Extending a CICS Web application using JCICS

This course provides Java application developers with a guide to CICS services, demonstrating how to access them using the JCICS API. Topics covered include:

- Fundamentals of using the JCICS API
- Using JZOS to map native record-based language structures to Java data types
- Communicating with other CICS programs using a COMMAREA
- LINKing to a Java application deployed in Liberty
- Accessing CICS resources, for example VSAM and temporary storage queues
- Managing unit of work scope in Java
- Handling errors in Java

Students should be familiar with the concepts and techniques discussed in "Developing a RESTful Web application for Liberty in CICS" before taking this course.

http://www.redbooks.ibm.com/abstracts/crse0302.html?Open

Section 1 Lecture 1

Hello and welcome to the CICS Java course to extend a CICS web application using the JCICS API. My name is Ian Burnett and I'm part of the CICS development team, based in the IBM Hursley lab in the UK.

This course will show you how to extend your knowledge of Java development in CICS, in order to produce feature-rich applications that use a range of CICS features.

We will build upon knowledge gained in the "Developing a RESTful Web application for Liberty in CICS" course, and use your existing development environment to produce more complex Java applications.

We will start the course by looking at the fundamental concepts you need to understand as a Java developer when interacting with CICS. Using these fundamentals, we will create a simple piece of code to invoke CICS programs from the Liberty environment.

After we have understood the principles of invoking other CICS programs to manipulate data, we will look at several resources available in CICS and study the APIs used to interact with those resources directly from Java.

Another aspect of CICS applications is that they adhere to the ACID properties of a transaction, and the ability to delineate resource updates into units of work is also available to Java applications. We will look at how CICS provides unit-of-work support, and also how that fits with the JCICS API.

Developing applications that are enterprise-ready, requires robust programming practices to gracefully handle application and system errors. Our final topic will be to cover the concepts and practicalities of handling error conditions using Java.

Our code samples are available from the CICSdev Github repository and provide fully-working examples of using the JCICS APIs that we will be discussing, along with additional supporting material.

The repository shown contains many Java source files that use the JAX-RS interface, in order to provide a simple mechanism of invoking Java code in a CICS environment. All the examples use the HTTP GET verb, and return JSON data. RESTful applications would typically use a variety of HTTP verbs, however the sample code is not intended as a reference for the development of RESTful applications, but merely to allow easy testing using a simple web browser.

During this course, we will assume that you have a Liberty JVM server running in CICS. The sample code provided requires use of facilities defined by the JAX-RS specification, therefore your Liberty server.xml file should be configured to include this feature.

We will also assume your workstation is configured with a development environment that contains the necessary JCICS Java libraries. A dynamic web project is sufficient to package the code samples, and for simplicity, the Liberty drop-ins directory can be used for deployment, although CICS bundles may be used if required.

Refer to the "Developing a RESTful Web application for Liberty in CICS" course if you need further information on configuring Java in your CICS environment.

Thank-you for watching this course.

Section 2 Lecture 1

The ability for programs written in different languages to communicate and share resources whilst running as a single task is a key strength of CICS. As we discussed in a previous course the interface to a CICS program is defined using either a COMMAREA, or Channels and Containers.

Different languages define the layout of the COMMAREA or container using language specific structures.

The C language allows fields to be defined with data types such as characters, integers and floating points.

COBOL programs define storage layout using fields, represented by picture clauses, to specify the different numeric and character data types.

Usually these language structures are separated out into a source file referred to as a copybook in COBOL, or as a header file in the C language. In practical terms, a copy book often represents an entity such as an employee or a stock item.

As well as describing interfaces, a language structure can also be used to describe the layout of data stored in sequential records such as VSAM files or Temporary Storage Queues.

To interface with existing programs from Java we need a means of accessing data within structured records.

When using the JCICS API to link to a CICS program COMMAREA data is passed as a byte array.

Using the standard Java language constructs, it is possible to construct, populate, and read from this byte array directly. However, there are several problems with this approach.

Firstly, manual byte array manipulation is time consuming to develop and error prone.

Secondly, a record may contain an element in a format that is not natively supported by the standard Java libraries. For example, packed decimal fields are common on z/OS, but have no Java data type equivalent.

Finally, manipulation of individual bytes within a byte array does not fit very well with the object-oriented programming model of Java.

The solution to these problems is to generate a Java bean from the copybook with accessor methods which get and set the fields automatically converting data as required.

The generated bean provides a mapping between the object-oriented Java programming model, and the record orientated programming model.

We will cover the generation of these beans in a following lecture.

