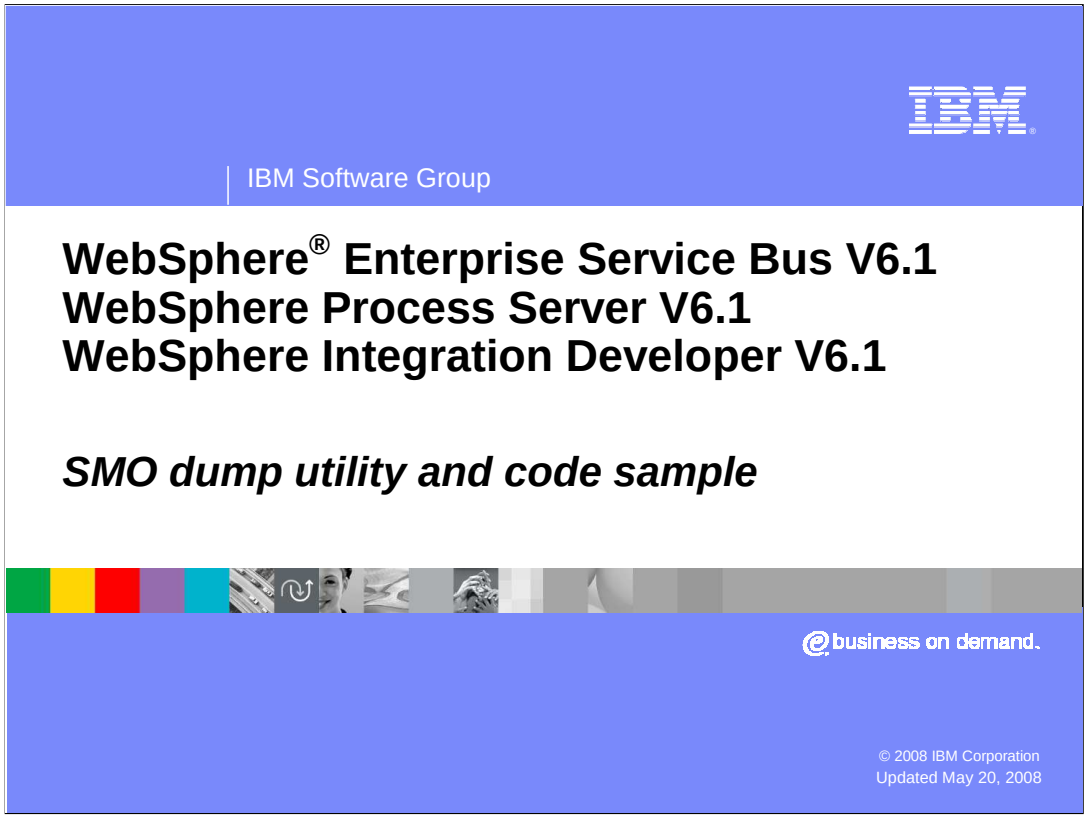


This presentation provides information about a DataObject dump utility useful for dumping a service message object (SMO). It describes the use of the utility for SMOs and looks at the implementation, which serves as an example of SMO traversal.

Goals

- Introduce a DataObject dump utility
  - Useful for dumping the service message object (SMO)
  - Provide you with the code in a project interchange file
  - Describe how to use it
- Enhance your understanding of the SMO structure
  - Showing you sample dump output
  - Describe code from dump utility that traverses the SMO
- Enable you to examine the SMO in your own mediation flows

The goal of the presentation is to introduce you to a DataObject dump utility that is used for dumping the contents of the service message object. You are provided with a link to a project interchange file that you can import into WebSphere Integration Developer for your own use. How to instrument a mediation flow to call the dump utility is described.

The utility can be used to dump any DataObject. However, the focus of its use in this presentation is for the dumping of an SMO with the intention of enhancing your understanding of the SMO structure and contents. The presentation shows you a few sample output dumps of major sections of the SMO. Following that, the code for the dump utility is examined to show you how to write code that can traverse an SMO.

The project interchange file for this utility is being provided to you because the best way to enhance your understanding of the SMO is for you to use the dump utility in your own mediation flows. Being able to see what is really in the SMO will help you tremendously as you design, develop and debug your mediation flows.

