Development and Debugging Java CICS Applications

Dennis Weiand
IBM

Thursday, March 3, 2011, 11:00-12:00
Session #8274

Abstract

- There's never been a better time to take advantage of Java technology in CICS. Learn about the latest and greatest Java features being incorporated into CICS, including Java 6, multi-threaded JVMs, and usability enhancements to help provide a smoother Java experience. Discover how to configure CICS Java support and debug your live Java programs running in CICS TS using the Rational Developer for System z (RDz).

Trademarks

- The following terms are trademarks of the International Business Machines Corporation or/and Lotus Development Corporation in the United States, other countries, or both:
  - Redbooks(logo)™, AIX®, alphaWorks®, CICS®, DB2®, IBM®, IMS™, Informix®, MQSeries®, VisualAge®, WebSphere®
- The following terms are trademarks of other companies:
  - Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation.
  - Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc.
  - CORBA, CORBAServices, and IIOP are trademarks of the Object Management Group, Inc.
  - UNIX is a registered trademark of The Open Group in the United States and other countries.
  - Other company, product, and service names may be trademarks or service marks of others.

SHARE in Anaheim 2011

Notices

- This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this presentation in other countries.
- INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PRESENTATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILTY OR FITNESS FOR A PARTICULAR PURPOSE.
- This information could include technical inaccuracies or typographical errors. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this presentation at any time without notice.
- Any references in this presentation to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

SHARE in Anaheim 2011

Agenda

- A bit of history
- Types of Java applications
- JCICS classes
- Java infrastructure
- Debugging
- JVMServer
- A few words on tuning

Notes:

- This session will be a quick tour of some of the Java-CICS related issues when running Java in the CICS environment.
- We only have an hour so we won't be able to spend much time on any one area.
- The intent of this session is really just to give you an idea of some of the areas surrounding the use of Java applications under the control of CICS. Being armed with some of the base concepts will give you a better understanding of what areas you want to research further as you proceed down the path of using Java in CICS.
- Your primary resource for information related to the use of Java in CICS is the CICS InfoCenter. It provides in-depth, up-to-date information on all aspects of Java in CICS. Additionally, it has several references to related information.
- Your next source of information for Java infrastructure issues will be the many articles and IBM documentation related to the Java Virtual Machine (JVM). While –all- JVMs (that want to claim they support Java) must provide a common environment to the Java programs that execute in that JVM, how the JVM handles this task is up to the vendor that supplies the JVM. This is how the JVM vendor can differentiate themselves in the Java area. Since CICS requires the use of the IBM-supplied JVM, areas such as garbage collection (GC) policies, just-in-time (JIT) compilation will be IBM-specific, and IBM documentation must be accessed for options to control these areas.
- While this presentation does list CICS-specific areas that a Java programmer will need to know when writing Java applications that run in a CICS environment, this presentation does not address any coding of the Java language itself. Many resources are available for learning the Java language, plus when faces with a Java programming language-related task, the Internet is a great source of ideas and coding examples.

History of Java in CICS

- **CICS TS V1.3 (1998)**
  - Java programs in Java Virtual Machine (JVM)
  - IIOP request inbound to Java programs
  - Basic CICS API available with JCICS classes
- **CICS TS V2.1 (2001)**
  - EJBs (Session Beans)
  - Java programming enhancements
- **CICS TS 2.2 (2002)**
  - EJB and Java programming enhancements
  - Better Java and EJB statistics
  - More CICS API supported in JCICS classes
- **CICS TS 2.3 (2003)**
  - CCI support and Java performance enhancements
  - CICS Web Support added to JCICS classes
- **CICS TS 3.1 (2005)**
  - Web Services Support, plus more CICS API in JCICS classes
  - Channels and Containers - alternative to 32K COMMAREA restriction
- **CICS TS 3.2 (2007)**
  - Support for Java 5, Configuration file changes
  - Continuous JVM
  - IPIC Communications (allows channels and containers with CICS TG V7.1 and above)
- **CICS TS 4.1 (2009)**
  - Support for Java 6
  - JVMSERVER resource added
  - Can use Java in Dynamic Scripting applications (2010)

Notes:

- Java is the name of the 5th largest island in Indonesia. Its capital city is Jakarta. As one of the most densely populated places in the world, it has a population of over 130 million people. The island of Java has a long history of growing coffee and Java coffee is one of Java's products. In the USA, the term 'Java' is commonly used as slang for coffee in general.

- Java is also the name of a programming language developed by James Gosling at Sun Microsystems (now Oracle Corporation). Although the language's roots are as early as 1991, when it was intended to be imbedded in 'set-top' box for use with interactive television, Sun Microsystems released Java 1.0 in 1995 with the promise of 'write once, run everywhere'. The 'write once, run everywhere' is achieved by a compilation of the Java program code into 'bytecode' which runs inside a Java Virtual Machine (JVM). The JVM provides the interface between the bytecode and the hardware/operating system on which the JVM runs. While interpretation of the bytecode must be the same in any JVM, multiple vendors have written their own JVM, so tuning of the JVM often becomes vendor-specific.

- CICS started supporting Java as a programming language in 1998 with the release of CICS TS V1.3

- The Enterprise JavaBean (EJB) specification, originally developed by IBM around 1997, was adopted by Sun Microsystems and made available as EJB 1.0 around 1999. CICS provided support for EJBs in 2001 with the release of CICS TS V2.1 in 2001. Although EJBs are great for some applications, CICS provides many ways to communicate from an EJB environment outside of CICS to application programs running in a CICS environment, so use of EJBs in CICS hasn't become as popular as originally expected. IBM, during the announcement of CICS TS V4.1, announced that the current level of EJB support in CICS (EJB V1.1, session beans only) would not be improved.

- CICS has continued to support current levels of the JVM available on z/OS, with Java 6 being the most current supported Java level.

Where Can Java be used?

- **Regular CICS program (JCICS applications)**
  - Initial program of a transaction
    - Started by a user at terminal or EXEC CICS START
  - Target of an EXEC CICS LINK
    - Target of EXCI call or DPL including ECI
  - Target of a Web Service request
  - Pipeline Handler
  - Target of a REST request
  - Target of EXEC CICS XCTL
  - Program named in EXEC CICS HANDLE ABEND command
  - Initialization program or shutdown program
  - Some User-Replaceable-Modules (URM)
- **Dynamic Scripting applications**
- **Inbound IIOP stateless CORBA objects**
- **Enterprise JavaBeans (EJB V1.1-Sessionbeans - stabilized)**

Notes:

- Java can be used in many places in CICS. Predominately, its use is as a 'regular' CICS program and receives its input via a COMMAREA, or channels and containers (as of CICS TS V3.1).
- Although Java includes classes to allow communications and other environment interactions, one of the purposes of having a CICS environment is to have CICS control security, transactionality, and other aspects of your application environment. Because of this, Java programs, as with all programming languages used in the CICS environment, should interact with CICS instead of interacting directly with the operating system.
- The CICS-provided classes that a Java application programmer uses to have their Java program interact with CICS are commonly referred to as the JCICS classes. Care should be taken as to the design of Java programs that run under CICS. Classes that interact with CICS can be grouped such that the main body of the Java application is still 'write once, run anywhere'.
- Support for EJBs in CICS has been stabilized at the EJB V1.1, session-bean only, level. Before EJBs came on the scene, CORBA objects were available. CORBA objects communicate using RMI/IIOP (Remote Method Invocation over Internet Inter-orb Protocol). The RMI/IIOP is also the communications technique used between EJBs (in any environment).
- The addition of the CICS Dynamic Scripting Feature Pack has added an additional way to add Java to your CICS environment.

Java in CICS: Application Issues

- **How Programs get started**
- **How your program is passed Data**
  - COMMAREA or channel and containers
- **Interacting with the environment**
  - DFHEIB, ASSIGN
- **Interacting with CICS resources**
  - TD, TSQs, Files, Programs, channels and containers, Start
  - Passing Data to CICS
  - Getting Data back from CICS
- **Exception Handling**
  - RESP, RESP2

The Java programmer needs to understand these issues to function well in the CICS environment.