During this course, it will be useful to use a sample copybook to demonstrate how and where byte arrays are used in the JCICS API.

The sample materials provide an example COBOL copybook that defines several fields in an 80-byte record.

The record represents an entry in a stock warehouse. In our application, a file is used to hold thousands of these records, representing the total stock inventory for a warehouse.

As we explore the JCICS API in this course, we will be using the sample record to demonstrate communication between various application components.

Looking in more detail at the 80-byte record defined in the sample, we can see several fields.

Firstly, an 8-byte field is used as the part ID, it represents the unique key for this record within the stock database. No two parts within the database have the same part ID.

A second 8-byte field is used to represent the ID of the supplier of that part. This would likely reference an entry in another file, which contained a list of all the suppliers used.

Each part has a unit price, that is defined using the packed decimal format.

Two dates are stored in the record – the last time an order was placed, and the next time an order will be placed for this part.

A numeric field contains information about the number of these parts that are currently held in the warehouse.

Finally, a description of this part is stored in the second half of the record in text format.

The supporting material contains another sample copybook to represent a supplier. This will be useful later when demonstrating cases where relationships exist between records.

In the next lecture, we will look at the JZOS toolkit and how the record generator component can be used to map from our sample COBOL copybook to an object-oriented Java class.

Section 2 Lecture 2

Java records are used to map native record based language structures to Java data types. IBM supplies two Java record generation tools.

One option is the J2C record importer provided with IBM Developer for z Systems Enterprise Edition within the J2C feature.

The other option is the JZOS Record Generator which we will describe in this course.

The JZOS record generator creates a Java class, based on a copybook from COBOL or a DSECT from assembler. In our example, we will generate a Java class from the stock part copybook that we looked at in the previous lecture. Generating a Java record is a two-stage process.

The first stage is to compile the COBOL program specifying the ADATA parameter to generate the ADATA file.

The second stage is to run the JZOS record generator using the ADATA file as input, to generate a Java source file

The Java source file can be included in a java project or can be compiled directly into a Java library.

Including the generated Java source file in the Java application project is useful when the generated code needs further customization such as changing encodings for specific fields. To do that requires importing the generated Java source file into your Eclipse development environment, and then adding the JZOS library to the project build path. However, if the COBOL copybook changes and record needs to be regenerated - this will require development effort to update the source in the Eclipse development environment.

An alternative approach is to compile the generated source file and package it into a Java archive, or jar file, which can be included with the Java application. This use of a compiled jar file avoids the requirement to add the JZOS libraries to the Eclipse workspace and allows multiple records to be packaged within a single library. Another advantage of this approach is that it can be automated and the JAR file placed in a repository for dependency management.

The sample application that accompanies this course follows the “compile and JAR” approach. It supplies a library jar file containing a compiled record class that corresponds to the stock part copybook.

In this section, we have looked at how the interface to CICS programs can be defined, and how Java applications can use record generators to map to this interface. We then showed how the JZOS record generator could be used to build a Java record from the compiler ADATA. Finally, we showed how to package the Java record into a library for distribution to developers.

In the next section, we will look at using this generated record when linking to a CICS COBOL program from a Java application.

For more details on these steps, including a worked example with complete JCL, have a look at the “Building Java records from COBOL with IBM JZOS” on the CICS developer center.

Thank you for watching.

Section 3 Lecture 1

In the “Developing a RESTful Web application for Liberty in CICS” course, we looked at the concept of a CICS program.

A program is a unit of compiled code that provides application logic, and can be developed using one of many languages supported in the CICS environment, including Java.

The terminology used when calling from one CICS program to another is known as a link. The link command passes control to the target program and waits for it to complete.

A link differs from a Java method call in that the target program can be implemented in any language supported by CICS and can also reside on a different CICS system.

A Java method call however is considerably more lightweight than a link command

When using the JCICS API, first create an instance of the Program class, and specify the name of the target program using the setName() method.

To issue the link simply invoke the link method on the program object. Here we do not pass any arguments on the link method so no data is passed to the target program.

When communicating from one program to another, there are two main ways to pass data --- either through the use of a commarea - or through the use of channels and containers.

Now let’s look at passing a commarea on a link command

A commarea is an area of memory used to pass data between two programs.

From a CICS perspective, a commarea is just a sequence of bytes, CICS does not attempt to format the data structure and just facilitates the transfer of the data between the programs.

The data structure is described using an interface, where the fields are defined using a language specific structure such as a COBOL copybook as we discussed in the previous lecture.

For a Java application to communicate with an application written in another language we can use a byte array to represent the commarea and to hold the data to be transferred. As noted in the previous section, generated Java classes can be used to more easily construct the required byte array.