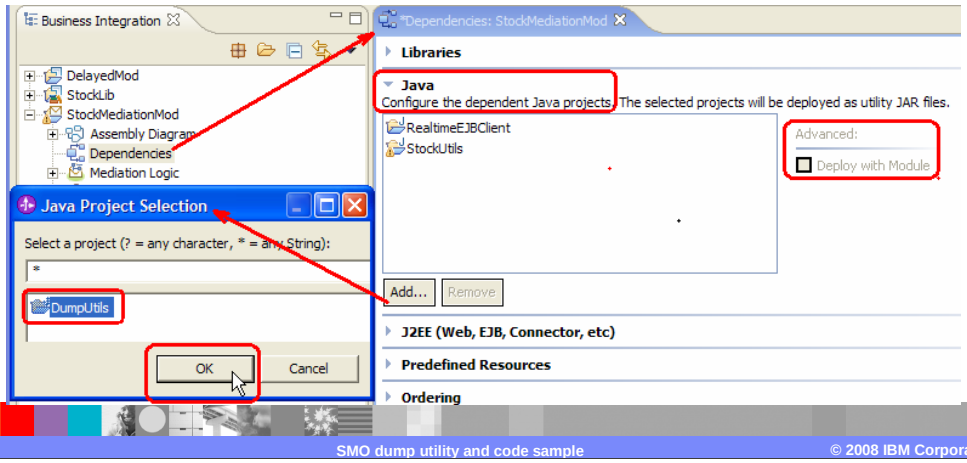
Section

How to use the dump utility

The first section of the presentation shows you how you can use the dump utility to dump the contents of the SMO.

Include dump utility in your mediation module

- To use the dump utility in your flow, you must:
  - **Download the DumpUtilsPI.zip** project interchange file
  - Import it into WebSphere Integration Developer
  - In your mediation module, add it as a Java™ project dependency



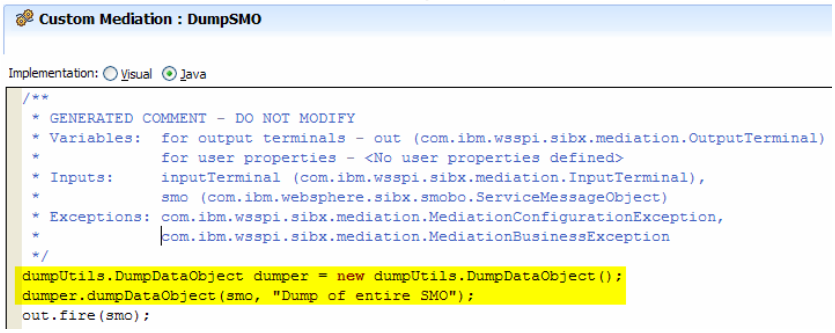
The dump utility is packaged in a WebSphere Integration Developer project interchange file named DumpUtilsPI.zip. You can download this using the link on this slide. There is also another link to it provided on the same page you used to launch this presentation.

Once you have downloaded the file, you need to import it as a project interchange file into your WebSphere Integration Developer workspace.

In order to use the utility in your mediation module a dependency to it must be created. The screen capture in the slide basically shows what you need to do. Start by opening the dependencies editor for the mediation module and then open the Java section, which allows you to add dependencies to Java projects. Click the Add… button to open the Java Project Selection dialog and then select the DumpUtils project and hit OK. This adds the project to the list of Java projects used by this module. Click on the DumpUtils project in the list and check to ensure that the Deploy with Module check box has been selected.

Call dump utility from a custom mediation

- Calling the dump utility
  - Create a custom mediation with Java snippet
  - Add the Java code to call the dump utility



- See README-UsageNotes.txt in DumpUtils project
  - Documents usage
  - Can cut and paste the code into the Java snippet

In your mediation flow, you can call the dump utility using a custom mediation primitive with a Java snippet implementation.

There are two lines of code that you need to add, which are highlighted in the screen capture on this slide. The first line creates the object that provides the dump functionality and the second line uses the object to perform the dump of the SMO. The method to call is named dumpDataObject and it takes two parameters, the first being the DataObject to dump and the second is an optional comment that is included at the beginning of the dump.

