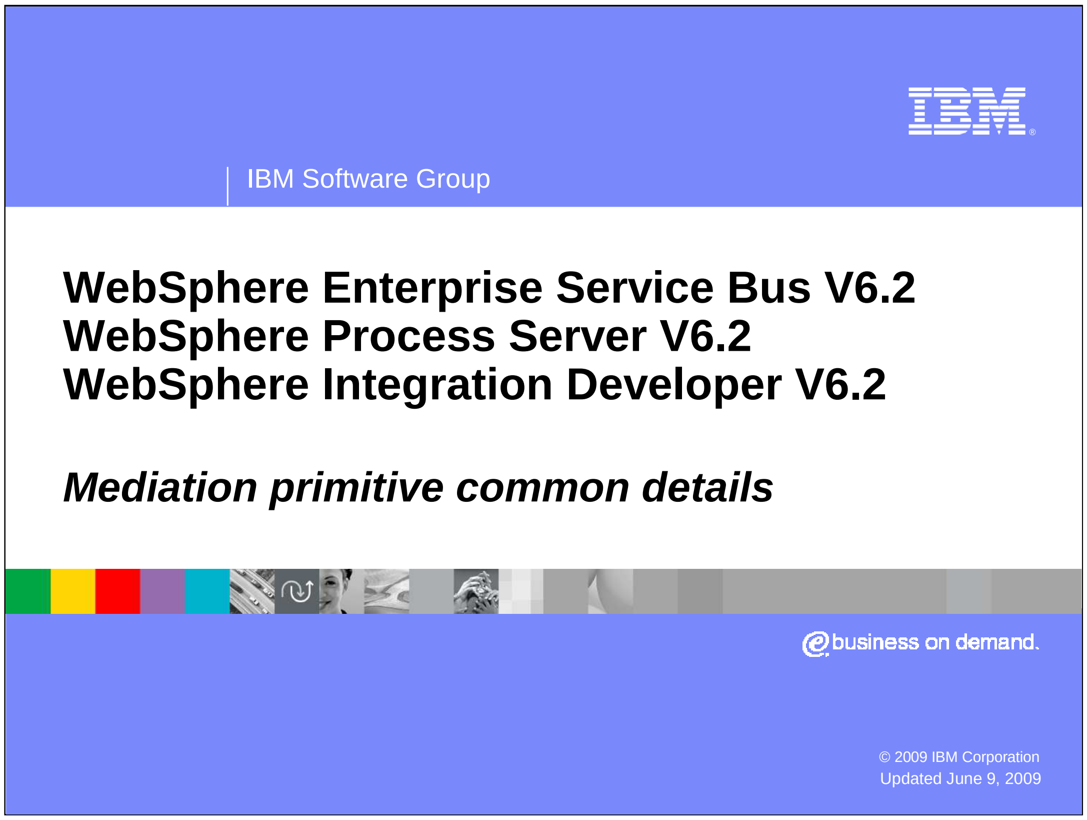


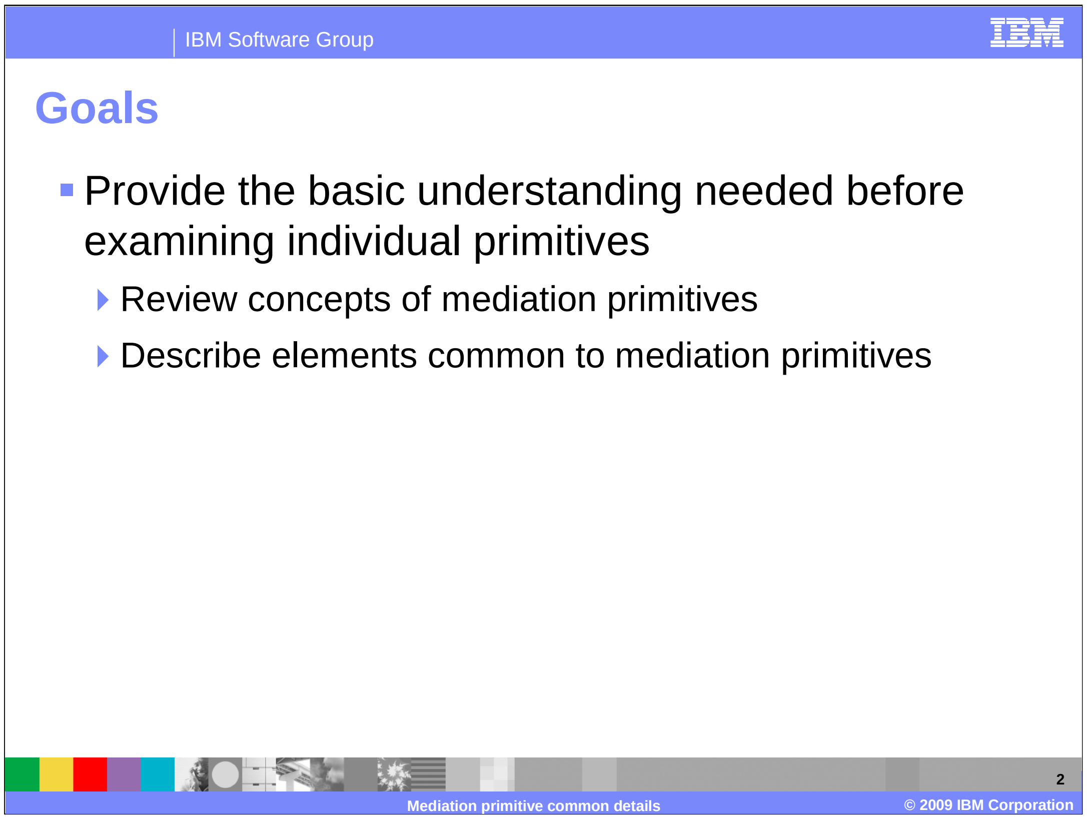
IBM Software Group

WebSphere Enterprise Service Bus V6.2
WebSphere Process Server V6.2
WebSphere Integration Developer V6.2

Mediation primitive common details

business on demand.

