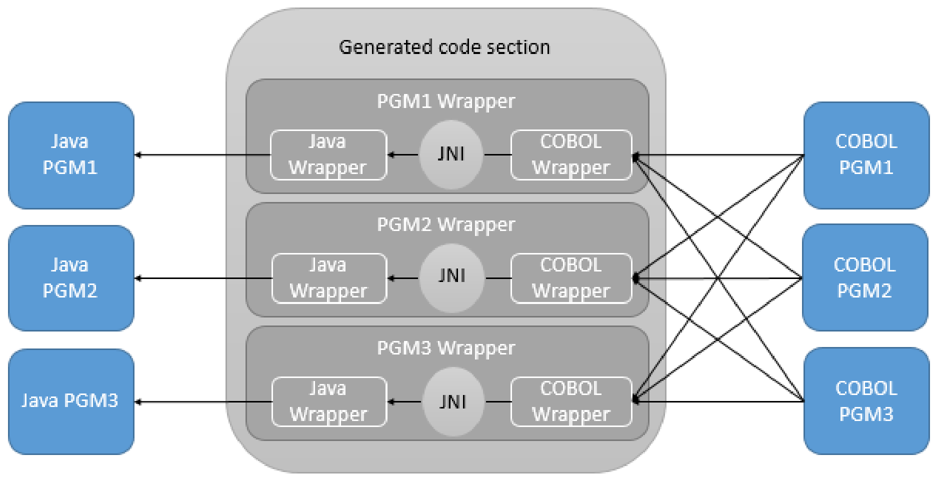
5.5. Considerations using native COBOL with JNI

The complexity of JNI calls can be a little bit overwhelming for an average programmer. You should consider taking this complexity to a framework, generate your JNI code, or generate both COBOL and Java code according to information that is stored in a metadata repository. If you choose the last approach to generate both COBOL and Java code based on some metadata, you must develop your own solution because there are no existing products that would do the generation for you.

5.5.1. Generating JNI code

Generating the JNI calls for COBOL is one way to hide the complexity. You could use Java classes with annotations to fill necessary information for the generator, or you generate the COBOL as well as the Java classes via your own domain-specific language.

Figure 23. Generating JNI calls for COBOL



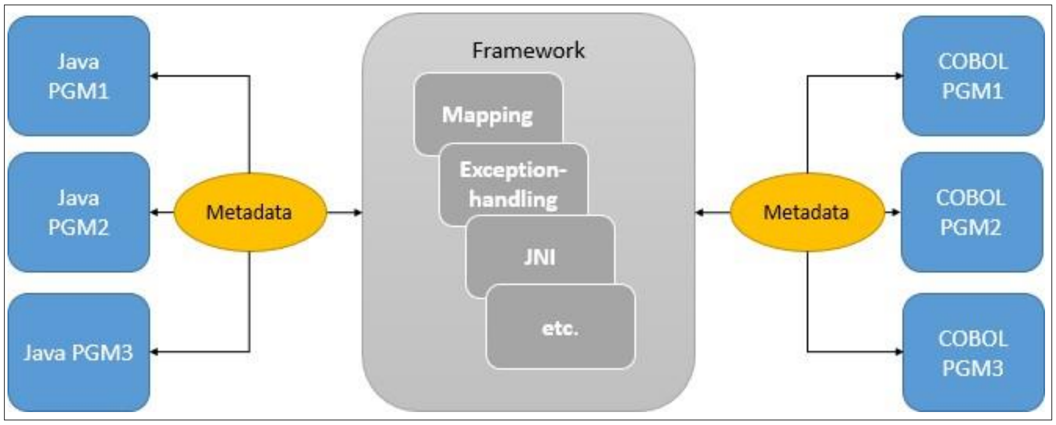
The disadvantage of this method is, you generate many artifacts that you have to manage. If you want to, for example, change something in your generated code, you would have to keep track of all your generated applications with JNI calls, generate your new code, and test it.

5.5.2. Framework

Another way to hide complexity is to write some framework code to take the complexity from the programmers.

The biggest challenge doing this is the mapping of the COBOL copybooks and the Java objects. The best solution in this case is to model the data types and generate copybooks, Java objects, and the associated mapper.

Figure 24. Framework



5.5.3. Using static methods

Use static methods instead of object methods because creating objects via JNI is a more troublesome task than simply calling a static method.

As an example, we try to call a method with no parameters or return values. What you now need to do first is the same procedure regardless of static or no static (except that for the JNI service you need to call `GetStaticMethodID` or `GetMethodID)`, you need to get the references for the class and the method you want to call via JNI services.

Now if you want to call the static method, congratulations-- you have everything you need. Just use the JNI service `CallStatic<`*`type`*`>Method` with the right references, and the method will be executed. In the following example, *`<type>`* is replaced with Void because there is no return value.

```
           Call CallStaticVoidMethod
                Using By Value JNIEnvPtr
                               Class-Ref
                               Static-Method-Ref
```

If you don't want to call the static method, then you need some additional work. First you need to create an object from the class. An object is created when you call the constructor from this specific class. To call the constructor you have to get the reference via JNI service `GetMethodID` with the method name "`<init>`" and the right signature.

```
           Move '<init>' To Method-Name
      *    No parameters for constructor => (),
      *    return value always V => void
           Move '()V'    To Signature
      *    Now convert from EBCDIC to UTF-8
…
```

```
           Call GetMethodID
                Using By Value JNIEnvPtr
                               Class-Ref
                               Method-Name-UTF8-Ptr
                               Signature-UTF8-Ptr
                    returning Constructor-Ref
```

When you have the constructor you have to create a new object with the `NewObject` service, which needs the constructor and the parameters.

```
           Call NewObject
                Using By Value JNIEnvPtr
                               Class-Ref
                               Constructor-Ref
                    returning Local-Object-Ref
      *              Add constructor parameters here if needed
```

After these steps you can finally call the method by using the JNI service `Call<type>Method` (just like the `Call<type>StaticMethod`, here we replace <type> with Void).

```
           Call CallVoidMethod
                Using By Value JNIEnvPtr
                               Object-Ref
                               Method-Ref
```

Note: If you want to use your Object again you should make it global via the JNI service `NewGlobalRef and` remove the local reference with `DeleteLocalRef` because local references are freed after the return of the native method.

```
           Call NewGlobalRef
              Using By Value JNIEnvPtr
                             Local-Object-Ref
              Returning Object-Ref
…
           Call DeleteLocalRef
              Using By Value JNIEnvPtr
                             Local-Object-Ref
```

5.6. Connecting Java and PL/I

The connection between PL/I and Java is a bit different compared to COBOL. There is no implicit attachment to the JVM, so some of the things that the COBOL run time does need to be coded explicitly by the PL/I programmer.

The main difference for PL/I in comparison to COBOL is that PL/I distinguishes between main and sub modules. Since the JVM in IMS regions is brought up as part of a CEEPIPI environment, only PL/I subs are allowed in JVM-enabled IMS dependent regions. This applies also to the “main” program.

This means that current PL/I main module will fail to execute in an IMS region that is configured to include a JVM. On the other hand, “main” PL/I programs that are compiled as subroutine modules will fail to execute in an IMS region without the JVM. The LE enclave will be built by the application and therefore is required to be a PL/I main module.

5.7. Examples

This section contains all the examples discussed in this chapter. Each example includes a COBOL program, a Java program, and the commands to compile, link, and run the programs.

5.7.1. Example 1: “Hello world” from COBOL

The following is a simple COBOL program that prints “Hello from COBOL “.

```
helloTest.cbl
Process pgmname(longmixed),lib,dll,thread
       IDENTIFICATION DIVISION.
       PROGRAM-ID.    "Java_HelloTest_printHello" is recursive.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       LINKAGE SECTION.
       01 ENV-PTR   USAGE POINTER.
       01 OBJECT-REF  PIC S9(9) BINARY.
       PROCEDURE DIVISION USING BY VALUE ENV-PTR
                                OBJECT-REF.
              Display " >> Hello from COBOL ".
              GOBACK.
```

HelloTest.java

```
public class HelloTest
{
   public native void printHello();
   public static void main(String argv[]) throws Exception
  {
    System.loadLibrary("Hello");
    HelloTest newHelloTest = new HelloTest();
    newHelloTest.printHello();
      return;
   }
}
```

The COBOL program is compiled and linked as follows, where a typical installation path for COBOL is `/usr/lpp/cobol and` a typical installation path for Java is `/usr/lpp/java` Replace with your own paths for your environment.

```
/usr/lpp/cobol/bin/cob2 -c -qdll,thread helloTest.cbl

/usr/lpp/cobol/bin/cob2 -bdll -o libHello.so helloTest.o
/usr/lpp/java/IBM/J7.1/bin/j9vm/libjvm.x /usr/lpp/cobol/lib/igzcjava.x
helloTest.cbl

/usr/lpp/java/IBM/J7.1/bin/javac HelloTest.java

/usr/lpp/java/IBM/J7.1/bin/java -Djava.library.path=. HelloTest
```

5.7.2. Example 2: Java calling procedural COBOL with JNI service calls (USS)

stringTest.cbl

```
Process pgmname(longmixed),lib,dll,thread

       IDENTIFICATION DIVISION.
       PROGRAM-ID.    "Java_StringTest_printStrings" is recursive.

       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.

       INPUT-OUTPUT SECTION.
       DATA DIVISION.
       Local-storage section.
       01 I pic s9(9) binary.
       01 P pointer.
       01 SAelement pic s9(9) binary.
       01 SAelementlen pic s9(9) binary.
       01 Sbuffer pic X(50) .
```

```
01 rc Pic s9(9) Comp-5.



LINKAGE SECTION.
01 SA PIC S9(9) BINARY.
01 SAlen pic s9(9) binary. 01 ENV-PTR USAGE POINTER.
01 OBJECT-REF PIC S9(9) BINARY.

COPY JNI SUPPRESS.

PROCEDURE DIVISION USING BY VALUE ENV-PTR //two JNI 'hidden' parms

                              OBJECT-REF
                              SA
                   RETURNING SAlen.

Set address of JNIEnv to ENV-PTR
Set address of JNINativeInterface to JNIEnv



Call GetArrayLength using by value ENV-PTR SA
   returning SAlen
Display " >> COBOL method entered array len: " SAlen

Perform varying I from 0 by 1 until I = SAlen
   Call GetObjectArrayElement
      using by value ENV-PTR SA I
      returning SAelement
   Call "GetStringPlatformLength"
      using by value ENV-PTR
                     SAelement
                     address of SAelementlen
                     0
      returning rc
Display "Returned from GetStringPlatformLength"
If rc Not = zero Then
Display "Error occurred retrieving len of jstring object"
Stop run
Else
Display "The length of returned string is:"
SAelementlen
End-if
Call "GetStringPlatform"
```

```
                 using by value ENV-PTR
                                SAelement
                                address of Sbuffer
                                length of Sbuffer
                                0
              Display Sbuffer(1:SAelementlen)
           End-perform

            .
           GOBACK.
```

StringTest.java

```
class StringTest
{
   static {
     System.loadLibrary("StringTest");
   }

static native int printStrings(java.lang.String[] SA);

public static void main(String argv[]) throws Exception
   {
    int i;
    String[] a = new String[3];
    a[0] = "John";
    a[1] = "White";
    a[2] = "1234567890";
    StringTest st = new StringTest();
    i = StringTest.printStrings(a);
    System.out.println("Number of elements printStrings read:" +i);
    return;
   }
}
```

The COBOL program is compiled and linked as follows, where a typical installation path for COBOL is `/usr/lpp/cobol and` a typical installation path for Java is `/usr/lpp/java` Replace with your own paths for your environment.

```
/usr/lpp/cobol/bin/cob2 -c -qdll,thread -I /usr/lpp/cobol/include
stringTest.cbl
```

```
/usr/lpp/cobol/bin/cob2 -bdll -o libStringTest.so  stringTest.o
/usr/lpp/java/IBM/J7.1/bin/j9vm/libjvm.x /usr/lpp/cobol/lib/igzcjava.x
-I /usr/lpp/cobol/include stringTest.cbl
```

```
/usr/lpp/java/IBM/J7.1/bin/javac StringTest.java

/usr/lpp/java/IBM/J7.1/bin/java -Djava.library.path=. StringTest
```

5.7.3. Example 3: Java calling OO COBOL with class definition (USS) TSTJNI.cbl

```
cbl dll,thread,lib,pgmname(longmixed)
       Identification division.
       Class-id. TSTJNI inherits Base.
       Environment Division.
       Configuration section.
       Repository.
           Class Base is "java.lang.Object"
           Class stringArray is "jobjectArray:java.lang.String"
           Class jstring is "java.lang.String"
           Class TSTClass is "TSTJNI".

       Identification Division.
       Factory.
         Procedure division.
         Identification Division.
         Method-id. "foo".
         Data division.
         Local-storage section.
         01 I pic s9(9) binary.
         01 P pointer.
         01 SAelement object reference jstring.
         01 SAelementlen pic s9(9) binary.
         01 Sbuffer pic X(50) .
         01 rc Pic s9(9) Comp-5.

         Linkage section.
         01 SA object reference stringArray.
         01 SAlen pic s9(9) binary.

        COPY JNI.
```

```
Procedure division using by value SA
    returning SAlen.

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

   Call GetArrayLength using by value JNIEnvPtr SA
      returning SAlen
   Display " >> COBOL method entered array len: " SAlen

   Perform varying I from 0 by 1 until I = SAlen
      Call GetObjectArrayElement
         using by value JNIEnvPtr SA I
         returning SAelement
      Call "GetStringPlatformLength"
         using by value JNIEnvPtr
                        SAelement
                        address of SAelementlen
                        0
         returning rc
      Display "Returned from GetStringPlatformLength"
      If rc Not = zero Then
      Display "Error occurred retrieving len of jstring object"
      Stop run
      Else
      Display "The length of returned string is:"
      SAelementlen
      End-if
      Call "GetStringPlatform"
         using by value JNIEnvPtr
                        SAelement
                        address of Sbuffer
                        length of Sbuffer
                        0
      Display Sbuffer(1:SAelementlen)
   End-perform


   .
 End method "foo".
 End factory.
 End class TSTJNI.
```

CallingCobol.java

```
class CallingCobol
{
   public static void main(String argv[]) throws Exception
   {
      int i;
      String[] a = new String[3];
      a[0] = "John";
      a[1]  =  "White";
      a[2] = "1234567890";

      System.out.println("In Java main, before calling cobol");
      i = TSTJNI.foo(a);
      System.out.println("Number of elements foo read:" + i);
      System.out.println("In Java main, after calling cobol.");
      return;
   }
}
```

The COBOL program is compiled and linked as follows, where a typical installation path for COBOL is `/usr/lpp/cobol and` a typical installation path for Java is `/usr/lpp/java` Replace with your own paths for your environment.

```
/usr/lpp/cobol/bin/cob2 -c -qdll,thread -I /usr/lpp/cobol/include
TSTJNI.cbl
```

```
/usr/lpp/cobol/bin/cob2 -bdll -o libTSTJNI.so  TSTJNI.o
/usr/lpp/java/IBM/J7.1/bin/j9vm/libjvm.x /usr/lpp/cobol/lib/igzcjava.x
-I  /usr/lpp/cobol/include  TSTJNI.cbl
```

```
/usr/lpp/java/IBM/J7.1/bin/javac TSTJNI.java

/usr/lpp/java/IBM/J7.1/bin/javac CallingCobol.java

/usr/lpp/java/IBM/J7.1/bin/java -Djava.library.path=. CallingCobol
```

5.7.4. Example 4: OO COBOL application invoking Java using INVOKE (USS)

helloTest.cbl

```
       Process thread,pgmname(longmixed)
       IDENTIFICATION DIVISION.
        PROGRAM-ID. "HELLOWORLD" is recursive.
      *
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER. RM-COBOL.
       OBJECT-COMPUTER. RM-COBOL.

       Repository.
           Class HelloJ is "HelloJ".
       DATA DIVISION.
       FILE SECTION.

       PROCEDURE DIVISION.

       MAIN-LOGIC SECTION.
       BEGIN.
           DISPLAY " " .
           INVOKE HelloJ "sayHello".
           DISPLAY "Hello world!".
           STOP RUN.
       MAIN-LOGIC-EXIT.
           EXIT.
```

HelloJ.java

```
class HelloJ {
   public static void sayHello() {
     System.out.println("Hello World, from Java!");
   }
}
```

The COBOL program is compiled and linked as follows, where a typical installation path for COBOL is `/usr/lpp/cobol and` a typical installation path for Java is `/usr/lpp/java`. Replace with your own paths for your environment.

```
/usr/lpp/cobol/bin/cob2 -c -qthread,dll  helloTest.cbl
```

```
/usr/lpp/cobol/bin/cob2 -bdll  helloTest.cbl -o hello
/usr/lpp/java/IBM/J7.1/bin/j9vm/libjvm.x
/usr/lpp/cobol/lib/igzcjava.x -I /usr/lpp/cobol/include
```

```
/usr/lpp/java/IBM/J7.1/bin/javac HelloJ.java
```

```
export _CEE_RUNOPTS="XPLINK(ON)"

export
LIBPATH=/usr/lpp/java/IBM/J7.1/bin:/usr/lpp/java/IBM/J7.1/bin/classic:
$LIBPATH

export CLASSPATH=<path to Class>:$CLASSPATH
```

5.7.5. Example 5: Java calling procedural COBOL (MVS)

```
//TSTJNI JOB ,
//  TIME=(1),MSGLEVEL=(1,1),MSGCLASS=H,CLASS=A,REGION=100M,
//  NOTIFY=&SYSUID
//*
// SET COBPRFX='SHARE.IGY410'
// SET LIBPRFX='CEE'
//*
//COMPILE EXEC PGM=IGYCRCTL,
// PARM='RENT,PGMN(LM),DLL,EXPORTALL'
//SYSLIN    DD DSNAME=IBARON.REDBOOK.OBJECT(TSTJNI),DISP=SHR
//SYSLIB    DD DSN=IBARON.REDBOOK.SOURCE,DISP=SHR
       //JNI copy book location
//SYSPRINT DD SYSOUT=*
//STEPLIB  DD DSN=&COBPRFX..SIGYCOMP,DISP=SHR
//          DD DSN=&LIBPRFX..SCEERUN,DISP=SHR
//SYSUT1    DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT2    DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT3    DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT4    DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT5    DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT6    DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT7    DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSIN     DD *
       Process dll,thread,lib,pgmname(longmixed)
       IDENTIFICATION DIVISION.
       PROGRAM-ID.    "Java_Hello_dis" is recursive.
       ENVIRONMENT DIVISION.

       DATA DIVISION.
       LINKAGE SECTION.

       COPY JNI.

       01 ENV-PTR              USAGE POINTER.
       01 OBJECT-REF  PIC S9(9) BINARY.


        PROCEDURE DIVISION USING BY VALUE ENV-PTR
                                  OBJECT-REF.

              DISPLAY "Hello from COBOL".
              GOBACK.

/*
//LKED EXEC PGM=IEWL,
//     PARM='RENT,LIST,XREF,LET,MAP,DYNAM(DLL),CASE(MIXED)'
//SYSLIB    DD DSN=&LIBPRFX..SCEELKED,DISP=SHR
//          DD DSN=&LIBPRFX..SCEELKEX,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTERM  DD SYSOUT=*
//OBJMOD    DD DSN=IBARON.REDBOOK.OBJECT,DISP=SHR
//SYSLMOD  DD  PATH='/u/ibaron/sandbox/Redbook/JavaCobol/libTSTJNI.so',
//              PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//              PATHMODE=(SIRWXU,SIWUSR,SIRGRP)
//SYSLIN    DD *
  INCLUDE OBJMOD(TSTJNI)
  INCLUDE '/usr/lpp/java/IBM/J7.1/bin/classic/libjvm.x'
  INCLUDE '/usr/lpp/cobol/igy410/lib/igzcjava.x'
```

Hello.java

```
class Hello {
   static {
      System.loadLibrary("TSTJNI");
   }

   static native void dis();
```

```
                              public static void main(String[ ] args) {

     Hello hello = new Hello();
     hello.dis();
   }
 }
```

The Java program is compiled and run as follows, where a typical installation path for Java is `/usr/lpp/java` Replace with your own path for your environment.

```
/usr/lpp/java/IBM/J7.1/bin/javac Hello.java

/usr/lpp/java/IBM/J7.1/bin/java -Djava.library.path=. Hello
```

5.7.6. Example 6: Java calling OO COBOL with class definition (MVS)

```
//TSTJNI JOB ,
//  TIME=(1),MSGLEVEL=(1,1),MSGCLASS=H,CLASS=A,REGION=100M,
//  NOTIFY=&SYSUID
//*
// SET COBPRFX='SHARE.IGY410'
// SET LIBPRFX='CEE'
//*
//COMPILE EXEC PGM=IGYCRCTL,
// PARM='SIZE(5000K)'
//SYSLIN   DD DSNAME=IBARON.REDBOOK.OBJECT(TSTJNI),DISP=SHR
//SYSLIB   DD DSN=IBARON.REDBOOK.SOURCE,DISP=SHR
//SYSPRINT DD SYSOUT=*
//STEPLIB  DD DSN=&COBPRFX..SIGYCOMP,DISP=SHR
//         DD DSN=&LIBPRFX..SCEERUN,DISP=SHR
//         DD DSN=IBARON.REDBOOK.SOURCE,DISP=SHR
//SYSUT1   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT2   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT3   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT4   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT5   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT6   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT7   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSJAVA DD PATH='/u/ibaron/sandbox/Redbook/JavaCobol/TSTClass.java',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=SIRWXU,
// FILEDATA=TEXT
//SYSIN    DD *
       cbl dll,thread,lib,pgmname(longmixed)
       Identification division.
       Class-id. TSTClass inherits Base.
       Environment Division.
       Configuration section.
       Repository.
           Class Base is "java.lang.Object"
           Class stringArray is "jobjectArray:java.lang.String"
           Class jstring is "java.lang.String"
           Class TSTClass is "TSTClass".

       Identification Division.
       Factory.
         Procedure division.
*
         Identification Division.
         Method-id. "foo".
         Data division.
         Local-storage section.
         01 I pic s9(9) binary.
         01 P pointer.
         01 SAelement object reference jstring.
         01 SAelementlen pic s9(9) binary.
         01 Sbuffer pic X(50) .
         01 rc Pic s9(9) Comp-5.
         Linkage section.
         01 SA object reference stringArray.
         01 SAlen pic s9(9) binary.
         Copy "JNI" suppress.



        Procedure division using by value SA
            returning SAlen.
           Set address of JNIEnv to JNIEnvPtr
           Set address of JNINativeInterface to JNIEnv
           Call GetArrayLength using by value JNIEnvPtr SA
              returning SAlen
           Display " >> COBOL method entered array len: " SAlen
           Perform varying I from 0 by 1 until I = SAlen
              Call  GetObjectArrayElement
                 using by value JNIEnvPtr SA I
                 returning SAelement
           Call "GetStringPlatformLength"
                 using by value JNIEnvPtr
                                SAelement
                                address of SAelementlen
                                0
                 returning rc
              Display "Returned from GetStringPlatformLength"
              If rc Not = zero Then
              Display "Error occurred retrieving len of jstring object"
              Stop run
              Else
              Display "The length of returned string is:"
              SAelementlen
              End-if
              Call "GetStringPlatform"
                 using by value JNIEnvPtr
                                SAelement
                                address of Sbuffer
                                length of Sbuffer
                                0
              Display Sbuffer(1:SAelementlen)
           End-perform

           .
         End method "foo".
         End factory.
         End class TSTClass.
```

```
/*
//LKED EXEC PGM=IEWL,PARM='RENT,LIST,LET,DYNAM(DLL),CASE(MIXED)'
//SYSLIB   DD DSN=&LIBPRFX..SCEELKED,DISP=SHR
//         DD DSN=&LIBPRFX..SCEELKEX,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTERM  DD SYSOUT=*
//OBJMOD   DD DSN=IBARON.REDBOOK.OBJECT,DISP=SHR
//SYSLMOD  DD  PATH='/u/ibaron/sandbox/Redbook/libTSTClass.so',
//              PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//              PATHMODE=(SIRWXU,SIWUSR,SIRGRP)
//SYSUT1   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSDEFSD DD DUMMY
//SYSLIN   DD *
  INCLUDE OBJMOD(TSTJNI)
  INCLUDE '/usr/lpp/java/IBM/J7.0/bin/classic/libjvm.x'
  INCLUDE '/usr/lpp/cobol/igy410/lib/igzcjava.x'
```

CallingCobol.java

```
class CallingCobol
{
   public static void main(String argv[]) throws Exception
   {
      int i;
      String[] a = new String[3];
      a[0] = "John";
      a[1]  =  "White";
      a[2] = "1234567890";

      System.out.println("In Java main, before calling cobol");
      i = TSTClass.foo(a);
      System.out.println("Number of elements foo read:" + i);
      System.out.println("In Java main, after calling cobol.");
      return;
   }
}
```

Submit job on MVS. Two files are generated on the USS: `TSTClass.java` and `libTSTClass.so`

```
/usr/lpp/java/IBM/J7.1/bin/javac CallingCobol.java

/usr/lpp/java/IBM/J7.1/bin/javac TSTClass.java

/usr/lpp/java/IBM/J7.1/bin/java -Djava.library.path=. CallingCobol
```

5.7.7. Example 7: COBOL invoking Java via INVOKE (MVS)

```
//TSTHELLO JOB ,
//  TIME=(1),MSGLEVEL=(1,1),MSGCLASS=H,CLASS=A,REGION=0M,
//  NOTIFY=&SYSUID
//*
// SET COBPRFX='SHARE.IGY410'
// SET LIBPRFX='CEE'
//*
//COMPILE EXEC PGM=IGYCRCTL,
// PARM='SIZE(5000K),LIB'
//SYSLIN   DD DSNAME=IBARON.REDBOOK.OBJECT(TSTHELLO),DISP=SHR
//SYSPRINT DD SYSOUT=*
//STEPLIB  DD DSN=&COBPRFX..SIGYCOMP,DISP=SHR
//         DD DSN=&LIBPRFX..SCEERUN,DISP=SHR
//SYSUT1   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT2   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT3   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT4   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT5   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT6   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT7   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSIN    DD *
       cbl dll,thread,lib,pgmname(longmixed)
       Identification division.
       Program-id. "TSTHELLO" recursive.
       Environment division.
       Configuration section.
       Repository.
           Class HelloJ is "HelloJ".
       Data Division.
       Procedure division.
           Display "COBOL program TSTHELLO entered"
           Invoke HelloJ "sayHello"
           Display "Returned from java sayHello to TSTHELLO"
           Goback.
       End program "TSTHELLO".
/*
//LKED EXEC PGM=IEWL,PARM='RENT,LIST,DYNAM(DLL),CASE(MIXED)'
//SYSLIB   DD DSN=&LIBPRFX..SCEELKED,DISP=SHR
//         DD DSN=&LIBPRFX..SCEELKEX,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTERM  DD SYSOUT=*
//OBJMOD   DD DSN=IBARON.REDBOOK.OBJECT,DISP=SHR
//SYSLMOD  DD DSNAME=IBARON.REDBOOK.GOSET(TSTHELLO),DISP=SHR
//SYSDEFSD DD DUMMY
//SYSLIN   DD *
  INCLUDE OBJMOD(TSTHELLO)
  INCLUDE '/usr/lpp/java/IBM/J8.0/bin/classic/libjvm.x'
  INCLUDE '/usr/lpp/cobol/igy410/lib/igzcjava.x'
/*
//GO EXEC PGM=TSTHELLO,
//         PARM=' /ENVAR("_CEE_ENVFILE=/u/ibaron/sandbox/ENV"),
//               XPLINK(ON) POSIX(ON)'
```

```
//STEPLIB   DD DSN=&LIBPRFX..SCEERUN2,DISP=SHR
//          DD DSN=&LIBPRFX..SCEERUN,DISP=SHR
//          DD DSN=IBARON.REDBOOK.GOSET,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTERM  DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//CEEDUMP  DD SYSOUT=*
//SYSUDUMP DD DUMMY
//JAVAOUT  DD PATH='/u/ibaron/sandbox/javaout',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=(SIRUSR,SIWUSR,SIRGRP)
//JAVAERR  DD PATH='/u/ibaron/sandbox/javaerr',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=(SIRUSR,SIWUSR,SIRGRP)
```

HelloJ.java

```
class HelloJ {
   public static void sayHello() {
     System.out.println("Hello World, from Java!");
   }
}
```

1. Compile the Java program.

```
/usr/lpp/java/IBM/J7.1/bin/javac HelloJ.java
```

2. Prepare the ENV file
   - `PATH=/usr/lpp/java/IBM/J8.0/bin:$PATH`
   - `LIBPATH=/usr/lpp/java/IBM/J8.0/bin:/usr/lpp/java/IBM/J8.0/bin/classic`
   - `CLASSPATH=.:<PathToClass>`
3. Submit job in MVS.

5.8. Summary

In this chapter we described how JNI defines the infrastructure between Java and LE- compliant languages, and discussed the main considerations in the context of Java and COBOL interoperability in z/OS. Main considerations were discussed for both Enterprise COBOL and native COBOL. Specific COBOL and Java examples are provided to demonstrate the techniques. Interoperability with PL/I was also discussed.

5.9. Related resources

The following list provides useful resources that provide more detailed description of the topics that are covered in this section.