The last line provides the SMO that is propagated from the output terminal when the custom mediation completes. This is required by the custom mediation primitive implementation and is included in the code skeleton when the custom mediation primitive is first created.

There are use notes provided in the DumpUtils project that are located in the read me file referenced on the slide. It contains documentation for the utility, including code snippets that can be cut and pasted directly into your custom mediation primitive.

Dumping selected elements of the SMO

- Utility can dump selected elements in the SMO
  - Element cannot be a leaf, must be a DataObject
  - When using ServiceMessageObject APIs, need to cast to DataObject
- Examples:

```
//--- Dump all of the headers
dumper.dumpDataObject((DataObject)smo.getHeaders(),
                      "Dump SMO headers");
//--- Dump jist the JMS headers
dumper.dumpDataObject(smo.getDataObject("/headers/JMSHeader"),
                      "Dump JMS headers");
//--- Dump the failure info on a flow from a fail terminal
dumper.dumpDataObject((DataObject)smo.getContext().getFailInfo(),
                      "Dump failure info");
//--- Dump the correlation context
dumper.dumpDataObject((DataObject)smo.getContext().getCorrelation(),
                      "Dump correlation context");
//--- Dump the message body
dumper.dumpDataObject((DataObject)smo.getBody(),
                      "Dump message body");
//--- Dump an order data object contained within the message body
dumper.dumpDataObject(smo.getDataObject("/body/submitOrder/order"),
                      "Dump order data object contained in message body");
```


The utility is able to dump any DataObject and therefore you do not need to dump the entire SMO. Any element of the SMO can be dumped provided it is a DataObject rather than a leaf element. When using the ServiceMessageObject APIs, the returned element's type does not inherit from DataObject even when that part of the SMO is in fact a DataObject. Therefore, when using the ServiceMessageObject APIs you need to cast the returned value to DataObject.

This slide shows several example invocations of the utility. There are two basic approaches that are shown. One is to dump an element of the SMO using the getDataObject method on the input parameter to the custom mediation primitive, which is called smo. The string passed to this operation is an XPath expression to the element you want to dump. The other approach is to use the ServiceMessageObject APIs on the input parameter smo and to then cast the returned value to the type DataObject.

The first example dumps all the headers in the SMO using the ServiceMessageObject API approach, while the second example dumps just the JMS header and uses the DataObject API approach.

The third example is useful in a flow wired to a fail terminal because it is dumping the failInfo element that gets populated when a mediation primitive raises an exception.

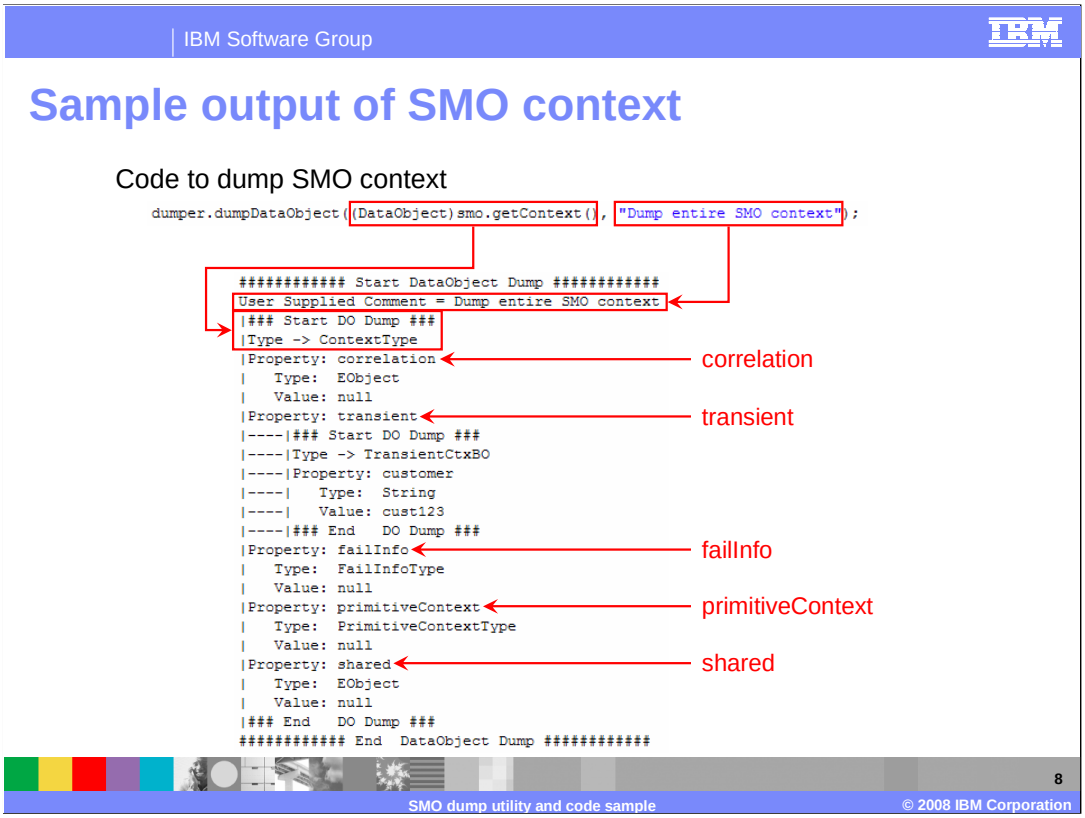
The fourth example dumps the correlation context.

