



IBM Software Group

**WebSphere® Enterprise Service Bus V6.0.2**  
**WebSphere Process Server V6.0.2**  
**WebSphere Integration Developer V6.0.2**

***Custom mediation primitive***



@business on demand.

© 2007 IBM Corporation  
Updated April 13, 2007

This presentation provides a detailed look at the Custom Mediation primitive.

## Goals

- Understand the custom mediation primitive details



Custom mediation

- ▶ Overview of function
- ▶ Use of terminals
- ▶ Definition of properties
- ▶ Structure of primitive and mediation module assembly
- ▶ Error handling
- ▶ Examine custom code samples

2

Custom mediation primitive

© 2007 IBM Corporation

The goal of this presentation is to provide you with a full understanding of the custom mediation primitive.

The presentation assumes that you are already familiar with the material presented in the **Mediation primitive common details** presentation and the **Common details – Promoted properties** presentation. These two presentations serve as a base for understanding mediation primitives in general.

In this presentation, an overview of the custom mediation primitive is provided along with information about the primitive's use of terminals and its properties. The structure of the custom mediation primitive in relation to the mediation module assembly is then described. Finally, the error handling characteristics are presented and example usages of a custom mediation primitive are provided.

## Overview of function

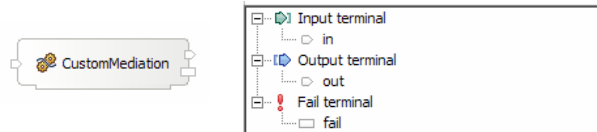
- Enables use of custom mediation logic
  - ▶ Used when no built-in primitive provides a needed function
- Logic implemented in Java™ and defined as:
  - ▶ Visual snippet
  - ▶ Java snippet
  - ▶ Invoke of a reference on mediation flow component
    - Normal service component architecture (SCA) reference
    - Reference wired in the SCA assembly diagram to either
      - Java component
      - Import (normal import, only knows target interface and protocol, not implementation type)
- Similar to other mediation primitives in behavior
  - ▶ Usage of terminals, wiring, exception processing

The custom mediation primitive enables you to define your own custom mediation logic for use when the built-in primitives do not provide the needed functionality. The logic is implemented in Java and there are three approaches to doing this. The custom mediation can be implemented as a visual snippet, a Java snippet or as an invoke of a Reference defined on the Mediation Flow Component. The Reference is a normal Service Oriented Architecture Reference, so it can be wired in the assembly diagram to an SCA Java component or to an Import.

The general behavior of a custom mediation primitive is similar to that of the built-in primitives, relative to its use of terminals, wiring within the mediation flow and handling of exceptions.

## Terminals

- Terminals:
  - ▶ Input terminal
  - ▶ One output terminal
  - ▶ Fail terminal
- Message type of input and output terminal
  - Same type – for manipulation of values within a message
  - Different type – for changing format of the message body



The custom mediation primitive has one input terminal, one output terminal and a fail terminal. The message type of the output terminal can be for the same message type as the input terminal or for a different message type. When the message types are different, the Java code in the custom mediation must modify the structure of the body of the message to conform with the output terminal type. Shown here is a custom mediation primitive with its terminals and the terminals as seen in the properties view.

IBM Software Group

## Properties

Custom Mediation : CustomMediation

Implementation:  Visual  Java  Invoke

Root: /body

Signature: commonj.sdo.DataObject execute(commonj.sdo.DataObject input1)

- Implementation
  - ▶ Select if a Visual snippet, Java snippet or Invoke of SCA Reference
  - ▶ Details panel unique for each implementation type
- Root
  - ▶ The portion of the Service Message Object (SMO) to be passed
  - ▶ Valid values are only: / and **/body**
- Signature
  - ▶ This is only a comment providing the signature of the call
  - ▶ input1 parameter
    - DataObject matching Root property and message type of the in terminal
  - ▶ returns
    - DataObject matching Root property and message type of the out terminal
    - Implementation responsible for returning appropriately constructed DataObject

5

Custom mediation primitive © 2007 IBM Corporation

The screen capture at the top of the slide shows the Details panel of the Properties view for a custom mediation primitive.