1. The Java Native Interface
   http://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html
2. Enterprise COBOL for z/OS Programming Guide: Developing object-oriented programs - Communicating with Java methods:
   p607, https://www.ibm.com/docs/en/SS6SG3_4.2.0/com.ibm.entcobol.doc_4.2/PGandLR/igy3pg50.pdf
3. Enterprise COBOL for z/OS Programming Guide: Developing object-oriented programs - Writing object-oriented programs:
   p561, https://www.ibm.com/docs/en/SS6SG3_4.2.0/com.ibm.entcobol.doc_4.2/PGandLR/igy3pg50.pdf
4. z/OS 2.1.0 - z/OS MVS- z/OS Batch Runtime Planning and User's Guide - Application structure and build considerations for COBOL and Java -DLL considerations for COBOL and Java considerations for COBOL and Java:
   https://www.ibm.com/docs/en/zos/2.1.0?topic=java-dll-considerations-cobol
5. New Ways of Running Batch Applications on z/OS: Vol 4 IBM IMS Section 2.9.1: How Java can call COBOL and vice versa
   http://www.redbooks.ibm.com/redbooks/pdfs/sg248119.pdf
6. Enterprise COBOL for z/OS Programming Guide: Compiling and debugging your program - Compiling, linking, and running OO applications:
   https://www.ibm.com/docs/en/cobol-zos/6.1?topic=program-compiling-linking-running-oo-applications
7. Enterprise COBOL for z/OS 4.2.0 - Programming Guide -Compiling and debugging your program - Compiling, linking, and running OO applications - Compiling, linking, and running OO applications under z/OS UNIX:
   https://www.ibm.com/docs/en/cobol-zos/6.1?topic=clroa-compiling-linking-running-oo-applications-under-zos-unix
8. Enterprise COBOL for z/OS 6.1.0 - Programming Guide -Compiling and debugging your program - Compiling, linking, and running OO applications - Compiling, linking, and running OO applications in JCL or TSO/E:
   https://www.ibm.com/docs/en/cobol-zos/6.1?topic=clroa-compiling-linking-running-oo-applications-in-jcl-tsoe

9. *The Java Native Interface: Programming Guide and Specification.* Sheng Liang, 1999.

10. *Essential JNI: Java Native Interface*. Rob Gordon, 1998.

CHAPTER 6

6. Infrastructure, setup, and scenarios

In this chapter, we will cover the configuration considerations and provide configuration examples for existing workloads when IMS™ regions are JVM-enabled to allow for Java interoperability. We will examine and share our experience about JVM setup in the IBM® Language Environment®, JVM configuration for a classic IMS TM environment, setup for external access to Db2® data, and related scenarios when IMS Java™ Dependent Region resource adapter and WebSphere® optimized local adapters are involved.

6.1. Environment description

First we will consider the environment restrictions and requirements for language interoperability to run in z/OS®.

6.1.1. JVM startup

Unlike in the initial implementations of JVMs in IMS regions, at the time of this writing, the JVM is fully initialized after starting the IMS dependent region. There is no difference if it's a Java Dependent Region (JBP, JMP), BMP or MPP region. The penalty for the first transaction to suffer from the JVM startup is not applicable anymore.

6.1.2. JVM persistency and abend penalties

The JVM in IMS regions is made persistent, which implies a couple of considerations, such as a persistent Language Environment (LE) enclave and a possible need for CANCEL_PGM to simulate the LE enclave termination when an IMS schedule (after processing one or more IMS messages) ends.

Be aware of the impact abends, rollbacks, and pseudo abends have in a JVM-enabled IMS region. If an IMS abend (including pseudo-abends such as IMS U0777 abends) occurs, the LE enclave is terminated and, as a result, the JVM is destroyed and needs to be rebuilt. If there are workloads that frequently create abends or pseudo-abends in the IMS message processing regions, there will be a lot more CPU usage than before, because the JVM is initialized after the abend.

6.1.3. Language Environment restrictions for Java interoperability

There are some known restrictions for COBOL/Java interoperability. One is that it requires the IBM LE setting XPLINK(ON). But AMODE24 and VS

COBOL modules will not work with XPLINK(ON) and produces a runtime exception including a message.

An example of the message for a call to VS COBOL module with XPLINK(ON) is the IGZ0186S message:

```
IGZ0186S An attempt was made to run a VS COBOL II program with
the run-time option XPLINK(ON). The program name is program-
name.
```

At one client site, the conversion of the remaining 30 AMODE24 assembler modules to AMODE31 took about two weeks. They did no separate testing, since testing of the AMODE31 assembler routines was part of the tests for COBOL and Java interoperability.

In order to support languages other than COBOL, such as PL/I or Assembler, it is required to have a LE compatible version of PL/I and/or a LE compatible/enabled assembler routine as the caller (this is true only for the module that is invoked by IMS to run the transaction; usually IMS PSB name is the module name). IMS has added a new diagnostic message in case there are non-LE compliant routines:

```
DFS650E NON-LE COMPLIANT PROGRAM IN PERSISTENT JVM ENVIRONMENT,
NAME=entry_point
```

This message should help you find the module that is not LE-compliant. In most cases the entry_point is equal to the module name.

It can be very difficult to find occurrences of old modules still used in production, especially if the modules are object code only and the source is not available in the shop. Some modules have been used for decades. An easy way (but perhaps not the best way) is to enable the JVM in the IMS regions in the test environment and wait for abends. A customer, for example, had an assembler stub that linked to every IMS Enterprise COBOL main program, which caused the actual LE-enabled module to be treated as non-LE program by the run time. This basically shows that getting rid of these inheritances from old days might not be as easy as it sounds.

Please note, that in some cases also ISV or IBM tools that use the IDENTIFY function cause the message above, but only show the name of the entry point. There was a case where a checkpoint restart tool with its main method caused the DFS650E message above, but the name shown was the name of the main IMS program, which was LE complaint. It was necessary to look at the dump to discover, that the checkpoint restart tool in use was the causer of that problem.

To restate the above, non-LE modules can be used in that environment, but they cannot be the first module to be called by IMS. The reason for this is, that CEEPIPI is used to make the JVM persistent and reusable across IMS transactions and schedules, the CEEPIPI subfunction add_sub is used to add and start a module. add_sub is only supported with LE-compliant modules and subroutines. Since COBOL modules compiled by Enterprise COBOL compiler are both MAIN and SUB, there is nothing to do on the COBOL side. However, since LE-compliant assembler and PL/I modules can only be either MAIN or SUB, they are required to be SUBs in Java interoperable environments that use CEEPIPI, such as IMS regions. In addition the calling chain is for example LE COBOL -> non LE assembler and return to the caller works, but the call sequence LE COBOL -> non LE assembler -> LE COBOL will not work in the JVM environment.

If library routine retention (LRR) is used, then XPLINK(ON) requires to use the CEELRRXP version in the IMS PREINIT PROCLIB member. Otherwise the IMS region will abend.

The LE ERRCOUNT parameter needs to be set to 0. The reason is, that the JVM for z/OS sometimes generates TRAP instructions, which cause the LE error count to increase and cause abends when the error count is reached.

See also the following doc on JVM restrictions:

- Known issues and limitations, IBM SDK, Java Technology Edition V7: https://www.ibm.com/docs/en/sdk-java-technology/7?topic=reference-known-issues-limitations
- Known issues and limitations, IBM SDK, Java Technology Edition V8: https://www.ibm.com/docs/en/sdk-java-technology/8?topic=support-known-issues-limitations

Since LE only supports interoperability or dynamic calls to 31-bit modules, a 64-bit JVM cannot be used as at the time of this writing. When setting up the environment paths, you should made sure that the paths point to a 31-bit JVM on z/OS.

In addition, it is not possible to dynamically call a DLL at all, if it was already dynamically called by a program which was not compiled with the DLL compiler option (which also happens when called from non-LE or no DLL assembler routines). You would get an IGZ0176S error, indicating that a call from a COBOL program compiled with the DLL compiler option failed because the program program-name was previously dynamically called by a COBOL program compiled without the DLL compiler option.

Program mask (from Processor Status word) and exception handling bits might be different if additional languages (a COBOL DLL also treated as new language, Java is written in C so this also applies here) are used. For example, if an existing COBOL calls Assembler scenario is moved to a JVM environment and enhanced with JNI calls this will likely cause the program mask to be different. In this case the result was as follows:

1. Language Environment neither saves nor restores the program mask setting across calls to Language Environment services or calls within the Language Environment.
   https://www.ibm.com/docs/en/zos/2.1.0?topic=environment-program-mask-conventions
2. The call from NODLL Cobol to the DLL COBOL was treated as new language (causing IGZCFCC calling CEEBADDM service to add a new member).
   The maskable program exceptions are enabled for all member languages represented in the root or main load module during Language Environment initialization. Each member language informs Language Environment of its program mask requirements, and Language Environment ORs all of the requirements together and sets the program mask during initialization. During termination, the program mask is reset by Language Environment to its value upon entry to Language Environment initialization.
3. While the enclave is running, the program mask is influenced by the callable service, CEE3SPM, and by members' requirements that are newly-added as a result of a dynamic call or fetch; this is handled by the CWI service CEE3ADDM.
4. If a program expects to work with a specific program mask, it needs to set it to the values it requires and reset it back, because of 1. This basically means, in an unlucky mix or with new compilers/languages, the program mask might be different than it was before.

6.1.4. Abend and error handling

If JVM is present in an IMS region, by default it registers many signal handlers. You need to make sure that all variables or structures are initialized in subroutines that are called many times during the lifetime of the IMS region containing the JVM. The LE enclave in JVM-enabled environments is persistent and only destroyed at abends or exceptions. We have seen badly initialized routines that resulted in rare 0C4 abends without a trace to help identify what caused them.

If there is a 0C9 (numeric exception) in the COBOL code, by default there is no 0C9 abend. Rather, there is a language environment abend such as U4038, or an IMS user abend such as a U101. When an error like this occurs, in order to turn that behavior off and produce a 0C9 abend message, the JVM requires a switch that disables the registration of POSIX handlers by the JVM.

This implementation also ensures that IMS abends such as 0476 and 0711 make it to the user without the LE and JVM handlers catching it. This allows the continued use of the abend and error handler mechanisms that are in place or used without the JVM being present.

<u>Important:</u> In order to get an abend with the real abend cause, for example 0C9, it is required to turn off the JVM's standard registration of POSIX handlers. This is done by using the `-Xsignal:userConditionHandler-percolate` JVM command line option.

Tools such as IBM Fault Analyzer can be used to display the root cause of the error, which is not easy for some LE abends.

Some clients have registered custom LE abend handlers, and if the use in mixed mode environments needs to be continued, then turning off the POSIX handler registration by the JVM is preferred. A recent IMS PTF that handles these types of errors was changed to better suit the needs of a production environment. Make sure you have the latest IMS maintenance applied.

Refer to the manuals for IBM 31-bit SDK for z/OS, Java Technology Edition, V8 and the SDK Guide at this link:
https://www.ibm.com/support/pages/java-sdk-products-zos#v8

6.1.5. z/OS memory configuration (IEFUSI)

JVMs require a large amount of storage. Limitations such as 32 MB will lead to JVMs not being able to start up at all or to fail. A JVM requires at least 128 MB to run basic Java code, so make sure that the address spaces where the JVMs run get enough main storage.

In one case where a customer ran the JVM in the production environment, JNI programming errors led to out-of-storage conditions in the JVM- enabled IMS regions. These regions then failed a normal region termination (MEMTERM) and entered a state where they could not be stopped by using IMS or z/OS commands, including FORCE. The main reason for this problem was that the modules required for doing the region termination could not be loaded into private storage.

A PMR that was opened to address this issue suggested to implement an IEFUSI exit to reserve 512k storage below 16MB to allow the region termination modules to be loaded. While this increased the chances for the successful region termination it is not a guarantee for region termination.
The recommendations from the PMR are shown in Example 2-15. It is suggested to use as a base the sample IEFUSI exit provided in member IEEUSI in SYS1.SAMPLIB and replace the statements.
Sample IEFUSI exit

From:

```
         USING REGION,R07        ADDRESSABILITY FOR REGION DSECT
         OI REGFLAGS,X'80'                SET THAT IEFUSI CONTROLS
REGIONS
         TM 0(R08),X'80'         V=R JOB?
         BO EXIT                 YES USE DEFAULT VALUES
         L R10,REGSZREQ          GET REQUESTED REGION SIZE
         LTR R10,R10                      IS IT ZERO
         BZ EXIT                 YES USE DEFAULT VALUES
         AL R10,N64K                      ADD N64K TO REGION SIZE BELOW
         ST R10,REGLIMB          SET IT AS REGION LIMIT BELOW
         MVC REGSIZB,REGSZREQ    SET REGION BELOW
         MVC REGSIZA,REG32MB              SET REGION SIZE ABOVE TO 32MB
```

To:

```
       MVC REGLIMA,REG32MB            SET REGION LIMIT ABOVE TO 32MB
```

By using the sample below:

```
          USING REGION,R07        ADDRESSABILITY FOR REGION DSECT
          OI REGFLAGS,X'80'            SET THAT IEFUSI CONTROLS
REGIONS
          TM 0(R08),X'80'         V=R JOB?
          BO EXIT                 YES USE DEFAULT VALUES
          L R09,16(R0)                 Obtain CVT Pointer
          L R11,560(R09)          Obtain GDA Pointer
          L R09,164(R11)          Obtain Region Size below
          L R11,REDUCLIM          Load subtract value
          SR R09,R11                   Subtract by 512KB
          L R10,REGSZREQ          GET REQUESTED REGION SIZE
          CR R09,R10                   Is Requested size bigger
          BNH CHNGLIM                  Yes do the change
          LTR R10,R10                  IS IT ZERO
          BZ CHNGLIM                   Yes do the change
          B EXIT                  Go to exit no change to req
          CHNGLIM ST R9,REGLIMB   Store REGLIMB to Parmlist
          SR R09,R11                   Subtract another 512KB
          ST R09,REGSIZB          Store REGSIZB to Parmlist
*
```

And then add in the constants section the 512KB subtraction value:
REDUCLIM DC X'00080000'

There is also a change of the REGION SIZE BELOW value because this value is being used for programs that do a variable *getmain* and tend to use all available storage.

For an example you can refer to the informational APAR II05315 with the title: ABEND878 OR ABEND40D DURING SMP/E WITH REGION=0M ORREGION=0K OR REGION=32M WITH NO IEFUSI.

Disclaimer: Since only some basic function testing has been performed for the codesample above, extensive testing should be done to the complete EXIT functionality before it is used in a production environment. The sample is provided as is.

6.1.6. Is CANCEL_PGM required?

Default setting when Java is enabled is CANCEL_PGM=N. Ensure that you understand what this means for your applications! Cancel is done on a unit of recovery level. For an explanation, refer to the section “Unit of work and unit of recovery”.

For default IMS processing, the LE enclave is created after the IMS application program is started. Depending on the IMS program configuration such as the PROCLIM parameter, the IMS application program processes one or more IMS messages and then either remains loaded in wait status (WAIT for Input/WFI, or Pseudo Wait for Input/PWFI=Y), or terminates immediately after the last IMS message was processed by giving back the QC status code. In both cases, IMS expects the application to end the loop-around GU call and to terminate (with GOBACK not EXIT). For the first case, the enclave termination occurs right after the transaction ends with GOBACK; for WFI/PWFI, it occurs when IMS decides to schedule a different transaction/module to the IMS region. Enclave termination will clean up everything and force a fresh module load for the next IMS unit of work.

Due to the fact that the JVM needs to be made persistent, when Java is enabled, the LE enclave (as a carrier for the JVM) has to stay up for the lifetime of the IMS region. This forces somehow a reusable environment, even if the code or modules are not reusable or working storage is not initialized. This basically means that the module needs to function correctly if working storage is not initialized to zeros when the routine is entered. So with the JVM there is only a single IMS schedule or unit of work for the whole lifetime of the JVM.

Different artifacts might also behave differently (for example, memory management for COBOL, Java, and JNI). Some common rules and generic settings do apply if you mix high-level languages with Java in Java-enabled regions.

With Java enablement, the region lifecycle changes (long-living LE enclave). In particular, the following constructs would survive program termination and need to be considered:

- Working storage. The storage lives as long as the LE enclave is up. Release of storage is enabled in CANCEL_PGM=Y environments, but only for cancelled modules.
- COBOL externals. COBOL externals live as long as the LE enclave is up.
- LE Heap storage requests. LE Heap storage requested by CEEGTST service lives as long as the LE enclave does. Check your application-based heap management for possible leaks. They would likely surface in this environment.
- Allocated storage through other methods, e.g. from stateless Assembler routines that do STORAGE OBTAIN and save it in COBOL working storage. Release of Storage is not enabled in CANCEL_PGM=Y environments, it will cause memory leaks there, since the COBOL working storage is released, without freeing the storage obtained in Assembler. There is no exit available to release this storage.

In addition CANCEL_PGM=Y also means that all modules that have been loaded during the last IMS transaction execution go through COBOL CANCEL processing (basically an IGZCXCC call). IMS code implements an LE load exit to track all loaded modules. This also means that if you bypass LE load with Assembler programs that do LOAD EP=MODULE, those modules will not be cancelled. The CANCEL progress is driven at syncpoint time. This could be during GU processing. It is not allowed to do COBOL CANCEL on active modules. A subroutine that implements the GU IOPCB meets that requirement, because it is in the active calling chain during syncpoint (MAIN -> GUIOPCB module). This module needs to be added to the CANCEL_PGM EXCLUDE list in order to avoid the U4038 abend that is accompanied by an IGZ0032S error message, which terminates the LE enclave.

The COBOL CANCEL statement will perform program cleanup before deleting the load module from the virtual storage. In particular, it will:

- Release the load module from the running LE enclave.
- Close all open files that are internal to the program.
- Free all program class storage associated with the program, which includes the program WORKING STORAGE, any internal control blocks, special registers, and others. However, it will not free external data records that are referenced by the program.
- Repeat the above program cleanup logic for all programs that are contained within the load module.
- Delete the load module from the virtual storage.

The net effect of this is, if the same program is called again with the COBOL CALL statement, the program will be entered in its initial state.

Please note that if modules with large working storage (e.g. 60M) are cancelled, this can lead to Out of Memory conditions due to LE Heap fragmentation. In this case, consider using the FREE parameter instead of KEEP for the LE option that defines the initial HEAP and the increments for expansion (HEAP option). Basically there are the following Options (1M, 1M, KEEP,...), (100M, 1M, KEEP,...), (1M, 100M, FREE,...), (1M, 100M, FREE,...) and (100M, 100M, FREE,...). Current testing shows there are advantages starting with a large initial Heap size and then increments that are freed (so use the FREE parameter).

There are, however, some exceptions, with the following four reasons for why a program will not get cancelled on a UOR level:

1. Program name is in the EXCLUDE list.
2. Program is active.
3. Transact PROCLIM is set to 0 or 1 (this check was added in APAR PI07418)
4. The program is a non-message driven BMP.

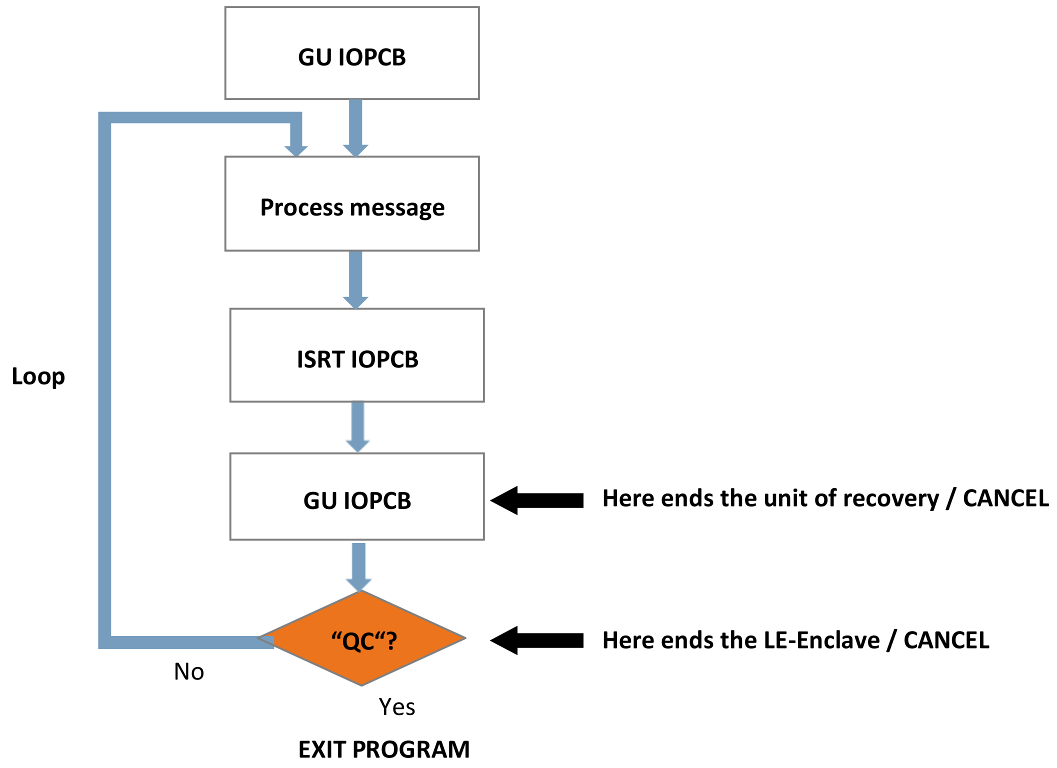
IMS preload is affected by CANCEL_PGM=Y. At cancel time, the module is removed from the LE enclave and needs to be copied/loaded into the LE enclave again on next usage/call. If there are programs that are frequently used and if there are no other reasons for them to be cancelled (e.g. no initialization of working storage), those modules should be added to the CANCEL_PGM Exclude list in order to avoid cancellation and reload and the accompanied I/Os. Enabling the CANCEL_PGM=Y environment should also be accompanied by a review of I/Os and a possible move of libraries to the linklist in order to exploit VLF/LLA to save I/Os.

Cancelling modules that aquire storage with other methods like explicit getHeap, getMain or storage obtain requests can cause memory leaks, if the modules are cancelled before the freeHeap, freeMain or storage release is issued for that main memory area. Cancel PGM or enclave termination does not free memory that was obtained via the storage or getmain macro.

6.1.7. Unit of work and unit of recovery

Basically, an IMS transaction is a unit of recovery and an IMS schedule is a unit of work.

Figure 25. Unit of work and unit of recovery


GU IOPCB
Process message
ISRT IOPCB
GU IOPCB
Loop
Here ends the unit of recovery / CANCEL
"QC"?
Here ends the LE-Enclave / CANCEL
No
Yes
EXIT PROGRAM

The Cancel implementation in IMS is done at the unit of recovery level, so you need to ensure that modules that are expected not to be canceled during the IMS schedule (unit of work) are in the exclude list. An example would be a subroutine that implements the GU to IOPCB and saves in the working storage some values that are expected to survive the unit of recovery boundary.

6.2. IMS TM classic scenario

In this section we will look at the classic IMS TM scenario with BMP and MPP. This section does not cover JMP and JBP.

To use Java in IMS, you need to configure two members in the IMS PROCLIB data set for JVM configuration. To do so, start by collecting the information required for your initial configuration.

6.2.1. Data to collect for the initial configuration

IMS needs the following information for applications that use Java. In order to set up the configuration properly, gather the following information for your environment:

- JAVA_HOME: Path to the ZFS directory containing a 31-bit Java SDK for z/OS
- DB2_JDBC_HOME: Path to ZFS directory pointing to the DB2 JDBC

libraries for the Db2 version and SMP/E maintenance level that IMS is connected to. The path usually contains the version number and "jdbc"in the name, such as `/usr/lpp/db2/db2a10_jdbc`

- IMS_JAVA_HOME: Path to ZFS directory pointing to the IMS TM and IMS JDBC libraries for the IMS version and SMP/E maintenance level that is in use. The path usually contains the version number and "imsjava" in the name, such as `/usr/lpp/ims/ims14/imsjava`
- APP_HOME: Path to the ZFS directory that contains the Java libraries (.jar files) that make up the application implementation or middleware classes.
- APP_JAR*nn*: Names of the actual application .jar files. Unfortunately there is no way to just specify a single directory; instead, it is required to explicitly name all application .jar files in the JVM configuration.

In the following configuration sample, the bold placeholders enclosed in "< >" are to be replaced with the actual values. At the time of this writing, there is no feature available that allows to set those variables globally.

6.2.2. IMS JVM configuration

There are two members in the IMS PROCLIB data set that are required to be set up:

- The environment member ENVIRON for the UNIX environment variables
- The JVM configuration member JVMOPMAS, which either directly contains the JVM settings (with length limitations) or points to a ZFS file that contains the JVM settings.

6.2.2.1. IMS JVM environment member

The IMS Message Region JCL points to this member by using the ENVIRON= execute parameter. Attached is the sample contents:

Example: ENVIRON Member

```
PATH=/bin:<JAVA_HOME>/bin:.
LIBPATH=/lib:/usr/lib:>
<JAVA_HOME>/bin/classic:>
<JAVA_HOME>/bin/j9vm:>
<DB2_JDBC_HOME>/lib
DB2JCC_CONN_REUSE=Y
UMASK=002
*UMASK=000
*CANCEL_PGM=Y,EXCLUDE=EXCLUDE
*DEBUG=Y
*DB2JCC_ESAF_THREAD_NOTIFICATION=YES
*ESAF_SIGNON_ACEE=YES
_CEE_DMPTARG=/SYSTEM/tmp/IMS
```

```
TMPDIR=/SYSTEM/tmp
JAVA_DUMP_TDUMP_PATTERN=GAEBLER.JVMDUMP.%job.D%y%m%d.T%H%M%S
JAVA_DUMP_OPTS=ONANYSIGNAL(JAVADUMP,SYSDUMP,HEAPDUMP,CEEDUMP)
```

Comments start with an asterisk.

For more information about the IMS related options, such as CANCEL_PGM, refer to the IMS docs:

https://www.ibm.com/docs/en/ims/15.2.0?topic=set-dfsjvmev-jvm-environment-settings-member

In addition, for JVM dump specific environment options, refer to the JVM docs: https://www.ibm.com/docs/en/sdk-java-technology/8?topic=options-xdump#filefilename

The CANCEL_PGM setting is required only if there are modules that do not initialize their working storage and rely on storage being zeroed out after module load. This is already discussed earlier in "Is CANCEL_PGM required?" section.

The UMASK setting is the mask for files created by the JVM, this includes logs and the temporary files that the JVM creates for the shared class loader cache. The default is UMASK=022 and as such the JVM creates files with permission 755. Since they are created with the userid that is active with the first use (first transaction user), they usually lead to the fact, that they cannot be used by other users. So UMASK can be set to 002 or 000, leading to file permissions of 775 or 777, which might be required to allow all users that run in that IMS regions to write or access those log and temporary files.

6.2.2.2. IMS JVM configuration member

The IMS Message Region JCL points to this member by using the JVMOPMAS= execute parameter. Below is the sample contents:

Sample JVMOPMAS member

```
-Xoptionsfile=/some/path/dfsjvmoptionsAPPx
```

In this example the configuration member just points to a file in ZFS.

One of the advantages of using a ZFS file is that a long classpath can be put into a single line instead of having to concatenate the records from the IMS PROCLIB. Furthermore, there is a limit on concatenation lengths, and the limit can only be bypassed by using the ZFS file. Often times as Java is adopted and integrated into an environment, the classpath is short and classpath length is not a problem, but as soon as the classpath gets larger, you would need to switch to a ZFS file. Therefore it is recommended to use the ZFS configuration for Java options from the beginning.

6.2.2.3. Alternate Environment and JVM configuration with STDENV

Beginning with IMS V14 there is the choice to use a JVM configuration similar to how the JZOS Java launcher defines environment variables and classpath.

The implementation is like this, that in case STDENV DD statement is present in the IMS dependent region JCL and this IMS region is Java enabled through the presence of a JVMOPMAS= and ENVIRON= parameter. The data from STDENV DD input is read, a shell script is launched and executed and on successful termination all present environment variables are taken as the input to the IMS JVM launcher as if they were specified in the ENVIRON member.

Attached is an example of how the STDENV DD statement can possibly be used:

Sample STDENV DD statement for IMS JVM configuration

```
//STDENV DD *
# This is a shell script which configures variables
. /etc/profile
###########################################################
# Customize below to match your installation:
# JAVA_HOME - The location of the SDK
###########################################################
export JAVA_HOME=/local/java/J8.0
export PATH=/bin:"$äJAVA_HOMEü"/bin:
export  LIBPATH=/lib:/usr/lib:"$äJAVA_HOMEü"/bin
export LIBPATH="$LIBPATH":"$äJAVA_HOMEü"/bin/classic
export LIBPATH="$LIBPATH":"$äJAVA_HOMEü"/bin/j9vm
export LIBPATH="$LIBPATH":/local/db2/db2a10_jdbc/lib
export LIBPATH="$LIBPATH":/local/ims/ims14/imsjava/lib
export LIBPATH="$LIBPATH":/local/mqm/V7R1M0/java/lib
#export JZOS_JVM_OPTIONS="$OPTS"
# Add the JZOS alphaWorks jars to the classpath
#for i in "$äJZOSAW_HOMEü"/*.jar; do
#CLASSPATH="$CLASSPATH":"$i"
#done
#export CLASSPATH="$CLASSPATH":/u/gaebler/CJ01imsdb2.jar
# Use this variable to supply additional arguments to main
#export JZOS_MAIN_ARGS=""
# Configure JVM options
export IBM_JAVA_OPTIONS="-Xoptionsfile=/u/gaebler/dfsjvmoptions"
# IMS Specific Options
export DB2JCC_CONN_REUSE=Y
```

Using this option allows to use other variables as part of the CLASSPATH or LIBPATH which do not need to be explicitly specified.

6.2.2.4. IMS JVM configuration file

The JVMOPMAS configuration member might point to this file. Attached is the sample contents:

Sample JVM options file pointed to by the JVMOPMAS Member

```
-Djava.class.path=<DB2_JAVA_HOME>/classes/db2jcc.jar:...
######################################################################
##
################            JVM Properties
################
######################################################################
##
-Xgcpolicy:gencon
-Xmx128M
-Xmso512k
-Xss256k
-Xms32M
-Xsignal:userConditionHandler=percolate
-Xnoagent
-Djzos.merge.sysout=true
```

Since this is a standard JVM on z/OS, all supported JVM options are allowed. For more information refer to the Java documentation for z/OS:
https://www.ibm.com/docs/en/sdk-java-technology/8?topic=options-specifying
As mentioned earlier, for the classpath it is required to specify and explicitly name all application .jar files that are required and used by the application:

Sample Java classpath

```
-
Djava.class.path=<IMS_JAVA_HOME>/imsjava/imsudb.jar:<IMS_JAVA_HOME>/ims
java/imsutm.jar:
<DB2_JAVA_HOME>/classes/db2jcc.jar:<DB2_JAVA_HOME>/classes/db2jcc_javax
.jar:<DB2_JAVA_HOME>/classes/db2jcc_license_cisuz.jar:<APP_HOME>/APP_JA
R01.jar:<APP_HOME>/APP_JAR02.jar:<APP_HOME>/APP_JAR03.jar:<APP_HOME>/AP
P_JARnn.jar
```

Furthermore, all -D options that might be required can be specified here, such as, but not limited to, the options shown in the following sample.