SHARE in Anaheim 2011

11

Notes:

- There are a few application-related issues that the Java application programmer needs to be aware of when writing Java programs that will run in a CICS environment. Your Java program, most of the time, will interact with the CICS environment, so these application-related issues address Java/CICS interaction. To be a successful Java programmer in the CICS environment, you will also need a basic understanding of the JCICS classes and how to use them to interact with your environment.

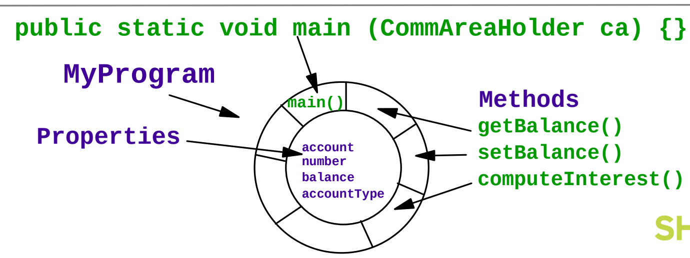
- These issues are listed on the slide and each issue will be discussed on upcoming slides.

SHARE in Anaheim 2011

12

CICS Java “Program” Start

- CICS Java “Program”
  - Start at 'public static void main(CommAreaHolder ca) ‘
    - COMMAREA is accessible in a CommAreaHolder
  - Start at 'public static void main(String[ ] args) ‘
- CICS IIOP programs start at specified method
- EJBs start at specified method
  - EJB support in CICS is stabilized
- CICS Dynamic Scripting
  - Instantiate appropriate class (constructor is invoked)
  - Execution starts at specified method



13

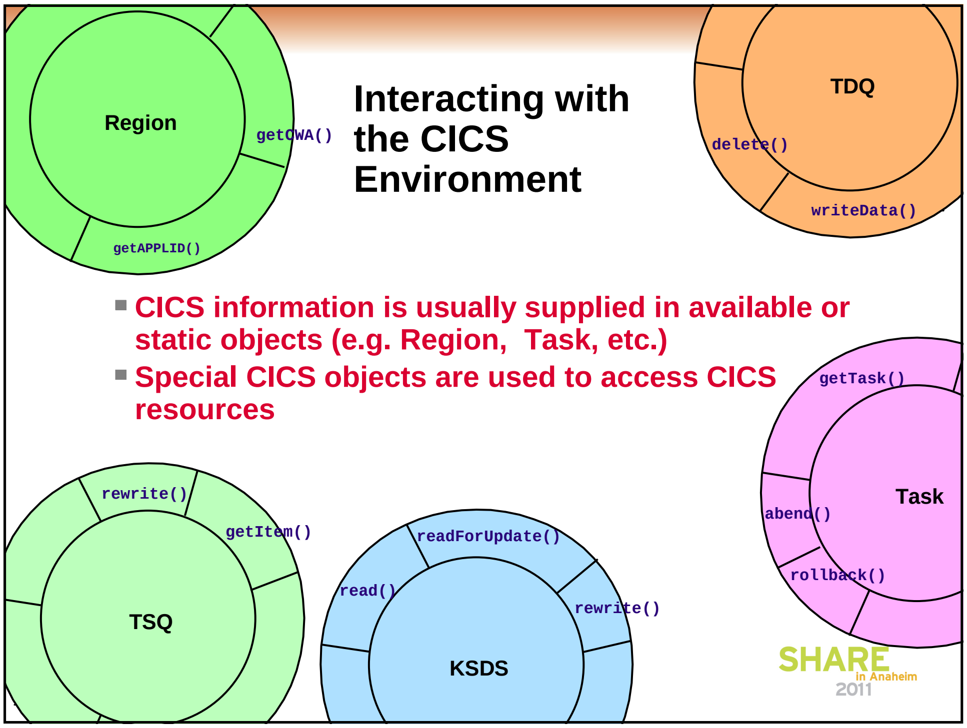
Notes:

- Java programs running in the CICS Dynamic Scripting environment are run in response to events that occur in that environment. From a logical perspective, this is the same way that any of your code (Java, PHP, or Groovy) is triggered to handle the various events in the Dynamic Scripting environment.
- For EJBs and CORBA objects, execution begins at the requested method.
- For “regular” Java programs in CICS, Java programs are started similarly to other “regular” programs in CICS (the use of the word “regular” was discussed on a previous slide).
- CICS is aware of ‘regular’ programs because of PROGRAM definitions. In the case of a Java program definition, the program is flagged as needing to run in a JVM, with execution starting in the specified class.
- Once the specified class is loaded into a JVM, CICS will look for a method signature in that class of “public static void main (CommAreaHolder ca)”. A method of this type will allow CICS to pass the method a COMMAREA which is a typical type of communications to other ‘regular’ CICS programs written in other languages. If the above method signature is not found, CICS will look for a method signature of “public static void main (String[ ] args)”. This is the normal method of invocation for a J2SE (Java Standard Edition) program. If invoked with the second signature, the application program will always receive an array of String objects whose length is 0 (zero).
- The ‘CommAreaHolder’ mentioned in the previous bullet is discussed on an upcoming slide.

14

Interacting with the CICS Environment

- CICS information is usually supplied in available or static objects (e.g. Region, Task, etc.)
- Special CICS objects are used to access CICS resources



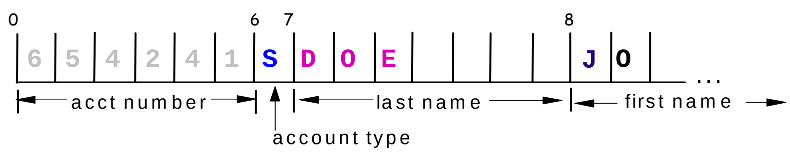
Notes:

- You interact with CICS from your Java program using a set of supplied objects that are specific to the CICS environment.
- There are static objects that let you refer to information that is general to the CICS environment. There are other objects that let you interact with a CICS resource.
- An example of a a static object is CICS's 'Region' object. Using this object, an application program could access items such as the application identifier of the CICS region, or the CWA (Common Work Area) that is available to all applications that run in CICS.
- An example of accessing the CICS region's application identifier is Region.getAPPLID() which would return a String with the region's application identifier.
- There is also a static Task object. The static Task object can be used to get a reference to details about the specific task (instance) of the currently running transaction. The way to get a reference the details of the currently running task is "Task myTask = Task.getTask();" Once you have a reference to your task-specific information, you can use that reference to access containers in the current channel (the channel that was passed to you) and also influence the outcome of the task, such as 'myTask.abend("MYAB");'.
- There are also objects that represent individual CICS resources such as VSAM files, Transient Data queues, and Temporary Storage queues.

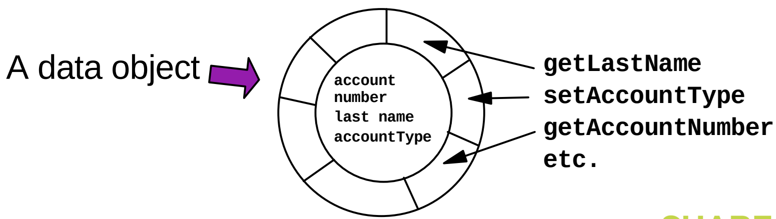
SHARE in Anaheim 2011

Data Areas

- **CICS COMMAREA**



- **Java**

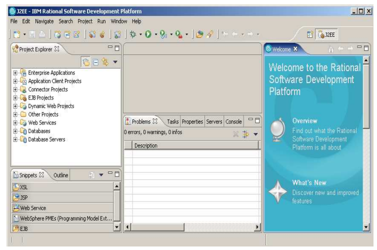


Notes:

- CICS's primary heritage has CICS providing support for non-object-oriented programming languages such as C, COBOL, PLI, and IBM Assembler.
- The non-object-oriented programming languages like COBOL use a series of characters that represent field-oriented information. The corresponding Java object representation is byte array. To bridge the gap between the normal getters/setters available to Java programs to access variable values and field references in other CICS languages, you can use some of the options listed on the following page.