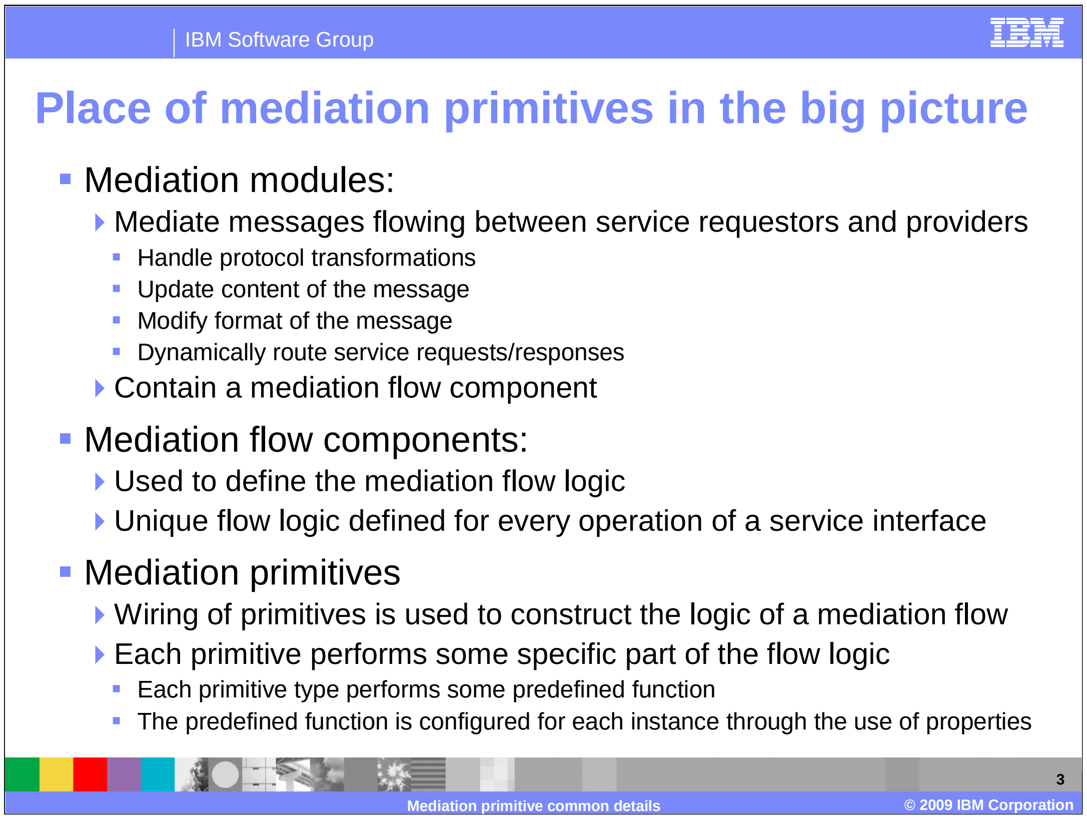
This presentation discusses those aspects of mediation primitives that are common to mediation primitives in general.



Goals

- Provide the basic understanding needed before examining individual primitives
  - Review concepts of mediation primitives
  - Describe elements common to mediation primitives

The goal of this presentation is to provide a basic understanding before discussing each of the individual mediation primitives. This is done by reviewing the concepts of mediation primitives and describing the elements that are common across many or all of the primitives.



IBM Software Group

Place of mediation primitives in the big picture

- Mediation modules:
  - Mediate messages flowing between service requestors and providers
    - Handle protocol transformations
    - Update content of the message
    - Modify format of the message
    - Dynamically route service requests/responses
  - Contain a mediation flow component
- Mediation flow components:
  - Used to define the mediation flow logic
  - Unique flow logic defined for every operation of a service interface
- Mediation primitives
  - Wiring of primitives is used to construct the logic of a mediation flow
  - Each primitive performs some specific part of the flow logic
    - Each primitive type performs some predefined function
    - The predefined function is configured for each instance through the use of properties

3

Mediation primitive common details © 2009 IBM Corporation

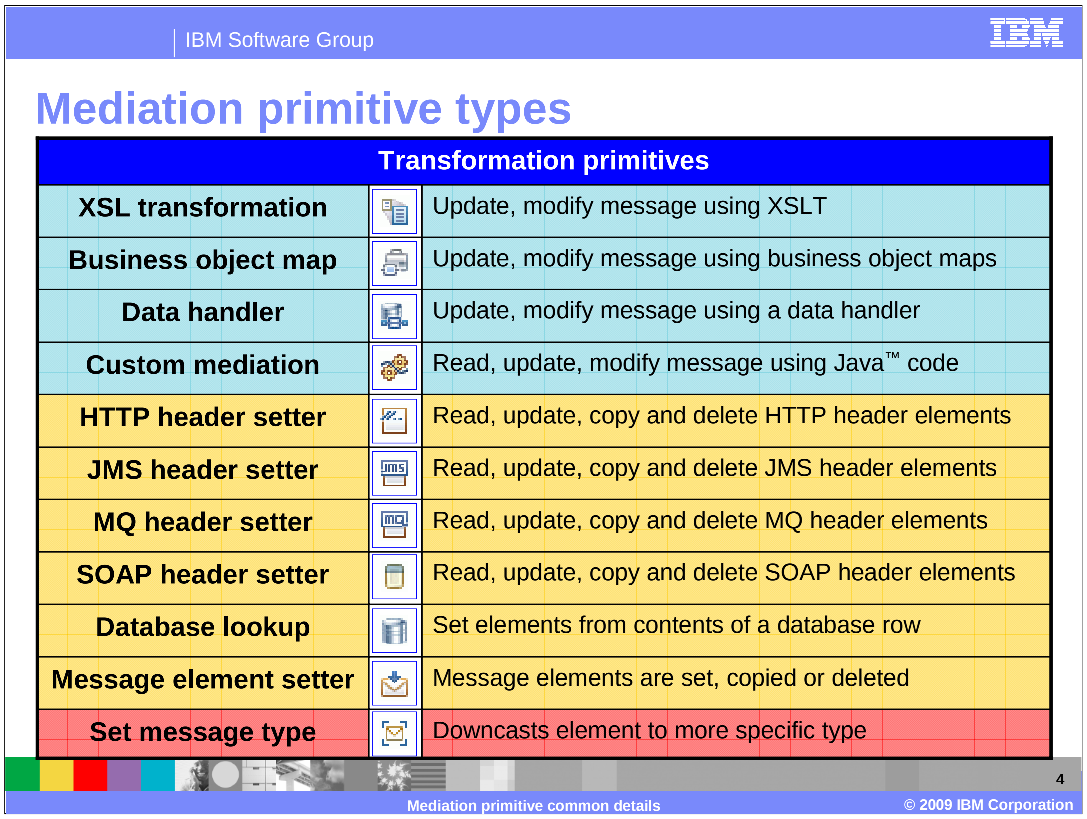
In order to understand mediation primitives, it is important to understand where they fit into the big picture of mediations.

Starting at the highest level of abstraction, there are mediation modules whose function is to mediate messages flowing between service requestors and service providers. Mediating a message might involve handling protocol transformations, updating the content of the message, modifying the format of the message or dynamically routing the message to an appropriate service provider.

The mediation module contains a mediation flow component, which is where the overall logic for the mediation is defined. For every operation defined on an input interface there is unique mediation flow logic for the operation’s request and response. The flow logic is defined within the mediation flow component using mediation primitives.

Each type of mediation primitive provides some predefined functional capability. Each instance of a mediation primitive has its predefined function configured through the use of properties. The property settings define how the primitive explicitly behaves in this specific instance. The overall logic of the flow is defined by wiring these configured mediation primitives together into a logical flow.

To summarize, the highest level of a mediation is the mediation module, which contains a mediation flow component, which contains mediation flows, which are composed of mediation primitives wired together to define the logic.



Mediation primitive types

Transformation primitives		
XSL transformation		Update, modify message using XSLT
Business object map		Update, modify message using business object maps
Data handler		Update, modify message using a data handler
Custom mediation		Read, update, modify message using Java™ code
HTTP header setter		Read, update, copy and delete HTTP header elements
JMS header setter		Read, update, copy and delete JMS header elements
MQ header setter		Read, update, copy and delete MQ header elements
SOAP header setter		Read, update, copy and delete SOAP header elements
Database lookup		Set elements from contents of a database row
Message element setter		Message elements are set, copied or deleted
Set message type		Downcasts element to more specific type

The next few slides introduce the various mediation primitive types. There are various ways that mediation primitive types can be organized. In WebSphere® Integration Developer, they are categorized into broad groupings based on the primary function of the primitive. They are categorized as transformation primitives, routing primitives, tracing primitives, error handling primitives and subflow primitives. That is the highest level of organization used on these slides. Another way to group primitives is according to their behavior and abilities for updating the service message object as it flows through the mediation. These slides use color coding to indicate this. Primitives that can change message type are shown in blue, those that update the SMO without changing the message type are shown in yellow and finally those that do not update the SMO are shown in red.