Sample JVM Options File that were in use

```
#-javaagent:/u/gaebler/imsstartup.jar

#-verbose

#-Dfile.encoding=UTF-8
#-verbose:gc

#-Xverbosegclog:/tmp/IMS/gclog.txt

#-Ddb2.jcc.override.currentSchema=DENIS

#-Ddb2.jcc.propertiesFile=/u/gaebler/DB2JccConfiguration.properties
#-

Ddb2.jcc.pdqProperties=captureMode(ON),executionMode(DYNAMIC),pureQuery
Xml(capture.pdqxml)

#-

Ddb2.jcc.pdqProperties=captureMode(OFF),executionMode(STATIC),allowDyna
micSQL(FALSE),pureQueryXml(capture.pdqxml)

-Xmaxf0.8

-Xminf0.3

-Xmx256M

#-Ddb2.jcc.traceLevel=-1

#-Ddb2.jcc.traceFileAppend=false

#-Ddb2.jcc.traceFile=/tmp/imsdb2jcctrace.trc
#-Xmn3M

#-Xmo13M

# profiling

#-agentlib:JPIBootLoader=JPIAgent:server=enabled;CGProf
#-agentlib:JPIBootLoader=JPIAgent:server=enabled;HeapProf

-Xmso512k

-Xss256k

-Xms32M

-Xsignal:userConditionHandler=percolate
#Xtrace:maximal={j9prt.422-423,j9scar.67-68,j9scar.141-

142,j9scar.159,j9scar.201-202,j9prt.423-424},output=/tmp/signal.trc
#-Xdump:java+heap+snap:events=vmstart+user

#-Xdebug

#-Xdump:heap:events=user,opts=PHD

#Xdump:system+java+heap+snap:events=user,opts=PHD,request=exclusive+pre
pwalk+compact

#-Xnoagent

#-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=7778
#-

Xrunjdwp:transport=dt_socket,server=n,suspend=n,address=172.17.36.87:80
01

#** Other JVM Settings

-Xcodecache10M

-Xshareclasses:name=cobolims1

-Xscmx64M

-Xscminaot16M

-Xhealthcenter:port=9020
```

```
#-Xjit:optlevel=scorching
#-Xgc:splitheap
#-Xgcthreads4
#-Xdump:java+heap+system:events=throw,filter=java/lang/OutOfMemoryError
#-verbose:gc
#-Xverbosegclog:/tmp/verboseGC.log
#-Xgcpolicy:optavgpause
-Xgcpolicy:gencon
#-Xtgc:parallel
#-Xcheck:jni:all,trace,pedantic
#-Xcheck:jni:all,nowarn,noadvice
#-Xcheck:jni:trace
#-Xtrace:trigger=tpnid{j9jvmti.69,ceedump},iprint=j9jvmti.69
#-Xtrace:maximal={j9vm,mt},methods={*.*()},iprint=mt
#-Xtrace:trigger=tpnid{j9vm.138-141,jstacktrace},iprint=j9vm.138-141
#-verbose:jni
-Djzos.merge.sysout=true
#-agentlib:hprof=cpu=times
#-XsamplingExpirationTime600
-Djavax.management.builder.initial=
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.port=1099
```

Unfortunately there is limited space here to cover all of the options. Most of the options are used for tracing or debugging purposes; others are for function enablement (remote debugging, profiling, JConsole, Health Center, etc.) or product configuration (pureQuery, JVM performance options, etc.). In most cases, using your favorite search engine should help you uncover the purpose of the settings. In some cases, tracing for PMRs involves specific -Xtrace settings, so they should only be used or switched on when directed to do so by IBM Software Support.

Note: The classpath can be defined in multiple places, but it is recommended to place it in the IMS JVM configuration file when it is used or the IMS JVM configuration member.

6.2.3. IMS JVM related Language Environment configuration

Bringing the JVM into the IMS environment causes implicit setting of some z/OS Language Environment options that cannot be turned off. Those are, for example, POSIX(ON) and XPLINK(ON), and XPLINK(ON) in turn forces ALL31(ON).

Those settings lead to most of the restrictions that are mentioned earlier (such as VS COBOL II modules not working or AMODE24 routines not working).

More information about Language Environment options can be found in the "Language Environment restrictions for Java interoperability" section.

6.3. Db2, ESAF connection pooling, and plans

Db2 access from IMS is discussed here.

When Java comes into play in IMS regions, it basically uses the same physical database connection and required resources to connect to Db2 as traditional languages do. This means a Db2 plan with the same name of the IMS PSB is required, unless a resource translation table (RTT) module is used to map the PSB into a Db2 plan in other ways. A Db2 plan defines the Db2 resources that an application accesses. Each IMS application is associated with a plan name.

Note: Since Db2 JDBC just uses the existing physical Db2 connection, it is not allowed to create more than one JDBC Connection object in Java (subsequent getConnection calls will fail). It is recommended to create a Java singleton class that creates the Connection object if it is null or marked as closed and otherwise returns the existing JDBC Connection object.

Since Java uses dynamic SQL, the JDBC dynamic SQL packages SYS*xyznn* are required to be bound into the existing plan as well. Unfortunately, Db2 for z/OS documentation does not provide details about the packages, but the Db2 for Linux, Unix and Windows has a reference: https://www.ibm.com/docs/en/db2/11.5?topic=environment-bind-files-package-names

Some more information about the number of statements that can be supported and the number of packages and how to generate it can be found in the Db2 for z/OS documentation for the Db2Binder utility: https://www.ibm.com/docs/en/db2-for-zos/12?topic=installation-db2binder-utility

As a starter, for the test and development environment with little JDBC usage, only the SYSSTAT, SYSLH200 and SYSLN200 packages were needed (with the NULLID qualifier unless other current schema or collections were used and set).

IMS uses External Subsystem Attach Facility (ESAF) to use two-phase commit capable access to Db2. When adding Java to IMS, the same configuration is used (SSM member in IMS PROCLIB). By default IMS uses the DSNMIN10 module, which basically means that a CREATE THREAD plus SIGNON andTERMINATE THREAD are driven for every processed IMS message (unit ofrecovery), unless special options are used (MODE=MULT). This is a huge overhead compared to the way Java EE application servers use database connections, but this is due to the fact that the RACF user id is bound to the Db2 connection for security reasons.

As mentioned earlier, the Db2 JDBC driver uses the same physical connection to Db2 as traditional workloads (e.g. COBOL, PL/I) do. The Db2 JDBC driver just creates Java objects that provide logical access for Java to the existing physical connection. In order to conform to the JDBC standard, the physical connection has no callback into the Java objects. This means that if the physical connection changes, such as change of user id, the Java Db2 JDBC objects are not implicitly notified of the change. Therefore, IMS code has to call a reset method from the Db2 JDBC driver to inform the JDBC driver during sync point processing. This also causes the Java Db2 JDBC Connection object not to be reusable from the Java language. Since all SQL statements that are created by the Db2 JDBC driver are reset at TERMINATE THREAD time, all statement objects that were created based on the Java Db2 JDBC Connection object are invalidated, and the Db2 JDBC Connection object is marked as closed (it will not be null). This process again is a huge overhead.

The calling of the reset method of the Db2 JDBC driver is also known to cause some vendor monitor applications to report increased CPU usage as part of the IMS sync point processing or GU calls. Some monitor applications also report the CPU usage for the JDBC reset operation as part of the next transaction (which starts before the GU) and as such do not provide correct chargeback numbers.

6.3.1. Db2 connection pooling for IMS

There is an enhancement for IMS to pool connections to Db2 and to avoid doing the CREATE THREAD and TERMINATE THREAD for every IMS transaction (UOR). This enhancement will add JDBC Type 4-like connection pooling to the IMS TM environment.

In order to enable this functionality, the required maintenance (See the section Recommended Maintenance based on experiences in the Problem Determination Chapter) need to be applied and the DSNMIN20 module has to be used in the IMS subsystem configuration member.

DSNMIN20 provides Db2 pooling support for Db2 IMS attachment facility threads. The pool contains a maximum of 50 pool entries for the threads. Each pool entry is keyed off of a Db2 plan name plus the active user name. The pool is scoped at the IMS dependent region level.

If Db2 JDBC is to be used, the Db2 JDBC driver also has to be notified about connection pooling. This requires the new parameter DB2JCC_ESAF_THREAD_NOTIFICATION=YES to be added to the JVM environment configuration member. The Db2 connection pooling function also adds the reuse capability to the Db2 JDBC Java objects, which in turn also provides additional CPU savings in the mixed Java environment in IMS.

There are a couple of settings to configure. The maxStatements property in com.ibm.db2.jcc.DB2BaseDataSource should be set to a value greater than 0, and optionally set the KEEPDYNAMIC option to yes to use the DataSource interface for the JDBC connection to Db2. Check the product documentation for the settings.

Moreover, since the connection pool is now provided by the Db2 JDBC driver, using the singleton Java class that returns the same Db2 JDBC Connection object is no longer a good choice. Rather, every time a Db2 connection is needed, the `DataSource getConnection()` method should be called.

6.3.2. Sample IMS setup for using Db2 in a mixed mode environment

The subsystem member (SSM) member needs to set up in the IMS PROCLIB data set for IMS to initiate contact with the external subsystem.

Sample SSM member (no pooling)

```
SST=DB2,SSN=DB10,LIT=SYS1,ESMT=DSNMIN10,REO=R,CRC=-
```

Next step is to add the Db2 loadlibs to the IMS Control region.

Excerpt from IMS Control region for Db2 libraries

```
//******** EXTERNAL SUBSYSTEM STATEMENTS *************
//*
//* USER MAY OPTIONALLY ADD THE DFSESL DD CARD
//* FOR EXTERNAL SUBSYSTEM CONNECTION.
//*
//DFSESL   DD DISP=SHR,DSN=SYS1.IMS.V14.DYNALLOC
//         DD DISP=SHR,DSN=SYS1.IMS.V14.SDFSRESL
//         DD DISP=SHR,DSN=DSN1010.SDSNEXIT
//         DD DISP=SHR,DSN=SYS1.DSN.V100.SDSNLOAD
//*
```

Last but not least, the SSM member name needs to be added to the IMS dependent region JCL.

Excerpt from IMS dependent region member

```
//*
//IVP11M11 EXEC PROC=DFSMPR,TIME=(1440),
//          NBA=6,
//          OBA=5,
//          SSM=SSM,                SUBSYS CONNECTION
//          SOUT='*',               SYSOUT CLASS
//          CL1=015,                TRANSACTION CLASS 1
//          TLIM=10,                MPR TERMINATION LIMIT
//          SOD=,                   SPIN-OFF DUMP CLASS
//          IMSID=IMSA,        IMSID OF IMS CONTROL REGION
//          PREINIT=DC,             PROCLIB DFSINTXX MEMBER
//          PRLD=DC,                PROCLIB DFSMPLXX MEMBER
//          PWFI=Y,                 PSEUDO=WFI
//          JVMOPMAS=DFSJVMMC,      PSEUDO=WFI
//          ENVIRON=DFSJVMEV        PSEUDO=WFI
//*
```

For completeness, the Db2 loadlibs need to be added to the IMS dependent region procedure.

Excerpt from IMS dependent region procedure member

```
//STEPLIB   DD DSN=SYS1.IMS.V14.&SYS2.DYNALLOC,DISP=SHR
//          DD DSN=GAEBLER.FU508T.PGMLIB,DISP=SHR
//*         DD DSN=SYS1.IMS.V14.&SYS2.PGMLIB.PDSE,DISP=SHR
//          DD DSN=SYS1.IMS.V14.&SYS2.SDFSRESL,DISP=SHR
//          DD DSN=SYS1.IMS.V14.&SYS2.SDFSJLIB,DISP=SHR
//          DD DISP=SHR,DSN=DSN1010.SDSNEXIT
//          DD DISP=SHR,DSN=SYS1.DSN.V100.SDSNLOAD
//          DD DISP=SHR,DSN=SYS1.DSN.V100.SDSNLOD2
//          DD DISP=SHR,DSN=SYS1.SCEERUN
//          DD DISP=SHR,DSN=SYS1.SCEERUN2
//DFSESL    DD DISP=SHR,DSN=SYS1.IMS.V14.&SYS2.DYNALLOC
//          DD DISP=SHR,DSN=SYS1.IMS.V14.&SYS2.SDFSRESL
//          DD DISP=SHR,DSN=DSN1010.SDSNEXIT
//          DD DISP=SHR,DSN=SYS1.DSN.V100.SDSNLOAD
//          DD DISP=SHR,DSN=SYS1.DSN.V100.SDSNLOD2
//          DD DISP=SHR,DSN=SYS1.SCEERUN
//          DD DISP=SHR,DSN=SYS1.SCEERUN2
//*
```

Make sure to check that the IMS procedures were updated for use of the ENVIRON and JVMOPMAS parameters.

6.4. IMS Connect

For some customers, IMS Connect has become the main entry point of calling IMS transactions. While this gateway is not directly related to Java interoperability in IMS, there are some options that affect processing in IMS and in combination with access to Db2, especially the OTMA security options.

6.4.1. Security related to Db2 access and USS permissions

Enabling Java in IMS allows JDBC calls out of your IMS transactions. Typically one would use static SQL from COBOL with package-based authorization. JDBC, however, behaves different. Users need to be granted to a package that enables JDBC as such. Further authorization is done by checking the authenticated user within Db2 if she has access to the requested resource. If the transaction comes in through OTMA and IMS Connect, it is necessary that a reusable access control environment element (ACEE) is set up to enable secondary authorization in Db2, such as based on the connected RACF group. When this function is enabled, IMS will internally force OTMA security level to FULL for any dependent region with a persistent JVM environment and with defined ESAF/Db2 capability. This allows you to keep /SECURE OTMA CHECK enabled for the IMS system.

Also, the reusable ACEE feature adds the APPL= parameter to the RACROUTE calls issued by OTMA security (DFSYRAC0). The APPL= parameter is now set to the IMSID instead of blanks. This would allow the RACF admin to limit RACF LOGON statistics for that APPL to whatever is appropriate instead of syncing RACF DB for every incoming OTMA request. Syncing RACF DB for every incoming OTMA request has caused serious IMS throughput degradation in some customer environments.

The reusable ACEE feature forces OTMA security level FULL in Db2/ESAF capable persistent JVM regions only (i.e. Java-enabled regions with ENVIRON= and JVMOPMAS= startup parameters specified).

- The implementation here is that the ACEE will be cloned into the IMS dependent region by IMS and made available to Db2 in field TCBSENV so that Db2 does not have to issue a RACROUTE call to create the ACEE. Db2 should just use the ACEE in field TCBSENV.
- We want to allow administrators to specify daily statisticss for applications which include APPL= on their verify request by using the APPL profile
- To save I/O to the RACF database, specify RACF-INITSTATS(DAILY) anywhere in the APPLDATA field of appropriate APPL class profile.
    - If APPLDATA has RACF-INITSTATS(DAILY) and this is not the user’s first logon of the day, skip recording statistics.
    - If no RACF-INITSTATS(DAILY) in APPLDATA, record statistics as directed by the SETROPTS INITSTATS setting.

- The APPL class must be active and RACLIST'ed to have consistent results due to ACEE caching.
    - RACLIST is the current recommendation for the APPL class.
- OTMA Full forces ACEE of the dependent region to change for authorization reasons. This situation can result in RACF violations, because system resource is now accessible without started task (STC) identities. For example, this situation requires the userid that runs the IMS transaction to be allowed to write dump datasets or to be allowed to read `.jar` files. In addition, log files are created with a permission of 755 by the first userid that uses them. As a consequence, no other user can write to it. This restriction was removed with the UMASK parameter added by PTF for APAR PI63800.
- Java dumps are created by default with the high-level qualifier (HLQ) of the active user in the format of `%uid.JVM.TDUMP.%job.D%y%m%d.T%H%M%S`. You might use the JAVA_DUMP_TDUMP_PATTERN option to specify the appropriate HLQs for the dumps.

In Db2, at a minimum, the default Db2 security exit must be active in order to support the SecAutId mechanism.

6.5. Language Environment

There are several requirements by the JVM for options to be used in Language Environment, such as storage tuning (given the fact that the JVM needs something around 100MB of heap size to even come up) and side effects such as the trap instructions generated during the JVM JIT compilation, which cause abends when the LE ERRCOUNT is set to something other than 0.

See the known issues and limitations in the documentation: https://www.ibm.com/docs/en/sdk-java-technology/8?topic=support-known-issues-limitations

Below is a list of options that was used in a customer production environment:

Sample CEEOPTS DD statement with LE options

```
//CEEOPTS  DD *
ANYHEAP(2M,1M,ANY,KEEP)
HEAP(80M,4M,ANY,KEEP,1M,512K)
HEAPPOOLS(ON,8,10,32,10,128,10,256,10,1024,10,2048,10,0,10,0,
10,0,10,0,10,0,10,0,10)
STACK(64K,64K,ANY,KEEP,512K,256K)
STORAGE(00,NONE,NONE,8K)
THREADSTACK(OFF,64K,16K,ANY,KEEP,128K,128K)
TERMTHDACT(UAIMM)
ERRCOUNT(0)
```

```
RPTOPTS(OFF)
RPTSTG(OFF)
//*
```

Please note this is an example for a CANCEL_PGM=Y environment, where STORAGE(00,…) causes the storage to be zeroed out on a get heap request, which is usually not required if working storage and local storage were written to have initial values. For those environments the STORAGE option should be set to STORAGE(NONE,…).

For more information about the meaning of those options refer to the z/OS Language Environment documentation:

https://www.ibm.com/docs/en/zos/2.4.0?topic=options-using-language-environment-runtime

The options XPLINK(ON) and POSIX(ON) are set by the IMS runtime and do not need to be specified.

6.6. z/OS UNIX System Services (USS)

For USS, some of the parameters in the BPXPRMxx member of the SYS1.PARMLIB data set affect successful Java operation by imposing limits on resources that are required. We will not discuss them here, but make sure you check the following documentation: https://www.ibm.com/docs/en/sdk-java-technology/8?topic=systems-bpxprm-settings-zos-only

6.6.1. Threads and tasks

While IMS work is single-threaded, when adding the JVM to an IMS region, there will be two task control blocks (TCBs) in the address space. In addition, since the JVM uses POSIX threads and they need to be dubbed into the UNIX environment, z/OS monitors will show those threads and there is a small CPU overhead for managing those threads.

Some dump management tools will also struggle with dumps created in this environment, so check carefully if existing dump handling and debugging tools for traditional languages provides the necessary functionality when JVM is added to IMS regions.

6.7. IMS DB and DL/I

There is no special requirements for IMS DB access and IMS DB calls in a mixed-language environment. The only thing to watch out for is the fact that when switching between Java and the traditional language, such as DL/I, the other language might change or have changed the database position. So never rely on the DB position from the previous method invocation when accessing IMS DB.

For more information about IMS DB access from Java refer to:
https://www.ibm.com/docs/en/ims/15.2.0?topic=drivers-programming-ims-universal-jdbc-driver

6.8. Db2 Java stored procedures

By creating Db2 Java stored procedures and calling them from traditional IMS applications, there is a possibility to mix COBOL and Java. This scenario is a viable option when there is just a small portion of Java code that needs to be executed, which doesn't justify the resource and environment setup work required to add a JVM to IMS regions.

It is still necessary to marshall between the copybook and its byte array representation that is passed to the Java code. This could be done by using simple SQL types, Strings or byte arrays.

Until Db2 V11 or if the 31-bit JVM is used for the Db2 Java stored procedures, NUMTCB is equal to the number of JVMs in the z/OS Workload Manger-managed stored procedure address space. With Db2 V11 and using the 64-bit JVM, a single JVM can be used to exploit worker threads.

6.9. IMS synchronous program switch

With IMS Version 13 and later, there is a feature called IMS synchronous program switch, which allows an IMS transaction to call another IMS transaction (such as a Java-only IMS transaction running in a JMP region) and wait until that transaction finishes before continuing with the traditional language.

Refer to the IMS documentation for more information:
https://www.ibm.com/docs/en/ims/15.2.0?topic=data-implementing-synchronous-callout-function

There is still the requirement to marshall between the copybook and its byte array representation that is passed to the Java code.

6.10. IMS Java Dependent Region resource adapter

When interfacing with Java, all the functionality from the IMS Java Dependent resource adapter, such as Java wrappers for most of the IMS DB, TM and System calls, can be used in a mixed environment with Java.

However, please note that accessing a PCB from other languages might change the values associated with it, such as the destination for a modifiable alternate PCB or positioning for an IMS DB PCB.

6.11. WebSphere Optimized Local Adapters

By using WebSphere Optimized Local Adapters (WOLA) and EJBs and calling them from traditional IMS applications, there is also a possibility to mix COBOL and Java. This scenario is a viable option if there is just a small portion of Java code that needs to be executed, which doesn't justify the resource and environment setup work required to add a JVM to IMS regions. Furthermore, this option is also the first choice in case the Java code requires a special container with Java EE or EJB functionality because IMS is, from Java's point of view, just a J2SE container without Java EE capabilities.

There is still the requirement to marshall between the copybook and its byte array representation that is passed to the Java code.

For more information, refer to the WOLA section in the architecture chapter.

6.12. Summary

In this chapter, we shared our experience from working with clients about configuration considerations for IMS regions that are Java-enabled. Key configuration requirements or potential pitfalls are also discussed for various scenarios, from JVM setup in LE and in a classic IMS TM environment, to setup for external access to Db2® data, and to scenarios when use of Db2 stored procedures, IMS Java Dependent Region resource adapter, or WebSphere Application Server is involved.

CHAPTER 7

7. Problem determination

IBM® provides a set of monitoring and diagnostic tools in IBM Support Assistant to assist in the understanding, monitoring, and problem diagnosis of applications and deployments for Java runtime environments.

IBM® Developer for z/OS, formerly known as IBM Rational® Developerfor z Systems (RDz), as well as IBM Rational Application Developer (RAD) have an integrated Java™ debugging perspective that allows a developer to connect to a remote JVM and do interactive debugging of the application.

The IMS™ Batch Terminal Simulator tool also supports IMS Java batch applications (JBPs). Therefore, the debugging of DL/I calls is possible with the same tool that has existed for traditional IMS batch applications. The new version of IMS Batch Terminal Simulator has a new Eclipse-based graphical user interface that allows for more integration, for example, when the Java code is developed by using an Eclipse-based GUI.

For further details on IBM IMS Batch Terminal Simulator for z/OS refer to: https://www.ibm.com/products/ims-batch-terminal-simulator

7.1. Gathering data

The key step to troubleshooting and determining the root cause of a problem is data gathering. We will cover some commonly used tools that can help you gather diagnostic data.

7.1.1. IBM Support Assistant

IBM Support Assistant is a web-based application that helps you gather, organize, analyze, and diagnose issues with software. Additional tools can be installed as plug-ins into IBM Support Assistant.

For more information about installing and using IBM Support Assistant, see the documentation at: https://www.ibm.com/docs/en/support-assistant/5.0.0?topic=start-overview

Please note that there is still the out of service IBM Support Assistant V4.1 available for download, which is a Java fat client application and can be used and installed directly on a workstation and does not require a server. Users are encouraged to adopt IBM Support Assistant V5 Team Server as a replacement for the IBM Support Assistant V4 Workbench. The screenshots in this book are from the V4 Workbench version.

The Health Center can be used as a diagnostic tool for applications running in

7.1.2. IBM Health Center

The Health Center can be used as a diagnostic tool for applications running in the Java virtual machine (JVM). You can also use the Health Center to monitor the JVM. For installation information, see: https://www.ibm.com/docs/en/mon-diag-tools?topic=monitoring-diagnostic-tools-health-center

The configuration is simple. A port is defined in the JVM configuration, which is then used by the Windows-based tool to connect to the JVM. One drawback is the need for a different port for every IMS region. There is no management console or variable support to achieve this, so the best way is for every IMS Java region to have a separate configuration member that has a specific, hardcoded port number.

The z/OS JVM parameter `-Xhealthcenter` is used to configure the port. Select a port of choice and add this parameter to the Java configuration:

```
-Xhealthcenter:port=1982
```

The IMS region job log should then print this message:

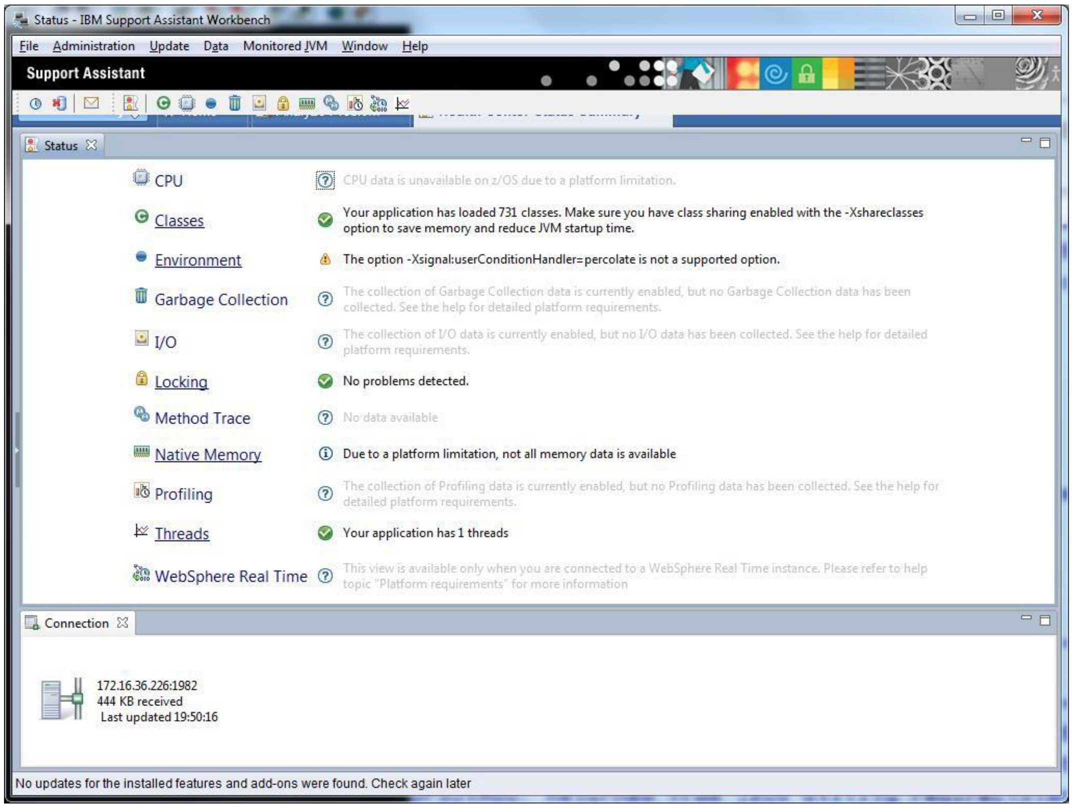
```
UTE115: Trace buffer discarded. The count of discarded buffers
is printed at VM shutdown.
```

To connect from the Health Center to the port, the following steps are required:

- Install IBM Support Assistant Workbench
- Install and enable IBM Health Center plug-in
- Start IBM Support Assistant
- Launch activity: Analyze problem
- Select IBM Health Center
- Connect to the JVM of a specific IMS region

The first two steps can be skipped, when the Health Center is already installed. The Health Center profiling view is displayed.

Figure 26. Health Center profiling view in IBM Support Assistant


Status - IBM Support Assistant Workbench
File Administration Update Data Monitored JVM Window Help
Support Assistant
Status
Connection

CPU	CPU data is unavailable on z/OS due to a platform limitation.
Classes	Your application has loaded 731 classes. Make sure you have class sharing enabled with the -Xshareclasses option to save memory and reduce JVM startup time.
Environment	The option -Xsignal:userConditionHandler=percolate is not a supported option.
Garbage Collection	The collection of Garbage Collection data is currently enabled, but no Garbage Collection data has been collected. See the help for detailed platform requirements.
I/O	The collection of I/O data is currently enabled, but no I/O data has been collected. See the help for detailed platform requirements.
Locking	No problems detected.
Method Trace	No data available
Native Memory	Due to a platform limitation, not all memory data is available
Profiling	The collection of Profiling data is currently enabled, but no Profiling data has been collected. See the help for detailed platform requirements.
Threads	Your application has 1 threads
WebSphere Real Time	This view is available only when you are connected to a WebSphere Real Time instance. Please refer to help topic "Platform requirements" for more information

```
172.16.36.226:1982
444 KB received
 Last updated 19:50:16
```

No updates for the installed features and add-ons were found. Check again later

<u>Note</u>: The Health Center is not a monitoring tool. It can only connect to and analyze data from one JVM or one IMS region. There is no aggregation functionality available.

7.1.3. IBM HeapAnalyzer

You can use IBM HeapAnalyzer to analyze heap memory leaks in Java applications. Possible failures in that situation could be either S0D4 abends or java.lang.OutOfMemoryError exceptions. You can download HeapAnalyzer and find the instructions for how to use it in IBM Support at: https://www.ibm.com/support/pages/ibm-heapanalyzer.

A heap dump in an IMS JVM-enabled environment can be generated with the standard procedures for generating heap dumps in UNIX or z/OS environments.
In order to do an analysis, it is required to generate a heap dump of the JVM while the workload that is causing the problem is running.

Here are the steps:

1. Set the -Xdump option as shown in the following example.

Example 3. Option to let create a heap dump in phd format by a user event

```
-Xdump:heap:events=user,opts=phd
```

2. In the environment member, add the option shown in the following example.

Example 4. Environment variable to be put into DFSJVMEV to enable Java heap dumps

```
IBM_HEAPDUMP=true
```

3. Use the kill -3 or kill -QUIT command followed by the process ID of the JVM and a heap dump can be produced.

   The PTF for APAR PM50971 prints the process ID (PID) for the JVM in the IMS job log as shown in the following example.