The next example is dumping the entire contents of the SMO body and is using the ServiceMessageObject APIs. Contrast this to the last example, which dumps a portion of the body using the DataObject APIs. Note that in this last example, there are no ServiceMessageObject APIs that provide traversal into the message body, so the DataObject APIs have to be used in this case.

Section

Sample output from the dump utility

In this section you are shown sample output from the dump utility. The dump output is directed to the SystemOut.log file for the server. It is also visible in the Console view of WebSphere Integration Developer.



This first example dumps the context section of the SMO.

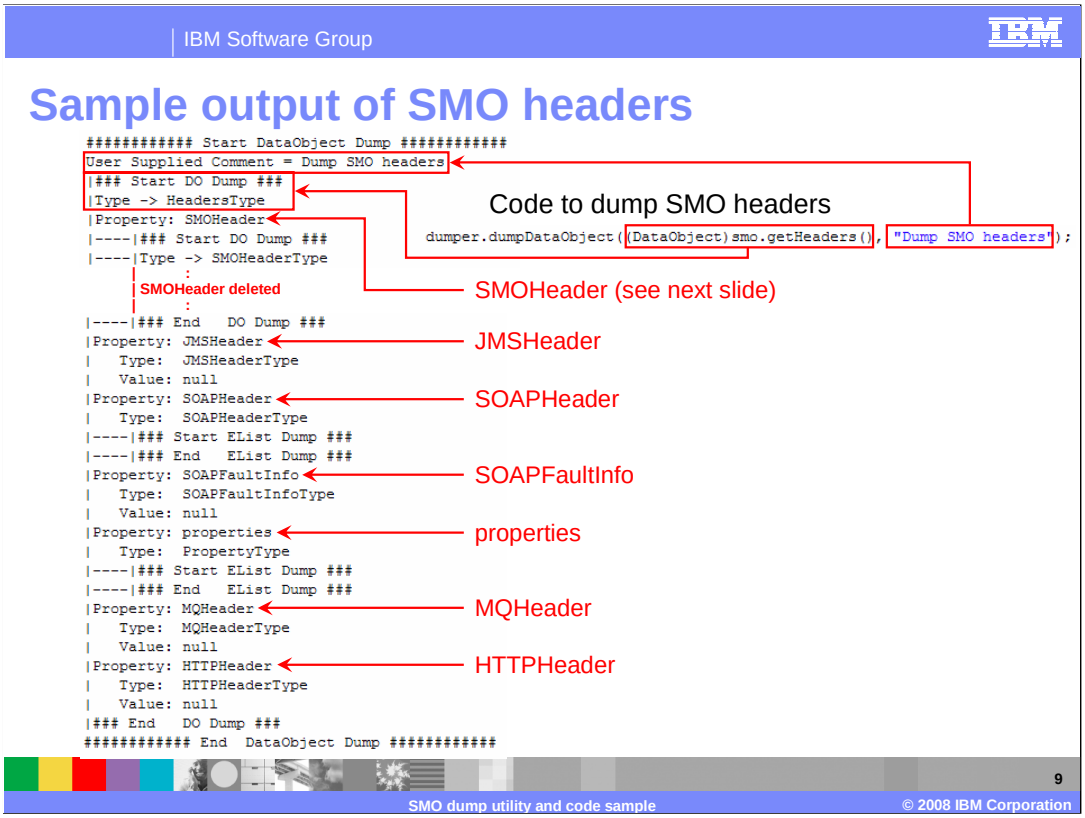
Looking at the example, notice that the second parameter to dumpDataObject supplies the comment “Dump entire SMO context”. This comment is placed at the beginning of the dump.