In this example, we look at passing a single record from our Java application into an existing CICS program using the COMMAREA interface. This record is defined by the standard stock part record format covered earlier.

When using the JZOS record generator tooling, we called our generated class “StockPart”. So, our first task is to create an instance of this class, and populate it with relevant data.

The JZOS record generator tooling defines getter and setter methods in the Java class, the names of these methods correspond to the fields of the original COBOL copybook.

Getter and Setter methods, otherwise known as “accessor” methods, allow a Java developer to use an object-oriented approach when building a record.

Once the Java object has been instantiated, we can convert it into a byte array using the getByteBuffer() method. This returns a serialized version of the structured record defined in the original copybook.

Now that we have the data ready, we can use this to link to the target program.

We create a new instance of the Program class, and specify the name of the target program that we wish to LINK to using the setName() method.

We then invoke the link method on our program object - this time passing the StockPart byte array

On successful return from the target program, any updates to the commarea will be reflected in the byte array.

Here, we take the updated byte array and construct a new StockPart object to represent the returned data. Subsequent code may then read values from this newly-constructed object using the accessor methods.

A commarea is a bidirectional transfer mechanism, and programs can use this area to send any format of data in either direction. Our previous example sent a record in the StockPart format, and received a record in the same format.

With a large enough data area, any record format can be returned.

In this example, we use a StockPart object to construct a byte array that will be passed to a target program. When called, that program will look up the supplier record and populate the commarea with a record format detailing the supplier of that part.

Having previously created a suitable JZOS class to map the record for a supplier - when the link method completes - we can generate a new instance of the supplier object from the returned byte array. The resulting object can be queried for the supplier information.

CICS offers an optimization on a link command by specifying the size of the data to the target program - a benefit also available from the JCICS API.

As an example, a part lookup routine may only need a 8-byte part number as input, but will require an 80-byte commarea to return the data. Specifying the optional int parameter after the byte array allows the caller to specify the data length used on the send.

In this example, a byte array of length 80 is created by the calling program, but the datalength is specified as 8. When CICS passes control to the target program, only the first 8 bytes are transmitted. The remainder of the 80-byte commarea is padded with zeroes. When the target program returns, all 80-bytes are sent back to the calling program.

This optimization can be used to reduce CPU and response times, especially where large commareas are required for return values or where links flow between CICS systems.

Recall that the other method of passing data from one program to another is by Channels and Containers. A channel is a conduit between programs, it typically holds one or more containers to be passed between the programs. A container is a named block of data.

Channels and containers offer the advantage that more than 32 KB of data can be passed. By contrast, commareas are confined to a 32 KB limit.

Multiple containers can be passed between programs within a channel. A channel is therefore analogous to a parameter list.

A worked example of using channels and containers in JCICS is available in the “Developing a RESTful Web application for Liberty in CICS” course. The sample materials for this course also contain an example of using channels and containers to pass data.

This lecture has looked at the ways in which the JCICS API can be used to LINK to another CICS program.

The sample source code provided with this course has examples of all the JCICS commands covered in this lecture.

The next lecture will look at how we can link from a CICS program into a Java component in a Liberty JVM server

Section 3 Lecture 2

WebSphere Liberty provides a Java Enterprise Edition or Java EE platform, and is designed to help developers create large-scale enterprise applications.

Using Link to Liberty, any CICS program can now call Java EE applications in a Liberty JVM server.

The CICS LINK command can invoke Plain Old Java Objects (POJOs) that have been deployed inside a Web application...just as if they were any CICS program.

To achieve this seamless LINK, a CICS defined Java annotation is applied to a Java method within the Liberty application. The annotation indicates that the method should be exposed as the entry point for a CICS Java program

When the application is installed CICS will detect the annotation meta-data and automatically create and install a PROGRAM resource for each entry point detected.

Once these PROGRAM resources have been created, then a CICS application written in any language can use the LINK command and invoke the desired Java method passing a channel as the interface. Note that a commarea cannot be passed in this scenario.

In this example, we have a class or POJO in a Web application that exposes the getSupplierInfo() method as an entry point. The @CICSProgram annotation is used to indicate this method should be made available as a PROGRAM resource. When the application is installed, a CICS program resource named GETSUPPL is automatically created, and is ready to be invoked.

Data is passed between calling program and the Liberty application using Channels and Containers.

Within the target Java method, JCICS is used to get a reference to the current channel and to obtain the data from containers provided by the caller.

Our sample code provides a complete implementation of the get supplier info method. The sample code could be extended to use Java EE capabilities such as JAX-RS client API to request information from a supplier using a remote, RESTful web service.