The **Implementation** property is a set of radio buttons used to define which of the three implementation types is used to implement this custom mediation. The choices are visual snippet, Java snippet or the invocation of an SCA Reference defined on the Mediation Flow Component. The contents of this panel varies depending upon which of these implementation types is chosen.


The **Root** property defines what portion of the Service Message Object (SMO) is passed to the operation. There are only two valid values for root, / (slash), which indicates to pass the entire SMO and **/body**, which indicates to pass only the body, the payload of the message,

The **Signature** is not really a property. It is a comment that documents the Java signature that is used to call the custom mediation. The input parameter, called input1, is a DataObject. Its contents are defined by the combination of the value of the Root property and the message type of the custom mediation's in terminal. The return value is also a DataObject. Its contents are defined by the combination of the value of the Root property and the message type of the custom mediation's out terminal. It is your responsibility to code the custom mediation so that an appropriately constructed DataObject is returned.

IBM Software Group IBM

## Properties for snippets

- Java snippet properties



Show Me

Implementation:  Visual  Java  Invoke


Root: /body

Signature: commonj.sdo.DataObject execute(commonj.sdo.DataObject input1)

```
System.out.println("The symbol parameter is: " + input1.getString("symbol"));
return input1;
```

- Visual snippet properties

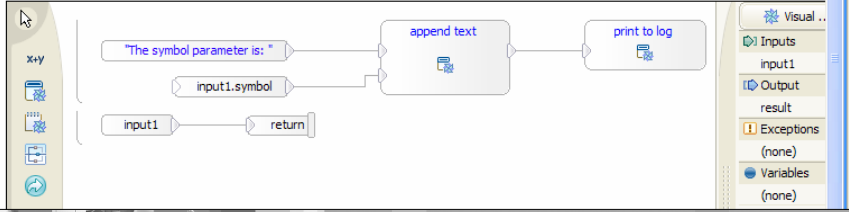


Show Me

Implementation:  Visual  Java  Invoke

Root: /body

Signature: commonj.sdo.DataObject execute(commonj.sdo.DataObject input1)



6

Custom mediation primitive © 2007 IBM Corporation

This slide takes a look at the Details panel of the Properties view for the two types of snippet implementations. The Java snippet implementation provides a text entry box which behaves like a mini Java editor, providing features such as content assist and context sensitive highlighting.


The Visual snippet implementation provides a visual snippet editor in which you can create the logic for your custom mediation. It consists of a palette, canvas and tray. In the center is the canvas, which is where you build up your logic through connecting visual constructs that provide various different capabilities. The shaded area to the left of the canvas is the palette. It contains icons representing the various constructs used to define the logic of your snippet. You click on these icons, drop them onto the canvas and wire them together to build your snippet. On the right is the tray, from which you can drag the input1 DataObject and drop it onto the canvas.

These two snippets are performing the same function. They write a message to the Console, or more specifically, to the SystemOut.log file. What gets written is a string which states, "The symbol parameter is: ", followed by the string value for symbol. Symbol is a String input parameter for the operation being mediated. The snippets then return the input1 DataObject unchanged.

Notice the Show Me icons in the slide. These are links to demonstrations illustrating how to build a custom mediation for each of the implementation types.

IBM Software Group IBM

## Properties for invoke



Show Me

Properties x Problems Servers

**Custom Mediation : CustomMediation**

Implementation:  Visual  Java  Invoke

Reference:

Operation:

Root: /body

- Reference
  - ▶ Reference on Mediation Flow Component in assembly diagram
  - ▶ The Reference needs to be wired to a Java component or an Import
  - ▶ Drop down box lets you select existing Reference or open a dialog to create a new one
  - ▶ Dialog for new Reference lets you use existing Interface or generate a new one
- Operation
  - ▶ Name of the operation to call on the Interface defined for the Reference
  - ▶ The operation can have any name
  - ▶ The signature of the operation must take as input and return a DataObject
  - ▶ The rules for the DataObject are the same as describe on a previous slide
- Root
  - ▶ As described on a previous slide

7

Custom mediation primitive © 2007 IBM Corporation

This slide takes a look at the Details panel of the Properties view for an Invoke implementation of a custom mediation.