Notice the first parameter is using the ServiceMessageObject APIs to get the SMO context and is then casting it to a DataObject. In the dump the DataObject is shown to be of type ContextType.

The context in the SMO has five properties. The first two are the correlation context and the transient context. Notice that the transient context in this flow was configured with a business object but that the correlation context was not. Also notice the last property, which is the shared context. In this flow it was not configured with a business object.

The property failInfo is not populated in this case but it is when this is a flow from a fail terminal.

The primitiveContext property is an area intended to be used by mediation primitive implementations. It is populated if there is an endpoint lookup or fan out primitive in the flow.

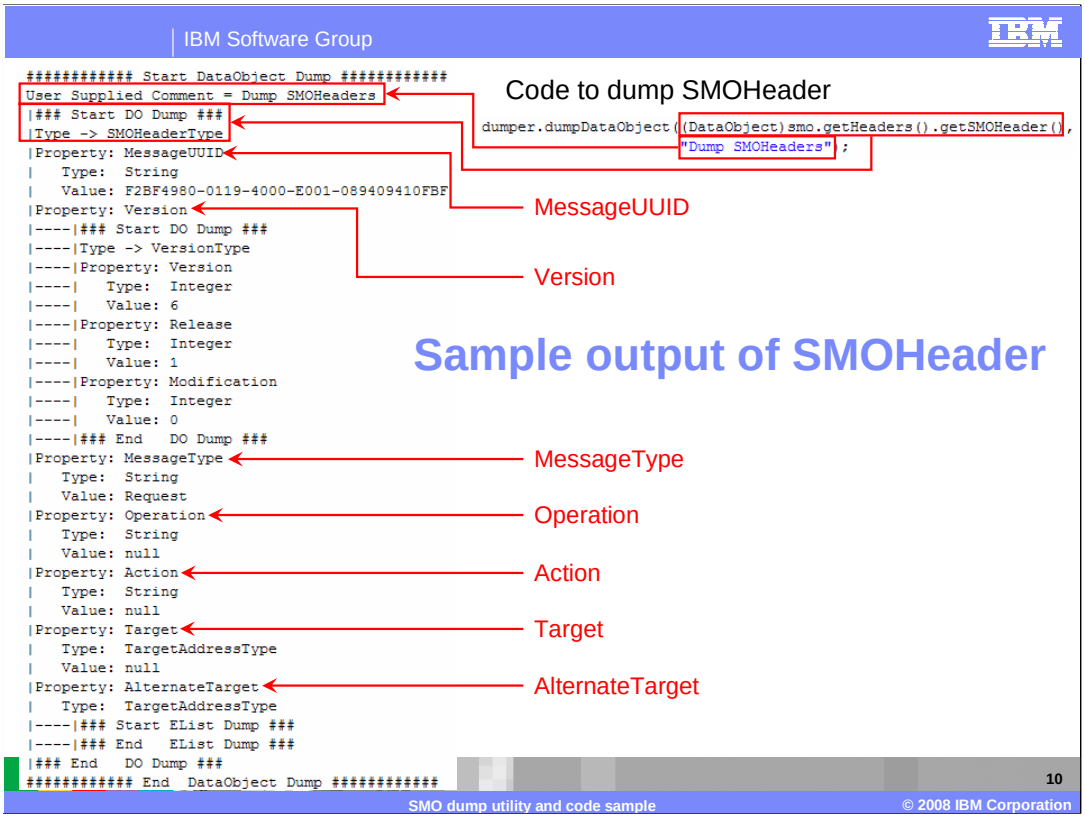


There are two slides used to show the SMO headers. This slide shows the entire SMO headers section, except that the details of the SMOHeader have been removed. They are shown on the next slide because there is not enough room to show the entire headers section on one slide.