From the perspective of the calling COBOL application, it has used the CICS LINK command to invoke a named program passing in data via a container, and received a response using a COBOL structure.

The POJO component in the web application was able to receive and process the containers and exploit Java EE7 functionality.

In this section, we have looked at how we can link to other CICS programs using methods on the JCICS Program class.

Using our generated record, we passed data to existing CICS applications using the commarea interface. The alternative method of passing data between programs - the CICS channels and containers support - was covered in the “Developing a RESTful Web application for Liberty in CICS” course.

Finally, we looked at how CICS can make Liberty applications available to traditional programs, using the Link to Liberty capability.

In the next section, we will look at how to access other CICS resources using the JCICS API.

Section 4 Lecture 1

CICS provides access to a range of resource types. Some of these resources like Temporary Storage Queues are unique to CICS, while others such as VSAM files are common across z/OS. In this section, we will be looking at accessing these resources using the JCICS API.

We'll come to the CICS-specific resources shortly, but first let's start with VSAM files.

In CICS, data management services are traditionally known as CICS file control. CICS file control offers you access to data sets that are managed by the Virtual Storage Access Method, or VSAM.

Basic direct-access method, or BDAM, data sets are not supported by the JCICS API, so this lecture will focus on VSAM files.

CICS file control lets you read, update, add, delete, and browse data in VSAM data sets. You can also access CICS shared data tables and coupling facility data tables using the file control APIs.

Files may be shared by applications within a CICS region, or across the wider z/OS sysplex.

A CICS application program reads and writes its data in the form of individual records. Each read or write request is made by a CICS command.

To access a record, the application program must identify both the record and the data set that holds it.

There are several methods that can be used to identify a record – which one to use depends on the type of VSAM file in use. We will look at the different types of VSAM file later.

An application uses a CICS file resource to identify a data set. This file resource has a unique name within the CICS region, allowing the application to remain agnostic of the underlying dataset name.

It is not necessary for an application to open or close a file explicitly, CICS manages the resource on behalf of all applications.

In this diagram, an application accesses a VSAM file using CICS file control.

The application identifies the data set to be used by referencing the CICS FILE resource named STOCK1.

The STOCK1 file resource is defined to reference the MVS dataset TEST dot DATA dot STOCK.

The application could later be deployed into a different system, where a file resource of the same name references a different MVS dataset.

CICS supports access to the following types of VSAM data set:

- Key-sequenced data set, or KSDS
- Entry-sequenced data set, or ESDS
- and Relative record data set, or RRDS

A key-sequenced data set has each of its records identified by a key. The key of each record is a field in a predefined position within the record. Each key must be unique in the data set.

An entry-sequenced data set is one in which each record is identified by its relative byte address or offset.

A relative record data set has records that are identified by their relative record number or order.

The VSAM datasets topic in the IBM Knowledge Center has a more comprehensive description of these different VSAM data sets.

Our example will use a key-sequenced data set.

Before we can use VSAM files, we need to define the dataset using the z/OS utility IDCAMS.

The example shows some sample input to the IDCAMS utility to define a VSAM file named TEST.DATA.STOCK. We will use this dataset in our example to store stock information from the StockPart record we generated using JZOS earlier.

The records keyword indicates the amount of space that the disk subsystem needs to reserve for the data set, while the INDEXED keyword specifies the VSAM data set is to be used for key-sequenced data.

The KEYS keyword specifies that each record has a unique key that is eight bytes in length and begins at offset zero. This matches the STOCK PART record we defined earlier.

Finally, the RECORDSIZE keyword specifies that on average, each record will be eighty bytes in length, with a maximum length of eighty bytes. While VSAM files support variable-length records, our simple application only uses fixed-length records.

For the full example, see the accompanying sample project materials in Github.

As described earlier, a CICS application is abstracted from the physical datasets it uses by means of a CICS FILE resource. In order to use our sample application, you will need to define and install a FILE definition into the CICS region.

For this example, most of the defaults will be suitable, with some exceptions.

The file resource definition requires a name of up to eight characters – the sample code uses the SMPLXMPL name.

If you do not specify the file as a DD statement in your CICS startup job, you will need to specify the physical dataset name using the DSNAME attribute on the definition.

To fully demonstrate the JCICS file control API, you should set all of the add, browse, delete, read, and update attributes to yes, otherwise CICS will throw an exception when the application attempts to perform an operation that is not permitted.

Now we are going to develop the web application to access the VSAM file.