The **Reference** property provides the name of a reference on the Mediation Flow Component in the assembly diagram. It is this reference through which the invoke of the custom logic occurs. In the assembly diagram, this reference needs to be wired to a Java component or to an import. When configuring this property, there is a drop down box from which you can select an existing reference or choose to create a new reference. If you chose to create a new reference, there is a dialog box that allows you to create one using an existing interface or to generate a new interface.


The **Operation** property is the name of the operation which is invoked. This must be an operation defined on the interface associated with the reference. It can be selected using a drop down box. The name of the operation does not matter, but the signature does. It must take a DataObject as input and return a DataObject. The contents of each DataObject is defined by the root property and the message type, as was described on a previous slide.

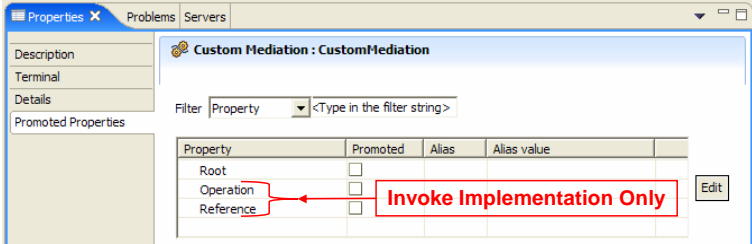
The **Root** property is the same as was previously described. It is either a / (slash) or /body.

The creation of a custom mediation with an invoke implementation involves not only defining the properties, but also updating the SCA assembly diagram for the Mediation Module. When you are adding a custom mediation to your flow it is important to understand the steps needed to do this. There are slides later in this presentation describing the steps in detail and you can also use the Show Me icon on the slide to link to a demonstration.

IBM Software Group IBM

## Promoted properties





| Property  | Promoted                 | Alias | Alias value |
|-----------|--------------------------|-------|-------------|
| Root      | <input type="checkbox"/> |       |             |
| Operation | <input type="checkbox"/> |       |             |
| Reference | <input type="checkbox"/> |       |             |

- Promotable
  - ▶ Root
  - ▶ Operation
  - ▶ Reference

Custom mediation primitive 8

© 2007 IBM Corporation

All of the properties for the custom mediation primitive are promotable.

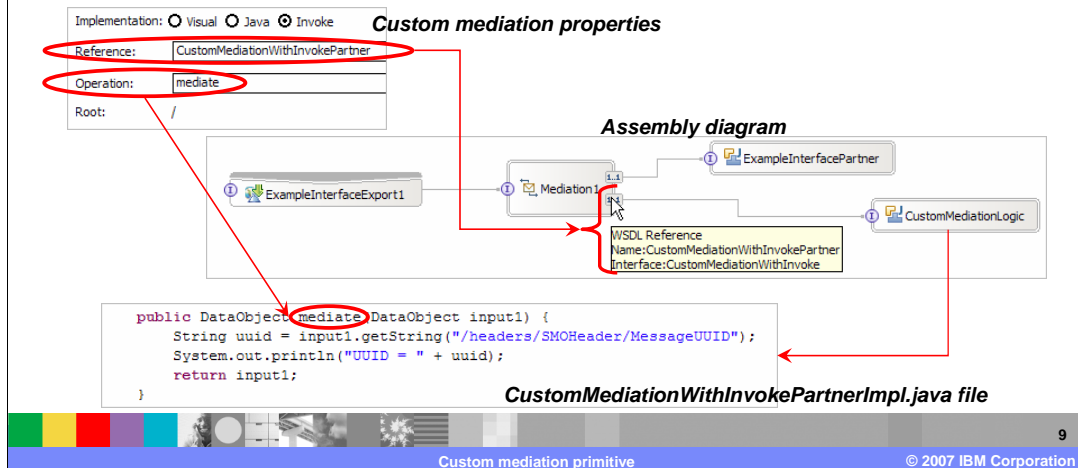
The **Root** property applies to all three implementation types. Care should be taken if promoting this property since almost all custom mediations have code that is dependent upon whether the root is specified as / (slash) or /body. Changing the root dynamically could cause the custom mediation logic to fail.