Similar to the example on the previous slide, this dump has a comment identifying what it is for. It states “Dump SMO headers,” which shows at the beginning of the dump.

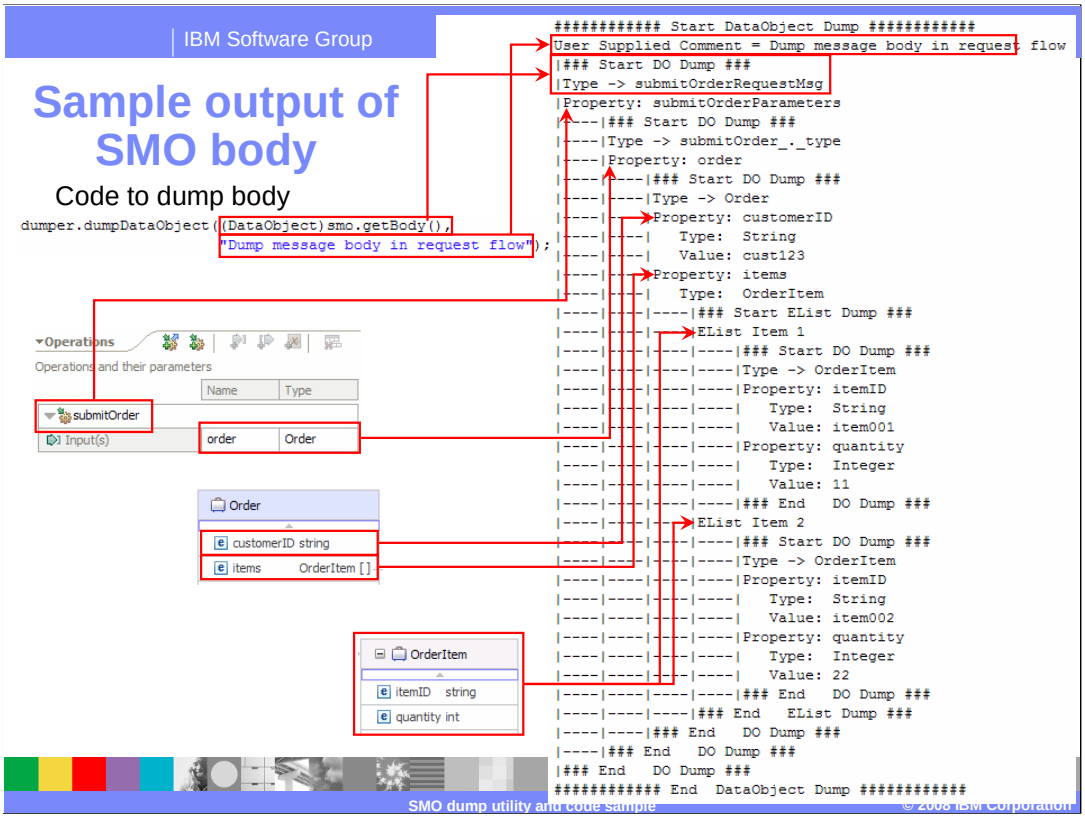
The first parameter to the dump uses the ServiceMessageObject APIs to get the headers section of the SMO. The DataObject shown in the dump is of type HeadersType.

In this particular example, none of the headers that are shown are populated. These are the JMSHeader, SOAPHeader, SOAPFaultInfo, properties, MQHeader and HTTPHeader. These are populated with the appropriate information based on the type of SCA binding that is configured on the export or import that initiated this flow. Examination of what is contained in the headers is one of the very useful applications of the dump utility.



This slide shows the SMOHeader contained within the SMO headers. In the invocation of the dump you can see that the ServiceMessageObject APIs are used to drill down to the SMOHeader, doing a getHeaders call followed by a getSMOHeader call. In the dump output it shows that the dump is for a DataObject of SMOHeaderType. The dump also contains the comment passed to it as the second parameter.