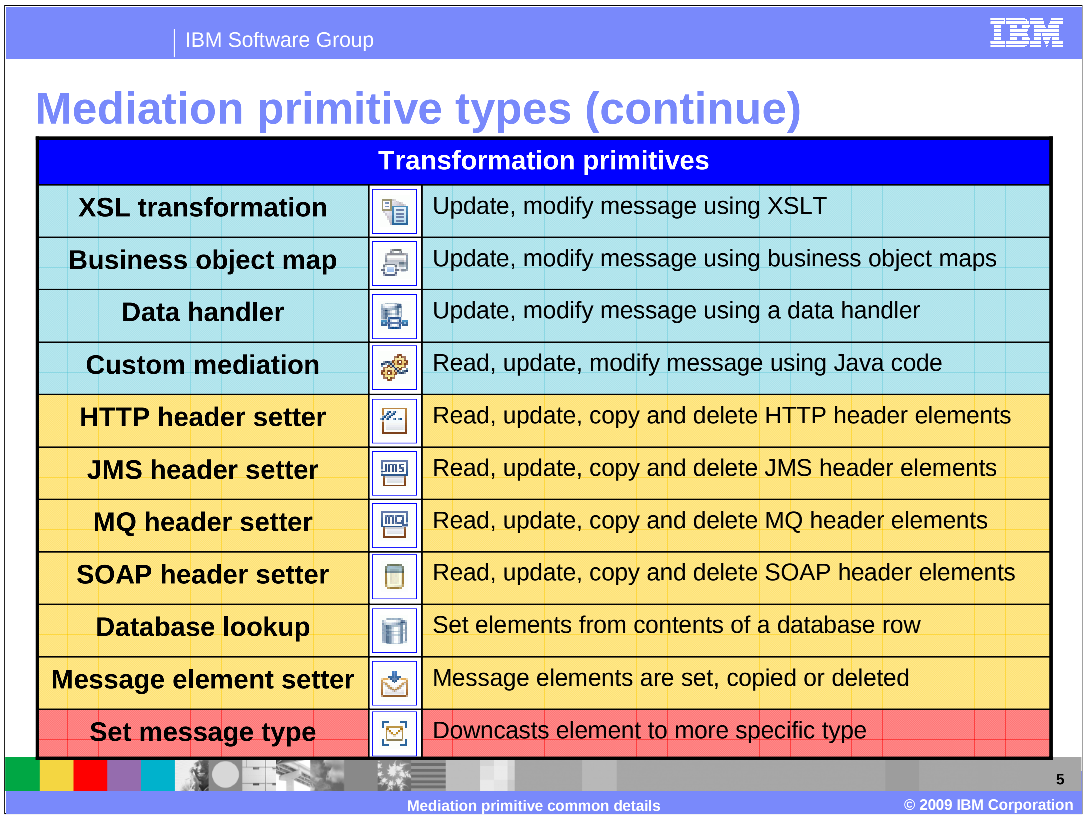
This slide looks at transformation primitives, starting with those that can change the message type. This is the truest sense of transformation, as the message type of the SMO is being transformed.

The **XSL transformation** primitive is used to update or transform messages using XSLT. This can be used to change the format of the message. An example of when the format needs to change is when the target provider has a different interface than the incoming message.

The **business object map** primitive is very similar in function to the XSL transformation primitive, but it uses business object maps rather than XSLT to perform the transformation. These are the same business object maps that are used in WebSphere Process Server within an interface map to perform parameter mapping. As a result, change logging and the relationship service are enabled.

The **data handler** primitive allows you to configure a data handler to transform the message. A data handler can be one supplied with the product or one that you provide, along with any configuration data that is needed by the data handler. These data handlers are the same as those used with SCA imports and exports.

The **custom mediation** primitive is used to do any message processing not covered by the other mediation primitives. This is done through Java code that can be written as a visual snippet or as a Java snippet.



IBM Software Group

IBM

Mediation primitive types (continue)

Transformation primitives		
XSL transformation		Update, modify message using XSLT
Business object map		Update, modify message using business object maps
Data handler		Update, modify message using a data handler
Custom mediation		Read, update, modify message using Java code
HTTP header setter		Read, update, copy and delete HTTP header elements
JMS header setter		Read, update, copy and delete JMS header elements
MQ header setter		Read, update, copy and delete MQ header elements
SOAP header setter		Read, update, copy and delete SOAP header elements
Database lookup		Set elements from contents of a database row
Message element setter		Message elements are set, copied or deleted
Set message type		Downcasts element to more specific type

5

Mediation primitive common details

© 2009 IBM Corporation

Continuing with the transformation primitives, the next grouping contains those primitives that update the contents of the SMO but do not actually transform the message type.

The first four, the header setter primitives, are all similar in nature. They provide you with an easy way to access protocol specific headers within the header section of the SMO. Although these headers can be accessed with other mediation primitives, these header setter primitives make it much easier because they are aware of how these protocol specific headers are structured. This allows you to focus on those aspects of the header you are interested in reading or updating, without having to be concerned with specifically how they are represented in the SMO.

The **HTTP header setter** primitive lets you access HTTP headers, which are grouped into control elements, standard elements and user elements.

The **JMS header setter** primitive lets you access JMS headers, which are grouped as standard elements and user elements.

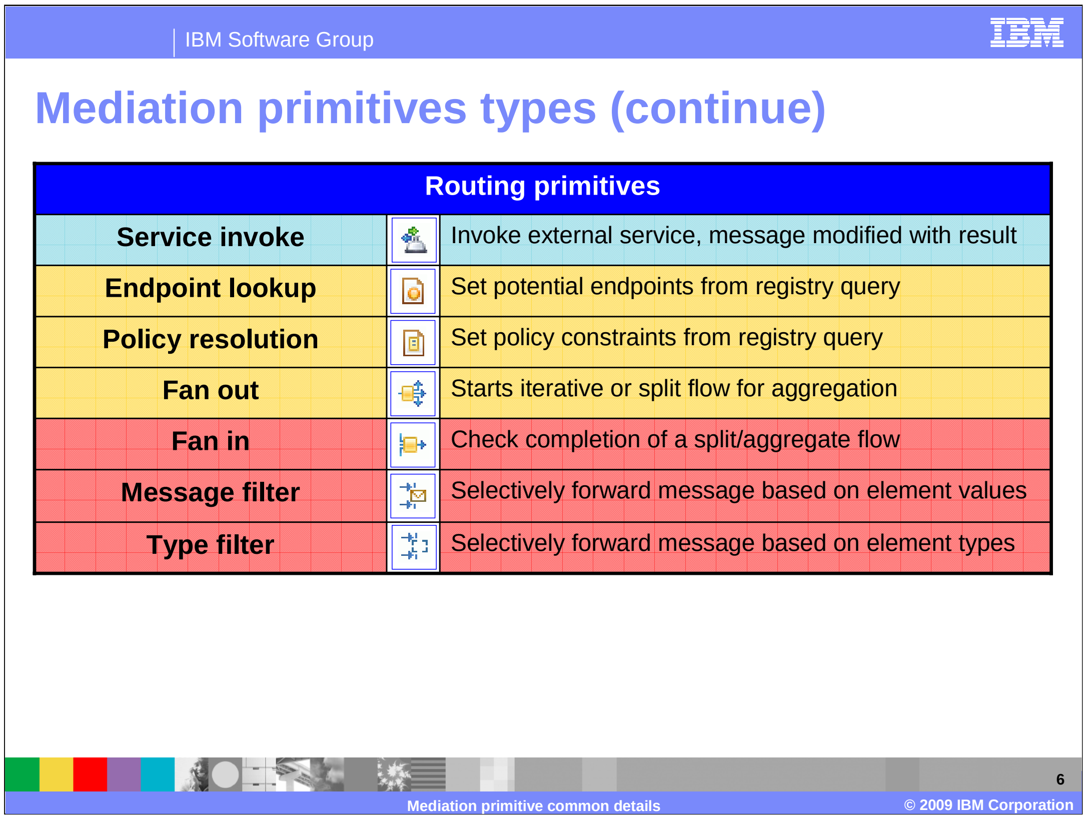
The **MQ header setter** primitive lets you access MQ headers. The headers exposed are the message descriptor, the CICS® bridge header, the IMS information header and the rules and formatting header two.