Data Areas – series of bytes to Java object

- **Generate Object (Recommended)**
  - Rational Application Developer (and RDz)
  - JZOS
- **Write generic Object classes**
  - Your code
- **Add code as needed**
  - Concatenate
  - Substring
  - Indexing into byte array
  - bytestream (see supplied CICS Samples)



Notes:

- One option to access the fields in a typical CICS-oriented data structure from your Java program is to use various Java methods to index into the byte array, or convert the byte array to a String and use the substring() method to access 'fields' in the data. Once you have the field data, you could convert the data to a data type you can work with such as integer, float, or BigDecimal. Some representations on z/OS, such as packed decimal do not have corresponding data types in Java so you would have to convert the bytes to a datatype usable in Java. Likewise, when preparing data to be given to CICS, you would need to take the data from its current form and place it in a byte array before giving the data to CICS.

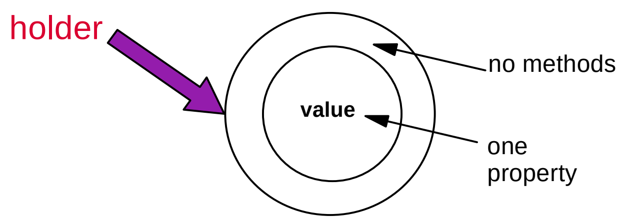
- Another option is to create a Java class dedicated to handling your data layout, commonly referred to as a data object. The data object would have getters and setters corresponding to the data in the record layout (or whatever you are working with), and would also have something like a getBytes() method to get the data as a series of bytes (a byte array) to be given to CICS.

- An easier way to approach this is to use the wizards available in Rational Application Developer (RAD) and Rational Developer for System z (RDz) to generate data objects for you. You can have a COBOL program as input to the CICS Java Data Binding wizard, select the data structure you want to work with, and have RAD/RDz generate a data object. The data object will have a getter and setter for each 'field' in the data structure, plus a getBytes() method for accessing the data as a byte array and a setBytes() methods for placing a byte array into the data object. The generated data object also understands data types such as packed decimal and will convert the information to/from a similar data type in Java.

- The z/OS Java implementation also comes with a set of classes referred to as the JZOS classes, which can be used for z/OS-specific activities like accessing a PDS. These JZOS classes can also generate data objects. You would compile your COBOL program using the ADATA compilation option which provides a description of the data areas in your program. You can use the ADATA data as input to JZOS classes that will generate the data object.

Holders: Getting Data from CICS

- **Byte arrays** are received from CICS in "Holders“
- Give CICS a **holder object**, CICS places the data in the holder
- Holders (except for one) have a single data property called "**value**" of type byte array
- The exception is the RetrievedDataHolder which contains additional data
- Examples: CommAreaHolder, ItemHolder, DataHolder, RecordHolder



Notes:

- Data to be given to CICS must be in a byte array. When you want information from CICS, you first create a ‘holder’ object. You give this holder object to CICS, and CICS will return the data you request in the holder object.
- There are various holder objects depending on the CICS resource you are accessing. The holder objects are similar in that they have a property, of type byte array, with the name ‘value’. Most holder objects only have the ‘value’ property. The RetrievedDataHolder, which is used to access information passed with a START command has several more fields.
- An example: If you wanted to access a record in a VSAM file, you would first create a RecordHolder. You would then, on a KSDS object, using a method called read(), pass the holder to CICS. The data from the VSAM file would be returned to you as a byte array. You would access this byte array in the ‘value’ property of the RecordHolder object.

CICS Java Exception Inheritance

SHARE Technology • Connections • Results

You 'try' a JCICS method, then 'catch' exceptions that correspond to CICS response codes

```
java.lang.
Object
 └► Throwable
     └► Error
         └► CicsError (**)
     └► RuntimeException
         └► CicsRuntimeException (**)
     └► Exception
         └► CicsException
             └► CicsConditionException
                 ├► NoSpaceException
                 ├► DuplicateRecordException
                 ├► InvalidProgramIdException
                 ├► NotAuthorisedException
                 ├► etc.
                 └► CicsResponseConditionException
                     ├► IOErrorException
                     ├► LengthErrorException
                     └► etc.
```

CICS exceptions are in com.ibm.cics.server.*

Examples of CICS responses, like **NOSPACE, DUPREC, PGMIDERR, NOTAUTH, IOERR, LENGERR**

(**)= More exceptions grouped under this exception

SHARE in Anaheim 2011

Notes:

SHARE Technology • Connections • Results

- An indication as to the completion status of a CICS command is returned to your application program in the EIB (Execute Interface Block) and also as 'resp codes' available to your program.
- There are Java exceptions for all of the conditions that can be returned from a CICS command. These exception extend the normal Throwable and Exception objects in Java, so normal Java exception processing can be used when interfacing with CICS.
- For example: Perhaps you have tried to read a VSAM record for a customer that doesn't exist. In your Java program you would 'try' the read operation, and 'catch' a RecordNotFoundException.
- See later slide for a coding example of using a try / catch in Java.

SHARE in Anaheim 2011

Java CICS Resource Access

- **CICS resources:** files, programs, temporary storage, transient data, etc. are represented by objects of the appropriate type
- Use **setName()** to specify resource name
  - set other characteristics as appropriate
- Invoke **method** specific to desired action
- Data to and from CICS is in **byte arrays**
- The byte arrays **from** CICS are in "**Holders**"

Notes:

- When accessing a CICS resource like a VSAM file, Temporary Storage, Transient Data, and Programs, you always follow a specific pattern.
- To access a CICS resource, you:
  - Instantiate an object of the type of resource you will access (e.g. a TSQ object if you want to access a Temporary Storage queue).
  - Set the name on the object to the name of the resource in CICS (using the setName() method)
  - Invoke a method on the object that indicates the desired action
  - You always pass data to CICS in byte arrays (e.g. the contents of a record to be written to a VSAM file)
  - You receive data from CICS by passing CICS a holder, into which CICS will place a byte array
- As with the traditional EXEC CICS commands, things don't always go as you would like. In Java, CICS returns conditions to your Java program as Exceptions. There are Exceptions that represent all of the exceptional conditions that can be returned by CICS.

CICS TSQ Read: Java

```
TSQ myTSQ = new TSQ();                    // Create TSQ object
myTSQ.setName("DENNIS");                  // Name of the Queue
ItemHolder myItem = new ItemHolder(); // Holder for data
int k;
try {
  for (k = 1; k < 6; k++) {
    myTSQ.readItem(k, myItem);
    System.out.println("Item contents: " +
      myItem.value.toString());
  }
} catch (ItemErrorException i) {
  System.out.println("Item " + k + "does not exist");
} catch (CicsConditionException c) {
  System.out.println("Unexpected condition: " + t);
}
```

- Use CICS-supplied Javadoc or your IDE's "code assist"
- IDE – for example RAD, RDz, or Eclipse

SHARE in Anaheim 2011

Notes:

SHARE Technology • Connections • Results

- This slide illustrates most of the concepts we have talked about.
- For this example code segment we will access information in a Temporary Storage queue.
- We first instantiate an object of type TSQ to give us an object representation of the Temporary Storage queue.
- We use the setName() method to indicate the name of the TS queue is "DENNIS". This is the name by which CICS knows this queue.
- Since we will be receiving data from a TSQ in CICS, we instantiate an object of type ItemHolder.
- To read a specific TSQ item, we use the readItem() method of the TSQ object, and also pass the ItemHolder object to CICS. If the contents of a TSQ item is successfully obtained, the contents of that TSQ item will be returned in the ItemHolder as a byte array.
- Notice that we have the readItem() method in a try/catch block. If the readItem() method is unsuccessful, the method will 'throw' an exception. The exception that is thrown will indicate the reason for failure (which could be as simple as having no more items in the queue).
- The first 'catch' block will be executed if the returned exception is an ItemErrorException (the indicated item doesn't exist).
- The 'catch' block for Throwable will be executed if any other exceptions are thrown from the readItem() method (since all exceptions inherit from Throwable).