Example 5. Sample output from the IMS job log printing the JVM PID

```
DFSJVM00: -Xmaxf0.8
DFSJVM00: -Xminf0.3
DFSJVM00: -Xmx64M
DFSJVM00: -Xmso512k
DFSJVM00: -Xss256k
DFSJVM00: -Xms32M
DFSJVM00: -Xcodecache10M
DFSJVM00: -Xshareclasses:name=cobolims1
DFSJVM00: -Xscmx64M
DFSJVM00: -Xscminaot16M
DFSJVM00: +++++++++++++++++++++++++++++++++++++++++
DFSJVM00: ++   End   Contents of -Xoptionsfile   ++
DFSJVM00: +++++++++++++++++++++++++++++++++++++++++
DFSJVM00: JVM initialization started:  Fri Apr 20 17:22:59.793  2012
DFSJVM00: JVM initialization complete: Fri Apr 20 17:23:00.118  2012
DFSJVM00: Process ID::::::: PID =2175
DFSJVM00: Parent Process ID: PPID=1
DFSJVM00: Process Group ID:: PGID=2175
```

   This makes it easy to send signals to the JVM with the `kill` command in order to create heap dumps in the format required for the IBM HeapAnalyzer.

4. Issue the kill -QUIT or kill -3 command with the PID from the IMS job log. The messages shown in the following example will display in the IMS job log. It indicates the successful creation of the PHD Heapdump file and other diagnostic information.

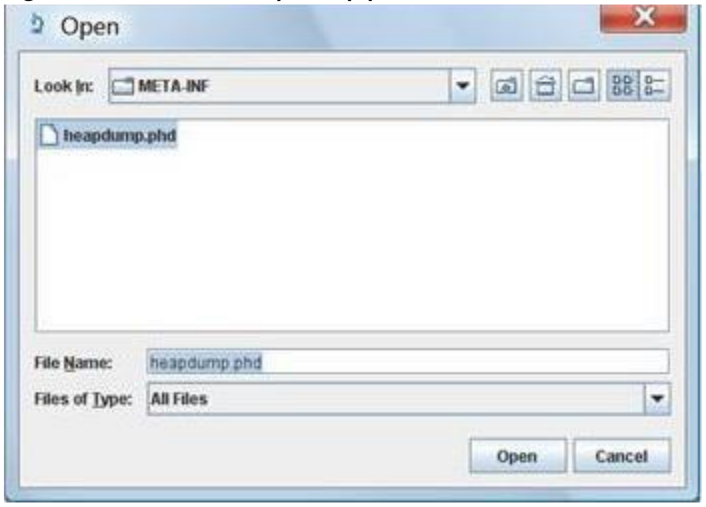
Example 6. Output indicating the creation of a successful heap dump

```
JVMDUMP039I Processing dump event "user", detail "" at 2012/07/03
15:35:58 - please wait.
JVMDUMP032I JVM requested System dump using
```

```
'GAEBLER.JVM.TDUMP.CJTSTMPP.D120703.T153558' in response to an event
IEATDUMP in progress with options
SDATA=(LPA,GRSQ,LSQA,NUC,PSA,RGN,SQA,SUM,SWA,TRT)
IEATDUMP success for DSN='GAEBLER.JVM.TDUMP.CJTSTMPP.D120703.T153558'
JVMDUMP010I System dump written to
GAEBLER.JVM.TDUMP.CJTSTMPP.D120703.T153558
JVMDUMP032I JVM requested Heap dump using
'/u/gaebler/heapdump.20120703.153558..50334771.0003.phd' in response to
an event
JVMDUMP010I Heap dump written to
/u/gaebler/heapdump.20120703.153558.50334771.0003.phd
JVMDUMP032I JVM requested Java dump using
'/u/gaebler/javacore.20120703.153558.50334771.0004.txt' in response to
an event
JVMDUMP010I Java dump written to
/u/gaebler/javacore.20120703.153558.50334771.0004.txt' in response to
an
JVMDUMP013I Processed dump event "user", detail "".
```

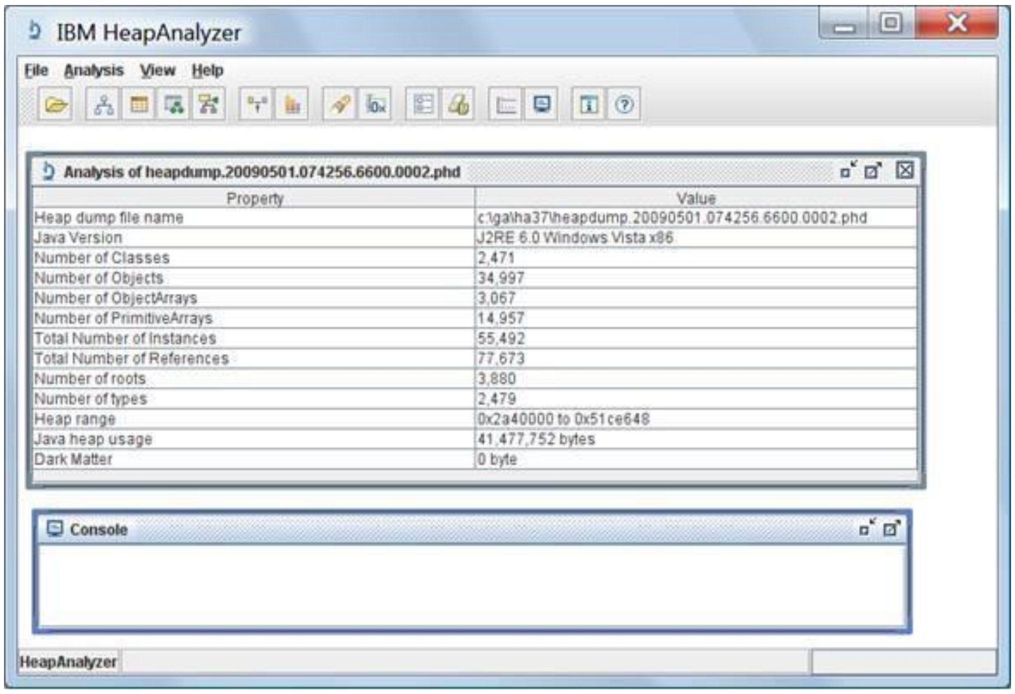
5. The .phd file can now be downloaded in binary format and loaded into the IBM HeapAnalyzer.
6. Start the IBM Support Assistant, and click Launch Activity -> Analyze Problem, select HeapAnalyzer and click Launch.
7. Select a heap dump file in the local file system by clicking Browse and choose the file under the Remote Artifact Browse tab as shown in the following figure.

Figure 27. Select the heap dump pdf file location



8. Select the heap dump file from the directory where the binary download of the .phd file was stored, and select the .phd file.
9. Click Next and the analysis will start. The IBM HeapAnalyzer will start and display a summary view of the data gathered.

Figure 28. Summary view of IBM HeapAnalyzer



Analysis of heapdump.20090501.074256.6600.0002.phd

Property	Value
Heap dump file name	c:\ga\ha37\heapdump.20090501.074256.6600.0002.phd
Java Version	J2RE 6.0 Windows Vista x86
Number of Classes	2,471
Number of Objects	34,997
Number of ObjectArrays	3,067
Number of PrimitiveArrays	14,957
Total Number of Instances	55,492
Total Number of References	77,673
Number of roots	3,880
Number of types	2,479
Heap range	0x2a40000 to 0x51ce648
Java heap usage	41,477,752 bytes
Dark Matter	0 byte

This example shows how a JVM running attached to IMS can be analyzed with standard tooling just like any other JVM.

7.1.4. Rational Agent Controller, Rational Profiling, and HealthCenter plugin

Starting with Rational Application Developer V9, both trace-based profiling and sample-based profiling are available. Both require the activation of JVM tool interface-based (JVMTI-based) libraries on z/OS and use the Rational Agent Controller address space on z/OS. The benefit is that not every IMS region requires its own port (like with using `-Xhealthcenter` parameter); instead, Rational Agent Controller has only one port and allows you to select the JVMs based on the process id (please remember that the process id of the JVM is printed in the IMS job log after the program is started).

For information on how to download and install the Rational Agent Controller V9, refer to: https://www.ibm.com/support/pages/node/313945

Please note that it is required to start the Java class in the IMS region by running the program as either batch or online program. It is not possible to start the program or the Java class execution from the Rational Application Developer Wizards.

The profiling is also available from IBM Developer for z/OS, but itrequires the license and the installation version that includes Rational Application Developer.

7.1.5.JConsole

Another way to perform diagnostics is to use JConsole. It is a free graphical tool that is available with the standard JDKs that allows you to monitor the behavior of Java applications. The JDK provides the functionality to allow JConsole to connect to it.

JConsole is a utility that is part of the standard JDKs on distributed platforms. However, the server part that delivers the information is also implemented in the JDK for z/OS.

7.1.5.1. Starting and running JConsole:

JConsole can be activated by adding the JVM options to the JVM configuration in the IMS region. JVM options required to activate JConsole are as follows:

```
-Djavax.management.builder.initial=
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.port=1099
```

When JVM is active, switch to the workstation. Look for the Java home directory of a Java SDK (the JRE does not come with JConsole) and go to the <JAVA_HOME>/bin directory. In that directory locate an executable called `jconsole.exe` or `jconsole` on UNIX platforms. It can be invoked by double clicking it or it can be invoked from the command line. Once it is invoked, a window opens for a new connection.

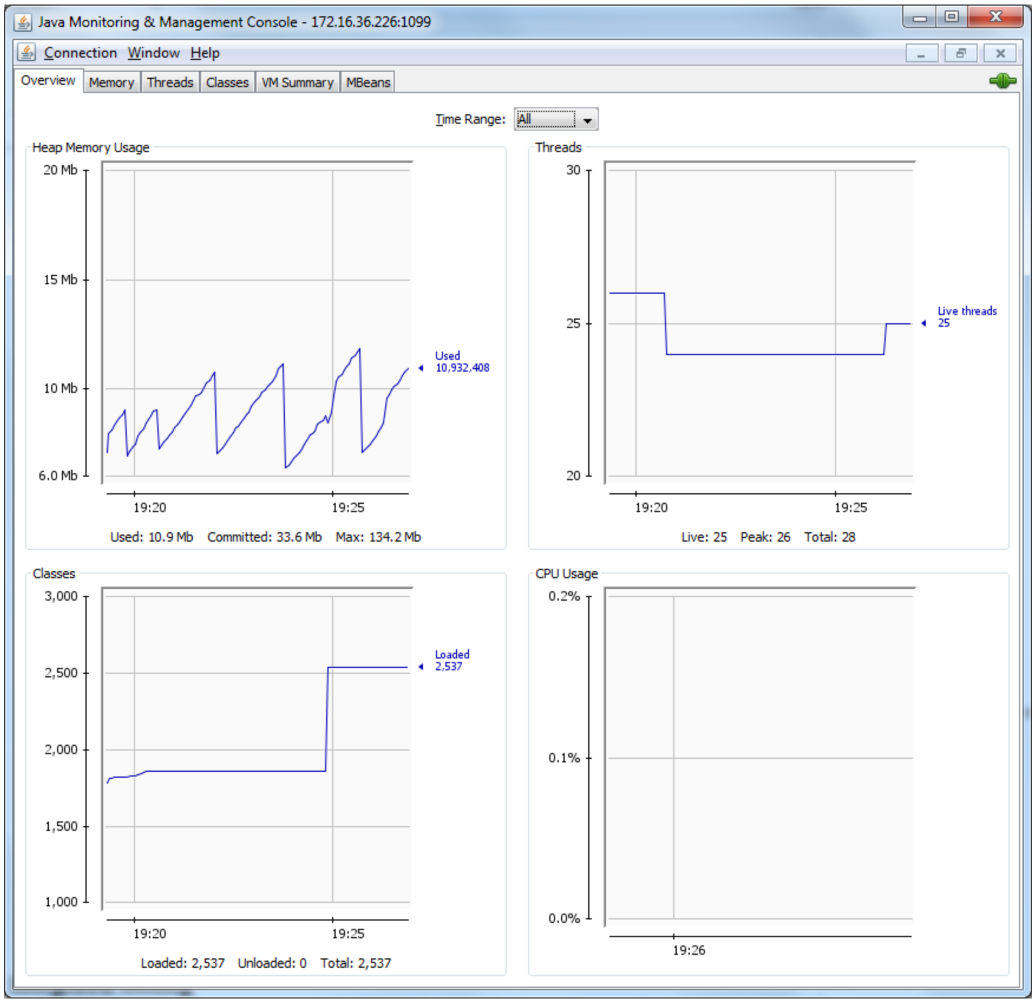
7.1.5.2. JConsole New Connection wizard

You need to enter the host name or IP address and the port number that was defined in the IMS regions JVM properties. The port should not be shared between multiple IMS regions. Each region requires its own port. Username and password are not required here because the security has been disabled with the configuration settings mentioned earlier. New versions try to connect using SSL, but when fail they ask to switch to the non-secured connection and it works.

After the connection is finished, the graphical interface opens. You can look at certain things in the JVM, such as heap, threads, and others.

Initially the Overview display opens.

Figure 29. Summary view from JConsole while running some workload



For more information about what diagnostics are possible with JConsole, refer to the documentation at: https://www.ibm.com/docs/en/sdk-java-technology/8?topic=reference-using-jconsole

Restriction: Not all functions that the JConsole provides can be used while connected to an JVM that was written by IBM and runs on z/OS.

7.2. Integrated tooling

Debugging is often done directly in the integrated development environment (IDE). In this section we provide an example on how to debug an IMS Java batch application running on the mainframe using IBM Developer for z/OS. The debugging of an IMS online application is similar.

7.2.1. IBM Developer for z/OS Java debugging

In order to enable debugging two additional Java configuration switches are required for JVM:

- Turn on the debug option and tell the JVM to be the server for the debug session
- Suspend execution of JVM until after a debug connection socket from the client is attached

The following steps show how to debug a Java BMP (batch message processing) region:

1. Issue the `-Xdebug` option to tell JVM to turn on debug mode and then issue the `-X runjdwp` option with the parameters: `server=y, suspend=y, address=7778`, as shown in the following example.

Example 7: JVM option for JVM wait

```
-Xdebug
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=7778
```

   These parameters tell the JVM that there is a server waiting for an incoming socket connections on port 7778 for a debugging session and to suspend any work until the debugging session with the Eclipse client (can also be RAD or RDz for Java debugging) is established. This means that without a connection the JVM will not do any work. Depending on the Eclipse version, it is possible that the JVM is initially suspended and will not start executing until resume has been clicked.

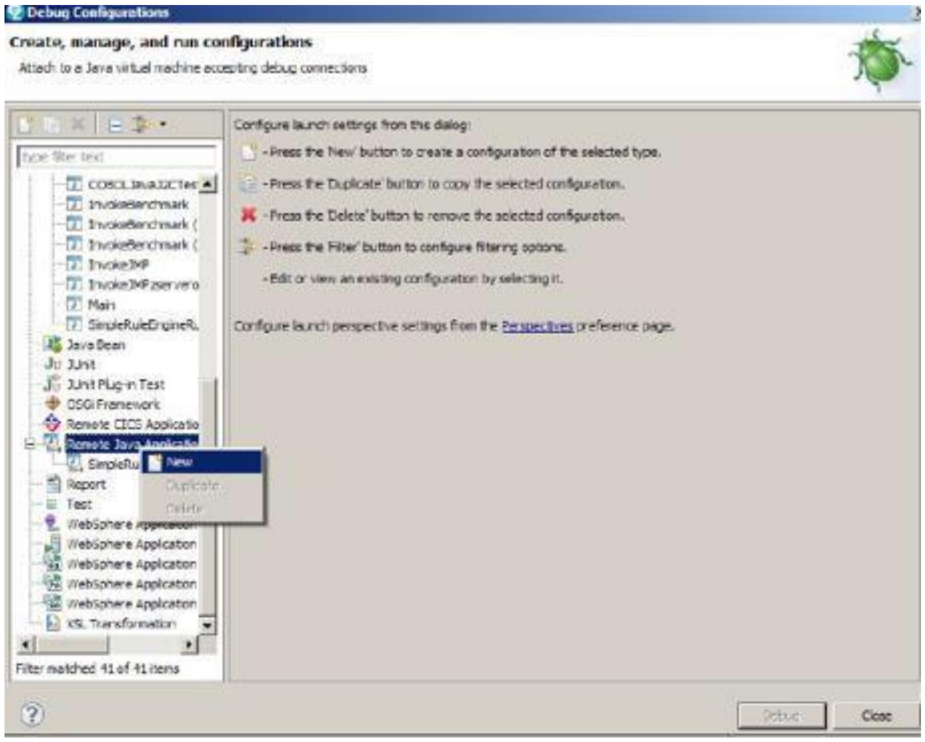
2. After the configuration change, start the IMS BMP. It will produce the following output:

   ```
   Listening for transport dt_socket at address: 7778
   ```

   We used a PSB which points to a program called SimpleRuleEngineRunner. Then it is required to select the Java class source in the RDz workspace.

3. Select **Debug As** > **Debug Configurations**. The Debug Configurations window opens.

Figure 30. Debug Configurations window



Create, manage, and run configurations

Attach to a Java virtual machine accepting debug connections

Configure launch settings from this dialog:

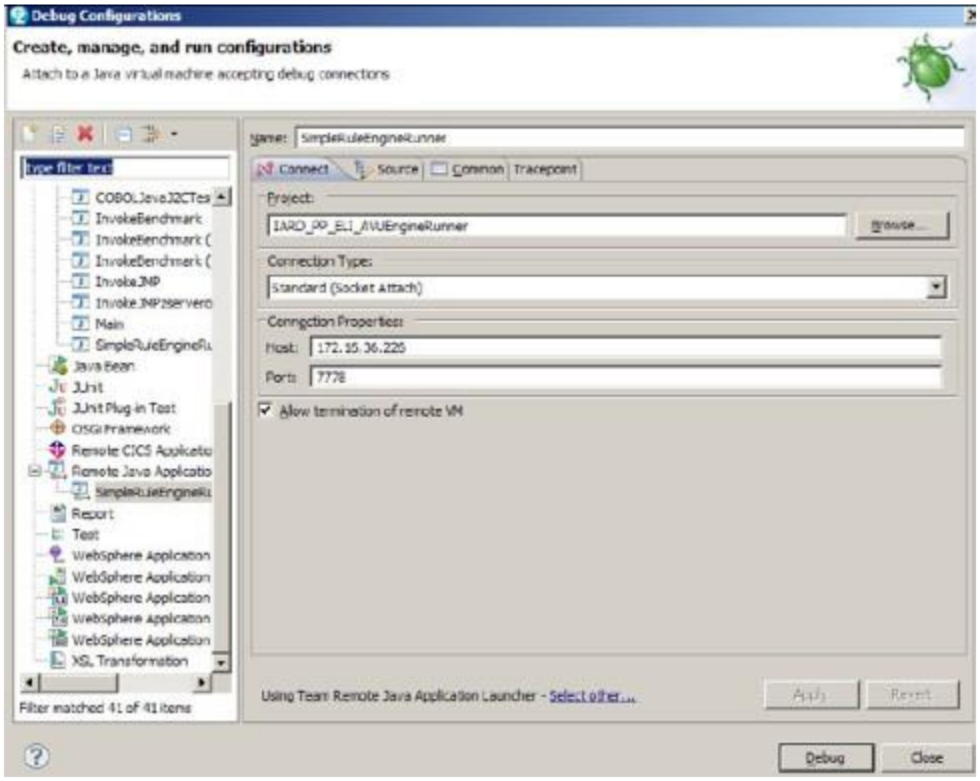
- Press the 'New' button to create a configuration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.

Configure launch perspective settings from the Perspectives preference page.

4. Fill out the server hostname or IP address and the port number. Select **Socket Attach** as the connection type, and check **Allow termination of the remote VM** to avoid hanging IMS BMPs at the end of program execution or if the debug session socket breaks.

5. Click **Apply** to save the values and then click **Debug** to connect to the waiting JVM on z/OS.

Figure 31. Remote Java application properties



Debug Configurations

Create, manage, and run configurations

Attach to a Java virtual machine accepting debug connections

Name: SimpleRuleEngineRunner

Project: IARD_PP_ELI_AVUEngineRunner

Connection Type: Standard (Socket Attach)

Connection Properties:

Host: 172.16.36.226

Port: 7778

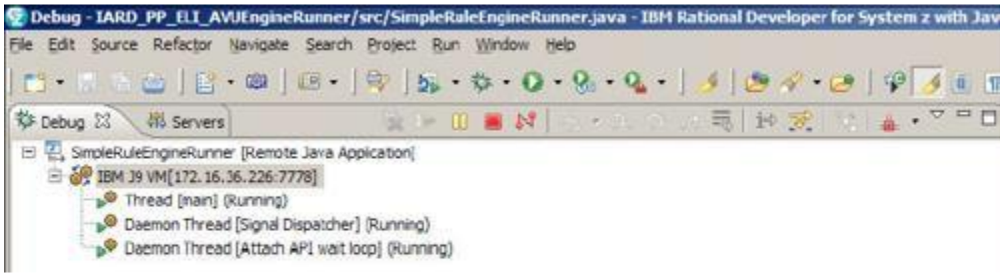
☑ Allow termination of remote VM

Using Team Remote Java Application Launcher - Select other...

Filter matched 41 of 41 items

Note: If there is no automatic switch to the debug perspective within RDz then switchthe perspective manually and the debug session will show up as shown in Figure 2-3

Figure 32. Remote debug connection



Debug - IARD_PP_ELI_AVUEngineRunner/src/SimpleRuleEngineRunner.java - IBM Rational Developer for System z with Jav

SimpleRuleEngineRunner [Remote Java Application]

IBM J9 VM[172.16.36.226:7778]

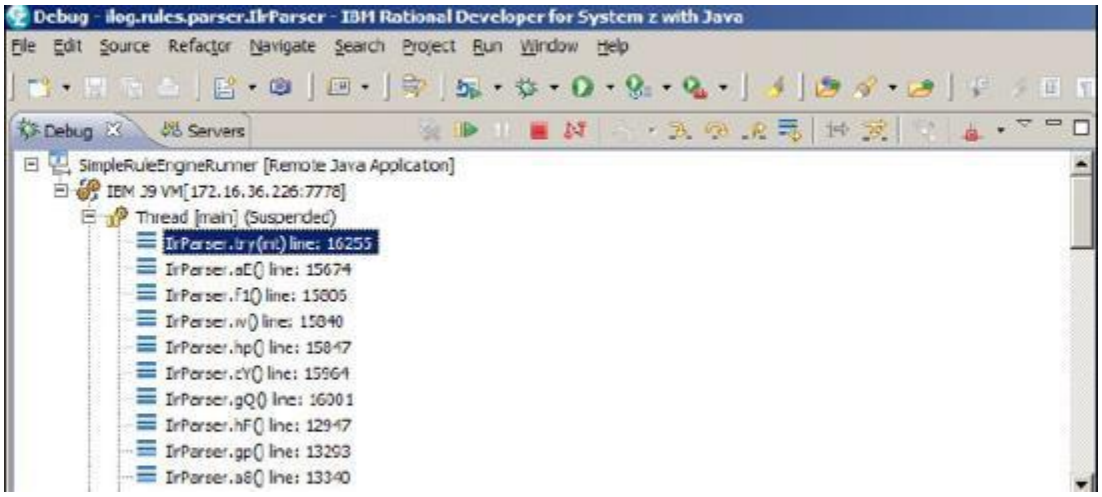
Thread [main] (Running)

Daemon Thread [Signal Dispatcher] (Running)

Daemon Thread [Attach API wait loop] (Running)

Initially because there is no breakpoint specified, the Java program will start running as soon as the debug client has connected to the JVM. If the network connection is slow enough or the suspend icon is clicked fast enough, the execution will be suspended and it can be determined where in the source code the execution has been suspended.

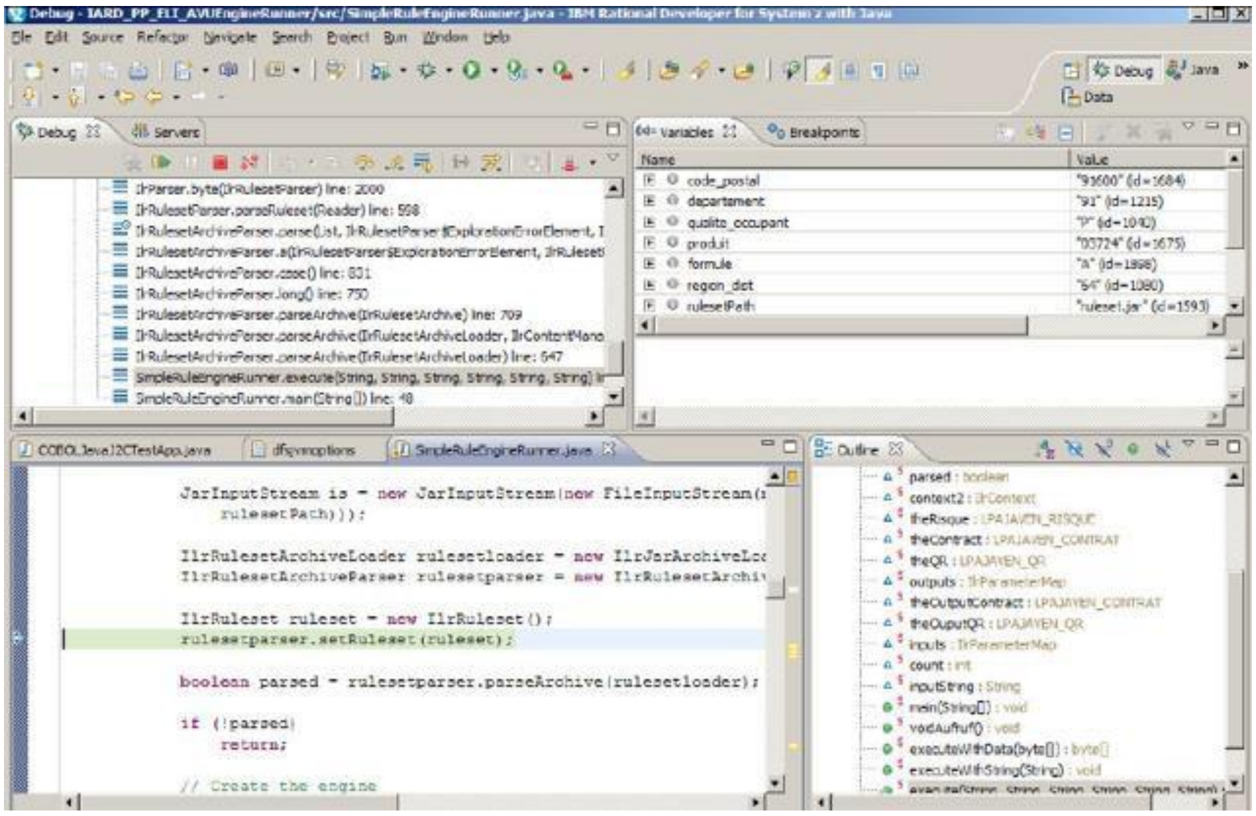
Figure 33. Suspended remote JVM execution


Debug - ilog.rules.parser.IlrParser - IBM Rational Developer for System z with Java
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Servers

```
SimpleRuleEngineRunner [Remote Java Application]
  IBM J9 VM[172.16.36.226:7778]
    Thread [main] (Suspended)
      IlrParser.try(int) line: 16255
      IlrParser.aE() line: 15674
      IlrParser.f1() line: 15805
      IlrParser.w() line: 15840
      IlrParser.hp() line: 15847
      IlrParser.cY() line: 15964
      IlrParser.gQ() line: 16001
      IlrParser.hF() line: 12947
      IlrParser.gp() line: 13293
      IlrParser.a8() line: 13340
```

6. Scroll down to the uppermost level class. You can see where in the programs source code the execution has been suspended. It also displays the current state/values of variables/objects.