The **SOAP header setter** primitive lets you access SOAP headers. Any XSD containing an element can be a SOAP header. Some are provided by standards such as WS-Security and others are user defined.

The **database lookup** primitive is used to access information from a database and insert it into the message. A field in the message is used as a key for the database access and selected fields from the resulting database row can be placed into the message.

The **message element setter** primitive can be configured to update elements of the SMO. Individual elements can be set to a specific value or can have their value deleted. Individual elements or sub-trees in the SMO can be set by copying the values from another location in the SMO. Arrays in the SMO can have an element appended.

The **set message type** primitive does not update the SMO contents, but is used to augment the message type. It is used in conjunction with loosely typed elements in the SMO, such as an XSD:anyType, allowing it to be downcast to a more specific type. This enables tools, such as the XPath expression builder, to represent the element to you as the more specific type.



Mediation primitives types (continue)

Routing primitives		
Service invoke		Invoke external service, message modified with result
Endpoint lookup		Set potential endpoints from registry query
Policy resolution		Set policy constraints from registry query
Fan out		Starts iterative or split flow for aggregation
Fan in		Check completion of a split/aggregate flow
Message filter		Selectively forward message based on element values
Type filter		Selectively forward message based on element types

The primitives in this next grouping are categorized as routing primitives. The **service invoke** primitive is the first of these and is the only one in this category that changes the message type. It is used to make a call from within a mediation flow to an external service defined on the mediation module assembly. The service can be defined by a Java component or by an import. The input terminal message type conforms to the input to the service and the output terminal message type conforms to the output from the service.

The **endpoint lookup** primitive is used to perform a query of the WebSphere Service Registry and Repository. The SMOHeader section of the SMO is updated with potential service endpoints that can be used by a callout node or service invoke primitive. Looked at from this perspective, the endpoint lookup is not actually doing the routing, but rather is providing the information used later in the flow to perform routing.

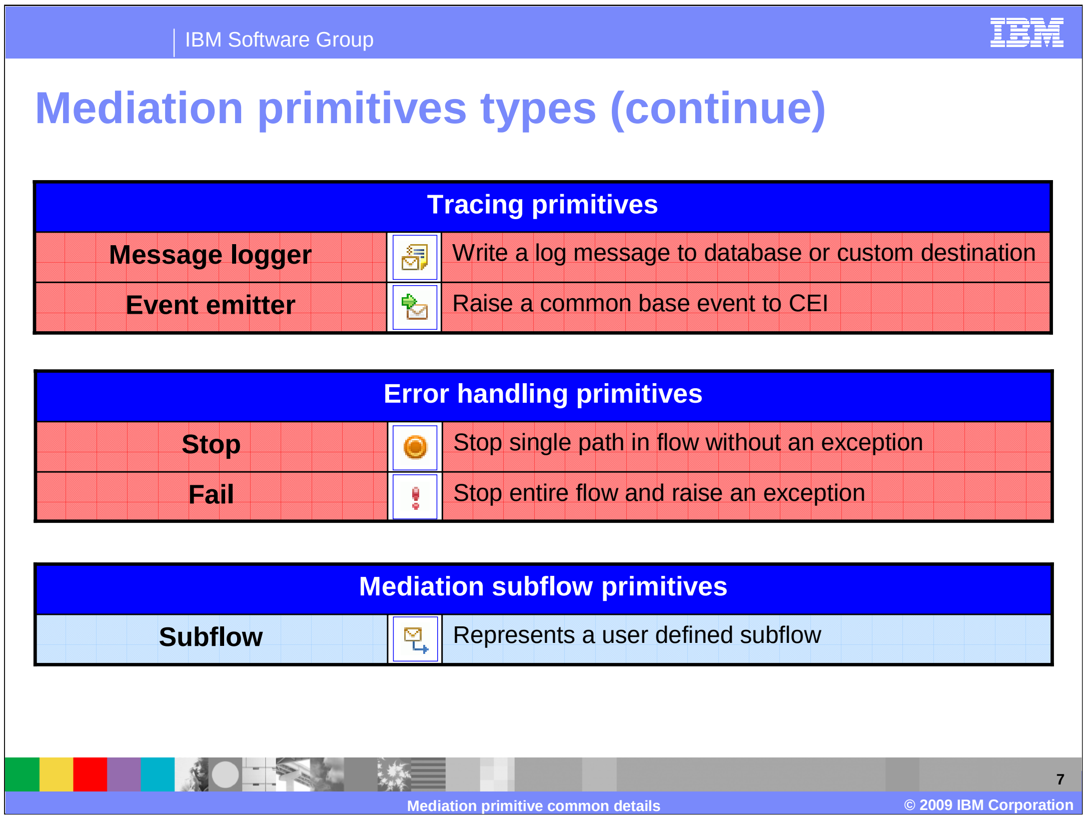
The **policy resolution** primitive is similar, also performing a query of the WebSphere Service Registry and Repository. Based on a conditional query, it looks up policy information and updates the dynamicProperty section of the SMO with property values. These property values are then used by subsequent primitives in the flow to influence their configured behavior.

The **fan out** primitive is used to start an iteration for a message splitting and aggregation flow. It can be used to process an array of repeating elements within the SMO and can also be used as the head of a flow with multiple paths. For a simple splitting scenario the fan out can be used without a fan in, but for a splitting and aggregation scenario it has an associated fan in. When used with an aggregation scenario, the primitive updates the FanOutContext section of the SMO.

The remaining routing primitives do not update the SMO. The **fan in** primitive is always used in conjunction with a fan out primitive as part of a message splitting and aggregation scenario. The primitive controls the flow based on the state of the flow and the fan in configuration information. Either the flow returns to the fan out for another iteration or the flow proceeds from the fan in, passing the aggregated results.

The **message filter** primitive is used to modify the path through a flow by selectively forwarding the message based on values of elements within the SMO. The primitive contains a table of simple XPath expressions, each associated with an output terminal defining where the message is forwarded. The message is forwarded based on evaluation results of the XPath expressions.

The **type filter** primitive is very similar to the message filter. It selectively forwards the message based on the evaluation of element types within the SMO. The primitive contains a table with elements and associated types, each associated with an output terminal defining where the message is forwarded. The message is forwarded when an element in the SMO is of the same type as that defined in the table.



IBM Software Group

Mediation primitives types (continue)

Tracing primitives	
Message logger	Write a log message to database or custom destination
Event emitter	Raise a common base event to CEI

Error handling primitives	
Stop	Stop single path in flow without an exception
Fail	Stop entire flow and raise an exception

Mediation subflow primitives	
Subflow	Represents a user defined subflow

7

Mediation primitive common details

© 2009 IBM Corporation

This slide looks at the remaining primitives, the tracing primitives and error handler primitives, which do not modify the SMO.