SHARE in Anaheim 2011

JCICS Classes

- **BMS and Terminal Control**
  - converse(), receive(), send(), sendControl(), sendText()
  - **no** SEND MAP, RECEIVE MAP, HANDLE AID or WAIT TERMINAL
- **Document API**
- **Common equivalents of** ASSIGN, ADDRESS, INQUIRE
- **FILE** Control, including BROWSE
- **LINK** and **XCTL** (no SUSPEND)
- **CANCEL**, **RETRIEVE**, **START**
- **Temp Storage and Transient Data**
- **ENQUEUE, DEQUEUE**
- **APPC mapped** conversations
- **Channels and Containers**
- **TRACE**
- **SYNCPOINT, ROLLBACK**
- **WEB, EXTRACT** (for HTTP requests and responses)
  - HTTPSession, HTTPRequest, HTTPResponse
  - **No** Support for Servlet API

SHARE in Anaheim 2011

29

Notes:

- This slide lists the CICS resources that a Java program using the JCICS classes can interact with.
- See the CICS InfoCenter for:
  - More details on what you can and cannot do
  - The various JCICS classes you would use to interact with the CICS resource
  - The various methods on the JCICS classes that take action on the resource
  - The exceptions that the methods can throw
- Java programs in CICS can interact with Web browser, provide responses to RESTful requests, and respond to most any HTTP request. However, the Java program would interact with CICS to get the input data, HTTP headers, and to send responses. Although CICS has an HTTPSession object, and HTTPRequest object, and an HTTPResponse object, these objects are different from those objects with a similar name that are part of the Servlet specification. CICS does not support the Servlet specification.

SHARE in Anaheim 2011

30

JCICS **Unsupported** Areas

- **BMS Send Map and Receive Map**
- **APPC unmapped conversations**
- **CICS Business Transaction Services**
- **DUMP services**
- **Journal services**
- **Storage services** (No GETMAIN – use normal Java storage management)
- **Timer services**
- **System Programmer Interface** (INQUIRE/SET/PERFORM, etc)

SHARE in Anaheim 2011

Notes:

- This slide lists the CICS resources that are not available to programs written in Java.
- Again, you can find more information on what resources or facilities can be accessed from a Java program and those resources and facilities that cannot.

SHARE in Anaheim 2011

© 2011 IBM Corporation

Your Java program in CICS can use…

- **JCICS**
  - Class library for access to CICS resources
- **WMQ API**
- **Not JMS**
- **Threads**
  - Can use Threads (not recommended), but all API calls must be from initial thread, stop Threads when done
- **JDBC 2.0 -** Includes SQLJ
- **JDBC access to IMS DB -** Requires IMS V7
- **SSL V3 and TLS –** Use CICS's support
- **XML4J or XERCES** classes and **other** APIs that Java does well

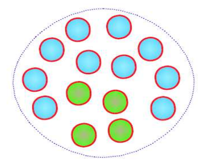
Notes:

- In addition to the JCICS classes, you can also use the MQ Java API (but not JMS).
- Although you can technically create threads in your Java program, it is highly discouraged.
- You can use JDBC and SQLJ to access DB2 data.
- You can use JDBC to access IMS data.
- Although Java itself contains SSL support, you should use CICS's support of SSL so that CICS can control the environment.
- You can also use other classes for activities that Java does well such as parsing XML.

Java in CICS TS - Traditional

- **Modes:** Single-use (REUSE=NO) or Continuous (REUSE=YES)
- CICS creates **pools** of JVMs and allocates JVM to application as needed
- JVMs run in CICS OTE (Open Transaction Environment) - their **own TCBs**

- **CICS TS V3.1**
  - SDK 1.4.2 – zAAP eligible – 31-bit mode
  - Can also use 'resetable' mode (REUSE=RESET)



- **CICS TS V3.2**
  - Java 5 or SDK 1.4.2 – zAAP eligible – 31-bit mode
  - Improved garbage collection
  - Better statistics and monitoring data

The steps to implement Java in CICS are well documented in the CICS InfoCenter

- **CICS TS V4.1**
  - Java 6 – zAAP eligible – 31-bit only
  - Added new **JVMSERVER** resource – Multi-threaded JVM

SHARE in Anaheim 2011

Notes:

- The Java environment is implemented for 'regular' Java programs by providing a pool of JVMs.
- When CICS receives a request to run a Java program, CICS selects a suitable JVM from the pool.
- If there are no available JVMs in the pool and we haven't reached the maximum size of the pool, CICS will create another JVM in the pool.
- CICS will only dispatch a single program to one of the JVMs in the pool at time. If there were 5 JVMs in the pool, CICS could only process 5 requests to run Java programs at the same time.
- You can run a JVM in the pool either in single-use mode (REUSE=NO) or in continuous mode (REUSE=YES). For single-use mode (which is sometimes handy for testing), CICS will create a JVM, have the JVM run the requested Java program, then destroy the JVM. In continuous mode, CICS has the JVM run the requested Java program, but doesn't destroy the JVM after the Java program has run.
- The exception to the above bullet is that when using CICS TS V3.1 and Java 1.4.2, you can use REUSE=RESET. This option, when selected, causes CICS to request a partial reset of the JVM. While in theory resetting only the application portion of the JVM seemed like a great way to have program isolation while lowering the cost of creating and destroying a JVM, in practice it is very difficult to write a program that only touches the application portion of the JVM. If an application touched more than the application portion of the JVM (which is the usual case), CICS destroys the JVM, causing it, in most cases, to work like REUSE=YES.
- With CICS TS V4.1 and the new JVMServer resource used by CICS Dynamic Scripting, CICS now has a multi-threaded JVM. This is explained more in later slides.

SHARE in Anaheim 2011

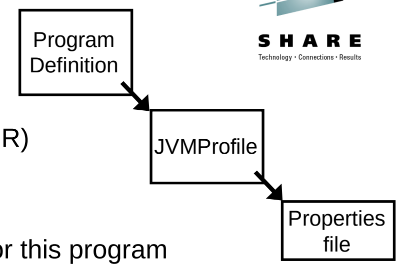
Java Infrastructure in CICS

- **Number** of JVMs in the pool
  - Limited by SIT parm MAXJVMTCBS, and available region space
- **Selection** of JVM from pool based on JVMProfile and key (user/CICS)
- **Space** for JVM is from CICS region (only have room for a finite number)
- **Characteristics** of JVM set in the Program Definition, JVM profile, and properties file
- **GC** as of V3.2 runs when heap reaches a threashold; separate task; CJGC
- Can specify **idle timeout** after which JVMs are removed from the pool
- If no JVMs available based on profile/key, a JVM will be **created or 'stolen'**
- JVMs are used 'by program', a single CICS task could use multiple JVMs
  - Go from Java program to Java program using traditional Java techniques (not CICS's LINK) to use least number of JVMs.
- Can **control the Pool** from SPI, CEMT, CICS Explorer, CPSM

Notes:

- CICS uses the open transaction environment (OTE) to run JVMs. Each JVM runs on a separate z/OS TCB, which is allocated from a pool of J8- and J9-mode Open TCBs, managed by CICS in the CICS address space. This pool of Open TCBs is called the JVM Pool. The priority of the J8- and J9-mode Open TCBs in the JVM pool is set lower than that of the main CICS QR TCB, to ensure that J8- and J9-mode activity does not affect the main CICS workload that is being processed on the CICS QR TCB.
- The CICS-JVM interface matches the EXECKEY of the JVM and its JVM Profile when selecting a JVM to use for a new program.
- CEMT and SPI commands are provided to inquire on the attributes of the JVMPOOL. Since CICS TS 2.3, it is also possible to inquire on the attributes of an individual JVM within the pool. SET commands are provided for the JVMPOOL to manipulate the pool as a whole.
- The PERFORM JVMPOOL command, introduced in CICS TS 3.2, enables starting a number of JVMs ahead of them being required, and terminating subsets of JVMs in the pool.
- Prior to TS 3.2 CICS would issue explicit Garbage Collection (GC) requests every 101 JVM uses. This was implemented to reduce the likelihood of GC occurring mid-transaction, thereby improving the average response time for EJBs. However, it never worked very well for regular Java applications. Regular Java applications saw both the CPU cost and the response time hit.
- In TS 3.2 CICS has been changed to issue explicit GC requests if the application heap has an occupancy rate greater than 85% at the end of a single use of that JVM. This GC is done in a separate system task. This has the advantage (when used properly) of keeping consistent CPU costs and response times for the applications, and gives greater ability to measure the GC costs.
- You can opt-out of explicit GC events entirely and just allow the JVM to do GC as and when it is required. This will give you better CPU costs overall, but may result in erratic CPU usage and response times for individual tasks.