Figure 34. Suspended remote Java program with source code position and variable/object values


Debug - IARD_PP_ELT_AVUEngineRunner/src/SimpleRuleEngineRunner.java - IBM Rational Developer for System z with Java
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Java
Data
Debug
Servers
Variables
Breakpoints
COBOLJavaJ2CTestApp.java
dfsjvmoptions
SimpleRuleEngineRunner.java
Outline

```
IlrParser.byte(IlrRulesetParser) line: 2000
IlrRulesetParser.parseRuleset(Reader) line: 598
IlrRulesetArchiveParser.parse(List, IlrRulesetParser$ExplorationErrorElement, I
IlrRulesetArchiveParser.a(IlrRulesetParser$ExplorationErrorElement, IlrRuleset
IlrRulesetArchiveParser.case() line: 831
IlrRulesetArchiveParser.long() line: 750
IlrRulesetArchiveParser.parseArchive(IlrRulesetArchive) line: 709
IlrRulesetArchiveParser.parseArchive(IlrRulesetArchiveLoader, IlrContextMana
IlrRulesetArchiveParser.parseArchive(IlrRulesetArchiveLoader) line: 547
SimpleRuleEngineRunner.execute(String, String, String, String, String, String) li
SimpleRuleEngineRunner.main(String[]) line: 48
```

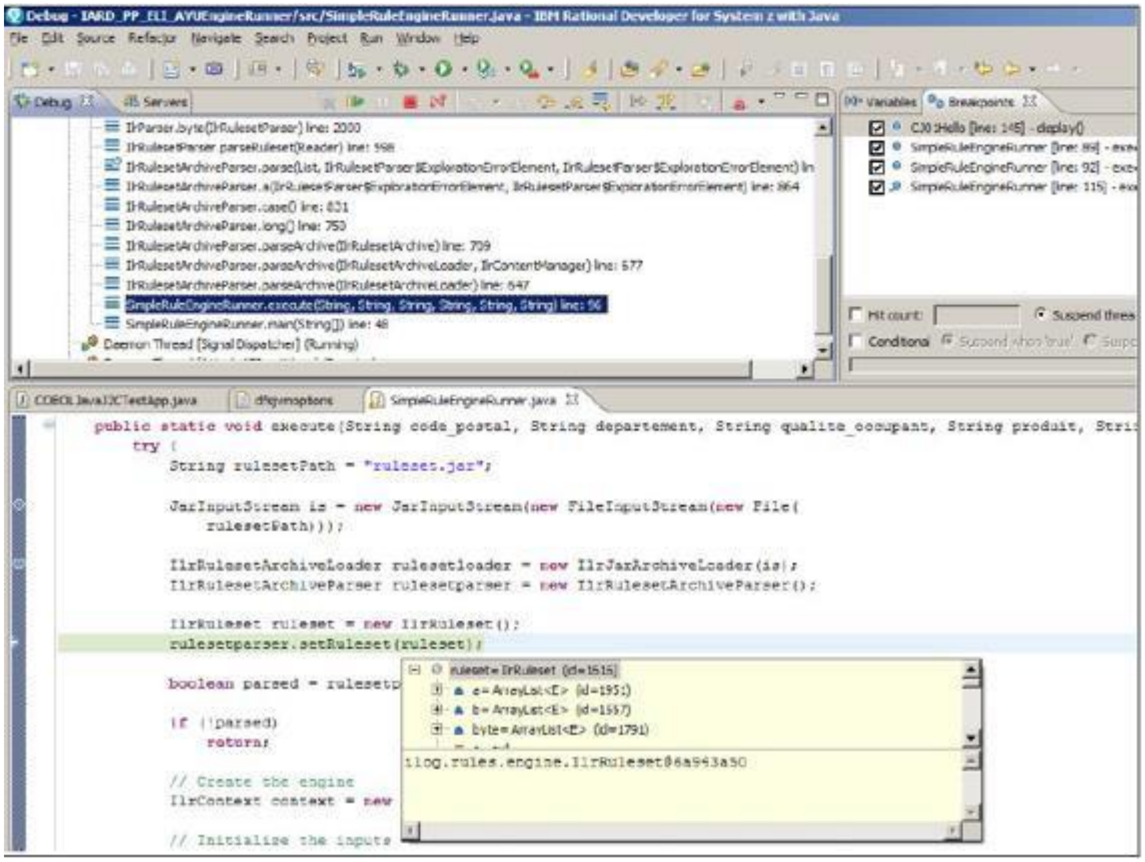
Name	Value
code_postal	"91600" (id=1684)
departement	"91" (id=1215)
qualite_occupant	"P" (id=1040)
produit	"03724" (id=1675)
formule	"A" (id=1898)
region_dst	"54" (id=1080)
rulesetPath	"ruleset.jar" (id=1593)

```
JarInputStream is = new JarInputStream(new FileInputStream(r
    rulesetPath)));

IlrRulesetArchiveLoader rulesetloader = new IlrJarArchiveLoa
IlrRulesetArchiveParser rulesetparser = new IlrRulesetArchiv

IlrRuleset ruleset = new IlrRuleset();
rulesetparser.setRuleset(ruleset);

boolean parsed = rulesetparser.parseArchive(rulesetloader);

if (!parsed)
    return;

// Create the engine
```

```
parsed : boolean
context2 : IlrContext
theRisque : LPAJAVEN_RISQUE
theContract : LPAJAVEN_CONTRAT
theQR : LPAJAVEN_QR
outputs : IlrParameterMap
theOutputContract : LPAJAVEN_CONTRAT
theOuputQR : LPAJAVEN_QR
inputs : IlrParameterMap
count : int
inputString : String
main(String[]) : void
voidAufruf() : void
executeWithData(byte[]) : byte[]
executeWithString(String) : void
```

7. When the execution is suspended or stopped at a breakpoint, the values of objects and variables can also be seen by moving the mouse over variables/objects.

Figure 35. Context sensitive source code view of current variable/object values at suspend/breakpoint time



Debug - IARD_PP_ELI_AYUEngineRunner/src/SimpleRuleEngineRunner.java - IBM Rational Developer for System z with Java
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Servers
Variables
Breakpoints
COBOL Java J2CTestApp.java
dfgvmoptions
SimpleRuleEngineRunner.java

```
IlrParser.byte(IlrRulesetParser) line: 2000
IlrRulesetParser.parseRuleset(Reader) line: 598
IlrRulesetArchiveParser.parse(List, IlrRulesetParser$ExplorationErrorElement, IlrRulesetParser$ExplorationErrorElement) lin
IlrRulesetArchiveParser.a(IlrRulesetParser$ExplorationErrorElement, IlrRulesetParser$ExplorationErrorElement) line: 864
IlrRulesetArchiveParser.case() line: 831
IlrRulesetArchiveParser.long() line: 753
IlrRulesetArchiveParser.parseArchive(IlrRulesetArchive) line: 709
IlrRulesetArchiveParser.parseArchive(IlrRulesetArchiveLoader, IlrContentManager) line: 677
IlrRulesetArchiveParser.parseArchive(IlrRulesetArchiveLoader) line: 647
SimpleRuleEngineRunner.execute(String, String, String, String, String, String) line: 96
SimpleRuleEngineRunner.main(String[]) line: 48
Daemon Thread [Signal Dispatcher] (Running)
```

```
☑ CJ0 1Hello [line: 145] - display()
☑ SimpleRuleEngineRunner [line: 89] - exe
☑ SimpleRuleEngineRunner [line: 92] - exe
☑ SimpleRuleEngineRunner [line: 115] - exe
```

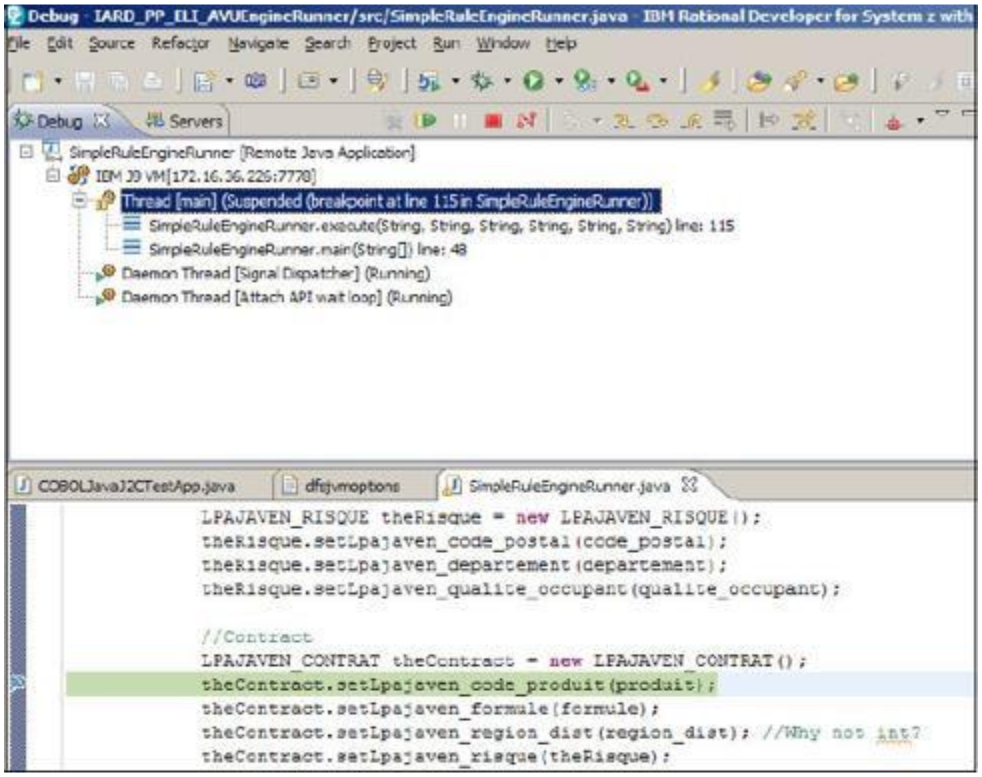
☐ Hit count: [ ] (●) Suspend threa
☐ Conditional (●) Suspend when 'true' ( ) Susp

```
    public static void execute(String code_postal, String departement, String qualite_occupant, String produit, Stri
        try {
            String rulesetPath = "ruleset.jar";

            JarInputStream is = new JarInputStream(new FileInputStream(new File(
                rulesetPath)));

            IlrRulesetArchiveLoader rulesetloader = new IlrJarArchiveLoader(is);
            IlrRulesetArchiveParser rulesetparser = new IlrRulesetArchiveParser();

            IlrRuleset ruleset = new IlrRuleset();
            rulesetparser.setRuleset(ruleset);

            boolean parsed = rulesetp

            if (!parsed)
                return;

            // Create the engine
            IlrContext context = new

            // Initialize the inputs
```

```
⊟ ruleset= IlrRuleset (id=1616)
   ⊞ ▲ e= ArrayList<E> (id=1951)
   ⊞ ▲ b= ArrayList<E> (id=1557)
   ⊞ ▲ byte= ArrayList<E> (id=1791)
ilog.rules.engine.IlrRuleset@6a943a50
```

8. When breakpoints are set and reached, the execution stops at the breakpoint and the line of code is displayed in the source view.

Figure 36. Debug session suspended at breakpoint with source code view


Debug - IARD_PP_ELI_AVUEngineRunner/src/SimpleRuleEngineRunner.java - IBM Rational Developer for System z with
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Servers
COBOLJavaJ2CTestApp.java
dfsjvmoptions
SimpleRuleEngineRunner.java

```
SimpleRuleEngineRunner [Remote Java Application]
  IBM J9 VM[172.16.36.226:7778]
    Thread [main] (Suspended (breakpoint at line 115 in SimpleRuleEngineRunner))
      SimpleRuleEngineRunner.execute(String, String, String, String, String, String) line: 115
      SimpleRuleEngineRunner.main(String[]) line: 48
    Daemon Thread [Signal Dispatcher] (Running)
    Daemon Thread [Attach API wait loop] (Running)
```

```
LPAJAVEN_RISQUE theRisque = new LPAJAVEN_RISQUE();
theRisque.setLpajaven_code_postal(code_postal);
theRisque.setLpajaven_departement(departement);
theRisque.setLpajaven_qualite_occupant(qualite_occupant);

//Contract
LPAJAVEN_CONTRAT theContract = new LPAJAVEN_CONTRAT();
theContract.setLpajaven_code_produit(produit);
theContract.setLpajaven_formule(formule);
theContract.setLpajaven_region_dist(region_dist); //Why not int?
theContract.setLpajaven_risque(theRisque);
```

9. When the step through ends, the JVM and the BMP on z/OS will terminate and the debug session ends.

7.2.2. Debugging with IBM z/OS Debugger (successor of IBM Debug Tool for z/OS)

IBM z/OS Debugger is a separately priced tool. At the time of this writing, the current Version is 14. It's the successor of IBM Debug Tool V13. It has both a 3270 user interface and an eclipse GUI plugin.

The COBOL side of the application usually also requires some sort of debugging. This can be done with the delayed debugging option of z/OS Debugger and by using the z/OS Debugger Eclipse plugin.

By default, Debug Tool starts a debug session at the first entry compile unit of the initial load module of an application. However, there are cases where the problem is in some compile unit (for example, `prog1`) inside the application that needs debugging.

Currently, you enter AT ENTRY prog1 and GO commands when the debug session starts.

However, in some complex applications, such as doing debugging after some JNI calls (COBOL to Java), it can take some significant time before `prog1` appears. In this case, you can use the delay debug mode to delay the starting of the debug session until Debug Tool recognizes `prog1`.

Debug Tool is in a dormant state during the delay debug mode and monitors only a few events. When Debug Tool recognizes prog1, Debug Tool comes out of the delay debug mode, completes the initialization, and starts the debug session.

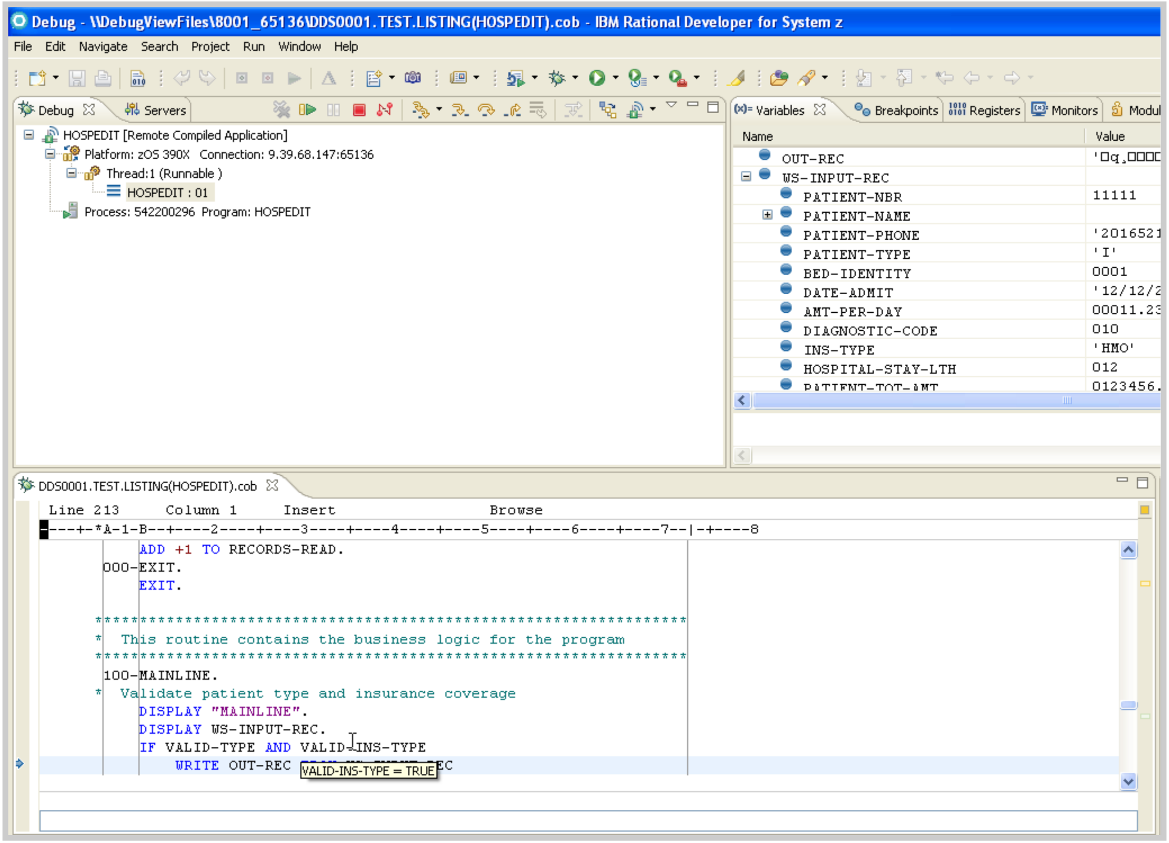
Details on how to use the Debug Tool Delay debugging are covered in the z/OS Debugger Documentation:
https://www.ibm.com/docs/en/debug-tool-for-zos/13.1?topic=spica-using-delay-debug-mode-delay-starting-debug-session

Information on how to download and install the Debug Tool plugin for Eclipse are covered here:
https://www.ibm.com/docs/en/debug-tool-for-zos/13.1?topic=guide-installing-debug-tool-plug-ins

Please note that using Debug Tool also requires setup on z/OS in order to be able to connect to z/OS Debugger from the z/OS Debugger Eclipse plugin.

This is how the debug session with a COBOL program looks like:

Figure 37. Debug session with a COBOL program


Debug - \\DebugViewFiles\8001_65136\DDS0001.TEST.LISTING(HOSPEDIT).cob - IBM Rational Developer for System z
File Edit Navigate Search Project Run Window Help
Debug
Servers
Variables
Breakpoints
Registers
Monitors
Modul

```
HOSPEDIT [Remote Compiled Application]
  Platform: zOS 390X   Connection: 9.39.68.147:65136
    Thread:1 (Runnable )
      HOSPEDIT : 01
  Process: 542200296  Program: HOSPEDIT
```

Name	Value
OUT-REC	'□q,□□□□
WS-INPUT-REC	
PATIENT-NBR	11111
PATIENT-NAME	
PATIENT-PHONE	'2016521
PATIENT-TYPE	'I'
BED-IDENTITY	0001
DATE-ADMIT	'12/12/2
AMT-PER-DAY	00011.23
DIAGNOSTIC-CODE	010
INS-TYPE	'HMO'
HOSPITAL-STAY-LTH	012
PATIENT-TOT-AMT	0123456.

DDS0001.TEST.LISTING(HOSPEDIT).cob

```
 Line 213      Column 1      Insert                 Browse
----+-*A-1-B--+----2----+----3----+----4----+----5----+----6----+----7--|-+----8
           ADD +1 TO RECORDS-READ.
       000-EXIT.
           EXIT.

      ******************************************************************
      *   This routine contains the business logic for the program
      ******************************************************************
       100-MAINLINE.
      *   Validate patient type and insurance coverage
           DISPLAY "MAINLINE".
           DISPLAY WS-INPUT-REC.
           IF VALID-TYPE AND VALID-INS-TYPE
               WRITE OUT-REC VALID-INS-TYPE = TRUE EC
```

Note: If JVM Debugging from Eclipse and z/OS Debugger Debugging for COBOL is active at the same time, you might need to switch between the perspectives to see which debugging part/language is currently active.

7.2.3. Looking at Performance with IBM Application Performance Analyzer for z/OS

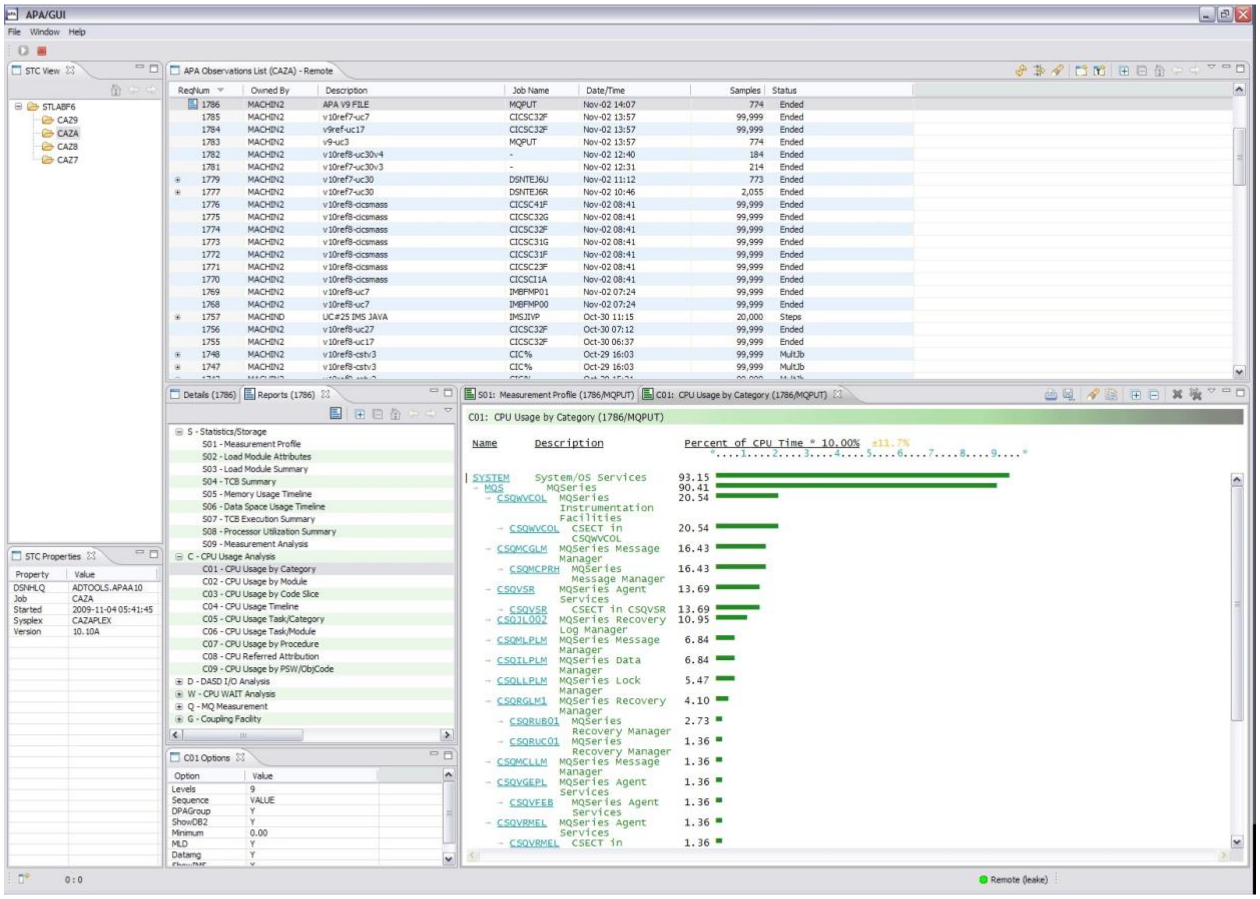
IBM Application Performance Analyzer is a separately priced tool whichcurrent Version is 14. It has both a 3270 user interface and an eclipse GUIplugin.

The IBM Application Performance Analyzer Plugin can be installed in any Eclipse Development environment and connect to z/OS. Please refer for instructions to: https://ibm.github.io/mainframe-downloads/eclipse-tools.html

Please note that using IBM Application Performance Analyzer also requires setup on z/OS in order to be able to connect to it from the IBM Application Performance Analyzer Eclipse plugin.

This is how the Eclipse plugin will display the results:

Figure 38. IBM Application Performance Analyzer Eclipse plugin



ReqNum	Owned By	Description	Job Name	Date/Time	Samples	Status
1786	MACHIN2	APA V9 FILE	MQPUT	Nov-02 14:07	774	Ended
1785	MACHIN2	v10ref7-uc7	CICSC32F	Nov-02 13:57	99,999	Ended
1784	MACHIN2	v9ref-uc17	CICSC32F	Nov-02 13:57	99,999	Ended
1783	MACHIN2	v9-uc3	MQPUT	Nov-02 13:57	774	Ended
1782	MACHIN2	v10ref8-uc30v4	-	Nov-02 12:40	184	Ended
1781	MACHIN2	v10ref7-uc30v3	-	Nov-02 12:31	214	Ended
1779	MACHIN2	v10ref7-uc30	DSNTEJ6U	Nov-02 11:12	773	Ended
1777	MACHIN2	v10ref7-uc30	DSNTEJ6R	Nov-02 10:46	2,055	Ended
1776	MACHIN2	v10ref8-cicsmass	CICSC41F	Nov-02 08:41	99,999	Ended
1775	MACHIN2	v10ref8-cicsmass	CICSC32G	Nov-02 08:41	99,999	Ended
1774	MACHIN2	v10ref8-cicsmass	CICSC32F	Nov-02 08:41	99,999	Ended
1773	MACHIN2	v10ref8-cicsmass	CICSC31G	Nov-02 08:41	99,999	Ended
1772	MACHIN2	v10ref8-cicsmass	CICSC31F	Nov-02 08:41	99,999	Ended
1771	MACHIN2	v10ref8-cicsmass	CICSC23F	Nov-02 08:41	99,999	Ended
1770	MACHIN2	v10ref8-cicsmass	CICSCI1A	Nov-02 08:41	99,999	Ended
1769	MACHIN2	v10ref8-uc7	IMBFMP01	Nov-02 07:24	99,999	Ended
1768	MACHIN2	v10ref8-uc7	IMBFMP00	Nov-02 07:24	99,999	Ended
1757	MACHIND	UC#25 IMS JAVA	IMSJIVP	Oct-30 11:15	20,000	Steps
1756	MACHIN2	v10ref8-uc27	CICSC32F	Oct-30 07:12	99,999	Ended
1755	MACHIN2	v10ref8-uc17	CICSC32F	Oct-30 06:37	99,999	Ended
1748	MACHIN2	v10ref8-cstv3	CIC%	Oct-29 16:03	99,999	MultJb
1747	MACHIN2	v10ref8-cstv3	CIC%	Oct-29 16:03	99,999	MultJb

Property	Value
DSNHLQ	ADTOOLS.APAA10
Job	CAZA
Started	2009-11-04 05:41:45
Sysplex	CAZAPLEX
Version	10.10A

- S - Statistics/Storage
  - S01 - Measurement Profile
  - S02 - Load Module Attributes
  - S03 - Load Module Summary
  - S04 - TCB Summary
  - S05 - Memory Usage Timeline
  - S06 - Data Space Usage Timeline
  - S07 - TCB Execution Summary
  - S08 - Processor Utilization Summary
  - S09 - Measurement Analysis
- C - CPU Usage Analysis
  - C01 - CPU Usage by Category
  - C02 - CPU Usage by Module
  - C03 - CPU Usage by Code Slice
  - C04 - CPU Usage Timeline
  - C05 - CPU Usage Task/Category
  - C06 - CPU Usage Task/Module
  - C07 - CPU Usage by Procedure
  - C08 - CPU Referred Attribution
  - C09 - CPU Usage by PSW/ObjCode
- D - DASD I/O Analysis
- W - CPU WAIT Analysis
- Q - MQ Measurement
- G - Coupling Facility

Option	Value
Levels	9
Sequence	VALUE
DPAGroup	Y
ShowDB2	Y
Minimum	0.00
MLD	Y
Datamg	Y

```
C01:  CPU Usage by Category (1786/MQPUT)

Name       Description          Percent of CPU Time * 10.00%  ±11.7%
                                 *....1....2....3....4....5....6....7....8....9....*

| SYSTEM     System/OS Services     93.15
 - MQS         MQSeries            90.41
   - CSQWVCOL  MQSeries            20.54
               Instrumentation
               Facilities
     - CSQWVCOL  CSECT in          20.54
                 CSQWVCOL
   - CSQMCGLM  MQSeries Message    16.43
               Manager
     - CSQMCPRH  MQSeries          16.43
                 Message Manager
   - CSQVSR    MQSeries Agent      13.69
               Services
     - CSQVSR    CSECT in CSQVSR   13.69
   - CSQJL002  MQSeries Recovery   10.95
               Log Manager
   - CSQMLPLM  MQSeries Message     6.84
               Manager
   - CSQILPLM  MQSeries Data        6.84
               Manager
   - CSQLLPLM  MQSeries Lock        5.47
               Manager
   - CSQRGLM1  MQSeries Recovery    4.10
               Manager
     - CSQRUB01  MQSeries           2.73
                 Recovery Manager
     - CSQRUC01  MQSeries           1.36
                 Recovery Manager
   - CSQMCLLM  MQSeries Message     1.36
               Manager
   - CSQVGEPL  MQSeries Agent       1.36
               Services
     - CSQVFEB   MQSeries Agent     1.36
                 Services
   - CSQVRMEL  MQSeries Agent       1.36
               Services
     - CSQVRMEL  CSECT in           1.36
```

There are similar views on the ISPF views which display the same data as the Eclipse plugin.

7.3. Understanding JNI problems

In most cases JNI problems occur because wrong parameter types, wrong amount of parameters, or Strings that are not UTF-8 are passed as input parameters to the JNI calls.

Most of those mentioned problems can be found with the `-Xcheck:jni` JVM option and its sub-options. The sub-options are referenced here: https://www.ibm.com/docs/en/sdk-java-technology/8?topic=jni-debugging

On a JNI check, the JVM might terminate, which can cause address spaces to terminate in some environments. The pedantic check also displays warnings that normally would just execute the call, but nevertheless, it's a good idea to fix all those errors before moving in production. There have been cases where errors or warnings cause exceptions with a new release of Java, and/or the JNI calls suddenly don't work anymore.

7.4. List of known problems and solutions

This section is about problems and error conditions that were experienced working with customers on Java interoperability projects during testing, development, and production phases.

7.4.1. Recommended maintenance based on experiences

The following list of APARs should be applied. This information is based on the experience at the time of this writing and is not complete and provided as is and without warranty.

- APAR PI56284 for z/OS LE runtime (memory leak with COBOL CANCEL)
- APAR PI37886 for z/OS LE runtime (NODLL link of COBOL with JNI fails)
- APAR PI36777 (5.1)/PI36780 (5.2) for Enterprise COBOL (NODLL link of COBOL with JNI fails)
- APAR PI29595 for IMS (on module not found S806 condition, the module that is not found should be printed)
- APAR PI61759 (V13), PI61769 (V14) for IMS (JMP performance degradation over time)
- APAR PI63800 for adding the UMASK parameter
- APAR PI64142 for adding 64bit JVM support for JMP and JBP applications
- APAR PI63952 (V5.1), PI44915 (V5.2, V6.1) for Enterprise COBOL (latest JNI.CPY)
- APAR PI59142 for Enterprise PL/I (latest IBMZJNI include)
- APAR PI40084 for z/OS Language Environment (high CPU usage for CEEPIPI with XPLINK)
- For ESAF thread pooling/connection pooling between IMS and Db2:
  - APARs PI60400 and PI69365 (IMS V14)
  - APARs PI61982, PI61983, PI64956, PI67609, PI69765, PI73734 and PI75155 (Db2 V11)
  - APARs PI66963, PI74192, PI74193, PI74243, and PI79119 (Db2 V12)
  - APARs PI67610 (JCC V4.21.40), PI76338 (JCC V4.22.37), PI67609 (JCC

V3.71.32) and PI76340 (JCC V3.72.30)

7.4.2. IGZ0032S at j9sl_close_shared_library with COBOL DLLs in call chain

If a COBOL-to-Java-to-COBOL calling chain is implemented, which requires the COBOL called by Java to start with a COBOL DLL, on shutdown of the IMS region, an IGZ0032S error is produced.

Check that the Enterprise COBOL V4.2 PTF UK90600 (APAR PM72434) is applied.

7.5. Summary

During your project development lifecycle, it's important to check and apply the latest maintenance to avoid unnecessary problems. When problems occur, you can use IBM Support Assistant and various tools that can be plugged into IBM Support Assistant to help gather and analyze data. JConsole that comes with the Java JDK is a handy tool for monitoring the behavior of Java applications. You can also use the Java debugging functions in your IDE.

CHAPTER 8

8. Case study: Bringing Java to a COBOL-based banking system

Introducing Java™ in IMS™ MPRs and BMPs is the way to establish Java as the next-generation language for the Mainframe. It is a natural evolution instead of revolution, allowing an organization to "evolve over time" from a mainly COBOL-based system to a mainly Java-based system without disruption.