The first tracing primitive is the **message logger** that is used to log all or part of the contents of the message. The primitive provides two different options. The first option logs the message to a message log database that is identified through configuration of the primitive. The other option allows you to specify a custom logging implementation based on the Java logging APIs. A default implementation of this is provided that logs to a file.

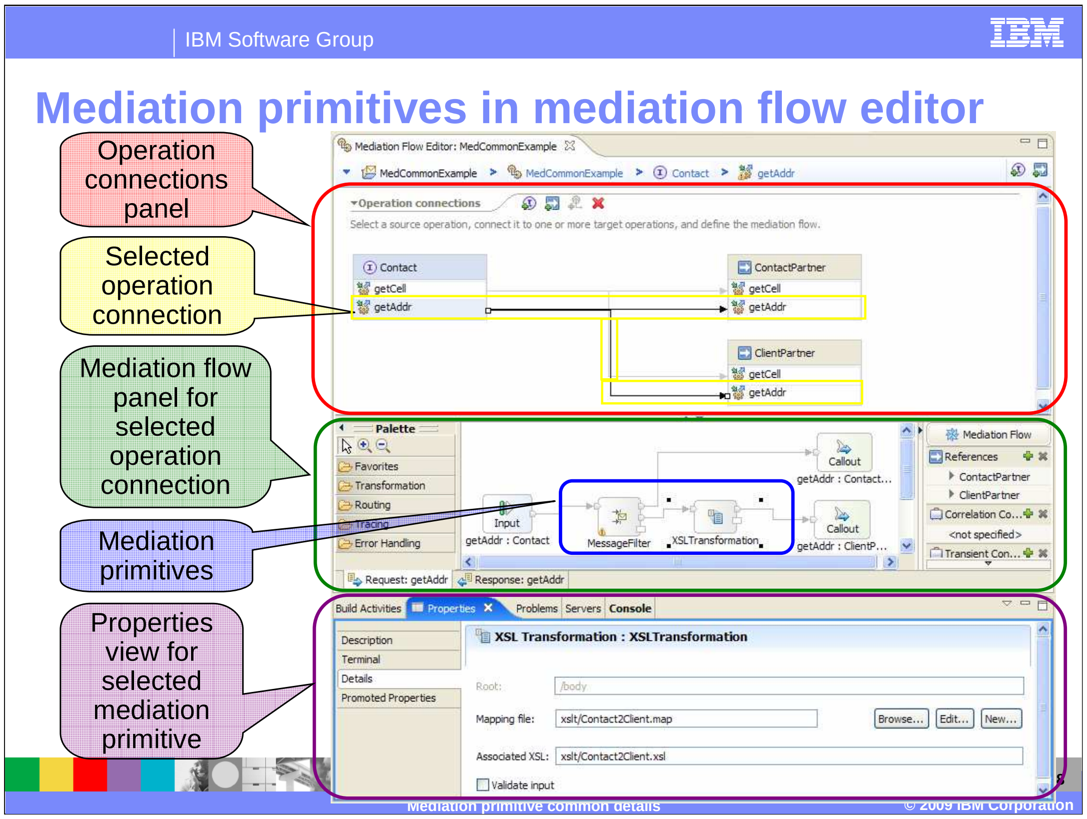
The **event emitter** primitive is used to raise an event containing all or part of the contents of the message. The event is emitted as a common base event, which is handled by the common event infrastructure.

The next category is the error handling primitives.

The **stop** primitive is used to stop an individual path through the mediation flow without raising an exception or affecting other paths through the flow.

The **fail** primitive is used for error conditions and will stop the entire mediation flow and cause an exception to be raised.

Finally, there is the mediation **subflow** primitive, which represents flow logic that you have provided as a subflow. A mediation subflow is a preconfigured set of mediation primitives that are wired together to create a common pattern or use case. The logic of the subflow determines if the SMO contents are update and if the message type is changed.



Mediation primitives in mediation flow editor

Select a source operation, connect it to one or more target operations, and define the mediation flow.

XSL Transformation : XSLTransformation

Root: /body

Mapping file: xslt/Contact2Client.map

Associated XSL: xslt/Contact2Client.xsl

☐ Validate input

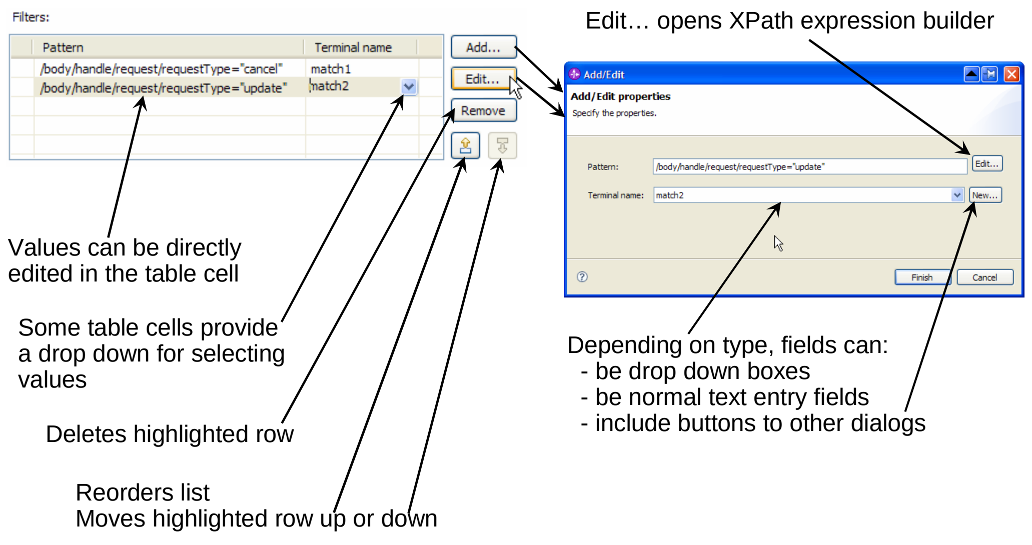
Shown here is a screen capture of the mediation flow editor.

The top panel of the editor is for operation connections and contains the input and output interfaces along with all of their operations. Every operation on the input interface must be wired to one or more operations on the output interfaces.

By selecting a particular operation connection, the mediation flow logic for that input operation is shown in the mediation flow panel of the editor. This panel has tabs that can be used to display the mediation logic for either the request flow or the response flow. Within a flow, mediation primitives are wired together between the nodes to define the logic of the flow.

Selecting any specific mediation primitive in the editor displays the properties for that primitive in the properties view, which is in the bottom panel. This is where the properties are specified to configure the behavior of the primitive.

Properties view – edit a table based property



Filters:

Pattern	Terminal name
/body/handle/request/requestType="cancel"	match1
/body/handle/request/requestType="update"	match2

Add/Edit

Add/Edit properties

Specify the properties.

Pattern: /body/handle/request/requestType="update"

Terminal name: match2

Edit… opens XPath expression builder

Values can be directly edited in the table cell

Some table cells provide a drop down for selecting values

Deletes highlighted row

Reorders list
Moves highlighted row up or down

Depending on type, fields can:
- be drop down boxes
- be normal text entry fields
- include buttons to other dialogs

Several of the primitives have a property that is represented as a table. This slide describes the common aspects of these table based properties and the mechanics of editing rows in the table. This example happens to use the filters property of the message filter primitive as an example.