CICS Java Support



- **Program Definition**
  - Specifies execution class
  - Specifies key to run in (CICS or USER)
  - Points to JVMProfile file
- **JVMProfile (HFS file)**
  - Characteristics of the JVM needed for this program
  - Specifies application classpath and libpath info
  - Specifies JVM Mode (single-use, continuous)
  - Standard Java options (e.g. -Xmx=32M – size of transient and middleware heaps for persistent reusable JVM)
  - Points to properties file (optional)
  - Documented in CICS System Definition Guide
- **Properties File (HFS file)**
  - EJB properties
    - JNDI Provider, JDBC driver, J2EE security

Notes:

- A sample program definition and JVMProfile file are on the upcoming Slides
- The JVM properties file probably won't be needed (you can specify properties using the normal -Dname=value convention used on all other platforms within the JVMProfile).
- See the CICS InfoCenter for more information on the program definition, JVMProfile file, and properties file.

CICS Program Definition

```
OBJECT CHARACTERISTICS
  CEDA  View PROGram( SAMPLE02 )
   REMOTEName     :
   Transid        :
   EXECUtionset   : Fullapi   Fullapi | Dplsubset
  JVM ATTRIBUTES
   JVM            : Yes                  No | Yes
   JVMClass       : com.ibm.cics.example.sample02
   (Mixed case)   :
                  :
                  :
                  :
   JVMProfile     : DFHJVMPR
   (Mixed Case)
  JAVA PROGRAM OBJECT ATTRIBUTES
   Hotpool        : No                   No | Yes
```


Notes:

- The JVM and JVMCLASS keywords on the program definition were added in CICS TS 1.3 and remain the same.
- The JVMPROFILE keyword was added in CICS TS 2.1.
- In CICS TS 2.3 the CEDA panel was changed to accept mixed case input for JVMCLASS and JVMPROFILE irrespective of the upper case translation setting for the terminal.
- HOTPOOL is no longer supported.

JVMProfile

- **Continuous JVM**
- **Properties file pointer**
- **LIBPATH**
- **CLASSPATH**
- **JVM options**
- **No shared Class cache**
- **Example shown is for V3.2 and V4.1**

```
####################################################
# Selected parts of a JVMProfile
####################################################
CICS_HOME=/usr/lpp/cicsts/cicsts32
JAVA_HOME=/usr/lpp/java/J1.5
WORK_DIR=/tmp/cicsaor2/work.
REUSE=YES
CLASSCACHE=NO
STDIN=dfhjvmin
STDOUT=dfhjvmout
STDERR=dfhjvmerr
GC_HEAP_THRESHOLD=85
IDLE_TIMEOUT=30
# Can specify LIBPATH_PREFIX= and LIBPATH_SUFFIX=
# Can specify CLASSPATH_PREFIX= and CLASSPATH_SUFFIX=
-Xms16M
-Xmx32M
-Xoss4M
-Xss512K
```


Notes:

- The HFS directory that contains the JVMPROFILE is specified via a new SIT parameter JVMPROFILEDIR.
- JVMPROFILE parameters were added in CICS TS 3.2. IDLE_TIMEOUT (if a JVM is idle for this amount of time it is removed from the pool), and GC_HEAP_THRESHOLD (the heap size after which CICS will run the CJGC transaction to perform Garbage Collection).
- JVM options use the standard conventions from all other platforms (eg -Xmx32M rather than Xmx=32M). CICS doesn't validate parameters beginning with '-' characters, it just passes them through to the JVM unchanged. Therefore any new or undocumented JVM parameters are automatically supported in CICS (this makes problem determination much easier).
- CICS can now build LIBPATH itself in most cases.
- The Profile parameter JVMPROPS specifies an optional system properties file in HFS
- You will probably not need the JVMPROPS file since you can specify properties using the normal -Dname=value convention in the JVMProfile file.
- There are many new usability tweaks such as adding the JVMProfile name to most SJ domain messages and improved error detection for configuration problems that resulted in PMRs in the past.
- EXEC CICS INQUIRE JVMPROFILE - returns HFS path name, shared classcache and REUSE values.
- JVM Profile parameters are documented in the CICS System Definition Guide.

Debugging / Profiling

- **Editors** – RDz – RAD – Eclipse
- **CEDF / CEDX**
- **CICS messages and abends**
- **stdout and stderr**
- **CICS Trace**
- **Interactive debugger -** JPDA – works with any Java debugging tool
- **Profiling -** See JINSIGHT in developerWorks Web site
- **JVM method tracing**
  - ibm.dg.trc.print (set to 'mt' to invoke method trace)
  - ibm.dg.trc.methods (indicate what to trace)
- **Dumps** (Javadump Heapdump, CEEdump, Sysdump)
- **Infrastructure Tuning** - IBM Support Assistant – Heathcenter – others – more later

SHARE in Anaheim 2011

45

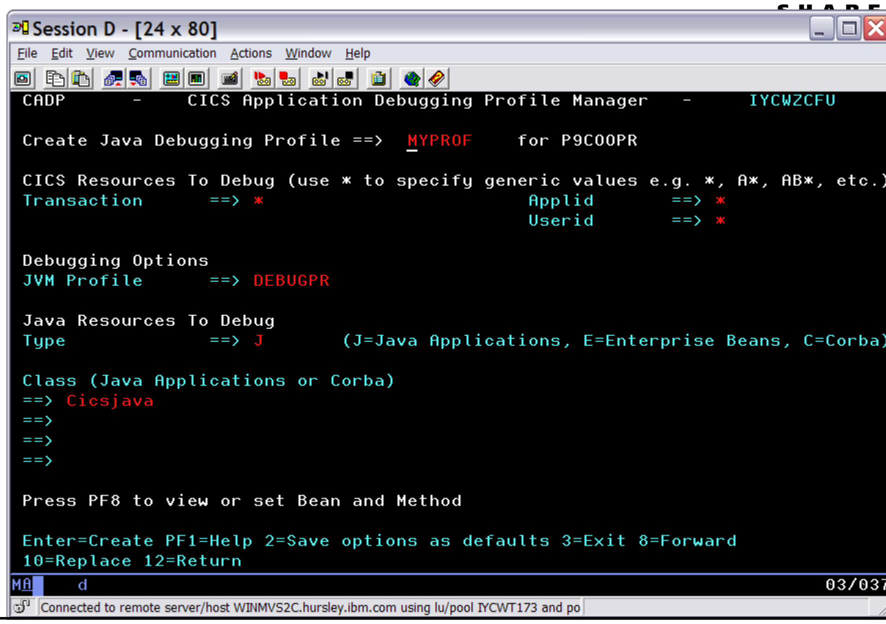
Notes:

- To a large degree your approach to Java application debugging will be the same as your approach to debugging CICS applications in other languages. There will, of course, be a few twists because of Java, plus their are some tools available to help you with your Java application that aren't available to other languages.
- There are Java editors that provide extensive syntax checking. IBM's RAD and RDz also do syntax checking as you type, provide code assist, provide several types of refactoring support, and provide suggestions to the problems the editor finds. Additionally, when you save the file, RAD/RDz compile the Java program. So unlike COBOL where the first real syntax check is done when you submit a batch job to compile your program, the Java editor gives you immediate feedback, assistance, and suggestions. This is also true if you are developing CICS programs on your workstation. You can download the dfjcics.jar file which contains all of the JCICS API, and after you save your program on your workstation, and transfer the compiled program to z/OS, you are ready to test your program in CICS. This eliminates batch compiles and because of the Java compiler's close type checking, eliminates common logic errors.
- If your Java program uses CICS API, you can walk through it with CEDF/CEDX, look for CICS messages and abend codes, and look for messages that Java or your application program writes to stdout and stderr.
- Source line debugging is available using RAD/RDz or any JPDA enabled debugger.
- Java method trace and CICS tracing are available.
- Profiling allows you to see your application programs performance characteristics.
- Several tools are available for infrastructure tuning assistance for garbage collection and other areas.