The next two properties, **Operation** and **Reference**, apply only to the invoke implementation type. Promoting these properties could enable the modification of the flow logic by having the custom mediation run different code.



## Complete picture for invoke

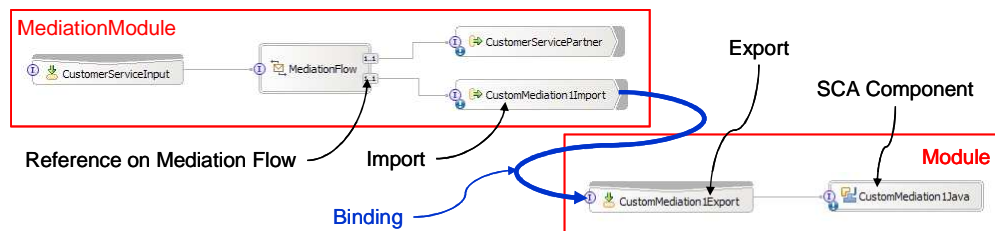
- Custom mediation with invoke implementation
  - ▶ Custom mediation defines a reference and operation to call
  - ▶ Mediation flow component contains the reference
  - ▶ Reference wired to Java component with operation implementation



The complete picture of a custom mediation using the invoke implementation is shown on this slide. In the upper left is the Details panel of the Properties view, showing the Reference and Operation properties. In the center is the assembly diagram, showing the Mediation Flow Component that contains the Reference identified by the properties. It is wired to a Java component whose implementation is shown in the lower part of the slide and whose operation is the one identified by the custom mediation properties.

## Using an import

- Alternative assembly diagram with import
  - ▶ Package complex custom mediation logic as a service
  - ▶ Use that service from many different mediation modules
  - ▶ Implement the service a non-Java SCA component



There is an alternative structure where the Reference in the assembly diagram is wired to an Import rather than a Java component. The properties in the custom mediation would be the same, identifying the Reference and Operation used to call the custom logic. The only difference is on the assembly diagram with the Import replacing the Java Component.

The main reason for following this approach is to be able to package the custom mediation logic as a service. This allows it to be called from many different mediation modules and makes it possible to implement the logic as something other than a Java component.

## Invoke implementation steps

- Creating “invoke” custom primitives requires multiple steps
  - ▶ Steps vary depending upon starting state
    - Is there an interface/operation already defined
    - Does the component already exist on the assembly
    - Is the reference defined on the mediation flow component
    - Is the reference wired to the component
- Steps to create from scratch (when nothing currently exists)
  - ▶ Drop a custom mediation primitive on the mediation flow editor
  - ▶ Wire the custom primitive
    - This ensures the in/out terminal message types are defined
  - ▶ In the properties/details panel
    - Select the “invoke” implementation type
    - In the drop down for “reference” select “add a new reference”
    - The “add reference” dialog opens
  - ▶ (Continue) →

The steps needed to configure your custom mediation with an invoke implementation vary depending upon the starting state of your implementation. Things that influence the steps needed would include whether or not there is an existing interface and operation you are utilizing and whether the Java component or import already exist on the assembly diagram. Other factors include if the reference has already been defined on the Mediation Flow Component and if the reference has already been wired.

This slide and the next bring you through the specific steps needed when creating a custom mediation from scratch when nothing already exists. This is probably the most common case.

You start by dropping the custom mediation primitive on the Mediation Flow editor canvas. Wiring the custom mediation next ensures that the in and out terminal message types are defined, as the message types are implied by the terminals they are wired to. In the Details panel of the Properties view, select the Invoke implementation type, and use the drop down box for Reference and select “Add a new reference” which opens the Add Reference dialog.

The remaining steps are found on the next slide.