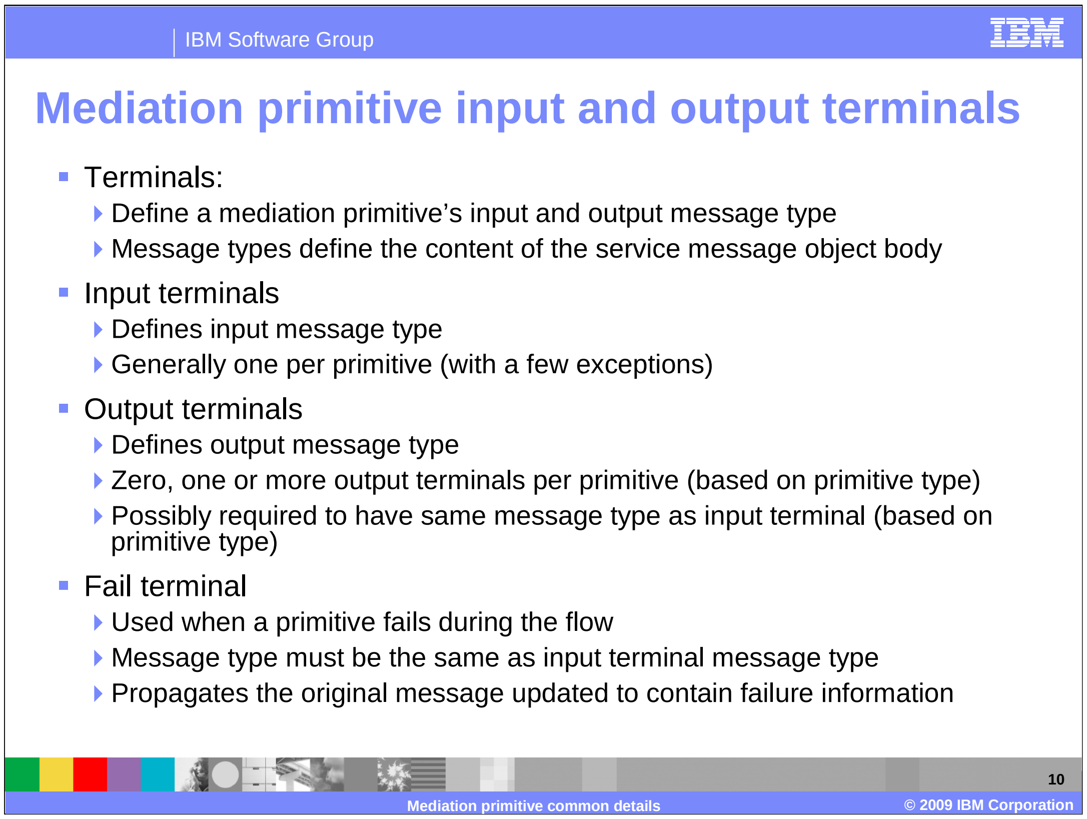
The **Add/Edit properties** dialog is used to edit a single row in the table, containing a field for each column. It is accessed by hitting the **Add…** button or by selecting an existing row from the table and hitting the **Edit…** button. When a column in the row represents an XPath expression, the dialog contains an **Edit…** button. This enables you to use the XPath expression builder dialog to define the expression to be evaluated.

Other fields in the Add/Edit properties dialog might contain drop down boxes, or they might be simple text entry fields. Which type is used depends upon the content expected for the column the field represents. Additionally, some types include a **New…** button that will present an appropriate dialog for creating a new instance of the appropriate type.

In some cases, Individual cells in the table can be edited directly without using the Add/Edit properties dialog. Also, the cell might contain a dropdown box from which you can select the value for the cell.

The **Remove** button deletes the selected row from the table.

For some table properties, the order of the rows is important, and for others it is not. In either case, there are up and down arrows that can be used to move a selected row up or down within the table.



Mediation primitive input and output terminals

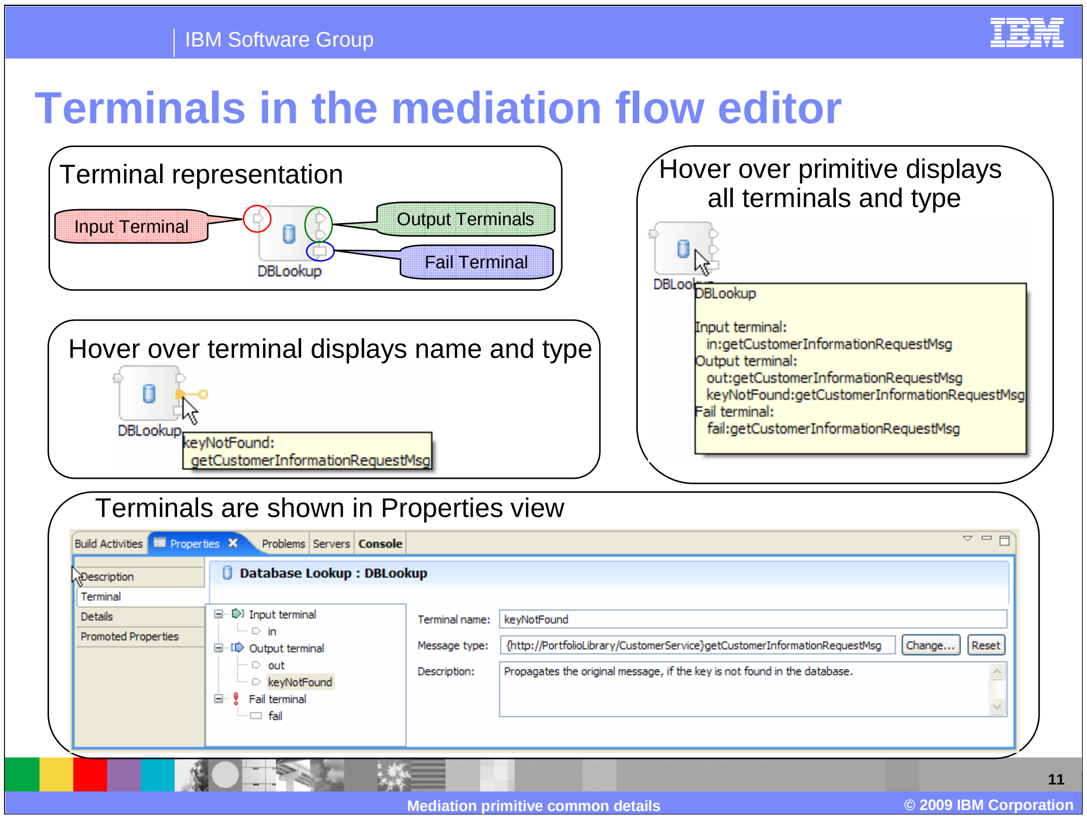
- Terminals:
  - Define a mediation primitive’s input and output message type
  - Message types define the content of the service message object body
- Input terminals
  - Defines input message type
  - Generally one per primitive (with a few exceptions)
- Output terminals
  - Defines output message type
  - Zero, one or more output terminals per primitive (based on primitive type)
  - Possibly required to have same message type as input terminal (based on primitive type)
- Fail terminal
  - Used when a primitive fails during the flow
  - Message type must be the same as input terminal message type
  - Propagates the original message updated to contain failure information

All mediation primitives have terminals, which are used to define the input and output of the primitive, specifically identifying the message type that flows through the terminal. The message type is defined by the structure of the service message object body that is present in that part of the flow.