The sample class VSAM KSDS file resource provides a number of methods to demonstrate the various aspects of accessing a VSAM file.

The simplest of these methods is the “write new record” method which creates a new stock part and adds it to the VSAM file.

As with the examples in the “Developing a RESTful Web application for Liberty in CICS” course, we use a simple RESTful interface to invoke the Java code.

The CICSApplication class specifies a base application path of “rest”, therefore any RESTful resources accessed in this web application will use this string as a root URI.

The VSAM KSDS file resource class specifies a relative resource URI of ksds using the Path annotation, while the write new record method specifies a path of “write” relative to the parent resource.

For the write new record method, we would therefore use the URI “rest/ksds/write” relative to the base URI of the application when installed into CICS.

Only the HTTP GET verb is recognized in this application, as indicated by the GET annotation on the method.

The write new record method returns a Stock Part Collection object, which is serialized to JSON as directed by the Produces annotation on the class. The Stock Part Collection object is a simple wrapper object that contains a list of Stock Part objects.

To create a record suitable for insertion into the VSAM file, the generate method in the stock part helper class creates an instance of the JZOS-generated Stock Part class and populates it with sample data.

The write new record method takes the generated record and converts this to a byte array using the JZOS get byte buffer method.

Another method in the stock part helper class extracts the record key as a byte array. Recall that the key for the stock part record is eight bytes in length and begins at offset zero. The helper class obtains the byte array from the Stock Part object and extracts the first eight bytes as a new byte array.

In this example, we are accessing a key-sequenced dataset, or KSDS file. To reference this file using JCICS, we create an instance of the KSDS class and specify the name of the CICS file resource.

The write method on the KSDS class takes two byte arrays as parameters – the key and the record for writing.

The CICS transaction would automatically be committed on normal completion of the CICS task, but as a simple demonstration of managing units of work, the sample application uses the Task commit method to explicitly complete the current unit of work.

Finally, the query file method is invoked, which returns an instance of the stock part collection class.

The query file method performs a file browse operation, starting at the very first record in the file, and reading each record into memory. Each record is converted to a stock part object and added to the stock part collection instance. When serialized to JSON, this provides a simple means of viewing the contents of the sample VSAM file and how it is affected by each file control operation.

For clarity, the necessary error-handling logic has not been included here, but is provided in the sample code. We will cover error-handling later in the course.

The updateRecord() method provides an example of updating a record in a VSAM file.

Firstly, the KSDS file is referenced by creating a new instance of the KSDS class, as previously done in the write new record method.

To read data from a file, a record holder object is required. The read for update method on the KSDS class is then called, passing in three parameters.

The first parameter is the key of the record that we wish to read from the VSAM file. In this case, we supply a key of all zeroes, created using a method in the stock part helper class. A key of all zeroes is used in our application, simply to obtain the first record in the absence of real data.

The second parameter specifies that the application will read the record with a key equal to, or greater than, the key supplied.

The final parameter is the record holder instance, which will be used to receive the data that results from the VSAM file read operation.

On successful completion of the read for update method, the record holder instance will contain a byte array representing the record in the data set, as located by the specified key. The get value method extracts this byte array, and it is passed to the constructor of the generated stock part class.

A second stock part instance is created using the generate method we covered earlier. The part id, or key, from the read record is copied into the newly-generated record. This step is used as a simple demonstration of where business logic would amend values in a real-world application.

We then update the record in the VSAM file by calling the rewrite method, supplying the full record as a byte array.

Note that for a rewrite operation, we do not need to supply the key, as this was established on the previous read for update operation.

Finally, we explicitly call the task commit method to release any locks we hold against the file.

When executing the sample code, repeatedly accessing the rest/ksds/update method by pressing refresh in your browser will show the first record in the file being updated with new sample data.

The delete record method in the VSAM KSDS file resource class provides an example of deleting a record in a VSAM file.

The flow of the method is very similar to the update record example, with a read for update operation finding the correct record, and then the delete method being called to delete the record from the file.

In this lecture, we have looked at how we can use the JCICS API to access VSAM KSDS files, using our generated JZOS class to provide an object-oriented means of accessing structured records on disk. Many of the concepts covered here are equally applicable to ESDS and RRDS files. See the JCICS Javadoc documentation for a complete reference on the syntax required to access these types of VSAM files.

In the next lecture, we will look at how Java applications can access CICS temporary storage queues.

Section 4 Lecture 2

CICS temporary storage queues, or TSQs, are a means of storing a sequence of data items.

Data in a TSQ can be stored in main memory, on disk, known as auxiliary TSQs, or shared via the sysplex coupling facility.