## Tool Support – Invoke implementation steps

- ▶ In the “Add Reference” dialog
  - Select “Generate a new interface for this reference”
  - Accept defaults or change values for Module, Interface & Operation names
  - Select either “/” or “/body” for the value of root
  - Hit OK
- ▶ Save the mediation flow
- ▶ In the Assembly diagram
  - Using the Mediation Flow Component pop-up menu select:
    - “Synchronize Interfaces and References” → “from Implementation”
    - A new Reference appears on the Mediation Flow Component
  - Using the pop-up menu for the new Reference select:
    - “Wire References to New” → “Components”
    - A new SCA component without implementation appears in the Assembly diagram
  - Using the pop-up menu for the new component select:
    - “Generate Implementation...” → “Java”
    - The Java editor opens containing a new Java code skeleton
- ▶ Implement the custom logic in the Java code

Using the Add Reference dialog, select “Generate a new interface for this reference”. You can either accept the defaults or change the values presented for Module, Interface and Operation names. Then select the value for Root and hit OK to end that dialog. The mediation flow should now be saved before switching to the assembly diagram.

On the assembly diagram, get a pop-up menu for the Mediation Flow Component. Select “Synchronize Interfaces and References”, followed by “from Implementation”, which causes a new Reference to appear on the Mediation Flow Component. Now get a pop-up menu for the new Reference and select “Wire References to New”, followed by “Components”, which creates a new SCA component wired to the Reference. Now use the pop-up menu for the new SCA component and select “Generate Implementation”, followed by “Java”. This opens the Java editor with a code skeleton ready for you to use to implement the custom logic.

## Error processing

- **MediationRuntimeException thrown for**
  - ▶ No operation or service reference specified in the properties
  - ▶ No matching reference exists on the Mediation Flow Component
- **MediationRuntimeException (Fail terminal flow)**
  - ▶ Note difference in behavior
    - Unlike most MediationRuntimeExceptions, these fire the Fail terminal if wired
  - ▶ The reference on the Mediation Flow Component is not wired
  - ▶ Custom code returns null
    - Must return a DataObject
    - Beware, default code is generated with **returns null;**
- **Any exception throw by the custom Java code**
  - ▶ Thrown as is, not wrapped by any MediationXxxxxException type
  - ▶ If the Fail terminal is wired that flow is followed

The error processing details and considerations are examined in this slide.

A `MediationRuntimeException` will be thrown when the operation or service reference name has not been specified in the properties or if the specified reference does not exist on the Mediation Flow Component.

A `MediationRuntimeException` can also occur for other conditions that are detected when processing the custom mediation primitive. Unlike most other `MediationRuntimeExceptions`, these will result in the Fail terminal flow being followed if the Fail terminal is wired. One of the conditions where this can occur is when the reference on the Mediation Flow Component is not wired. Another cause of this is when the custom code returns a null rather than a `DataObject`. This could be a common mistake because the generated code stub returns null and must therefore be changed to return a `DataObject`.

The custom code can throw any exception and it is not wrapped by any of the mediation specific exceptions. If the Fail terminal is wired, that flow is followed, otherwise the exception is thrown and the mediation flow is terminated.

## Custom code example

- A typical scenario
  - ▶ Custom mediation used to construct a key before a database lookup
  - ▶ Data from the body of the message is manipulated to construct key
  - ▶ Key value is placed into the transient context
- Scenario for this code example:
  - ▶ Database lookup needs first two digits of the customer ID as a key

```
public DataObject mediate(DataObject input1) {
    // Get the Service Message Object
    ServiceMessageObject smo = (ServiceMessageObject)input1;
    // Get the transient context
    ContextType context = smo.getContext();
    DataObject transientContext = (DataObject)context.getTransient();
    // Get the customerID from the body
    DataObject body = (DataObject)smo.getBody();
    DataObject customerInfo = (DataObject)body.get("getCustomerInformation");
    String customerID = (String)customerInfo.get("customerID");
    // Move the prefix of the customerID to the transient context
    transientContext.setString("accountLocationID", customerID.substring(0,2));
    // Return the data object
    return (DataObject)smo;
}
```