Typically, there is just one input terminal per primitive and it defines the input message type. Exceptions to this are the custom mediation primitive, allowing a variable number of input terminals, and the fan in primitive, which has two defined input terminals.

An output terminal defines the output message type. The number of output terminals varies by the primitive type. A primitive type can have zero, one, two or a variable number of output terminals. For many mediation primitives, the output terminal must be of the same message type as the input terminal. This is because the primitive is not capable of changing the structure of the SMO body. However, for primitives that can change the SMO body structure, the output terminal can be for a different message type.

The fail terminal is used when the mediation primitive fails in some way while processing the message. Because the original message is propagated when there is a failure, the fail terminal is always for the same message type as the input terminal. The message is updated to contain information about the failure.



Terminals in the mediation flow editor

Terminal representation

Hover over terminal displays name and type

keyNotFound:
getCustomerInformationRequestMsg

Hover over primitive displays all terminals and type

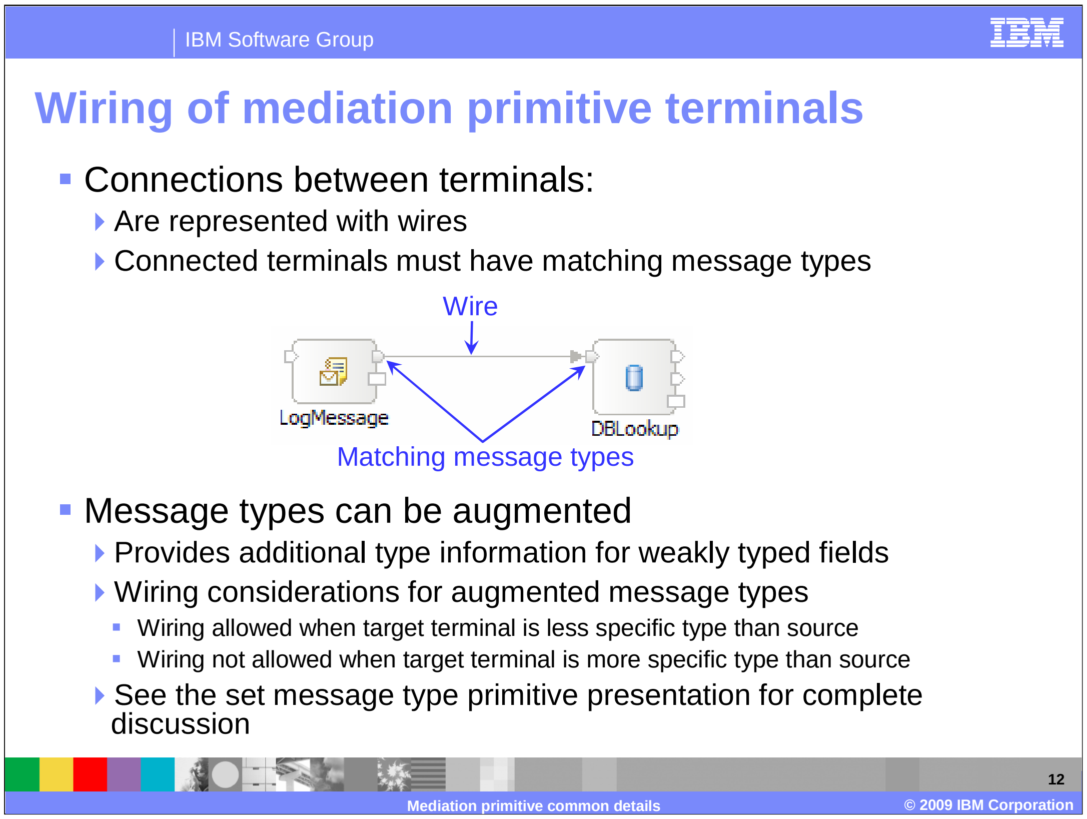
DBLookup

Input terminal:
in:getCustomerInformationRequestMsg
Output terminal:
out:getCustomerInformationRequestMsg
keyNotFound:getCustomerInformationRequestMsg
Fail terminal:
fail:getCustomerInformationRequestMsg

Terminals are shown in Properties view

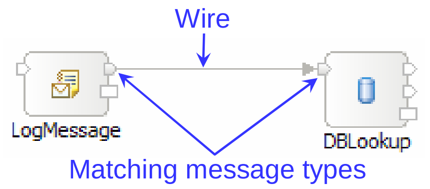
Terminal name: keyNotFound
Message type: {http://PortfolioLibrary/CustomerService}getCustomerInformationRequestMsg
Description: Propagates the original message, if the key is not found in the database.

This slide examines how terminals are represented in the mediation flow editor. Starting in the upper left is a mediation primitive. It has an input terminal, which is always on the left side; output terminals, which are on the right side; and the fail terminal, which is the lower terminal on the right side. Notice that the fail terminal has a different shape than either the input or output terminals. Moving down to the illustration in the left center, the behavior when hovering the mouse pointer over a terminal is illustrated. When this is done, a popup opens specifying the name of the terminal and the message type associated with that terminal. The illustration on the upper right shows you what happens when you hover the mouse pointer over the primitive. A popup opens specifying the name of the primitive along with the name and message type of all the terminals for that primitive. Finally, on the bottom is a screen capture of the terminal tab in the properties view of the mediation primitive. Selecting any terminal in the list on the left displays the name and type of the terminal on the right. Notice that the message type here is the fully qualified type rather than the short version of the type shown in a popup. Also notice the **Change…** button that opens a dialog for modifying the message type associated with the terminal.



Wiring of mediation primitive terminals

- Connections between terminals:
  - Are represented with wires
  - Connected terminals must have matching message types