The data items in a TSQ have a maximum size of just under thirty two kilobytes, and can be accessed in any order

TSQs are referenced using a name of up to 16 characters in length. They can be defined as being recoverable or non-recoverable CICS resources.

Temporary storage queues do not need to be defined in advance of an application using them, and can be used to share data across applications.

Support for temporary storage queues is provided in the JCICS API by using the TSQ class.

In our sample we are using a TSQ from within a CICS Liberty Web-application and because HTTP requests are stateless, we'll need some means of persisting the TSQ name across multiple HTTP requests.

To provide a simple-to-use method of accessing the same temporary storage queue from a browser across multiple requests, the queue name is stored in an HTTP session object.

Every time a browser issues a GET request against the temporary storage resource class, the get queue method extracts the queue name from the HTTP session object, and returns a JCICS TSQ object that corresponds to the correct queue for that browser session.

Examining the flow in more detail, the first time a browser accesses the sample application, it issues an HTTP GET request against the rest slash tsq slash write URI, and has no cookie token to send to the server. The get queue method in the application finds no cookie has been sent, and hence no HTTP session is present.

The generate queue name method is invoked to create a new, unique queue name. This queue name is stored in an HTTP session object, which is persisted automatically by the Liberty runtime.

The write new record method will write an element to this new queue, and then return to the client. When the Liberty container sends the JSON response back to the browser, it also includes an HTTP session cookie as part of the flow.

Subsequent requests from the browser will flow this HTTP session cookie to the application, and this cookie is used by the Liberty server as a key to retrieve the HTTP session object that corresponds to the browser instance making the request.

Before we examine the JCICS API in detail, let's take a brief look at some of the sample code's helper methods.

The name of the queue used by our sample application is created by the generate queue name method. This creates a queue name, based on a known prefix and the current time.

The sample temporary storage resource class provides examples of accessing a TSQ using the JCICS API,

The write new record method demonstrates how to write a single element to a temporary storage queue.

Here we use the get queue method to obtain an instance of the JCICS TSQ class.

This TSQ object references the CICS temporary storage queue that is in use by the current browser session, using the mechanism described earlier.

A sample record is created using the stock part helper class we covered when looking at VSAM file access, and a byte array is obtained using the JZOS get byte buffer method.

Writing a single element to a TSQ is achieved by simply calling the write item method, passing the data to be written as a byte array.

In a similar manner to the VSAM examples, a stock part collection object is returned by the method, and is serialised by the Liberty server as a JSON response.

This collection object is created by the query queue method, which performs a browse across the queue, adding each queue element to the collection.

There are two important points to note from the query queue method.

The first is that elements in a temporary storage queue are numbered from element one, unlike many constructs you may be used to in Java.

The second is that items are read from a TSQ using an Item Holder instance. This item holder object performs a similar function to the record holder we discussed earlier for VSAM files.

As in the previous VSAM example, the update record method will update the first element in the queue.

The get queue method is again used to obtain the TSQ corresponding to the current browser session, and a new sample stock part record is created using the stock part helper class.

The TSQ is updated through the use of the rewrite item method in the TSQ class, which takes two parameters – the item index and the new data to write to the queue.

After updating the queue, our query queue method is called to return the contents of the queue as a JSON response.

Individual elements cannot be deleted from a temporary storage queue – it is only possible to delete an entire queue.

The delete method in the TSQ class takes no arguments and deletes the corresponding temporary storage queue.

The delete queue method in our sample code gives a simple example of obtaining a TSQ object and then deleting the underlying queue.

In this lecture we have covered the basic concepts of a temporary storage queue, and looked at how we can use the JCICS API to access these queues from Java.

Thank you for watching.

Section 4 Lecture 3

One of the fundamental concepts of transaction processing is that resource updates adhere to the atomicity, consistency, isolation, and durability principles, commonly known as the ACID properties of a transaction.

Resources accessed by a program are managed within a transactional scope known as a unit of work. All recoverable resources accessed within a single unit of work are updated according to the ACID principles.

A unit of work may either complete successfully, known as a commit, or unsuccessfully, known as a roll back.

In a CICS task, an active unit of work is always present.

A unit of work begins automatically at the start of a CICS task, before control is passed to the application. Upon successful completion of a CICS task, the unit of work is automatically committed.

If a CICS task terminates abnormally, then the active unit of work is rolled-back and any updates to recoverable resources are undone.

Updates to recoverable CICS resources made from Java applications will be subject to the same unit of work constraints as other CICS programs.

It is also possible for an application to mange the unit of work boundaries.

The JCICS commit and rollback methods in the Task class can be used to complete the current unit of work and update resources accordingly.