14

Custom mediation primitive

© 2007 IBM Corporation

One of the typical scenarios for a custom mediation is constructing a key that can be used to perform a database lookup. In such a scenario, the custom mediation can access various data elements in the SMO and use them to construct a key value. That key value can then be placed into the transient context to be used by a subsequent database lookup. In this particular scenario, the first two digits of a customer number are extracted to be used as a key. Looking at the code, you see that the mediate operation takes a DataObject as input and returns a DataObject. In this example, the root property was specified as / (slash), indicating that the DataObject passed in and returned is the entire SMO. The first step is to cast the input parameter input1 to a ServiceMessageObject type and assign it to a ServiceMessageObject variable called smo. This variable can then be used to perform type safe SMO specific operations. The next step is to obtain the transient context, which is done in two steps. The getContext operation is performed on smo, which returns the context portion of the SMO and is assigned to a ContextType variable called context. Next, the getTransient operation is performed on context to obtain the transient context, which is assigned to a DataObject type called transientContext. The customerID must now be obtained from the body of the SMO. Using the smo variable, the getBody operation returns the body of the message, which is assigned to a DataObject called body. Using the generic get operation on body the getCustomerInformation property is obtained and similarly the customerID is then obtained from it. A substring of the first two digits of customerID is obtained and placed into the accountLocationID element of the transientContext using the generic setString operation. Finally the original SMO is returned with the transient context updated to contain the key value.

## Throwing an exception

### Scenario

- ▶ Check customer ID for validity
- ▶ Throw MediationBusinessException when invalid
- ▶ Exception contains user defined message insert

```
// Method signature has "throws" clause added
public DataObject mediate(DataObject input1) throws MediationBusinessException {
    // Get the customer ID
    DataObject body = (DataObject) input1.get("body");
    DataObject customerInfo = (DataObject) body.get("getCustomerInformation");
    String customerID = (String) customerInfo.get("customerID");
    // Perform whatever logic is needed to validate the customerID
    if (customerID.length() < 6) {
        // Throw exception if invalid
        String msg = "MyOwnErrorCode: CustomerID is too short, ID=" + customerID;
        throw new MediationBusinessException(msg);
    }
    // CustomerID is OK, return
    return input1;
}
```

### Resulting Log

```
[1/18/06 17:48:50:354 CST] 0000004b ExceptionUtil E CNTR0020E: EJB threw an unexpected (non-declared) exception during invocation of method
"transactionNotSupportedActivitySessionSupports" on bean "BeanId(CustomerRoutingMediationApp#CustomerRoutingMediationEJB.jar#Module, null)".
Exception data: com.ibm.wsspi.sibx.mediation.MediationBusinessException: MyOwnErrorCode: CustomerID is too short, ID=22222
at sca.component.java.impl.CustomMediation1PartnerImpl.mediate(CustomMediation1PartnerImpl.java:49)
```

Exception Type

Code where error occurred

User defined message

15

Custom mediation primitive

© 2007 IBM Corporation

The code example on this slide examines custom code that throws a `MediationBusinessException`, containing a user defined message insert, if the `customerID` does not pass a validity check.

The operation has the same signature as the previous example, except that it now has a `throws` clause to declare that the operation can throw a `MediationBusinessException`. In the code, the `customerID` is obtained from the SMO by drilling down to it using the generic `DataObject` get operation. First the body is obtained from `input1`, which contains the SMO. Then `getCustomerInformation` is obtained from the body and finally the `customerID` is obtained from the `getCustomerInformation`. Validation checking is then done on the `customerID`. In this example case, it is only checking to make sure its length is not less than six digits. If it is less than six digits the `MediationBusinessException` containing the user defined error message is thrown. Assuming that the Fail terminal of the custom mediation primitive has not been wired, the mediation flow is terminated and a log message written. An example of such a log is shown at the bottom of the slide. Notice that the log contains the exception type and the user defined error message. It also identifies the custom mediation code as the source of the exception.

## Summary

- Examined the custom mediation primitive details



Custom mediation

- ▶ Overview of function
- ▶ Use of terminals
- ▶ Definition of properties
- ▶ Structure of primitive and mediation module assembly
- ▶ Error handling
- ▶ Examine custom code samples

In summary, this presentation provided details regarding the custom mediation primitive, along with an overview of its function and information about the primitive's use of terminals and its properties. The structure of the primitive in relationship to the mediation module assembly was also described. Finally, information about error handling was presented and two code examples were examined.



## 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](#)



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

EJB, 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 NON-INFRINGEMENT. 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.

© Copyright International Business Machines Corporation 2007. All rights reserved.

Note to U.S. Government Users - Documentation related to restricted rights-Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract and IBM Corp.