Fiducia & GAD IT AG (Fiducia & GAD) provides IT services to all 1,100 cooperative banks in Germany and also to private banks and companies in other industries. Our strategy is to introduce Java in IMS as an additional programming language, with the goal to evolve from a COBOL-based system to a Java-based system with minimum cost. We introduced Java alongside COBOL in IBM® IMS on IBM Z to accelerate the creation of new services and extend the life and value of our existing applications.

In this chapter, we will describe such a modernization project in a large banking system. From functional requirements, the infrastructure landscape, environment settings, to application development and security, we will share our experience, provide examples and highlight the lessons learned.

8.1. Introducing Java in existing IMS MPRs and BMPs

With the long-term goal of evolving our mainframe application infrastructure from COBOL-centric to Java-centric in mind, we started out by identifying our strategies and functional prerequisites. We also examined our existing IMS landscape to identify the parts that must be Java-enabled.

8.1.1. Strategies

Our short-term and mid-term strategies are:

- Expand the existing application set through:
  - Reuse of existing services
  - Implementing new functions in Java and using it everywhere
  - Use of third-party software inside IMS
- Use technologies and languages where skill is widely available and new generation is willing to use.
- Seek every opportunity to build applications which, over time, would

become platform independent.

8.1.2. Functional prerequisites

The following functional prerequisites are identified to ensure calls across systems can be supported efficiently.

- IMS must handle the Unit of Work (UOW) over IMS DB, Db2® for z/OS® and IBM MQ for both access types: from COBOL to Java, and Java to COBOL. This means issuing a DL/I call, a Db2 static call, a Db2 dynamic call, and an MQ call from COBOL, and a JDBC type 2 call or an MQ JMS call from Java, is all handled by the same Unit of Work.
- Must have the ability to do cascading calls (from COBOL to Java to COBOL to Java).
- Must have the ability to exchange data between COBOL and Java in an efficient way (JNI-based).
- Output from COBOL (DISPLAY statements) and Java (`printf()` statements) must appear chronologically in the same output DD statement. The JVM settings `-Djzos.merge.sysout=true` must be in effect to enable it.
- Errors in the JVM context must be percolated back to IMS. This is enabled with the JVM option `Xsignal:userConditionHandler=percolate`. We will discuss more about JVM options in Section 8.2.3 on page 158.

8.1.3. Technical overview of the existing IMS landscape that must be Java-enabled

Our environment has the following characteristics:

- All transactions are written in COBOL with some assembler routines.
- All programs are 31-bit enabled.
- All transactions are stateless (no 3270, no scratchpad area, etc.).
- All online transactions are accessed via IMS Connect or the MQ-OTMA bridge.

Those transactions are classified in 3 categories: high performance transactions, standard performance transactions, and batch programs (BMPs).

8.1.3.1. High performance Transactions

These transactions are written reentrant, reusable. This means that the next consecutive execution of the same transaction in one message processing region (MPR) reuses the heap memory as it was left by the precedent execution. As a result, the transaction and all used programs inside must ensure that the memory (heap) is properly initialized at the beginning. These transactions are executed in either wait for input (WFI) or pseudo-WFI MPRs.

8.1.3.2. Standard performance transactions

Standard transactions are transactions that are expected to be executed in a cleaned environment. After each transaction execution, the LE enclave is canceled and rebuilt at the next execution. This approach leads to a clean LE heap.

8.1.3.3. Batch processing (BMP)

Batch procession programs define the major and critical workload of the system. These programs need to perform fast with a focus on elapsed time. Due to the nature of batch processing, no LE environment cleanup is necessary.

8.2. Enabling IMS for Java: Experiences and how-to's

With our strategies, functional prerequisites, and existing IMS landscape that must be Java-enabled clearly identified, we were ready to embark on our Java enablement journey. This section covers environment settings and helpful tips in various areas, including RACF, UNIX System Services (USS), JVM, MPRs, and BMPs, as well as release management for JVM and JDBC drivers.

8.2.1. How Java is embedded in classic IMS regions

The introduction of Java to MPRs and BMPs leads to several technical changes inside the regions and might cause behavior changes of exiting transactions.

For example, the JVM has to stay resident inside the MPR. It is started during the initialization phase of the MPR. IBM uses the LE (CEEPIPI) mechanism to implement it. Because the JVM itself conforms to the POSIX standard, the address spaces must be in the POSIX(ON) mode.

8.2.2. Environment enablement

To enable the environment for Java, you might find the following setup helpful.

8.2.2.1. Minimizing I/Os on RACF datasets

To minimize the I/Os on RACF datasets, you can reduce writing of RACF statistics and switch to a daily processing mode. If you omit this step, RACF will write LOGON statistics for every Db2 request that is issued from IMS via External Subsystem Attach Facility (ESAF).

- Configure the usage of the IMSID or any other constant (not spaces) in the APPL field of the RACROUTE macro in DFSDCxxx (parameter SAPPLID=).

- For OTMA/IMS Connect requests, apply the fix for APAR PI64496 to also reuse ESAF/Db2 threads.
- Within the RACF APPLDATA field of that APPL profile, specify RACF-INITSTATS(DAILY). If RACF-INITSTATS(DAILY) is not defined in the APPLDATA field, recording of statistics is done as directed by the SETROPTS INITSTATS setting.
- APPL class must be active and RACLIST'ed to have consistent results due to ACEE caching.
- The Db2 Sign-on exit routine DSN3@SGN must be modified to exploit the APPL field.

8.2.2.2. POSIX(ON) and ALL31(ON) related issues

The POSIX(ON) setting has its side effects. Switching to the POSIX(ON) mode leads to differences in condition handling that are now no longer z/OS- but UNIX-conforming, and new and different abends can occur. The ALL31(ON) LE runtime option presumes your application no longer contains modules with AMODE(24). Those must become AMODE(31).

8.2.2.3. UNIX System Services settings

z/OS UNIX System Services (OMVS subsystem) settings must be taken care of, especially the MAXPROCUSER setting. MAXPROCUSER specifies the maximum number of processes that a single OMVS user ID (UID) is allowed to have active at the same time, regardless of how the process became a USS process. The range is 3 to 32 767.

The following sample shows all options that were set:
z/OS Console Command: `D OMVS,O`

Output:

```
RESPONSE=MVSx
BPXO043I 14.18.49 DISPLAY OMVS 952
OMVS     0011 ACTIVE             OMVS=(PA,P2,20,V0)
CURRENT UNIX CONFIGURATION SETTINGS:
MAXPROCSYS      =       1500    MAXPROCUSER     =       1000
MAXFILEPROC     =      64000    MAXFILESIZE     = NOLIMIT
MAXCPUTIME      = 2147483647    MAXUIDS         =        200
MAXPTYS         =        256    MAXIOBUFUSER    =       2048
MAXMMAPAREA     =       4096    MAXASSIZE       = 2147483647
MAXTHREADS      =        500    MAXTHREADTASKS  =        500
```

```
 MAXCORESIZE      =    4194304     MAXSHAREPAGES    =   32768000
```

```
 IPCMSGQBYTES     = 2147483647     IPCMSGQMNUM      =      10000
 IPCMSGNIDS       =        500     IPCSEMNIDS       =       4096
 IPCSEMNOPS       =      32767     IPCSEMNSEMS      =         50
 IPCSHMMPAGES     =      25600     IPCSHMNIDS       =        500
 IPCSHMNSEGS      =       1000     IPCSHMSPAGES     =     786432
 SUPERUSER        = BPXROOT        FORKCOPY         = COPY
 STEPLIBLIST      = /etc/steplib
 USERIDALIASTABLE=
 PRIORITYPG VALUES: NONE
 PRIORITYGOAL VALUES: NONE
 MAXQUEUEDSIGS    =       1000     SHRLIBRGNSIZE    =   67108864
KERNELSTACKS     = BELOW
```

The following sample shows the system-wide parmlib limits:

Command: `D OMVS,L`

Output:

```
 RESPONSE=MVSR
 BPXO051I 17.51.00 DISPLAY OMVS 624
 OMVS      0011 ACTIVE             OMVS=(22,V0)
 SYSTEM WIDE LIMITS:         LIMMSG=NONE
                   CURRENT  HIGHWATER     SYSTEM
                     USAGE      USAGE      LIMIT
 MAXPROCSYS            620        774       1210
 MAXUIDS                61         78        200
 MAXPTYS                 0          2        256
 MAXMMAPAREA           138        138       4096
 MAXSHAREPAGES        1050       1050   32768000
 IPCMSGNIDS              4          5        500
 IPCSEMNIDS              1          1       4096
 IPCSHMNIDS              3          3        500
 IPCSHMSPAGES          228        228     786432
 IPCMSGQBYTES          ---       1204     262144
 IPCMSGQMNUM           ---          3      10000
 IPCSHMMPAGES          ---         75      25600
 SHRLIBRGNSIZE    67108864   67108864   67108864
 SHRLIBMAXPAGES          0          0       4096
 MAXUSERMOUNTSYS        13         13        100
 MAXUSERMOUNTUSER        2          6         10
 MAXPIPES               76        533      15360
```

The following example shows the current parmlib limits for process ID 16777456: Command: `D OMVS,L,PID=`*`nnnnnnnn`*
Output:

```
 BPXO051I 17.55.40 DISPLAY OMVS 874
 OMVS     0011 ACTIVE             OMVS=(22,V0)
 USER     JOBNAME  ASID        PID        PPID STATE    START     CT_SECS
 XCK2034  XCK20348 0407   67437668    67437273 1CI----- 17.55.00       .0
   LATCHWAITPID=         0 CMD=obrowse
 PROCESS LIMITS:         LIMMSG=NONE
                   CURRENT  HIGHWATER    PROCESS
                     USAGE      USAGE      LIMIT

 MAXFILEPROC             3         11      64000
 MAXFILESIZE           ---        ---    NOLIMIT
 MAXPROCUSER             3          7        500
 MAXQUEUEDSIGS           0          1       1000
 MAXTHREADS              0          0      10000
 MAXTHREADTASKS          0          0      10000
 IPCSHMNSEGS             0          0       1000
 MAXCORESIZE           ---        ---    4194304
 MAXMEMLIMIT           10M        10M      4096M
IPCSHMNSEGS                                                0
0               500
MAXCORESIZE                                              ---
---         4194304
MAXMEMLIMIT                                                0
0   16383P
```

8.2.3. JVM

As of this writing, for IMS 13, only 31-bit JVM can fit in IMS MPRs and IMS BMPs. For IMS 14, 64-bit JVM is enabled for JMP and JBP applications through APAR PI64142.

8.2.3.1. Setting a z/OS-wide default Java version

To gain flexibility in the management of the environment, one key lesson we learned through the process is that it is best to install Java in a dedicated file system that conforms to agree-upon naming rules. Users and application developers would use only aliases and symbolic links in their application (or JCL, etc.). For example, they can use a symbolic link such as `/prod/java/inuse` that would result in the physical name, for example, `/prod/java/J8.0.3`

For more discussions, see "JVM release management" later in this chapter.

8.2.3.2. JVM version

The following examples show a symbolic link of Java in /prod/java/inuse, and how Java version is identified.

```
… >cd /prod/java/inuse/bin
MVSR:/prod/java/inuse/bin >./java -version
java version "1.8.0"
Java(TM) SE Runtime Environment (build pmz3180sr3fp20-20161019_02(SR3
FP20))
IBM J9 VM (build 2.8, JRE 1.8.0 z/OS s390-31 20161013_322271 (JIT
enabled, AOT enabled)
J9VM - R28_Java8_SR3_20161013_1635_B322271
JIT  - tr.r14.java.green_20161011_125790
GC   - R28_Java8_SR3_20161013_1635_B322271
J9CL - 20161013_322271)
JCL - 20161018_01 based on Oracle jdk8u111-b14
MVSR:/prod/java/inuse/bin >
```

8.2.3.3. IMS settings

Java-related IMS settings are spread over two configuration members, DDname ENVIRON and JVMOPMAS, and a USS file that is specified through the `Xoptionfile` option. These settings are described below.

8.2.3.3.1. IMS JVM settings for high-performance transactions

For high-performance transactions, we have the **ENVIRON** member set with the PATH, LIBPATH and JVMOPTIONS as follows:

```
PATH=/bin:/prod/java/inuse/bin:.
LIBPATH=/lib:/usr/lib:/prod/java/inuse/lib:>
/prod/java/inuse/lib/s390:>
/prod/java/inuse/lib/s390/classic:>
/prod/db21/jdbc/lib:>
/prod/jvmti_path
CANCEL_PGM=N
```

With CANCEL_PGM set to No, IMS will not clean up the program and all its subprograms that it invokes during execution. This means the working storage area will remain intact for the next program execution.

For the member in the IMS.PROCLIB data set that’s specified in the

JVMOPMAS parameter, we set the JVM options as follow:

```
-Xoptionsfile=/entw/appl/imsq/jbf/env/jvmParmIMS_endevor
```

```
-Djzos.merge.sysout=true
```

The **-Djzos.merge.sysout=true** option guarantees that at any point of time the COBOL DISPLAY output and Java `printf()` output are visible in the"OUTPUT" DDname.

The **–Xoptionsfile** option points to the JVM properties file, in our case, jvmParmIMS_endevor. This USS file contains the class path and the Java related option. Here is an example of the JVM properties file:

```
-Xgcpolicy:gencon
-Xmx256M
-Xmso512k
-Xss256k
-Xms32M
-Xsignal:userConditionHandler=percolate
-Xnoagent
```

In the above example:

- **-Xgcpolicy:gencon** specifies short garbage collections to limit the impact on the active transaction. Short-lived objects are handled differently than objects that are long-lived. Applications that have many short-lived objects can see shorter pause times with this policy.
- **-Xsignal:userConditionHandler=percolate** specifies that, in case of error, the JVM gives the control back to the caller, so the region terminates as usual in case of errors (comprehensible DUMP). Without this option, the JVM tries to handle the error itself and this leads to uncomprehensive error.
- **-Xnoagent** is the default, and disables support for the old JDB debugger.

8.2.3.3.2. IMS JVM settings for standard transactions

For standard transactions, our ENVIRON member setting is different:

```
PATH=/bin:/prod/java/inuse/bin:.
LIBPATH=/lib:/usr/lib:/prod/java/inuse/lib:>
/prod/java/inuse/lib/s390:>
/prod/java/inuse/lib/s390/classic:>
/prod/db21/jdbc/lib:>
/prod/jvmti_path
CANCEL_PGM=Y,EXCLUDE=JVMEXCL
```

IMS CANCEL_PGM is set to Yes, and the JVMEXCL member contains a list of load modules that are not "deleted" and "loaded." Keeping load modules in this list means that its working storage will not be deleted and reinitialized. Therefore the values of the precedent execution of those modules will be kept. The JVM options are set identically as those for the high performance transactions:

```
DFSJVM00: -Xgcpolicy:gencon
DFSJVM00: -Xmx256M
DFSJVM00: -Xmso512k
DFSJVM00: -Xss256k
DFSJVM00: -Xms32M
DFSJVM00: -Xsignal:userConditionHandler=percolate
DFSJVM00: -Xnoagent
```

8.2.3.3.3. IMS JVM settings for BMP regions

The ENVIRON member for BMPs has PATH, LIBPATH and JVMOPTIONS set as follows:

```
PATH=/bin:/prod/java/inuse/bin:.
LIBPATH=/lib:/usr/lib:/prod/java/inuse/lib:>
/prod/java/inuse/lib/s390:>
/prod/java/inuse/lib/s390/classic:>
/prod/db21/jdbc/lib:>
/prod/jvmti_path
CANCEL_PGM=Y,EXCLUDE=JVMEXCL
```

Likewise, IMS CANCEL_PGM is set to Yes, and the member JVMEXCL contains a list of load modules that are not “deleted” and “loaded”.

CANCEL_PGM has no effect on BMPs because the main COBOL program stays active for the whole execution of the BMP.

JVM options are set as follows:

```
DFSJVM00: -Xgcpolicy:optthruput
DFSJVM00: -Xmx256M
DFSJVM00: -Xmso512k
DFSJVM00: -Xss256k
DFSJVM00: -Xms32M
DFSJVM00: -Xsignal:userConditionHandler=percolate
DFSJVM00: -Xnoagent
```

- **`-Xgcpolicy:optthruput`** specifies the garbage collection policy to optimize the throughput.
- **`-Xsignal:userConditionHandler=percolate`** specifies that, in case of error, the JVM gives the control back to the caller, so the region terminates as usual in case of errors (comprehensible DUMP). Without this option,

the JVM tries to handle the error itself and this leads to uncomprehensive error.

- `-Xnoagent` is the default.

8.2.4. Message Processing Region settings

The LE settings must be adapted depending on your environment. The following settings fulfills our requirements. The same LE settings are used for both high performance and standard transactions.

8.2.4.1. MPRs settings for high performance transactions (production settings)

8.2.4.1.1. LE (language Environment)

```
ANYHEAP(2M,1M,ANY,KEEP)
HEAP(80M,4M,ANY,FREE,1M,512K)
HEAPPOOLS(ON,8,10,32,10,128,10,256,10,1024,10,2048,10,0,10,0,
10,0,10,0,10,0,10,0,10)
STACK(64K,64K,ANY,KEEP,512K,256K)
STORAGE(00,NONE,NONE,8K)
THREADSTACK(OFF,64K,16K,ANY,KEEP,128K,128K)
TERMTHDACT(UAIMM)
RPTOPTS(OFF)
RPTSTG(OFF)
```

- **`HEAP(80M,4M,ANY,FREE,1M,512K):`** 80M is the minimal footprint to start a JVM free to guarantee that the memory will not be too much fragmented over time (using “`KEEP`” instead of “`FREE`” might lead to errors when there is not enough contiguous memory available to fulfill a `getheap` request).
  - **STORAGE(00,NONE,NONE,8K)**: Sets to 00 in production to force a memory initialization to x’00’. This setting causes no issue in

    production because it is used to initialize the storage when a module is loaded. This initialization happens exactly once as each module is loaded exactly once and then reused (according to the CANCEL_PGM=NO behavior). In this context, this setting neither impact response time nor resource consumption.
  - **`RPTOPTS(OFF):`** Reports the LE settings at normal region termination (at enclave termination). Enable this option when needed, such as for debugging.
- **`RPTSTG(OFF):`** Reports storage utilization. It generates a lot of output and should only be used for tracing purposes. Enable this option when needed, such as for debugging.

8.2.4.1.2. IMS-related region setting

Our region setting is as follows:

```
STD PWFI=Y
STD PREINIT= …

RGN=768M
…
JVMOPMAS=JVAENDVR
ENVIRON=,
CEEJAVA= CEEJVM00

STD PWFI=Y
STD PREINIT=
…
//CEEOPTS  DD DSN=SCD.IMSQ000.A.PROCLIB(&CEEJAVA),DISP=SHR
```

Our region size is currently set to 768M. This size is adjusted depending on the situation and real needs. LE settings are injected via a CEEOPTS DDNAME.

8.2.4.2. MPRs settings for standard transactions (production settings)

In the following sections we will describe our MPR settings in the production environment for standard transactions.

8.2.4.2.1. Language Environment (LE)

The same settings for high performance transactions are used for standard transactions, but they lead to a different behavior (see STORAGE).

```
ANYHEAP(2M,1M,ANY,KEEP)
HEAP(80M,4M,ANY,FREE,1M,512K)
HEAPPOOLS(ON,8,10,32,10,128,10,256,10,1024,10,2048,10,0,10,0,
10,0,10,0,10,0,10,0,10)
STACK(64K,64K,ANY,KEEP,512K,256K)
STORAGE(00,NONE,NONE,8K)
THREADSTACK(OFF,64K,16K,ANY,KEEP,128K,128K)
TERMTHDACT(UAIMM)
RPTOPTS(OFF)
RPTSTG(OFF)
```

- **`HEAP(80M,4M,ANY,FREE,1M,512K)`:** `80M` is the minimal footprint to start a JVM free to guarantee that the memory will not be too much fragmented over time (using `KEEP` instead of `FREE` might lead to errors when there is not enough contiguous memory available to fulfill a `getheap` request).

- **`STORAGE(00,NONE,NONE,8K)`:** Sets to 00 to force a memory initialization to x'00'. After each transaction execution, the memory (heap) is freed, but it still contains the values from the last execution. Setting 00 ensures that memory is initialized by this value when getheap occurs. Avoiding this initialization might lead to errors when programs expect to deal with zeroed memory areas.

  Be aware that this initialization happens for each transaction that is executed. In this context, this initialization process happens after each transaction execution, which means it leads to a higher resource consumption and also might marginally impact the transaction duration. If you have identified and are very sure that certain transactions don't need to be initialized, that means those transactions fulfill the requirements to be executed in the high performance environment (transactions that are programmed as reentrant and reusable).
- **`RPTOPTS(OFF)`:** Reports the LE settings at normal region termination (at enclave termination).
- **`RPTSTG(OFF)`:** Reports storage utilization. This setting generates a lot of output and should only be used for tracing purposes.

8.2.4.2.2. Region settings

Our default region settings are shown in the following JCL:

```
RGN=768M
…
JVMOPMAS=JVAENDVR
ENVIRON=,
CEEJAVA= CEEJVM00

STD PWFI=Y
STD PREINIT=
…
//CEEOPTS  DD DSN=SCD.IMSQ000.A.PROCLIB(&CEEJAVA),DISP=SHR
```

Region size is currently set to 768M. This size is adjusted depending on the real needs. LE settings are injected via a CEEOPTS DDNAME.

8.2.5. BMP settings

Our BMP settings are described below.

8.2.5.1. LE (Language Environment)

```
PATH=/bin:/prod/java/inuse/bin:.
LIBPATH=/lib:/usr/lib:/prod/java/inuse/lib:>
/prod/java/inuse/lib/s390:>
```

```
/prod/java/inuse/lib/s390/classic:>
/prod/db21/jdbc/lib:>
/prod/jvmti_path
IBM_HEAPDUMP=true
JAVA_DUMP_TDUMP_PATTERN=HCA00.DUMP.A.%job.D%y%m%d.T%H%M%S.N&DS
```

8.2.5.2. Region setting

Our default region settings are shown in the following JCL:

```
RGN=768M
…
JVMOPMAS= JVA000QA
ENVIRON= JVE0B0Q0,
CEEJAVA= CEEJVM00
…
//CEEOPTS  DD DSN=SCD.IMSQ000.A.PROCLIB(&CEEJAVA),DISP=SHR
```

The default region size is currently set to 768M, but it can be increased up to 1.85GB to handle multiple jobs. This is the maximum amount authorized. Never use 0M, as in case of memory shortage errors, the system would not be able to find any memory for abend handling, potentially leading to severe problems.

8.2.6. Specific settings for development regions

For development regions, we use the same settings as in production for standard transactions as the baseline for test regions. One difference is the STORAGE setting is set to X'DB' instead of X'00'.

- **`STORAGE(DB,NONE,NONE,8K)`:** Sets to DB in the test environment to force a memory initialization to x'DB'. Setting the storage to a non-zero value (for example , "`STORAGE(DB…)`") in test environments helps you identify in the dump, in case of errors, the areas that were not correctly handled by some programs. This setting will help you identify the origin of errors more effectively.

8.2.7. JVM release management

The JVM is installed in a USS path by your system administrator. You have to configure those paths, PATH, LIBPATH, and CLASSPATH, in your IMS configuration. Assuming that you want to activate a new version of the JVM in the whole system, you would use links and system symbols. This approach gives your system administrator the opportunity to replace an old version with a new one without any change in your JCLs or PROCs. The system administrator only has to modify the setting of the symbolic link and system symbols.

You can make this change at any time. However, you must restart all concerned MPRs and other running processes after the change to use thenewest JVM.

8.2.7.1. Simplifying system-wide JVM upgrades

To simplify the JVM upgrades, build an alias composed of system symbols that points to the physical Java file system. Don't use a direct symbolic link (such as `/prod/java/inuse` linking to `/prod/java/J8.0.3`); Instead, compose the symbolic link name with system symbols.

For example, `/prod/java/inuse` is a symbolic link on `$SYSSYMR/&JAVAVERS` and `&JAVAVERS = "J8.0.3"`

By simply modifying the settings of the `&JAVAVERS` symbol, you are able to point instantly to another version of the JVM.

This mechanism allows a switch to another version of the JVM in concurrencyto the execution of active JVMs. Recycle all long-running processes using this symbolic link to avoid errors.

8.2.8. JDBC driver and release management

8.2.8.1. JDBC driver

Identifying JDBC related issues due to the invocation of an incorrect version ora misconfigured version can be difficult. Here is a sample job that identifies the installed version.

```
//JCCIMSQ  EXEC PGM=IKJEFT01,REGION=512M
//SYSTSPRT DD SYSOUT=*
//SYSOUT    DD SYSOUT=*
//STDOUT  DD SYSOUT=*
//STDERR  DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSTSIN  DD *
 BPXBATCH SH cd  /prod/db2u/jdbc/classes ; +
 java -cp ./db2jcc.jar com.ibm.db2.jcc.DB2Jcc -version
/*
```

The output prints out the version of the JDBC driver, such as "`The IBM Data Server Driver for JDBC and SQLJ 3.69.66.`"

8.2.8.2. Simplifying system-wide JDBD driver upgrades

To simply system-wide JDBC driver upgrades, you can use similar mechanisms as those for JVM management.

8.2.9. Defining a validation suite

To guarantee the immediate identification of configuration problems and its origin, define a validation suite with very simple test cases that must be executed before and after each change in Java-related configuration. Hand this suite out to those responsible for software changes. Use it whenever configuration changes are made in Java/JZOS, IMS, Db2/JDBC, and even z/OS.

This suite can consist of a batch job or a BMP with several steps to execute commands to display the configuration and typical application check points. For example:

1. Use the java –version and DB2Jcc -version commands to display the version.
2. Use a simple "echo" to validate Java.
3. Use a Db2 call via JDBC to validate Db2 and JDBC.
4. Use a crypto call via a defined cryptography provider to validate Java access to the cryptography hardware.

Take care to use the standard defined symbolic links and environment variables as they are set in the appropriate environment.

8.2.10. IMS class/region concept

Depending on your environment, you might have to adjust your concept of a region.

For IMS 13, since it only supports 31-bit JVM, you have to consider the size a region might get when a lot of things are loaded (non-Java modules, Java classes, heap for Java, and so on).

The following aspects would influence the new region concept.

- Connection pooling. New ESAF connection pooling supports 50 connections per MPR.
- Length of the Java class path. It makes sense to isolate functional blocks and manage a dedicated class path for each block. So a class path should contain exactly the JAR files that are needed for the execution of the transactions associated with a given IMS class. Those block-based class paths must be as small as possible (no unused overhead).
- The release model (versioning concepts) you want to support.

To implement a long-term viable solution, make a comparison with micro services. Each IMS class has to fulfill the following expectations, as compared to micro services.

- The API is the IMS input and output messages. All published versions of a transaction must be supported, that is, downward compatibility.

- The set of transactions contained in an IMS class can be separately released, independently from any other business services (hosted in other IMS classes).

8.3. Connecting COBOL with Java

We will now dive into COBOL and Java interoperability by providing a COBOL sample that calls a Java program, and a sample procedure for compiling the COBOL program. We will also discuss some tips and tricks from our experience.

8.3.1. Sample for COBOL calling Java that conforms to our specifications

8.3.1.1. Coding the COBOL parts

The sample consists of four files as shown in the following table.

Table 3. Sample files for COBOL calling Java

Program/file	Description
COBMAIN.cbl	The main program that will execute some Java code (JNIHelper.java). It calls JNIFIND using INIFINDI and INIFINDO.
JNIFIND.cbl	A subprogram to get the needed class and method references per JNI services.
JNIFINDI.cpy	The input copy for JNIFIND.cbl.
JNIFINDO.cpy	The output copy for JNIFIND.cbl. It returns the determined class or method reference.

8.3.1.1.1. COBMAIN.cbl

This is the main program that will execute some Java. The determination of the class and method reference, which is needed when we don't want to use INVOKE, will be done by the JNIFIND.cbl program in the Init sections of COBMAIN.cbl.

After the references for the Java class and method are available, you can call the method via a JNI call such as CallStatic<type>Method" or if you created anobjectCall<type>Method(replace<type> withthereturnvaluetype,Void if none). This happens in the CALLING-RETURNDATA section.