Once the current unit of work has been completed in CICS, a new unit of work is automatically created, ensuring that any resource updates always occur within a unit of work.

Where large numbers of resources are being updated in long-running tasks, if the application design permits, it is recommended that units of work are committed periodically. This will reduce the number of outstanding locks that are held by the application, and potentially improve overall system throughput.

The write new record method in our VSAM sample shows an example of explicitly committing the current CICS unit of work.

In this section, we have looked at the characteristics of several CICS resources, and how to access them using the JCICS API.

In this lecture we covered the CICS support for units of work, and how applications can control the scope of updates to recoverable resources.

In the next section, we will look at how Java applications can successfully handle application and system errors, in order to produce more robust applications.

Our sample application code provides examples of accessing key-sequenced VSAM files, CICS temporary storage queues, and also CICS transient data queues using JCICS.

Thank you for watching.

Section 5 Lecture 1

It is important to provide good error-handling logic in enterprise-class software to gracefully handle a range of application and system problems.

In this section, we will look at how Java handles errors in general, then apply that knowledge to the JCICS API.

During execution of an application, various error conditions may arise. These error conditions may have many root causes, but fundamentally they can be categorised into three main types: expected errors, unexpected errors, and fatal errors.

Expected errors are conditions that can be reasonably foreseen by an application developer. For example, attempting to read a non-existing record from a VSAM file is likely to be a condition that the application is expected to handle.

Unexpected errors are conditions that may occur, but are not normally expected to be handled by the application. For example, a program may attempt to access an element of a Java array which does not exist. This probably represents a programming error, and as a result, the application is likely to be in an indeterminate state.

Fatal errors are the most serious of all error conditions, and usually represent a situation from which an application is unlikely to be able to recover. An example of a fatal error occurring would be if the Java virtual machine failed to load a Java class required by an application.

Error conditions in the Java language are managed using the exception mechanism.

If a Java method encounters an error condition, then a special Java object called a Throwable is created to capture the state of the current thread at the time of the error, and this object is propagated to the caller of the current method – a process known as throwing an exception. Correspondingly, handling the error condition is known as catching an exception. A calling method uses a try keyword to indicate the scope of the error handling logic.

The Java language specifies that the objects used when propagating exceptions must be instances of a type which extends the Throwable class. The Throwable class itself has two direct subclasses – Exception and Error. There are many sub-classes of Exception. However, the RuntimeException subclass is a special case and is not classified as a checked exception as it represents unexpected runtime errors.

Together, the checked exception, runtime exception, and error classes represent error conditions in Java code that correspond to the expected, unexpected, and fatal error categories that we covered earlier.

In Java's error handling mechanism, expected errors extend the Exception class, and are known as checked exceptions.

The Java language specifies that any method which can throw a checked exception must declare this as part of the method signature, using the throws keyword.

Any Java code which calls a method declared as throwing a checked exception must either provide logic to catch the checked exception, or add the exception to it's own method signature, in order to propagate the exception further up the stack.

The constraints on checked exceptions are enforced at compile time, and failure to adhere to the specification will result in a compilation error.

Unexpected error conditions are represented by Java classes which extend the Runtime exception class, and are known as unchecked exceptions.

Unchecked exceptions are not subject to the compile time checking mandated for checked exceptions, although they can be caught if required.

Fatal error conditions are represented by Java classes which extend the Error class. It is considered poor programming practice to catch any exceptions that are subclasses of the Error class, as these rarely represent a condition from which the application will be able to recover.

Now lets take a look at how CICS error handling is managed for COBOL and other high level languages.

These languages that use the EXEC CICS API have three choices when handling error conditions: on a per-command basis, using a condition handler, or using an abend handler.

If a CICS command produces an error, then it will first try to return a response code to the application using the data area supplied in the RESP parameter of the CICS command.

If a RESP parameter has not been supplied, then CICS will search for an active condition handler. Condition handlers are registered in advance using the EXEC CICS HANDLE CONDITION command, and perform some form of application-specific recovery operation.

If no active condition handler is found, then CICS will abnormally end, or abend the task. There are many types of CICS abends defined, each corresponding to a specific error condition, and abends are classified by their four-character abend code.

When an abend occurs, CICS will search for an active abend handler that matches the generated abend. Similar to condition handlers, an abend handler routine is registered in advance using an EXEC CICS HANDLE ABEND command.

If an active abend handler cannot be located, then CICS will abnormally terminate the current task, which in turn will rollback any uncommitted updates to recoverable resources in the current unit of work.