SHARE in Anaheim 2011

46

Java Debugging with CADP



Notes:

- You don't have to use the CADP transaction, but it will probably be the easiest approach if you want to easily turn debugging off and on.
- CADP is a CICS-supplied transaction that tells CICS when to start a debugging environment for your application program.
- If you want to use a source line debugger with your Java program, you can:
  - Specify a JVMProfile file for your Java program that tells the JVM to add JPDA debugging support
  - Use CADP, so when certain conditions exist, CICS will switch your program's JVMProfile to a different JVMProfile that enables debugging (using JPDA debugging support)
- This slide shows a CADP screen and how you might specify when you want source line debugging to start with your Java program.
- When the JVM is stated with the appropriate options (see next slide), the JVM used in CICS can be used with an JPDA enabled debugger (like the one included with RAD and RDz).

Debugging / Profiling

- Create a Debugging **JVMProfile**:

```
# DEBUG options
-Xdebug=YES
-Xrunjdwp:transport=dt_socket,address=9.20.221.168:8000
REUSE=NO
```

- The address is the host and port number of your debugging session on the workstation
- Debugging Profile is dynamically enable based on the userid in use and the Java class that is invoked (CADP)
- Can have JVM contact debugger, or start and wait for the debugger to contact the JVM

SHARE in Anaheim 2011

49

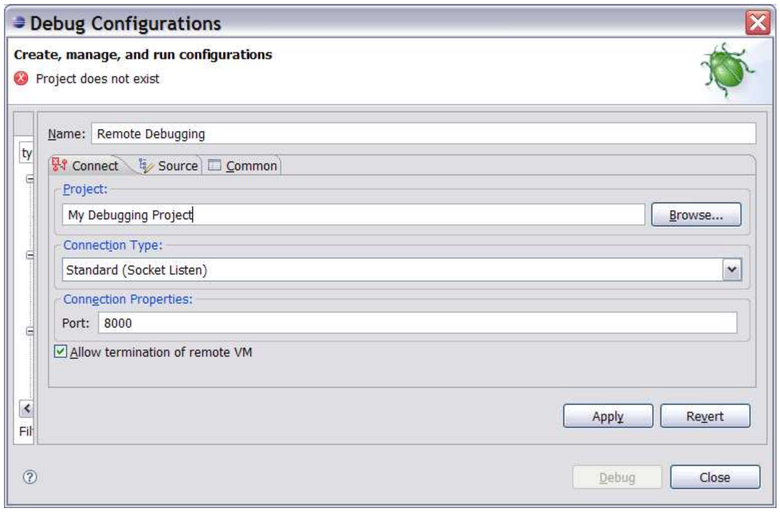
Notes:

- You don't have to debug using CADP, but it can make life a lot simpler. The setup for CADP involves creating three VSAM files and setting DEBUGTOOL=YES in the SIT. Further details are available in the CICS Information Center.
- There is a web based interface to CADP that can be used if you'd prefer not to work with 3270 terminals.
- You can cause CICS to use the debugging profile instead of the normal JVMProfile based on the Userid, Applid, Class name or transaction name that are in effect.
- If you want to set up a JVMProfile file with the parameters listed on this slide for debugging.
- You can let CADP switch your Java program to use your JVMProfile file your prepared for interactive debugging, or you can manually set your program's PROGRAM definition to point at your debugging JVMProfile file. Manually switching your JVMProfile file is prone to errors, so using CADP is likely to much easier.

SHARE in Anaheim 2011

50

Java Debugging with Eclipse



Create a debugging profile in Eclipse, remember to set the same port number as was specified in your debugging JVMProfile.

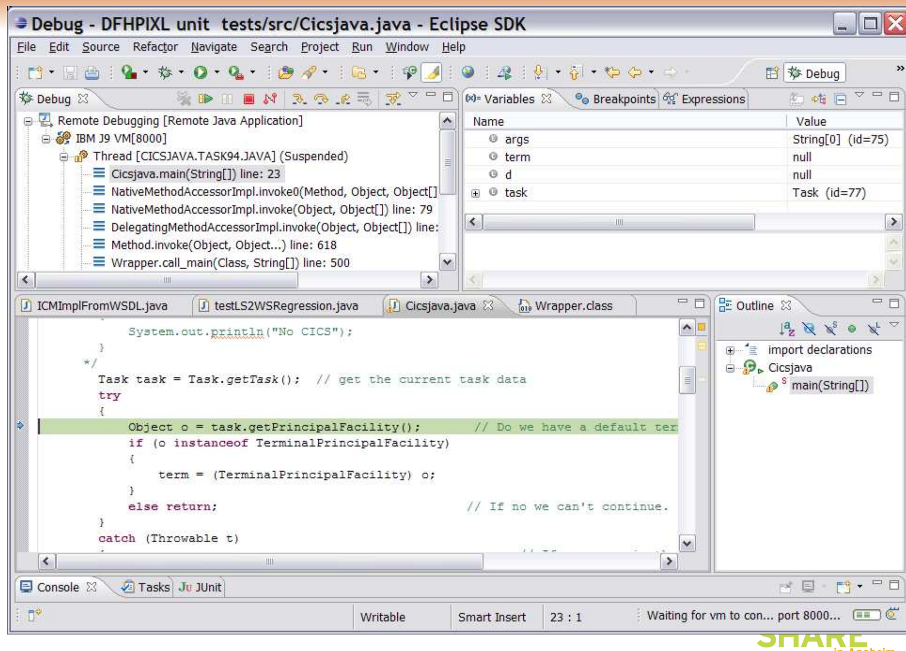
SHARE in Anaheim 2011

Notes:

SHARE Technology • Connections • Results

- In your JPDA debugger on your workstation you can take two different approaches:
  - You can have your debugger listen to be contacted by the JVM started in debugging mode to run your Java program
  - You can have your debugger contact the JVM that started and is waiting for your debugger to contact it
- The approach you take (have the JVM contact your debugger or having your debugger contact your JVM) will be determined on the statements you added to the JVMProfile file that specifies JPDA debugging for your program.

SHARE in Anaheim 2011



Notes:

- When the Java program executes in CICS a new JVM will be started that will attach to the debugging process on the work station.
- The Java code you will be debugging needs to be on your workstation.
- You will also need to set a breakpoint (on the slide notice the dot next to the source line in green). A breakpoint can be set by double clicking on the margin adjacent to line where you want interactive debugging to start.
- When a breakpoint it encountered, execution stops. You can see the various threads in your application (top left), you can display and change variable contents (top right), and you can step through your code, step into methods, step out of methods and do normal source line debugging. If you click the resume button, execution will resume until the next breakpoint is encountered to the end of the program is reached.
- Interactive debugging makes it appear that the Java code is running on your local workstation when in reality, it is running in a JVM under the control of CICS on z/OS.

JVMServer – Dynamic Scripting

- Dynamic Scripting dynamically creates
  - **JVMProfile file** – Placed in the region's JVMPROFILEDIR
  - **JVMServer** –
    - stdout and stderr files are in APP_HOME directory
    - Set your preferences in the zerocics.config
      - Defaults for resource
      - Per installation (settings specified override the defaults)
      - Per application (settings specified override installation defaults)
  - **TCPIPService** – to listen for input to your Dynamic Scripting App
  - **URIMAP** – directs all input from above TCPIPService to PIPELINE
  - **PIPELINE** – handler specified in pipeline config file passes the HTTP request to your JVMServer.
    - Pipeline config file and shelf directory are in APP_HOME/.zero/private/cics

55

Notes:

- As of June 22, 2010, CICS suports Dynamic Scripting via the CICS Dynamic Scripting Feature Pack.
- Dynamic Scripting runs a Project Zero application under CICS's control in a JVMServer resource.
- The JVMServer resource controls a multi-threaded JVM where your Dynamic Scripting application is loaded and runs.
- Interaction with this environment (creating, starting, and stopping) your application is done by using the UNIX System Services command line.
- Using the administrative interface for CICS Dynamic Scripting can seem like a bit of magic until you realize that CICS is using standard resource definitions and configurations, but is –dynamically-creating them for you.
- This slide lists the dynamically created CICS resource definitions and associated configuration files.
- You can influence the characteristics of the dynamically created resources and config files with information you place in:
  - Your Dynamic Scripting installation's config/zerocics.config file
  - Your Dynamic Scripting application's config/zero.config file
  - Your Dynamic Scripting application's config/zerocics.confg file
- The information you place in the above file can control the JVMServer's multi-threaded JVM's heap size, garbage collection policy and other JVM infrastructure characteristics.