Note that you can put the code with the JNI service calls in a subprogram so you have a clean separation of native COBOL and COBOL with JNI services.

```
       cbl map,lib,pgmname(longmixed)
       Identification division.
       Program-ID. "COBMAIN".
       Environment division.
       Configuration section.
       Repository.
           Class jmethodid is 'jmethodid'
           Class jclass is 'jclass'
           Class jbyteArray is 'jbyteArray'
           Class jException is 'java.lang.Exception'
           .
       Data Division.

*****************************************************************
      *      W O R K I N G   S T O R A G E   S E C T I O N

*****************************************************************
       Working-Storage Section.

*****************************************************************
      *      JNIFIND

*****************************************************************
       77 JNIFIND                     PIC X(08) Value 'JNIFIND'.
       01 JNIFIND-IN.
          Copy JNIFINDI.
       01 JNIFIND-OUT.
          Copy JNIFINDO.

*****************************************************************
      *      buffer fields for JNIFIND call

*****************************************************************
       77 WK-TEMP-NAME                PIC X(64).
       77 WK-TEMP-SIG                 PIC X(7).

*****************************************************************
       *    JNIHelper
```

```
*****************************************************************
       01 JNIHelper-Methods.
          05 returnData.
             49 returnData-data-id    PIC S9(09) COMP-5.
             49 returnData-data       Object Reference jbyteArray.

       77 Array-Size                  PIC S9(09) COMP-5.
       77 JavaException               Object Reference jException.

       01 COBMAINX External.
          05 COBMAINX-PEST            PIC X(08).
          05 COBMAINX-Referenzen.
             10 COBMAINX-Class        Object Reference jclass.
             10 COBMAINX-returnData   Object Reference jmethodid.


       77 COBMAIN-DATA                PIC X(20).
       77 COBMAIN-DATA-PTR            POINTER.

******************************************************************
      *      L I N K A G E   S E C T I O N

*****************************************************************
       Linkage Section.
          Copy JNI.

******************************************************************
      *      P R O C E D U R E   D I V I S I O N

*****************************************************************
       Procedure Division.
       BEGIN Section.
           Set Address Of JNIEnv To JNIEnvPtr
           Set Address Of JNINativeInterface To JNIEnv
           Set COBMAIN-DATA-PTR To Address Of COBMAIN-DATA
           If COBMAINX-PEST Not = 'COBMAIN' Then
      *      Init just once
             Perform A05
             Move 'COBMAIN' To COBMAINX-PEST
           End-If
           Perform PROCESSING
           .
       BEGINZ.
           Goback
           .

*****************************************************************
      * Getting the method and class references

*****************************************************************
       INIT Section.
           Perform INIT-PTR
           Set FC-ClassRef   To True
           Perform INIT-CLASS
           Set FC-MethodRef  To True
           Set ObjectRef To Address Of COBMAINX-Class
           Perform INIT-RETURNDATA
           .
       INITZ.
           Exit
           .

*****************************************************************
      *      Init temporary fields

*****************************************************************
       INIT-PTR Section.
           Set ClassName  To Address Of WK-TEMP-NAME
           Set MethodName To Address Of WK-TEMP-NAME
           Set Signature  To Address Of WK-TEMP-SIG
           .
       INIT-PTRZ.
           Exit
           .

*****************************************************************
      *      Init Class JNIHelper

*****************************************************************
       INIT-CLASS Section.
           Move 1 To ClassNameLen
           String 'com/example/JNIHelper'
                  Delimited By Size
             Into WK-TEMP-NAME
             With Pointer ClassNameLen
           End-String
           Subtract 1 From ClassNameLen
           Set Ref To Address Of COBMAINX-Class
           Call JNIFIND Using JNIFIND-IN
```

```
                                 JNIFIND-OUT
           Display 'COBMAIN : Getting reference for'
                   WK-TEMP-NAME(1:ClassNameLen)
                   ' successful'
           .
       INIT-CLASSZ.
           Exit
           .

*****************************************************************
      *      Init Method returnData

*****************************************************************
       INIT-RETURNDATA Section.
      *    set methodname
           Move 1 To MethodNameLen
           String 'returnData'
                  Delimited By Size
             Into WK-TEMP-NAME
             With Pointer MethodNameLen
           End-String
           Subtract 1 From MethodNameLen
      *    set methodsignature
           Move 1 To  SignatureLen
           String '(I)[B'
                  Delimited By Size
             Into WK-TEMP-SIG
             With Pointer SignatureLen
           End-String
           Subtract 1 From SignatureLen
           Set Ref To Address Of COBMAINX-returnData
           Call JNIFIND Using JNIFIND-IN
                                 JNIFIND-OUT
           Display 'COBMAIN : getting reference for '
                   WK-TEMP-NAME(1: MethodNameLen)
                   WK-TEMP-SIG (1: SignatureLen)
                   ' successful'
           .
       INIT-RETURNDATAZ.
           Exit
           .

*****************************************************************
      *      Execute Java
```

```
*****************************************************************
       PROCESSING Section.
           Move Zero To Return-Code
           Perform Varying Tally From 1 By 1
             Until Tally > 10 Or Return-Code > 0
                Perform CALLING-RETURNDATA
                Display COBMAIN-DATA
           End-Perform
           .
       PROCESSINGZ.
           Exit
           .

*****************************************************************
      *      Execute returnData

*****************************************************************
       CALLING-RETURNDATA Section.
      *      Init data-id
           Move Tally To returnData-data-id

*****************************************************************
      * call method returnData

*****************************************************************
           Call CallStaticObjectMethod
                Using By Value JNIEnvPtr
                               COBMAINX-Class
                               COBMAINX-returnData
                               returnData-data-id
                Returning returnData-data
           Perform CHECK-EXCEPTION

*****************************************************************
      * read return value returnData-data

*****************************************************************
           Call GetArrayLength
                Using By Value JNIEnvPtr
                               returnData-data
                Returning Array-Size
           Perform CHECK-EXCEPTION
           If returnData-data = null Then
      *      Getting a null Pointer from Java
             Move 8 To Return-Code
```

```
              Go To CALLING-RETURNDATAZ
            End-If
            If Length Of COBMAIN-DATA < Array-Size
            Then
              Move 4 To Return-Code
              Display 'Field is too small to hold array data'
              Display 'Fieldsize: ' Length Of COBMAIN-DATA
              Display 'Arraysize: ' Array-Size
              Move Length Of COBMAIN-DATA To Array-Size
            End-If
            Call  GetByteArrayRegion
                 Using By Value JNIEnvPtr
                                returnData-data
                                0
                                Array-Size
                                COBMAIN-DATA-PTR
            Perform CHECK-EXCEPTION
            Call DeleteLocalRef
                 Using By Value JNIEnvPtr
                                returnData-data
            Perform CHECK-EXCEPTION
            .
        CALLING-RETURNDATAZ.
            Exit
            .

*****************************************************************
      *      Exceptionhandling

*****************************************************************
        CHECK-EXCEPTION Section.
            Call ExceptionOccurred
                 Using By Value JNIEnvPtr
                 Returning JavaException
            If JavaException Not = Null Then
               Call ExceptionDescribe
                    Using By Value JNIEnvPtr
               Call ExceptionClear
                    Using By Value JNIEnvPtr
               Move 16 To  Return-Code
               GOBACK
            End-If
            .
        CHECK-EXCEPTIONZ.
        Exit
             .
        End Program "COBMAIN".
```

8.3.1.1.2. JNIFIND.cbl

This is a subprogram to get the needed class and method references per JNI services.

It has three functions:

1. FC-ClassRef: Uses the JNI service `FindClass` to get the java class reference
2. FC-MethodRef: Uses the JNI service `GetStaticMethodID` to search for the named method. It needs the class reference, which can be determined with the `FC-ClassRef` function.
3. FC-MethodID-NonStatic: Uses the JNI service `GetMethodID` and needs the class reference in the same way as the `GetStaticMethodID` service.

Note: When using this sample, replace the XXXX in the encoding section with the encoding of your runtime.

```
        Identification Division.
        Program-Id. "JNIFIND".


       ****************************************************
       *    PROGRAMM    JNIFIND
       *    ----------------------------------------
       *******************************************************
        Environment Division.
        Configuration Section.
        Repository.
            Class jexception    Is 'java.lang.Exception'
            Class jclass        Is 'jclass'
            Class jmethodid     Is 'jmethodID'.
        Data Division.
        Working-Storage Section.

        01  booleanResult               Pic X.
            88 boolean-true             Value X'01' Through X'FF'.
            88 boolean-false            Value X'00'.

       *----------------------------------------------------------------------------*
       *     FindClass Section
       *----------------------------------------------------------------------------*
```

```
 77  Class-LocalRef              Object Reference jclass.
 77  Class-NameLen               Pic S9(04) Comp-5.


*-----------------------------------------------------------------------------*
*    Encoding EBCDIC to Modified UTF-8
*-----------------------------------------------------------------------------*
 01 Encoding.
    05 Enc-String                Pic X(9999) Value Spaces.
    05 Enc-UTF16                 Pic N(9999) Usage National
                                             Value Spaces.
    05 Enc-MethodName            Pic X(9999) Value Spaces.


 01 Ex                           Object Reference jexception.


 Linkage Section.
 77 L-ClassRef                   Object Reference jclass.
 77 L-MethodID                   Object Reference jmethodid.

*--- Encoding Section
 77 L-String                     Pic X(9999).
 77 L-StringLen                  Pic S9(04) Comp-5.

*--- FindClass Section
 77 L-ClassName                  Pic X(9999).
 77 L-ClassNameLen               Pic S9(04) Comp-5.

 01 JNIFIND-IN.
    Copy JNIFINDI.
 01 JNIFIND-OUT.
    Copy JNIFINDO.

 Copy JNI.

 Procedure Division Using JNIFIND-IN
                          JNIFIND-OUT.
 BEGIN-Entry Section.
 BEGINA.
     Set Address Of JNIenv To JNIEnvPtr
     Set Address Of JNINativeInterface To JNIenv
     Perform PROCESSING
     .
 BEGINZ.
     Goback
     .
```

```
*-----------------------------------------------------------------------------*
*    Steuerung durch Funktionscode
*-----------------------------------------------------------------------------*
 PROCESSING Section.
     Evaluate True
       When FC-ClassRef
         Set Address Of L-ClassName    To ClassName
         Set Address Of L-ClassNameLen To Address Of ClassNameLen
         Set Address Of L-ClassRef     To Ref
         Perform FindClass
       When FC-MethodRef
         Set Address Of L-ClassRef     To ObjectRef
         Set Address Of L-MethodID     To Ref
         Perform GetStaticMethodID
       When FC-MethodID-Nonstatic
         Set Address Of L-ClassRef     To ObjectRef
         Set Address Of L-MethodID     To Ref
         Perform U15-GetMethodID
       When Other
         Move 8 To Return-Code
     End-Evaluate
     .
 PROCESSINGZ.
     Exit
     .

 FindClass Section.
     Set Address Of L-String To Address Of L-ClassName
     Set Address Of L-StringLen To Address Of L-ClassNameLen
     Perform Encoding

     Call FindClass
        Using By Value JNIEnvPtr
                       Address Of Enc-String
        Returning Class-LocalRef
     Perform Check-Exception

     Call NewGlobalRef
        Using By Value JNIEnvPtr
                       Class-LocalRef
        Returning L-ClassRef
     Perform Check-Exception
```

```
    Call DeleteLocalRef
       Using By Value JNIEnvPtr
                      Class-LocalRef
    Perform Check-Exception
    .
FindClassZ.
    Exit
    .

GetStaticMethodID Section.
    Set Address Of L-String To MethodName
    Set Address Of L-StringLen To
       Address Of MethodNameLen
    Perform Encoding
    Move Enc-String To Enc-MethodName

    Set Address Of L-String To Signature
    Set Address Of L-StringLen To
       Address Of SignatureLen
    Perform Encoding


    Call GetStaticMethodID
         Using By Value JNIEnvPtr
                        L-ClassRef
                        Address Of Enc-MethodName
                        Address Of Enc-String
              Returning L-MethodID
    Perform Check-Exception
    .
GetStaticMethodIDZ.
    Exit
    .

GetMethodID Section.
    Set Address Of L-String To MethodName
    Set Address Of L-StringLen To
       Address Of MethodNameLen
    Perform Encoding
    Move Enc-String To Enc-MethodName

    Set Address Of L-String To Signature
    Set Address Of L-StringLen To
       Address Of SignatureLen
    Perform Encoding
    Call GetMethodID
         Using By Value JNIEnvPtr
                        L-ClassRef
                        Address Of Enc-MethodName
                        Address Of Enc-String
              Returning L-MethodID
    Perform Check-Exception
    .
GetMethodIDZ
    Exit
    .

Encoding Section.
*   replace the encoding XXXX with your codepage
    String Function National-Of(L-String(1:L-StringLen) XXXX)
       Delimited By Size
       Into Enc-UTF16
    End-String

    String Function Display-Of(Enc-UTF16(1:L-StringLen) 1208)
       Delimited By Size
       x'00' Delimited By Size
       Into Enc-String
    End-String
    .
EncodingZ.
    Exit
    .

Check-Exception Section.
    Call ExceptionCheck Using By Value JNIEnvPtr
                        Returning booleanResult

    If boolean-true
      Call ExceptionOccurred Using By Value JNIEnvPtr
                             Returning Ex
      Call ExceptionDescribe Using By Value JNIEnvPtr
      Call ExceptionClear Using By Value JNIEnvPtr
      Move 16 To Return-Code
      Goback
    End-If
    .
Check-ExceptionZ.
```

```
           Exit
           .

       End Program "JNIFIND".
```

8.3.1.1.3. JNIFINDI.cpy

This is the input copy for JNIFIND.cbl.

For FC-CLASSREF, only CLASSNAME and CLASSNAMELEN need to be filled.

For FC-METHODREF and FC-METHODID-NONSTATIC, you need to fill SIGNATURE, METHODNAME and their length fields. Also, OBJECTREF needs to be set to the Java class from which the method comes.

```
           05 FC                     VALUE 'C'
                                     PIC X(01).
                88 FC-CLASSREF       VALUE 'C'.
                88 FC-METHODREF      VALUE 'M'.
                88 FC-METHODID-NONSTATIC VALUE 'N'.
           05 CLASSNAME              USAGE POINTER VALUE NULL.
           05 CLASSNAMELEN           VALUE +0 USAGE COMP-5
                                     PIC S9(04).
           05 METHODNAME             USAGE POINTER VALUE NULL.
           05 METHODNAMELEN          VALUE +0 USAGE COMP-5
                                     PIC S9(04).
           05 SIGNATURE              USAGE POINTER VALUE NULL.
           05 SIGNATURELEN           VALUE +0 USAGE COMP-5
                                     PIC S9(04).
           05 OBJECTREF              USAGE POINTER VALUE NULL.
```

8.3.1.1.4. JNIFINDO.cpy

This is the output copy for JNIFIND.cbl. It returns the determined class or method reference.

```
           05 REF                    USAGE POINTER VALUE NULL.
```

8.3.1.2. Compiling the COBOL stuff

8.3.1.2.1. Procedure COMPJNI

In the compile procedure, set the variables to the right data sets and execute it with the member name you want to compile. This procedure uses the

INCLUDE statement to include two members, IGZCJAVA and LIBJVM, in LLQ.SYSDEFSD.DATA.

```
//COMPJNI   PROC MEMBER=,                              < Name MEMBER-Member
//     SRCLIB='LLQ.SOURCE.CNTL',                       < MEMBER Library
//     LOADLIB='LLQ.TEST.LIB',                         < SYSLMOD
//     DBRMLIB='LLQ.TEST.DBRMLIB',
//     SYSDEBUG='LLQ.SYSDEBUG.DATA',
//     SYSDEFSD='LLQ.SYSDEFSD.DATA',
//     SYSLIB='LLQ.TEST.SYSLIB',
//     COPYLIB='LLQ.COPYBOOK.LIB'
//*******************************************
//* Compile
//*******************************************
//COMPILE  EXEC PGM=IGYCRCTL,PARM=OPTFILE
//SYSLIN    DD  DSN=&LOADS,SPACE=(CYL,(10,2)),
//              STORCLAS=ALTEMP,DISP=(,PASS)
//SYSIN     DD DISP=SHR,DSN=&SRCLIB.(&MEMBER.)
//SYSOPTF DD *
  NOADV,APOST,FLAG(I,W),NOC(E),LIB,MAP,RENT,NOTERM,
  NOSEQ,XREF,AWO,CP(273),NOOPT,DATA(31),SIZE(50000K),
  TEST(NOHOOK,SEP),OFFSET
//SYSPRINT DD SYSOUT=*
//DBRMLIB  DD  DSN=&DBRMLIB.(&MEMBER.),DISP=SHR
//SYSDEBUG DD DISP=SHR,DSN=&SYSDEBUG.(&MEMBER.)
//STEPLIB  DD  DSN=SCD.DB2U888.A.DSNEXIT,DISP=SHR
//          DD  DSN=SCD.DB2U888.A.DSNLOAD,DISP=SHR
//          DD  DSN=SYS1.IGY.V421.SIGYCOMP,DISP=SHR
//SYSLIB    DD DISP=SHR,DSN=&COPYLIB.
//          DD  DSN=SYS1.CEE.SCEESAMP,DISP=SHR
//SYSJAVA  DD PATH='/prod/vjh/jbf/jni/src/&MEMBER..java',
//             PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//             PATHMODE=(SIRWXU,SIRWXG,SIRWXO),
//             FILEDATA=TEXT
//SYSUT1    DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT2    DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT3    DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT4    DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT5    DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT6    DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT7    DD UNIT=VIO,SPACE=(CYL,(1,1))
//**********************************************************************
*
//*       PRELINK
```

```
//*********************************************************************
*
//PRELINK  EXEC PGM=EDCPRLK
//STEPLIB DD DISP=SHR,DSN=SYS1.CEE.SCEERUN
//SYSMSGS DD DISP=SHR,DSN=SYS1.CEE.SCEEMSGP(EDCPMSGE)
//INCLIB2  DD  DSN=&SYSDEFSD.,DISP=SHR
//SYSIN DD DISP=(OLD,DELETE),
//         DSN=&LOADS
//      DD DDNAME=SYSIN2
//SYSMOD DD DISP=(NEW,PASS),
//          DSN=&LOADS2,
//          UNIT=VIO,
//          SPACE=(32000,(30,30)),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSOUT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSIN2 DD *
  INCLUDE INCLIB2(IGZCJAVA)
  INCLUDE INCLIB2(LIBJVM)
//*********************************************************************
***
//*   Link
//*********************************************************************
***
//LINK     EXEC PGM=IEWL,PARM=('RENT,LIST,LET,DYNAM(DLL),CASE(MIXED)',
// 'AMODE=31,RMODE=ANY,REUS')
//SYSLIB   DD DSN=&SYSLIB.,DISP=SHR
//         DD DSN=SYS1.CEE.SCEELKED,DISP=SHR
//         DD DSN=SYS1.CEE.SCEELKEX,DISP=SHR
//         DD DISP=SHR,DSN=SYS1.CSSLIB
//SYSPRINT DD SYSOUT=*
//SYSTERM  DD SYSOUT=*
//SYSLMOD  DD  DISP=SHR,
//             DSN=&LOADLIB.(&MEMBER.)
//SYSDEFSD DD DUMMY
//SYSLIN   DD  DSN=&LOADS2,UNIT=WORK1,DISP=(OLD,DELETE)
```

Note: LLQ.SYSDEFSD.DATA contains the two members IGZCJAVA and LIBJVM(delivered by IBM)

8.3.1.2.2. Content of IGZCJAVA

```
 IMPORT CODE,IGZCJAVA,'GetClassObject'
 IMPORT CODE,IGZCJAVA,'GetEnvPtr'
```

```
IMPORT CODE,IGZCJAVA,'GetInstanceData'
IMPORT CODE,IGZCJAVA,'GetJVMPtr'
```

8.3.1.2.3. Content of LIBJVM

```
IMPORT  CODE,'libjvm.so','addHarmonyPortLibToVMArgs'
IMPORT  CODE,'libjvm.so','deregisterj9scarWithTrace'
IMPORT CODE,'libjvm.so','exitHook'
IMPORT CODE,'libjvm.so','findDirContainingFile'
IMPORT CODE,'libjvm.so','findDirUplevelToDirContainingFile'
IMPORT DATA,'libjvm.so','g_sigaction'
IMPORT CODE,'libjvm.so','getj9bin'
IMPORT CODE,'libjvm.so','initializeSyscallInterruptMechanism'
IMPORT CODE,'libjvm.so','isFileInDir'
IMPORT CODE,'libjvm.so','jio_fprintf'
IMPORT CODE,'libjvm.so','jio_snprintf'
IMPORT CODE,'libjvm.so','jio_vfprintf'
IMPORT CODE,'libjvm.so','jio_vsnprintf'
IMPORT DATA,'libjvm.so','j9copyright'
IMPORT DATA,'libjvm.so','j9scar_group0'
IMPORT DATA,'libjvm.so','j9scar_group1'
IMPORT DATA,'libjvm.so','j9scar_group2'
IMPORT DATA,'libjvm.so','j9scar_group3'
IMPORT DATA,'libjvm.so','j9scar_group4'
IMPORT DATA,'libjvm.so','j9scar_group5'
IMPORT DATA,'libjvm.so','j9scar_tpids0'
IMPORT DATA,'libjvm.so','j9scar_tpids1'
IMPORT DATA,'libjvm.so','j9scar_tpids2'
IMPORT DATA,'libjvm.so','j9scar_tpids3'
IMPORT DATA,'libjvm.so','j9scar_tpids4'
IMPORT DATA,'libjvm.so','j9scar_UtActive'
IMPORT DATA,'libjvm.so','j9scar_UtLevels'
IMPORT DATA,'libjvm.so','j9scar_UtModuleInfo'
IMPORT DATA,'libjvm.so','j9scar_UtTraceVersionInfo'
IMPORT CODE,'libjvm.so','post_block'
IMPORT CODE,'libjvm.so','postInterruptFileOperation'
IMPORT CODE,'libjvm.so','pre_block'
IMPORT CODE,'libjvm.so','preInterruptFileOperation'
IMPORT CODE,'libjvm.so','registerj9scarWithTrace'
IMPORT CODE,'libjvm.so','registerInterruptable'
IMPORT CODE,'libjvm.so','shutdownSyscallInterruptMechanism'
IMPORT CODE,'libjvm.so','unregisterInterruptable'
IMPORT CODE,'libjvm.so','z_compareAndSwapUDATA'
IMPORT CODE,'libjvm.so','z_compareAndSwapU32'
IMPORT CODE,'libjvm.so','z_issueReadBarrier'
IMPORT CODE,'libjvm.so','z_issueReadWriteBarrier'
IMPORT CODE,'libjvm.so','z_issueWriteBarrier'
IMPORT CODE,'libjvm.so','z_CX8Field'
IMPORT CODE,'libjvm.so','CX8Field'
IMPORT CODE,'libjvm.so','DestroyJavaVM'
IMPORT CODE,'libjvm.so','DLLinit'
IMPORT CODE,'libjvm.so','GetStringPlatform'
IMPORT CODE,'libjvm.so','GetStringPlatformLength'
IMPORT CODE,'libjvm.so','GetXUsage'
IMPORT CODE,'libjvm.so','JNI_a2e_vsprintf'
IMPORT CODE,'libjvm.so','JNI_CreateJavaVM'
IMPORT CODE,'libjvm.so','JNI_GetCreatedJavaVMs'
IMPORT CODE,'libjvm.so','JNI_GetDefaultJavaVMInitArgs'
IMPORT CODE,'libjvm.so','JVM_Accept'
IMPORT CODE,'libjvm.so','JVM_ActiveProcessorCount'
IMPORT CODE,'libjvm.so','JVM_AllocateNewArray'
IMPORT CODE,'libjvm.so','JVM_AllocateNewObject'
IMPORT CODE,'libjvm.so','JVM_Available'
IMPORT CODE,'libjvm.so','JVM_ClassDepth'
IMPORT CODE,'libjvm.so','JVM_ClassLoaderDepth'
IMPORT CODE,'libjvm.so','JVM_Close'
IMPORT CODE,'libjvm.so','JVM_Connect'
IMPORT CODE,'libjvm.so','JVM_ConstantPoolGetClassAt'
IMPORT CODE,'libjvm.so','JVM_ConstantPoolGetClassAtIfLoaded'
IMPORT CODE,'libjvm.so','JVM_ConstantPoolGetDoubleAt'
IMPORT CODE,'libjvm.so','JVM_ConstantPoolGetFieldAt'
IMPORT CODE,'libjvm.so','JVM_ConstantPoolGetFieldAtIfLoaded'
IMPORT CODE,'libjvm.so','JVM_ConstantPoolGetFloatAt'
IMPORT CODE,'libjvm.so','JVM_ConstantPoolGetIntAt'
IMPORT CODE,'libjvm.so','JVM_ConstantPoolGetLongAt'
IMPORT CODE,'libjvm.so','JVM_ConstantPoolGetMemberRefInfoAt'
IMPORT CODE,'libjvm.so','JVM_ConstantPoolGetMethodAt'
IMPORT CODE,'libjvm.so','JVM_ConstantPoolGetMethodAtIfLoaded'
IMPORT CODE,'libjvm.so','JVM_ConstantPoolGetSize'
IMPORT CODE,'libjvm.so','JVM_ConstantPoolGetStringAt'
IMPORT CODE,'libjvm.so','JVM_ConstantPoolGetUTF8At'
IMPORT CODE,'libjvm.so','JVM_CurrentClassLoader'
IMPORT CODE,'libjvm.so','JVM_CurrentLoadedClass'
IMPORT CODE,'libjvm.so','JVM_CurrentTimeMillis'
IMPORT CODE,'libjvm.so','JVM_CX8Field'
IMPORT CODE,'libjvm.so','JVM_DefineClassWithSource'
IMPORT CODE,'libjvm.so','JVM_DumpThreads'
IMPORT CODE,'libjvm.so','JVM_ExpandFdTable'
IMPORT CODE,'libjvm.so','JVM_FindLibraryEntry'
IMPORT CODE,'libjvm.so','JVM_FindSignal'
IMPORT CODE,'libjvm.so','JVM_FreeMemory'
IMPORT CODE,'libjvm.so','JVM_GetAllThreads'
IMPORT CODE,'libjvm.so','JVM_GetCallerClass'
IMPORT CODE,'libjvm.so','JVM_GetClassAccessFlags'
IMPORT CODE,'libjvm.so','JVM_GetClassAnnotations'
IMPORT CODE,'libjvm.so','JVM_GetClassConstantPool'
IMPORT CODE,'libjvm.so','JVM_GetClassContext'
IMPORT CODE,'libjvm.so','JVM_GetClassField'
IMPORT CODE,'libjvm.so','JVM_GetClassLoader'
IMPORT CODE,'libjvm.so','JVM_GetClassMethod'
IMPORT CODE,'libjvm.so','JVM_GetClassName'
IMPORT CODE,'libjvm.so','JVM_GetClassSignature'
IMPORT CODE,'libjvm.so','JVM_GetEnclosingMethodInfo'
IMPORT CODE,'libjvm.so','JVM_GetInterfaceVersion'
IMPORT CODE,'libjvm.so','JVM_GetLastErrorString'
IMPORT CODE,'libjvm.so','JVM_GetManagement'
IMPORT CODE,'libjvm.so','JVM_GetPortLibrary'
IMPORT CODE,'libjvm.so','JVM_GetSystemPackage'
IMPORT CODE,'libjvm.so','JVM_GetSystemPackages'
IMPORT CODE,'libjvm.so','JVM_GetThreadInterruptEvent'
IMPORT CODE,'libjvm.so','JVM_GC'
IMPORT CODE,'libjvm.so','JVM_GCNoCompact'
IMPORT CODE,'libjvm.so','JVM_Halt'
IMPORT CODE,'libjvm.so','JVM_InitializeSocketLibrary'
IMPORT CODE,'libjvm.so','JVM_InvokeMethod'
IMPORT CODE,'libjvm.so','JVM_IsNaN'
IMPORT CODE,'libjvm.so','JVM_LatestUserDefinedLoader'
IMPORT CODE,'libjvm.so','JVM_Listen'
IMPORT CODE,'libjvm.so','JVM_LoadLibrary'
IMPORT CODE,'libjvm.so','JVM_Lseek'
IMPORT CODE,'libjvm.so','JVM_MaxMemory'
IMPORT CODE,'libjvm.so','JVM_MaxObjectInspectionAge'
IMPORT CODE,'libjvm.so','JVM_MonitorNotify'
IMPORT CODE,'libjvm.so','JVM_MonitorNotifyAll'
IMPORT CODE,'libjvm.so','JVM_MonitorWait'
IMPORT CODE,'libjvm.so','JVM_NanoTime'
IMPORT CODE,'libjvm.so','JVM_NativePath'
IMPORT CODE,'libjvm.so','JVM_NewInstanceFromConstructor'
IMPORT CODE,'libjvm.so','JVM_OnExit'
IMPORT CODE,'libjvm.so','JVM_Open'
IMPORT CODE,'libjvm.so','JVM_RaiseSignal'
IMPORT CODE,'libjvm.so','JVM_RawAllocate'
```

```
 IMPORT CODE,'libjvm.so','JVM_RawCalloc'
 IMPORT CODE,'libjvm.so','JVM_RawFree'
 IMPORT CODE,'libjvm.so','JVM_RawMonitorCreate'
 IMPORT CODE,'libjvm.so','JVM_RawMonitorDestroy'
 IMPORT CODE,'libjvm.so','JVM_RawMonitorEnter'
 IMPORT CODE,'libjvm.so','JVM_RawMonitorExit'
 IMPORT CODE,'libjvm.so','JVM_RawRealloc'
 IMPORT CODE,'libjvm.so','JVM_Read'
 IMPORT CODE,'libjvm.so','JVM_Recv'
 IMPORT CODE,'libjvm.so','JVM_RecvFrom'
 IMPORT CODE,'libjvm.so','JVM_RegisterSignal'
 IMPORT CODE,'libjvm.so','JVM_RegisterUnsafeMethods'
 IMPORT CODE,'libjvm.so','JVM_Send'
 IMPORT CODE,'libjvm.so','JVM_SendTo'
 IMPORT CODE,'libjvm.so','JVM_SetLength'
 IMPORT CODE,'libjvm.so','JVM_Sleep'
 IMPORT CODE,'libjvm.so','JVM_Socket'
 IMPORT CODE,'libjvm.so','JVM_SocketAvailable'
 IMPORT CODE,'libjvm.so','JVM_SocketClose'
 IMPORT CODE,'libjvm.so','JVM_Startup'
 IMPORT CODE,'libjvm.so','JVM_SupportsCX8'
 IMPORT CODE,'libjvm.so','JVM_Sync'
 IMPORT CODE,'libjvm.so','JVM_Timeout'
 IMPORT CODE,'libjvm.so','JVM_TotalMemory'
 IMPORT CODE,'libjvm.so','JVM_TraceInstructions'
 IMPORT CODE,'libjvm.so','JVM_TraceMethodCalls'
 IMPORT CODE,'libjvm.so','JVM_UcsOpen'
 IMPORT CODE,'libjvm.so','JVM_Write'
 IMPORT CODE,'libjvm.so','JVM_ZipHook'
 IMPORT CODE,'libjvm.so','NewStringPlatform'
```

8.3.1.3. Coding Java

This is the Java class that provides COBMAIN.cbl with the needed data. As a simple example, this class returns selected data from a String array back to its caller.

8.3.1.3.1. JNIHelper.java

```
package com.example;

public class JNIHelper
{
    private static final String[] data = {
        "Datafield1",
        "Datafield2",
        "Datafield3",
        "Datafield4",
        "Datafield5",
        "Datafield6",
        "Datafield7",
        "Datafield8",
        "Datafield9",
        "Datafield10"};

    public static byte[] returnData(int dataPos)
    {
        if (dataPos > data.length || dataPos < 1)
        {
            return null;
        }
        return data[dataPos - 1].getBytes();
    }
}
```

8.3.1.4. Lessons learned: Avoid using DLLs

Restrictions for calling DLL programs are, among other things, mainly an LE restriction of calling the same program from non-DLL and DLL, or simply the need to make a static call. To remove these restrictions and creating a non-DLL program with JNI calls you need to avoid using the INVOKE statement. This is the only statement which requires the DLL option.