The SMOHeader is always populated with useful values in the MessageUUID, Version and MessageType properties, but the Operation and Action properties are typically null. The Target and AlternateTarget properties are populated when you have logic in the flow to set them for use with dynamic endpoint requests.



In this example, the ServiceMessageObject APIs are used to dump the body of the SMO on the request flow for a submitOrder operation. Notice that the DataObject type is submitOrderRequestMsg, which corresponds with the custom mediation primitive's input terminal message type. The submitOrder operation definition is shown on the slide. It takes an input parameter called order of type Order. The business object definition of Order is shown. Notice that it contains a field called items, which is an array of OrderItem objects, the definition for which is also shown. Examine the slide to see how the operation, its input parameter and the complex types it contains are shown in the DataObject dump.

Section

Dump utility code

Now that you are familiar with the output of the dump utility, this section examines the code. This provides some good sample code for showing how to traverse a DataObject.

Code for DumpDataObject – public methods

- dumpDataObject(DataObject inputDO, String comment)
  - Public method to call when including a comment
  - Surrounds dump of the inputDO with starting and ending comments

```
public void dumpDataObject(DataObject inputDO, String comment) {
    System.out.println(" ");
    System.out.println("############ Start DataObject Dump ############");
    if (comment != null)
        System.out.println("User Supplied Comment = " + comment);
    dumpDO(inputDO, "|");
    System.out.println("############ End  DataObject Dump ############");
    System.out.println(" ");
}
```

dumpDO
internal method that does the actual work

- dumpDataObject(DataObject inputDO)
  - Public method to call without providing a comment
  - Calls other dumpDataObject method with a null comment

```
public void dumpDataObject(DataObject inputDO) {
    dumpDataObject(inputDO, null);
}
```


This slide shows the two public methods that you can call to do a DataObject dump. They are both called dumpDataObject and take a DataObject as input. The one on the top of the slide also takes a second parameter containing a comment to put into the dump.

The method on the top of the slide is very straight forward. It prints start and end of dump messages, in between which is a call to a private method named dumpDO, which performs the actual dump of the DataObject.

The method on the bottom of the slide allows you to call for a DataObject dump without supplying a comment. It is implemented by calling the other method and supplying a comment string of null.

Code for DumpDataObject – private methods

- dumpDO(DataObject inputDO, String indent)
  - Recursive method, dumps the DataObject with indentation for nesting

```
private void dumpDO(DataObject inputDO, String indent) {
    if (inputDO != null) {
        System.out.println(indent + "### Start DO Dump ###");
        System.out.println(indent + "Type -> "
                + inputDO.getType().getName());
        List properties = inputDO.getType().getProperties();
        for (Iterator j = properties.iterator(); j.hasNext();) {
            Property p = (Property) j.next();
            System.out.println(indent + "Property: " + p.getName());
            String pName = p.getName();
            Object pValue = inputDO.get(pName);
            if (DataObject.class.isInstance(pValue)) {
                dumpDO((DataObject) pValue, "|----" + indent);
            } else  if (EObjectContainmentEList.class.isInstance(pValue)) {
                System.out.println(indent + "   Type:  " + p.getType().getName());
                dumpEList((EObjectContainmentEList) pValue, "|----" + indent);
            } else {
                System.out.println(indent + "   Type:  " + p.getType().getName());
                System.out.println(indent + "   Value: " + pValue);
            }
        }
        System.out.println(indent + "### End   DO Dump ###");
    }
    return;
}
```

indicate start of dump at this nesting level

print type of DataObject

get list of its properties

iterate, for each property

print property name

get property value

DataObject?, recursively dump

EList?, print type & call dumpEList to iteratively dump the list

all other properties, print type and value

indicate end of dump at this nesting level

This slide shows the dumpDO method, which does the real work for dumping a DataObject. As input it takes the DataObject to dump and an indentation string that is added to the front of each line of output. This enables the visual nesting of DataObjects when the routine is called recursively to dump DataObjects within DataObjects.

The routine allows for the possibility that the DataObject passed in is null, in which case it does nothing.