56

JVMServer

- **Multi-threaded JVM** added in CICS TS V4.1
- Used for **CICS Dynamic Scripting** (June 22, 2010)
- Uses T8 TCBs
- 1 to 256 threads per JVMServer (default=15)
- Max of 1024 threads per CICS region
- Uses JVMProfile file
- Statistics
  - Use count
  - Thread limit
  - Peak threads
  - Thread limit waits
  - Thread limit wait time
  - Peak thread limit waits

57

Notes:

- Each Dynamic Scripting application runs in its own JVMServer. The JVMServer in CICS is a multi-threaded JVM. All JVMs are multi-threaded, however in CICS's JVMServer, each of the threads used for application code are associated with a T8 TCB (the new TCB type for CICS TS V4.1). The reason for the T8 TCBs is that although you can create new threads in a JVM, CICS won't be aware of them unless they are mapped to T8 TCBs. A T8 TCB is needed for application code on the thread to be able to interact with CICS. So, if CICS creates threads in the JVM, T8 TCBs will be mapped to the threads and code running on those threads can interact with CICS. If an application programmer does a Thread.create() (or similar function), then the thread won't be mapped to a T8 TCB, CICS will be unaware of the thread, and code running on the thread cannot interact with CICS. (Bottom Line: application programmers are highly discouraged from creating their own threads).

- Hundreds of T8 TCB mapped threads can run in a single JVMServer. Each of these threads would be a concurrent path through the application and would be able to interact with CICS resources.

- The JVMServer server resource has a parameter where you can spacify the max number of threads (actually T8 TCBs) that can be allocated to the JVMServer (the THREADLIMIT() parm on the resource definition). The ThreadLimit on a JVMServer can be from 1-256 with the default being 15. There can be a maximum of 1024 threads for a CICS region. So if you want all of your JVMServers to allow for the maximum 256 threads, then you can only get 4 JVMServers per CICS region.

- These threads are for concurrent application usage. This means that a single JVMServer with 256 threads could handle multiple thousands of users (depending on your application's workload).

58

Shared Class Cache

- **Shared class cache** (different between 1.4.2 and Java 5/6)
- **Persists** across CICS restarts
- Does not survive an IPL
- **Faster** JVM startup
- **Lower** storage requirements (more JVMs per CICS region)
- Not suggested during testing (while you are making lots of changes)
- Changed classes are picked up by the share class cache, but a CICS JVM doesn't see the changes until it is phased out
- JIT'd classes are not cached
- Usually in /etc/javasharedresources
- Java commands for managing shared class cache
  - List, status, options to size

Notes:

- Class data sharing allows multiple JVMs to share a single space in memory.
- The Java™ Virtual Machine (JVM) allows you to share class data between JVMs by storing it in a cache in shared memory. Sharing reduces the overall virtual storage consumption when more than one JVM shares a cache. Sharing also reduces the startup time for a JVM after the cache has been created. The shared class cache is independent of any running JVM and persists until it is destroyed or the system is IPL'd.
- Using shared class cache is the default for the JVMServer.
- Using shared class cache is set to NO in the default DFHJVMPR used for JVMs in CICS's traditional JVM pool environment.
- Once you are done testing your Java program that runs in CICS's pooled JVM environment, you may want to look into running your Java program in a pooled JVM whose JVMProfile file has the shared class cache is set on.
- JIT'd classes are not cached.
- For more information on shared class cache, and how to display and control it, see the IBM Java InfoCenter and the various developerWorks articles detailing shared class cache.

JIT (Just In Time Compiler)

- **Enabled by default**
- **5 levels of JIT** – applicable to all recent IBM JVMs
  - noOpt
  - Cold
  - Warm
  - Hot
  - Veryhot
  - scorching
- When use count exceeds threshold, method is JIT'd
  - Can use default threshold or manually set it
- Can tell JVM to JIT at first class load (watch for long startup times)
- Can turn off JIT if you think JIT is causing an error
  - Can also turn off some optimizations
- Dynamic Scripting sets –Xquickstart
  - JIT compiler will use a lower optimization level by default to compile fewer methods, which can improve application startup time

Notes:

- The JIT compiler is enabled by default, and is activated when a Java method is called. The JIT compiler compiles the bytecodes of that method into native machine code, compiling it "just in time" to run. When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it. Theoretically, if compilation did not require processor time and memory usage, compiling every method could allow the speed of the Java program to approach that of a native application.
- JIT compilation does require processor time and memory usage. When the JVM first starts up, thousands of methods are called. Compiling all of these methods can significantly affect startup time, even if the program eventually achieves very good peak performance.
- In practice, methods are not compiled the first time they are called. For each method, the JVM maintains a call count, which is incremented every time the method is called. The JVM interprets a method until its call count exceeds a JIT compilation threshold. Therefore, often-used methods are compiled soon after the JVM has started, and less-used methods are compiled much later, or not at all. The JIT compilation threshold helps the JVM start quickly and still have improved performance. The threshold has been carefully selected to obtain an optimal balance between startup times and long term performance.
- After a method is compiled, its call count is reset to zero and subsequent calls to the method continue to increment its count. When the call count of a method reaches a JIT recompilation threshold, the JIT compiler compiles it a second time, applying a larger selection of optimizations than on the previous compilation. This process is repeated until the maximum optimization level is reached. The busiest methods of a Java program are always optimized most aggressively, maximizing the performance benefits of using the JIT compiler. The JIT compiler can also measure operational data at run time, and use that data to improve the quality of further recompilations.
- The JIT compiler can be disabled, in which case the entire Java program will be interpreted. Disabling the JIT compiler is not recommended except to diagnose or work around JIT compilation problems.

Garbage Collection

- **Garbage Collection** – 3 steps
  - Mark: Find all live objects in the system
  - Sweep: Reclaim unused heap memory to the free list
  - Compact: Reduce fragmentation within the free list
- **Four GC "policies"** – optimized for different scenarios
  - -Xgcpolicy:optthruput – for "batch" type apps
  - -Xgcpolicy:optavgpause – for apps with responsiveness criteria
  - -Xgcpolicy:gencon – highly transactional workloads
  - -Xgcpolicy:subpools – large systems with allocation contention
- **Healthcenter** and gcdiag (analyze GC and other aspects)
  - http://www.alphaworks.ibm.com/tech/gcdiag

63

Notes:

- As your usage of Java in CICS increases, Garbage Collection (GC) is something you need to be aware of and something that may need tuning. In the object-oriented world of Java, objects created by your Java program are dynamically allocated from a storage heap (memory specifically reserved for object allocation and deallocation). When objects are no longer used by your program (the objects are 'dereferenced'), the objects are available for GC so that the space they occupy can be used for other objects.
- GC contains 2 required steps and 1 optional step. During GC, the live (currently used) objects need to be found (the 'mark' phase), and no longer referenced objects returned to the free memory list (the 'sweep' phase). If only 'mark' and 'sweep' were done, memory would get fragmented with unusable space or not enough contiguous space for larger object allocations. The 'compact' phase defragments the heap memory.
- Traditionally, all activity in the JVM must stop during GC. This "stop the world' can cause erratic behavior and response times in your application if not controlled properly. The IBM JVM provides four GC 'policies' to deal with your application's workload characteristics.
- JVMs in CICS's pool of JVMs normally would use the default GC policy called 'optthruput' which is designed for maximum throughput. This is appropriate for JVMs in CICS's JVM pool since CICS gives them one request to deal with at a time, and we want that request to get through just as quickly as possible. When a request is complete, CICS will ask the JVM if the heap consumes greater than a specify threshold, and if true, CICS will run the CJGC transaction in that JVM. The CJGC transaction requests the JVM to do garbage collection. During that GC, since this is a CICS transaction and not a user transaction it is OK if we "stop the world" during GC. CICS decides when to do GC to maintain very quick consistent response time.
- The configuration file CICS dynamically creates for a multi-threaded JVMServer used for Dynamic Scripting specifies a GC policy called 'gencon' which is optimized for many short lived transactions that usually create smaller objects. The 'gencon' GC policy is more involved. Multiple helper threads are used for parallel mark and sweep, part of which is performed while the JVM is still processing user requests. Periodic 'compact' takes place, but even on a compact, multiple helper threads can get involved for parallel compact. We don't normally compact the whole heap at once. For transactional systems, newer, smaller objects are most likely to need GC more often so heap is divided into areas (for newer and older objects). If a newer object is passed over for GC multiple times, it is moved to the area for older objects where we apply GC less often. This isn't the end of what is done for 'gencon' as it has object nurseries and much more. Hopefully this short description is enough for you to appreciate that GC shouldn't be taken lightly, and that IBM has done a lot of research to minimize the affects of GC for your workload.

64

JVM heap size

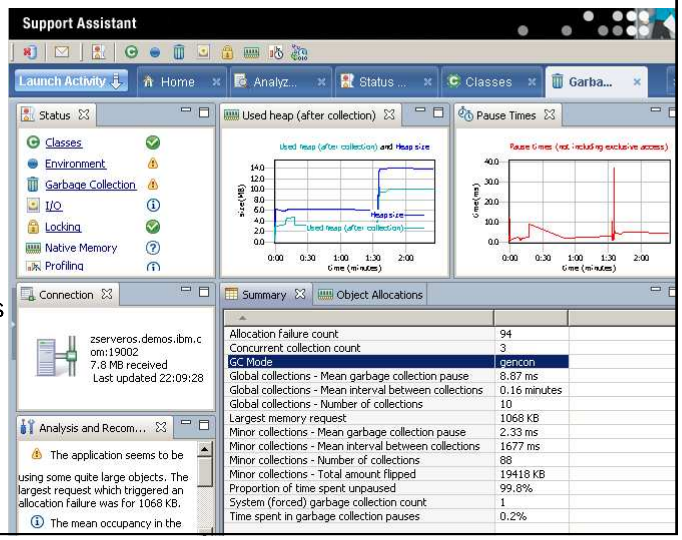
- Use **healthcenter** or **verbosegc** to set heap size
- **-Xms**
  - Should be big enough to avoid allocate failures from the time the application starts to the time it becomes ‘ready’.
  - Rule of thumb: shouldn’t be any bigger than needed for previous bullet
- **-Xmx**
  - Normal load: free heap after GC should be greater than minf (default is 30%)
  - There should be no OutOfMemory errors
  - Heaviest load: if free heap after GC is greater than maxf (Default is 70%), heap size if too big
- Heap too small = too frequent GC, too big = too much GC pause, heap > phyiscal memory = paging/swapping

Notes:

- This slide lists two of the most discussed parameters related to setting the size of the heap. These are the most looked at parameters for heap size tuning. The JVM will grow and shrink the size of the heap dynamically between the –Xms and –Xmx values while trying to maintain a balance between minimizing the time for GC and the frequency of GC.
- The CICS Dynamic Scripting zerocics.config file supplied with the installation files sets –Xmso128K. The –Xmso parameter sets the initial stack size for operating system threads.
- The CICS Dynamic Scripting zerocics.config file supplied with the installation files sets -Xiss64K. The –Xiss parameter sets the initial stack size for Java threads.
- The CICS Dynamic Scripting zerocics.config file supplied with the installation files sets -Xss256K. The –Xss parameter set the maximum stack size for Java threads.

Sources of tuning and diagnostic help

- Lots available
- Start with the **IBM Healthcenter**: realtime info -
  - Classes
  - Environment
  - GC
  - IO
  - Locking
  - Profiling
  - Gives recommendations



67

Notes:

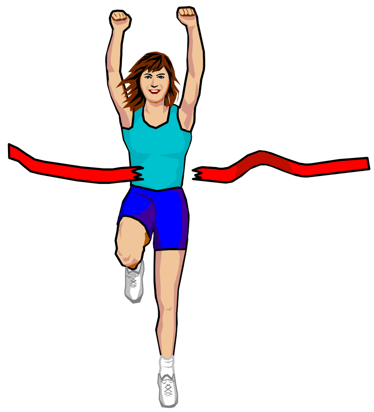
- For help looking at the performance characteristics of your JVM, take a look at the IBM Healthcenter.
- You download the IBM Support Assistant, then from it, install the HealthCenter.
- Provides lots of information in realtime with low overhead; even gives recommendations.
- See the Healthcenter documentation.

SHARE in Anaheim 2011

68

Summary

- A bit of history
- Types of Java applications
- JCICS classes
- Java infrastructure
- Debugging
- JVMServer
- A few words on tuning



Resources

- **IBM CICS Transaction Server for z/OS**
  - http://www.ibm.com/cics/
  - InfoCenter
  - Javadoc
- **IBM CICS Transaction Gateway** – **ibm.com**/software/htp/cics/ctg/
- **WebSphere Application Server** - **ibm.com**/software/webservers/appserv/was/
- **WebSphere MQ** - **ibm.com**/software/integration/mqfamily/index.html
- **Rational**
  - **ibm.com**/software/info/developer/rarwd/index.jsp
  - **ibm.com**/software/awdtools/developer/application/
- **CICS Manual:** Java Applications in CICS SC34-6238
- **Redbook** – Java Application Development for CICS SG24-5275
- **Diagnostic Guide:** http://www.ibm.com/developerworks/java/jdk/diagnosis/
- **Performance:** www.ibm.com/software/htp/cics/library/whitepapers/java_benchmark_whitepaper.pdf

Resources

- **IBM Java InfoCenter:** http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/index.jsp
- **Class Sharing:** http://www.ibm.com/developerworks/java/library/j-ibmjava4/
- **IBM Developer Kits:** http://www.ibm.com/developerworks/java/jdk/docs.html
- **IBM 31-bit SDK for z/OS, Java Technology Edition, V6:** http://www-03.ibm.com/systems/z/os/zos/tools/java/products/j6pcont31.html
- **IBM JDK Diagnosis Documentation:** http://www.ibm.com/developerworks/java/jdk/diagnosis/60.html
- **Java Troubleshooting:** http://publib.boulder.ibm.com/infocenter/javasdk/tools/index.jsp?topic=/com.ibm.java.doc.igaa/_1vg00011e17d8ea-1163a087e6c-7ffe_1001.html
- **IBM Developer Kits:** http://www.ibm.com/developerworks/java/jdk/docs.html
- **Visual Performance Analyzer:** http://www.alphaworks.ibm.com/tech/vpa
- **Garbage collection:** http://www.ibm.com/developerworks/forums/thread.jspa?messageID=13940438 http://www.ibm.com/developerworks/java/library/j-ibmtools2/index.html

Questions?

Disclaimers

The workshops, sessions and materials have been prepared by IBM or the session speakers and reflect their own views. They are provided for informational purposes only, and are neither intended to, nor shall have the effect of being, legal or other guidance or advice to any participant. While efforts were made to verify the completeness and accuracy of the information contained in this presentation, it is provided AS IS without warranty of any kind, express or implied. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, this presentation or any other materials. Nothing contained in this presentation is intended to, nor shall have the effect of, creating any warranties or representations from IBM or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of IBM software.

References in this presentation to IBM products, programs, or services do not imply that they will be available in all countries in which IBM operates. Product release dates and/or capabilities referenced in this presentation may change at any time at IBM's sole discretion based on market opportunities or other factors, and are not intended to be a commitment to future product or feature availability in any way. Nothing contained in these materials is intended to, nor shall have the effect of, stating or implying that any activities undertaken by you will result in any specific sales, revenue growth or other results.
Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon many factors, including considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve results similar to those stated here.

All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics may vary by customer.