8.3.1.5. Lessons learned: Avoid using INVOKE

INVOKE is a convenient method to call Java but is at the same time expensive. If you remove the INVOKE statement, you need to call Java another way. That means you have to use JNI services such as `FindClass`, `GetStaticMethodID` and `GetMethodID`.

The downside of this is, you have to write a lot more code if you replace INVOKE with JNI calls, and you have to do the encoding from EBCDIC to UTF-8 yourself, which is normally handled by INVOKE. But just because you have more to write yourself, it doesn't mean you need more CPU resources to access Java. On the contrary, you can now cache the references of your classes and methods so you don't need to make unnecessary JNI calls, which INVOKE does every time.

8.3.1.6. Lessons learned: Use static Java methods

Why should you use static methods, instead of object methods? Because creating objects via JNI is more troublesome than just calling a static method.

As an example, we try to call a method with no parameters or return values. What you now need to do first is the same procedure regardless of static or no static (except for the JNI service you need to call `GetStaticMethodID` or `GetMethodID`), you need to get the references for the class and the method you want to call via JNI services.

Now if you want to call the static method, congratulations-- you have everything you need. Just use the JNI service `CallStatic<type>Method` with the right references and the method will be executed (we replace <*type*> with Void because we have no return value).

```
           Call CallStaticVoidMethod
                Using By Value JNIEnvPtr
                               Class-Ref
                               Static-Method-Ref
```

If you don't, then you need to take some additional steps. First you need to create an object from the class. An object is created when you call the constructor from this specific class. To call the constructor you have to get the reference via JNI service `GetMethodID` with the method name `<init>` and the right signature.

```
           Move '<init>' To Method-Name
      *    No parameters for constructor => (),
      *    return value always V => void
           Move '()V'    To Signature
      *    Now convert from EBCDIC to UTF-8
…
           Call GetMethodID
                Using By Value JNIEnvPtr
                               Class-Ref
                               Method-Name-UTF8-Ptr
                               Signature-UTF8-Ptr
                    returning Constructor-Ref
```

When you have the constructor, you have to create a new object with the `NewObject` service, which needs the constructor and the parameters.

```
           Call NewObject
                Using By Value JNIEnvPtr
```

```
                              Class-Ref
                              Constructor-Ref
                   returning Local-Object-Ref
      *            Add constructor parameters here if needed
```

After these steps you can finally call the method you want by using the JNI service `Call<type>Method` (just like the `Call<type>StaticMethod`, we replace `<type>` with Void).

```
           Call CallVoidMethod
                Using By Value JNIEnvPtr
                               Object-Ref
                               Method-Ref
```

Note: If you want to use your Object again, make it global via the JNI service NewGlobalRefand remove the local reference with DeleteLocalRefbecause local references are freed after the return of the native method.

```
           Call NewGlobalRef
              Using By Value JNIEnvPtr
                             Local-Object-Ref
              Returning Object-Ref
…
           Call DeleteLocalRef
              Using By Value JNIEnvPtr
                             Local-Object-Ref
```

8.3.2. Handling Java exceptions in COBOL

After each JNI service you need to check if an Exception occurred by using `ExceptionCheck`. If that is the case, you need to retrieve the exception via `ExceptionOccurred` and use `ExceptionClear` to signal to the JVM that the Exception was handled nicely. Optionally, you can print the stack trace via `ExceptionDescribe`.

```
       Check-Exception Section.
           Call ExceptionCheck Using By Value JNIEnvPtr
                               Returning booleanResult
           If boolean-true
             Call ExceptionOccurred Using By Value JNIEnvPtr
                                    Returning Ex
             Call ExceptionDescribe Using By Value JNIEnvPtr
             Call ExceptionClear Using By Value JNIEnvPtr
             Move 16 To Return-Code
             Goback
           End-If
           .
       Check-ExceptionZ.
           Exit
           .
```

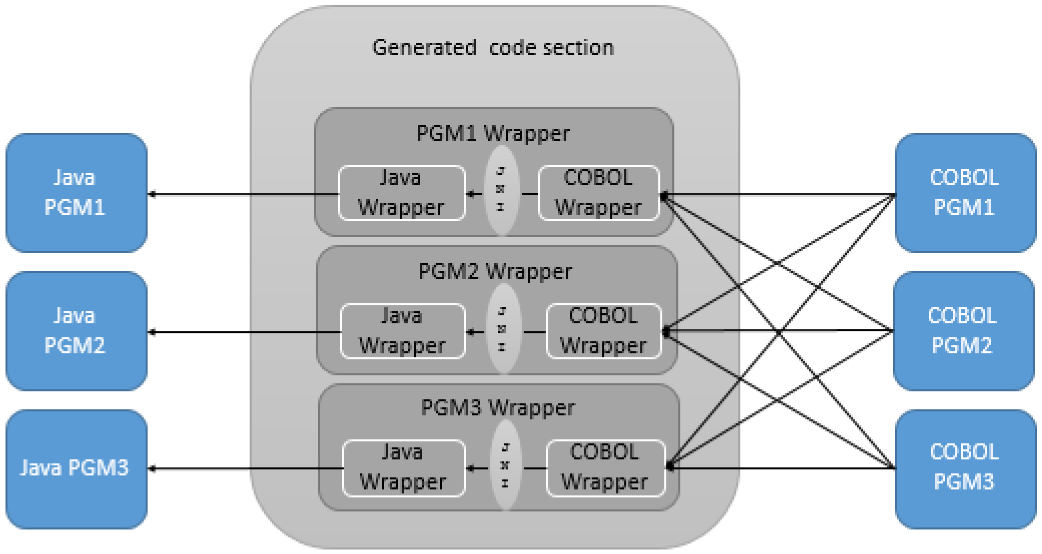
8.3.3. Wrapping JNI with our own static converters

In our initial attempt, we tried to hide the complexity by generating wrapper code using our own static converters.

8.3.3.1. Hiding the JNI complexity

Generating the JNI calls for COBOL is one way to hide the complexity. You could use Java classes with annotations to fill necessary information for the generator, or you generate the COBOL as well as the Java classes via your own domain-specific language (DSL).

Figure 39. Generating Java and COBOL wrapper code



8.3.3.2. Lessons learned

The disadvantage of this method is that you generate many artifacts that you will have to manage. If you want to, for example, change something in yourgenerated code, you would have to keep track of all your generated applications with JNI calls, regenerate your code, and test it. This restriction imposes higher code management efforts.

8.3.4. Wrapping JNI using a roll-your-own dynamic converter

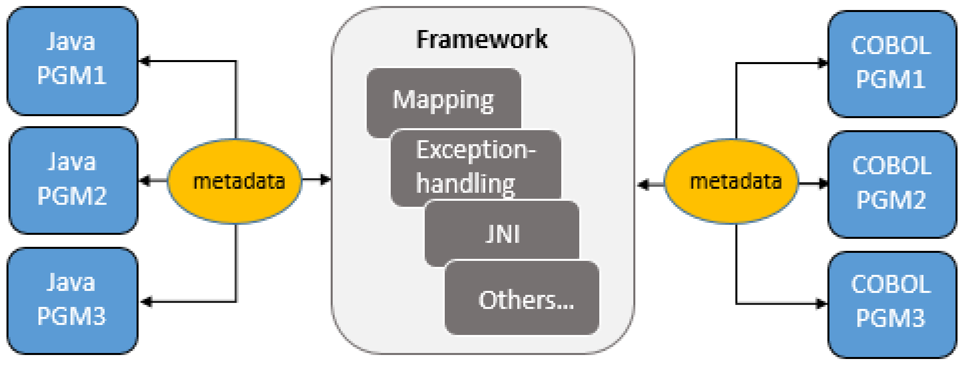
8.3.4.1. Reasoning for writing and using a custom converter

With our first attempt to combine COBOL and Java, we learned that we lose some flexibility in Java if we do not enhance things on the COBOL side. In Java it is common to use collections such as lists. A list has no limits in the number of elements it can store. In COBOL you can use tables with OCCURS statements, but you must define the largest number of elements you can store in the table. If you want to map a Java list to a COBOL table, you have to set limitations on the Java side.

This is the reason why we implemented a COBOL representation of a list as a subprogram. The COBOL list has no limitations in the number of stored elements and matches perfectly to the Java list.

To be able to map our COBOL data to Java with flexibility and vice versa, we designed our own mapping framework that is aware of the rules of how we store our data in COBOL.

Figure 40. Mapping framework for a custom converter

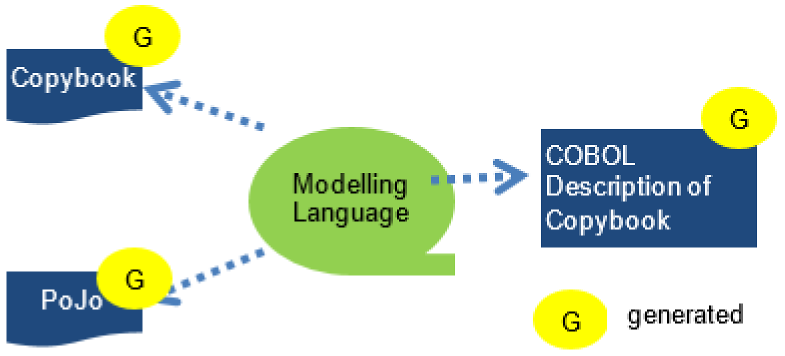


8.3.4.2. Implementation

First of all, we designed a XML-based modelling language to describe our data models. Based on these model we generate the COBOL representation based on a copybook and the Java representation based on the Plain Old Java Object (POJO) programming model.

Our generator also produces a description of the memory layout of the COBOL copybook. The description enables us to do introspection on the COBOL side.

Figure 41. Generating COBOL and Java representations using the modelling language



It is also accessible on the Java side and used to map the data from COBOL to Java and back.

8.3.5. Other important information

8.3.5.1. Data type designations

The following table contains a list of the JNI data types and their corresponding representations in Java and COBOL.

Table 4. Data type mapping from JNI to Java and COBOL

JNI designation	Java designation	COBOL object designation	COBOL declaration
B	byte	jbyte	PIC X
S	short	jshort	PIC S9(04)
I	int	jint	PIC S9(09)
J	long	jlong	PIC S9(18)
Z	boolean	jboolean	PIC X[1]
F	float	jfloat	COMP-1
D	double	jdouble	COMP-2
C	char	jchar	PIC N
L	Object	jobject	Object Reference[2]
S	String	jstring	Object Reference
[…[3]	…[]	j…Array	Object Reference

1 Two boolean conditions ("true" and "false") are represented by hex-values. **x'00'** = **false**, **x'01'** to **x'FF'** = **true**

2 Reference to the COBOL object e.g.: „**Object Reference jbyteArray**"

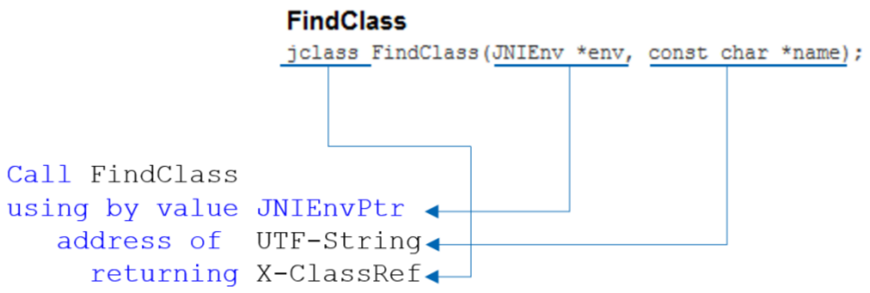
8.3.5.2. Sample JNI in COBOL using FindClass

The following example demonstrates how a JNI function is called from a COBOL program, as well as the declarations of the variables.

```
…
       Environment Division.
       Configuration Section.
       Repository.
           Class jclass        Is 'jclass'
           …
           .
…
       Working-Storage Section.
           …
           77 X-ClassRef                    Object Reference jclass.
           77 UTF-String                    Pic X(9999).
       Procedure Division Using …
           …
           Call FindClass using by value JNIEnvPtr
                             address of UTF-String
                              returning X-ClassRef
```

The following figure shows how the JNI function is mapped to a COBOL call.

Figure 42. Mapping FindClass to a COBOL call



8.4. Additional considerations for production usage

[3] „…" represents any element in the same column. For example, [… = [J, j…Array = jlongArray, …[] = long[]

In this section, we will describe challenges that need to be taken into consideration when this technology is staged into a production environment. Some things to consider include error handling and usage statistics.

8.4.1. Error handling in production

In a production environment, dumps will become larger, which means storage pools (DFSMS) sizing should be observed and eventually increased. In addition, new error codes as a result of the mandatory LE setting with POSIX(ON) need to be handled.

You will want to consider using a product, for example, IBM Fault Analyzer, to analyze the dump, and provide a how-to cookbook for the programmer who needs to analyze the issues.

8.4.2. Identifying CPU, zIIP, and zIIP on CPU (zICP) usage

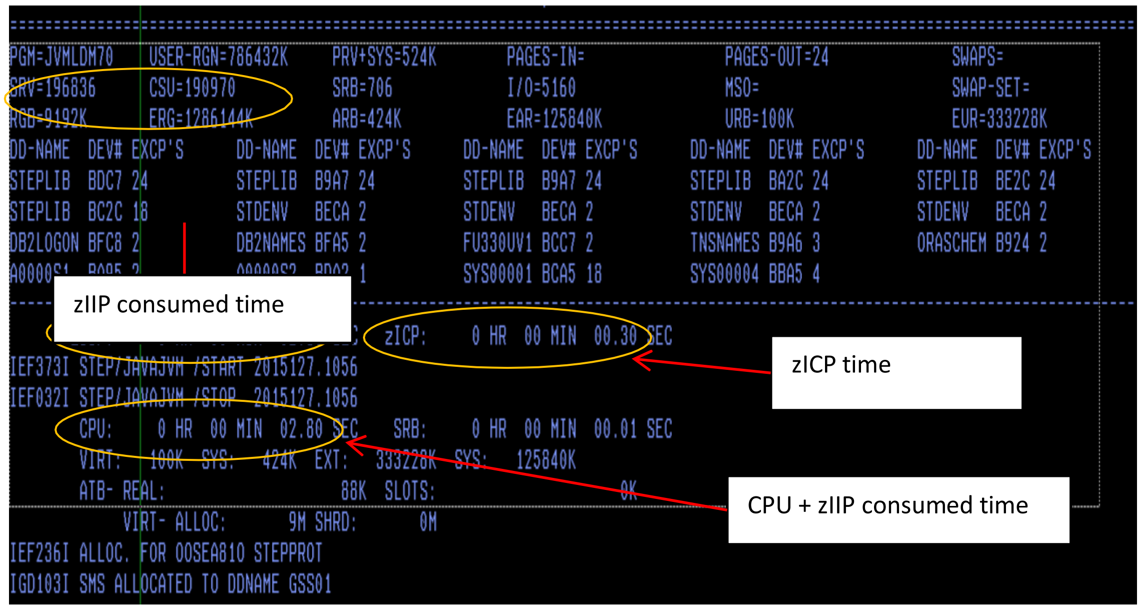
When Java is used substantially in your systems, the processing resource consumption is mainly on zIIP engines. z/OS Workload Manager (WLM) assigns CPU cycles when not enough zIIP resources are available. As a result, you have to monitor the zIIP-eligible workload that was executed on CPUs, and based on the workload, consider acquiring additional zIIP(s).

There are several ways to identify the zIIP and zICP.

8.4.2.1. Joboutpout statistics

You can show the resources on a job basis by enhancing the IEFACTRT exit. By default those values are not reported in the job output.

Figure 43. Output from the IEFACTRT exit

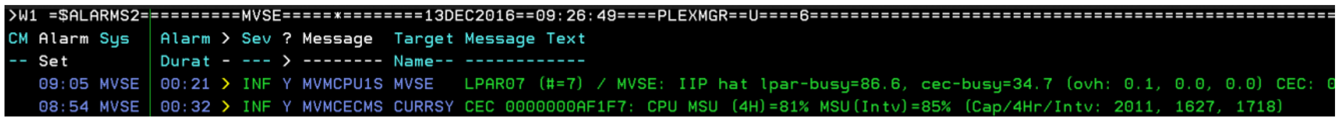


```
==========================================================================================================
PGM=JVMLDM70    USER-RGN=786432K     PRV+SYS=524K        PAGES-IN=             PAGES-OUT=24          SWAPS=
SRV=196836      CSU=190970           SRB=706             I/O=5160              MSO=                  SWAP-SET=
RGD=9192K       ERG=1286144K         ARB=424K            EAR=125840K           URB=100K              EUR=333228K
DD-NAME  DEV# EXCP'S      DD-NAME  DEV# EXCP'S      DD-NAME  DEV# EXCP'S      DD-NAME  DEV# EXCP'S      DD-NAME  DEV# EXCP'S
STEPLIB  BDC7 24          STEPLIB  B9A7 24          STEPLIB  B9A7 24          STEPLIB  BA2C 24          STEPLIB  BE2C 24
STEPLIB  BC2C 18          STDENV   BECA 2           STDENV   BECA 2           STDENV   BECA 2           STDENV   BECA 2
DB2LOGON BFC8 2           DB2NAMES BFA5 2           FU330UV1 BCC7 2           TNSNAMES B9A6 3           ORASCHEM B924 2
A0000S1  B0A5 2           A0000S2  BD02 1           SYS00001 BCA5 18          SYS00004 BBA5 4
----------------------------------------------------------------------------------------------------------
                                              C  zICP:     0 HR  00 MIN  00.30 SEC
IEF373I STEP/JAVAJVM /START 2015127.1056
IEF032I STEP/JAVAJVM /STOP  2015127.1056
        CPU:     0 HR  00 MIN  02.80 SEC    SRB:     0 HR  00 MIN  00.01 SEC
        VIRT:   100K  SYS:   424K  EXT:   333228K  SYS:   125840K
        ATB- REAL:                  88K  SLOTS:                 0K
             VIRT- ALLOC:      9M SHRD:      0M
IEF236I ALLOC. FOR 00SEA810 STEPPROT
IGD103I SMS ALLOCATED TO DDNAME GSS01
```

8.4.2.2. Real-time alerting when thresholds are reached

You could use a commercial product or your own implementation to analyze the zIIP and zICP usage. In our environment, based on BMC MainView information, a real-time alerting solution is implemented. A typical message issued looks as follows:

Figure 44. Real-time alerting messages

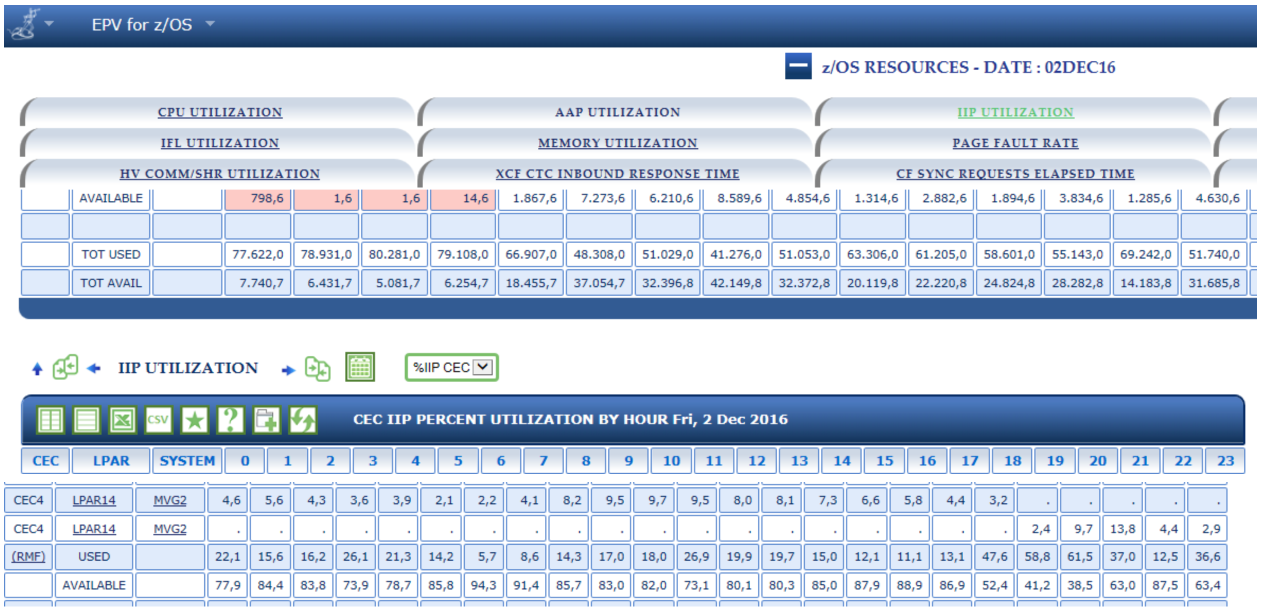


```
>W1 =$ALARMS2=========MVSE=====*========13DEC2016==09:26:49====PLEXMGR==U====6==================================================
CM Alarm Sys  | Alarm > Sev ? Message  Target Message Text
-- Set        | Durat - --- > -------- Name-- ------------
   09:05 MVSE | 00:21 > INF Y MVMCPU1S MVSE   LPAR07 (#=7) / MVSE: IIP hat lpar-busy=86.6, cec-busy=34.7 (ovh: 0.1, 0.0, 0.0) CEC: 0
   08:54 MVSE | 00:32 > INF Y MVMCECMS CURRSY CEC 0000000AF1F7: CPU MSU (4H)=81% MSU(Intv)=85% (Cap/4Hr/Intv: 2011, 1627, 1718)
```

8.4.2.3. Statistics view on IIP and ICP

You could use a commercial product or your own implementation to analyze the IIP and ICP usage. One such product is EPV for z/OS. This product gives you a view of the IIP workload. You can also define thresholds that initiate events you want to capture.

Figure 45. EPV for z/OS



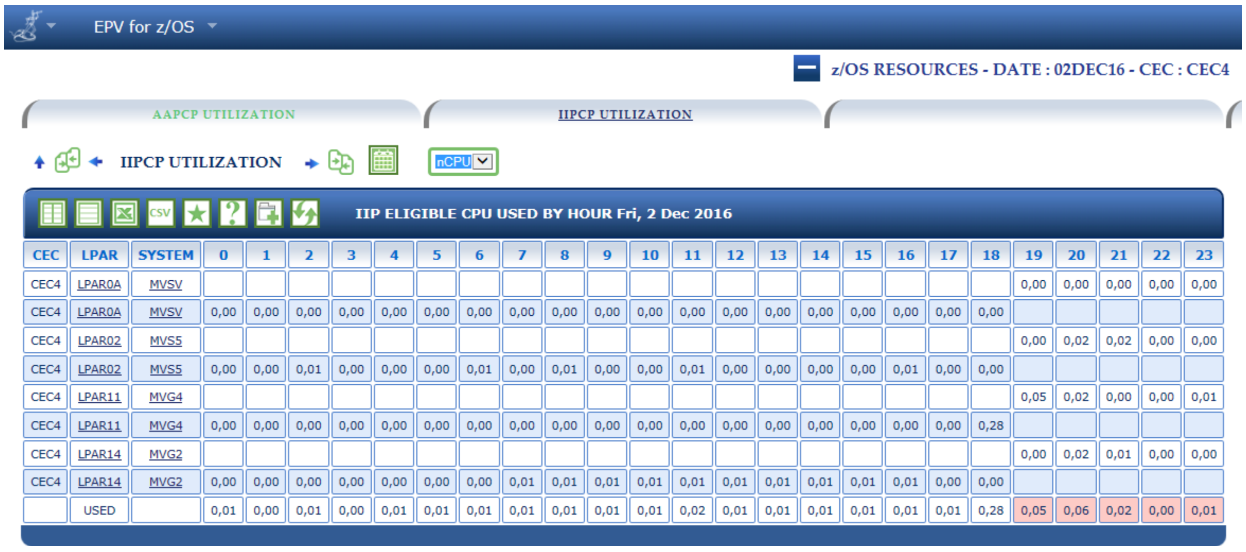
	AVAILABLE		798,6	1,6	1,6	14,6	1.867,6	7.273,6	6.210,6	8.589,6	4.854,6	1.314,6	2.882,6	1.894,6	3.834,6	1.285,6	4.630,6
	TOT USED		77.622,0	78.931,0	80.281,0	79.108,0	66.907,0	48.308,0	51.029,0	41.276,0	51.053,0	63.306,0	61.205,0	58.601,0	55.143,0	69.242,0	51.740,0
	TOT AVAIL		7.740,7	6.431,7	5.081,7	6.254,7	18.455,7	37.054,7	32.396,8	42.149,8	32.372,8	20.119,8	22.220,8	24.824,8	28.282,8	14.183,8	31.685,8

CEC IIP PERCENT UTILIZATION BY HOUR Fri, 2 Dec 2016

CEC	LPAR	SYSTEM	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
CEC4	LPAR14	MVG2	4,6	5,6	4,3	3,6	3,9	2,1	2,2	4,1	8,2	9,5	9,7	9,5	8,0	8,1	7,3	6,6	5,8	4,4	3,2	.	.	.	.	.
CEC4	LPAR14	MVG2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	2,4	9,7	13,8	4,4	2,9
(RMF)	USED		22,1	15,6	16,2	26,1	21,3	14,2	5,7	8,6	14,3	17,0	18,0	26,9	19,9	19,7	15,0	12,1	11,1	13,1	47,6	58,8	61,5	37,0	12,5	36,6
	AVAILABLE		77,9	84,4	83,8	73,9	78,7	85,8	94,3	91,4	85,7	83,0	82,0	73,1	80,1	80,3	85,0	87,9	88,9	86,9	52,4	41,2	38,5	63,0	87,5	63,4

The following chart shows the number of IIP eligible workload executed on CPUs.

Figure 46. IIP eligible workload in EPV for z/OS


EPV for z/OS
z/OS RESOURCES - DATE : 02DEC16 - CEC : CEC4
AAPCP UTILIZATION
IIPCP UTILIZATION
IIPCP UTILIZATION
nCPU
CSV
IIP ELIGIBLE CPU USED BY HOUR Fri, 2 Dec 2016

CEC	LPAR	SYSTEM	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
CEC4	LPAR0A	MVSV																				0,00	0,00	0,00	0,00	0,00
CEC4	LPAR0A	MVSV	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00					
CEC4	LPAR02	MVS5																				0,00	0,02	0,02	0,00	0,00
CEC4	LPAR02	MVS5	0,00	0,00	0,01	0,00	0,00	0,00	0,01	0,00	0,01	0,00	0,00	0,01	0,00	0,00	0,00	0,00	0,01	0,00	0,00					
CEC4	LPAR11	MVG4																				0,05	0,02	0,00	0,00	0,01
CEC4	LPAR11	MVG4	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,28					
CEC4	LPAR14	MVG2																				0,00	0,02	0,01	0,00	0,00
CEC4	LPAR14	MVG2	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,00	0,00					
	USED		0,01	0,00	0,01	0,00	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,02	0,01	0,01	0,01	0,01	0,01	0,01	0,28	0,05	0,06	0,02	0,00	0,01

8.4.3. Accounting: New SMF records 29 & 121

For IMS V14, JVM usage statistics are captured as type-29, subtype-2 records in z/OS System Management Facility (SMF). Use the SMF 29 record to find Java-related data collected in an IMS V14 environment. Use the SMFINTERVAL parameter for the DFSJVMEV member or IMS procedures to specify the time interval in milliseconds to log JVM statistics.

SMF record type 121 covers the JZOS-based jobs, recording z/OS Java runtime performance statistics.

Values captured in the record are used by IBM and various third party vendor products for near-real-time analysis as well as reporting purposes.

8.5. Security considerations

All accesses to z/OS UNIX System Services files should be controlled via access control lists (ACLs) for security purposes. It is critical that you:

- Identify the roles that you need. Define those as RACF groups.
- Implement the adequate ACLs for the defined groups.
- Connect the RACF user to the adequate group(s).

8.5.1. Roles classification

When identifying user roles, you must consider all environments, from development, test, integration test, pre-production, to production. The following roles were identified in our environment:

1. Application development and testing. This role is for application development and testing.

2. Application deployment. This role is responsible for deploying the Java software and applications in the adequate environments

3. Application execution. This user role needs IMS region RACF USERID(s) in order to run Java applications.

4. Operational control. This role is responsible for operational tasks, such as starting IMS MPRs and controlling operations, and usually provides the first-level support.

5. Second- and third-level support. This role has the ability to access for a short time (a few minutes to a few hours) the production environment for problem determination purposes.

A prerequisite for the implementation of adequate role-based security mechanism is an adequate structure of your file system and path naming conventions.

8.5.2. Separation of concerns

A prerequisite for the implementation of adequate role-based security mechanism is an adequate structure of your file system and path naming conventions. Having separate directories and paths with a naming convention that clearly indicates the user roles not only facilitates access control, but also reduces confusion and eases support.

8.5.3. Transaction execution security

All transactions are executed under the USERID(s) set by the IMS Connect client. In our environment, the IMS OTMA security level is set to CHECK, which means for access to the requested transaction, the user ID is verified and checked against RACF for authorization in the control region.

Another thing to note is that both the USERID under which the MPRs are started and the USERID used for the transaction execution need an OMVS segment. Previously the USERID for transaction execution must have write access to the home file system of the USERID under which the region was started. With IMS V14 APAR PI64496, this requirement is lifted.

8.5.4. Accessing Db2

All access to Db2 is done with the technical user ID that is assigned to the IMS transaction by the IMS Connect client. Those users have to be granted on all related resources (views, tables, and so on).

8.6. Summary

We introduced Java in IMS to accelerate the creation of new services and extend the life and value of existing applications, with the goal to evolve from a COBOL-based system to a Java-based system. We set out on the modernization journey by identifying our strategies, carefully examining our existing infrastructure and environments, and clearly identifying our functional requirements. Through this project, we learned how to embed Java in classic IMS regions, what environment setting changes are involved, how Java and COBOL differences can be handled, what security issues should be considered, and how performance and processing resources can be monitored.

Notices

COPYRIGHT LICENSE

Trademarks