Now lets consider how CICS errors map to the JCICS API.

The JCICS exception hierarchy includes unchecked exceptions, checked exceptions, and fatal errors.

The key to the mapping between CICS errors and the JCICS exception hierarchy is understanding where these exceptions could be thrown, how they map to the CICS response codes, and what action you should take as an application developer.

Let's start by looking at checked exceptions.

Here we see the key classes involved in the JCICS exception hierarchy. Most JCICS methods are defined as throwing checked exceptions, and these checked exceptions represent the majority of conditions that an application may be expected to handle.

Each error condition from a CICS command is mapped to an exception, and this exception is a subclass of the CICS condition exception.

As an example, the CICS command to read an element from a temporary storage queue may return an ITEMERROR condition to indicate the queue did not contain the element requested.

In the JCICS API, an element is read from the TSQ using the read item method. This method signature declares it may throw (amongst others) an Item Error Exception, which is the JCICS API equivalent of the ITEMERROR condition.

See the query queue method in the temporary storage resource sample class to see how the ITEMERROR and QIDERR conditions are handled as expected errors, while any other error condition results in a failure of the request,

There are only a few unchecked exceptions in the JCICS API, and they all extend the CICS runtime exception class.

All of these exceptions represent conditions within CICS that should not be handled by an application.

Java code running in CICS should not catch these exceptions, either explicitly in a catch block, or implicitly by catching a superclass of these exceptions. Instead, they should be allowed to propagate out of the Java environment and back to CICS, where the unit of work can be rolled back.

Note included in this list is the AbendException which represents an abend of a CICS task, and should only be caught if you wish to develop your own abend handling routine.

It important to stress that having a try block that catches any superclass of the java dot lang dot exception class should not be used when invoking the JCICS API.

It important to stress that having a try block that catches any superclass of the java dot lang dot exception class should not be used when invoking the JCICS API.

<<Note to video team. You will probably need to pause the speaker here for a short while – the complexity of this slide wasn't really thought about when preparing the script. Thankfully I am off-screen.>>

The Java Error class represents a fatal error condition in the JVM that should not be caught by an application.

The same applies to the CICS error class, which extends java dot lang dot error. An application should not attempt to handle a fatal CICS error, but instead allow the exception to propagate back to the CICS environment to allow full recovery to take place.

If a Java exception such as a null pointer exception is allowed to propagate out of the Java code and back to the JVM server runtime, this is generally surfaced as one of the CICS abends starting with AJ.

Most commonly, an uncaught exception will result in an AJ04 abend, and the current unit of work will be rolled-back.

It is also possible for a Java application to issue an abend directly, using one of the abend or force abend methods in the Task class. This is similar in concept to the throwing of Java exception, as it can allows a CICS abend handler to take control of error processing.

The various forms of the abend method allow an application to optionally specify an abend code or if a dump is required.

The force abend methods provide the same options as their equivalent abend methods, but are equivalent to specifying the CANCEL keyword on the EXEC CICS ABEND command. Invoking a force

abend method will always terminate the task abnormally, and overrides any existing abend handlers that have been established for the task.

In our sample VsamKsdsFileResource class we use the following try catch block to handle CICS error conditions when deleting items from a VSAM file

The RecordNotFoundException is a checked exception but is a normal situation on the initial browse, as so can be ignored when browsing the file.

All other CICS condition exception are un expected, and in this situation we throw the JAX-RS InternalServerErrorException which will cause a HTTP server error to be returned to the caller.

This lecture has introduced the concept of checked and unchecked exceptions in Java, along with the throwing and catching mechanism used to propagate the exception objects.

We have seen how the JCICS API maps CICS error condition values to exceptions, and how CICS abends are processed.

Remember it is acceptable for an application to catch and handle CICS condition exceptions, but it strongly recommended that CICS runtime exceptions are allowed to propagate back to the CICS environment.

Finally, we looked at how Java exceptions are mapped to abends in CICS programs, along with the APIs required to issue abends directly from Java code.

Thank you for watching.

Section 6 Lecture 1

In this course, we have extended our knowledge of Java development in CICS.

We covered some of the fundamental concepts you need to understand as a Java developer when interacting with CICS, and then used these fundamentals to invoke CICS programs from the Liberty environment.

We then looked at the use of the JCICS API to access CICS resources including VSAM files and TSQs, and examined how Java applications can participate in CICS unit of work support.

Finally, we looked at the concepts and practicalities of handling error conditions using Java.

Many code samples, including the ones used in this course, are available on our CICS Dev git hub site.

This concludes our course around extending a CICS web application using JCICS.

Thank-you for watching.