When dumping a DataObject, it prints a line indicating that this is the start of the dump and then prints the type of the DataObject. Since a DataObject is composed of properties, a list of the contained properties is then obtained. An iterator is then used to iterate through the list of properties so that each property can be dumped. For each property, the name of the property is printed and the type of the property value is determined. If the property value is a DataObject, this operation is recursively called to dump it. If the property value is an EList, a method to dump the contents of the list is called. This method is examined on the next slide. If the property is neither a DataObject nor an EList, the type and value are printed. After all the properties are dumped an end of dump indicator is printed.

Code for DumpDataObject – private methods

- dumpEList(EObjectContainmentEList eList, String indent)
  - Recursive method, dumps the list with indentation for nesting
  - EObjectContainmentEList part of Eclipse Modeling Framework (EMF)
    - org.eclipse.emf.ecore.util.EObjectContainmentEList

```
private void dumpEList(EObjectContainmentEList eList, String indent) {
    if (eList != null) {
        System.out.println(indent + "### Start EList Dump ###");
        int count = 1;
        for (Iterator k = eList.iterator(); k.hasNext();) {
            Object item = k.next();
            System.out.println(indent + "EList Item " + count++);
            if (DataObject.class.isInstance(item)) {
                dumpDO((DataObject) item, "|----" + indent);
            } else if (EObjectContainmentEList.class.isInstance(item)) {
                dumpEList((EObjectContainmentEList) item, "|----" + indent);
            } else {
                System.out.println(indent + "   Item: " + item);
            }
        }
        System.out.println(indent + "### End   EList Dump ###");
    }
    return;
}
```

indicate start of dump at this nesting level

iterate, for each list item

print index for this item in the list

DataObject?, call dumpDO to iteratively dump the properties

EList?, recursively dump

print all other list items

indicate end of dump at this nesting level

The method on this slide is used to dump an EList, or more specifically an EObjectContainmentEList. Similar to the dumpDO routine, it can be called recursively to handle an EList within an EList. It also has support for the indentation string, allowing the output to have a visual indication of the nesting.

The EObjectContainmentEList class is part of the Eclipse Modeling Framework (EMF) and is defined in the package org.eclipse.emf.core.util.

If the EList passed in is null, the method returns immediately without any processing.

The method prints out start and end of EList dump messages around the dump of the list. The list is processed using an iterator to dump each item in the list, with each item being identified by an index number. If an item is a DataObject, the dumpDO method is called to dump it and if it is another EList, this method is called recursively. Otherwise, the item itself is printed.

Summary

- Introduced the DataObject dump utility
  - Provided the code in a project interchange file
  - Described how to use it
  - Showed sample dump output
  - Described the dump utility code
- Provided you with a tool you can use for:
  - Learning to gain a better understanding of the SMO
  - Examining the SMO during development and debug

In summary, this presentation provided you with a project interchange file that you can use to dump DataObjects. How to use the utility was described, along with sample output showing the results from dumping various sections of the SMO. The code used to perform the dump was then explained.

The dump utility has the potential to be useful to you. It can be used to gain a better understand of the structure and content of the SMO by dumping and examining the SMO for various different mediation flows. It can also be quite useful during the development and debugging of new mediation flows.

Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send e-mail feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_WBIV61_SMOCodeSample.ppt

This module is also available in PDF format at: ../WBIV61_SMOCodeSample.pdf

You can help improve the quality of IBM Education Assistant content by providing feedback.

Trademarks, copyrights, and disclaimers

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM WebSphere

A current list of other IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

Java, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This document could include technical inaccuracies or typographical errors. IBM may make improvements or changes in the products or programs described herein at any time without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business. Any reference to an IBM Program Product in this document is not intended to state or imply that only that program product may be used. Any functionally equivalent program, that does not infringe IBM's intellectual property rights, may be used instead.

Information is provided "AS IS" without warranty of any kind. THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted, if at all, according to the terms and conditions of the agreements (for example, IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products.

IBM makes no representations or warranties, express or implied, regarding non-IBM products and services.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. The actual throughput or performance that any user will experience will vary depending upon 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 throughput or performance improvements equivalent to the ratios stated here.