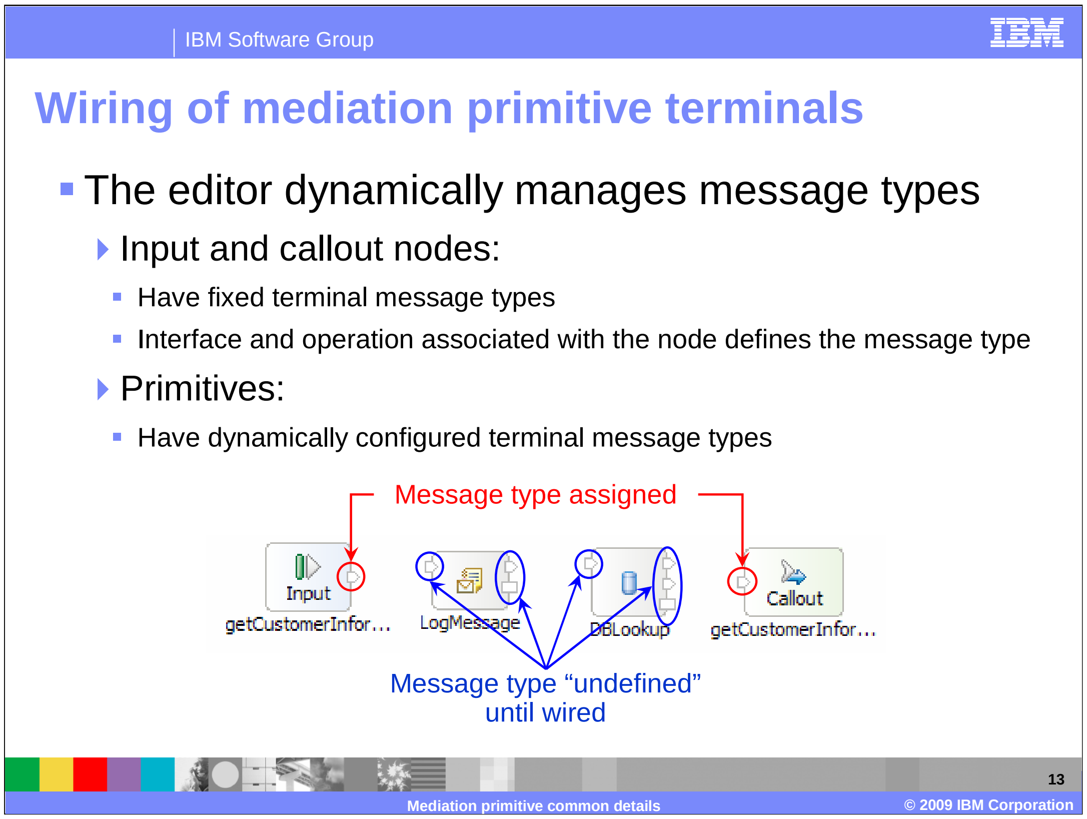


- Message types can be augmented
  - Provides additional type information for weakly typed fields
  - Wiring considerations for augmented message types
    - Wiring allowed when target terminal is less specific type than source
    - Wiring not allowed when target terminal is more specific type than source
  - See the set message type primitive presentation for complete discussion

The next few slides look at the behavior of the mediation flow editor relative to the assignment of message types to terminals during the process of wiring a mediation flow. As illustrated in the graphic, connections between terminals are represented with wires. When two terminals are wired together, they must have matching message types.

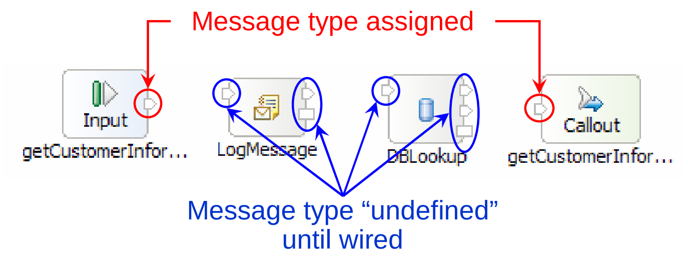
Message types can be augmented with additional type information. This is used when needing to provide additional type information for weakly typed fields, such as those defined as an xsd:anyType. When wiring terminals that have augmented message types, the two terminals are not required to have exactly the same augmentation. Wiring is allowed when the target terminal has a less specific type than the source terminal. However, if the target has a more specific augmented type than the source, they can not be wired together.

Augmented message types are not addressed in detail in this presentation. A complete discussion of augmentation of message types is provided in the set message type primitive presentation.

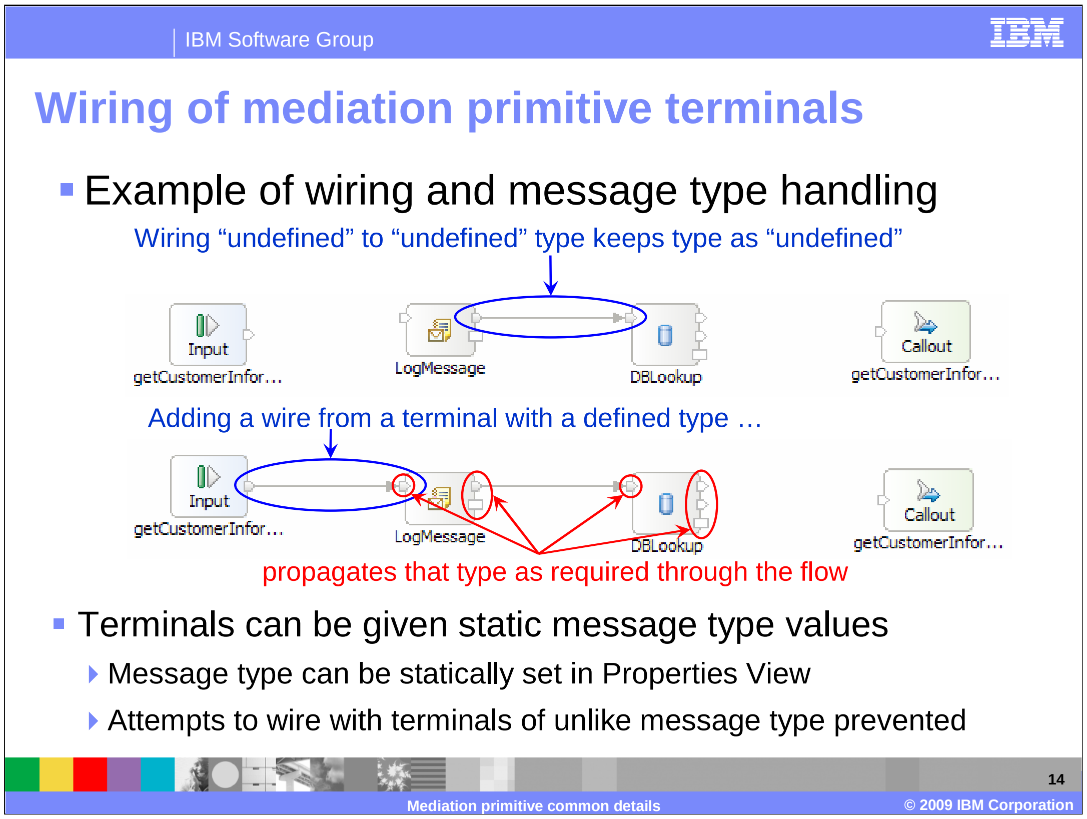


Wiring of mediation primitive terminals

- The editor dynamically manages message types
  - Input and callout nodes:
    - Have fixed terminal message types
    - Interface and operation associated with the node defines the message type
  - Primitives:
    - Have dynamically configured terminal message types

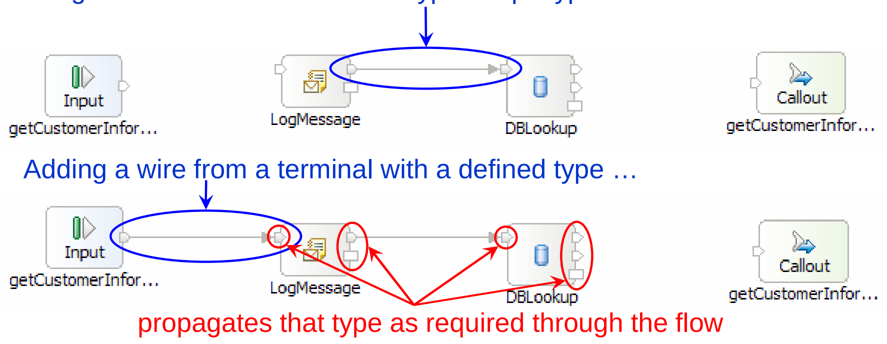


During the process of wiring terminals together, the editor dynamically manages the message types of the terminals. In the graphic, you see a mediation flow that has an input node on the left, a callout node on the right, and two mediation primitives in the middle. At this point, none of these are wired together. Notice that the nodes have message types assigned to their terminals, whereas the mediation primitives do not yet have message types assigned to their terminals.



Wiring of mediation primitive terminals

- Example of wiring and message type handling


Wiring "undefined" to "undefined" type keeps type as "undefined"
Input
getCustomerInfor...
LogMessage
DBLookup
Callout
getCustomerInfor...
Adding a wire from a terminal with a defined type …
Input
getCustomerInfor...
LogMessage
DBLookup
Callout
getCustomerInfor...
propagates that type as required through the flow

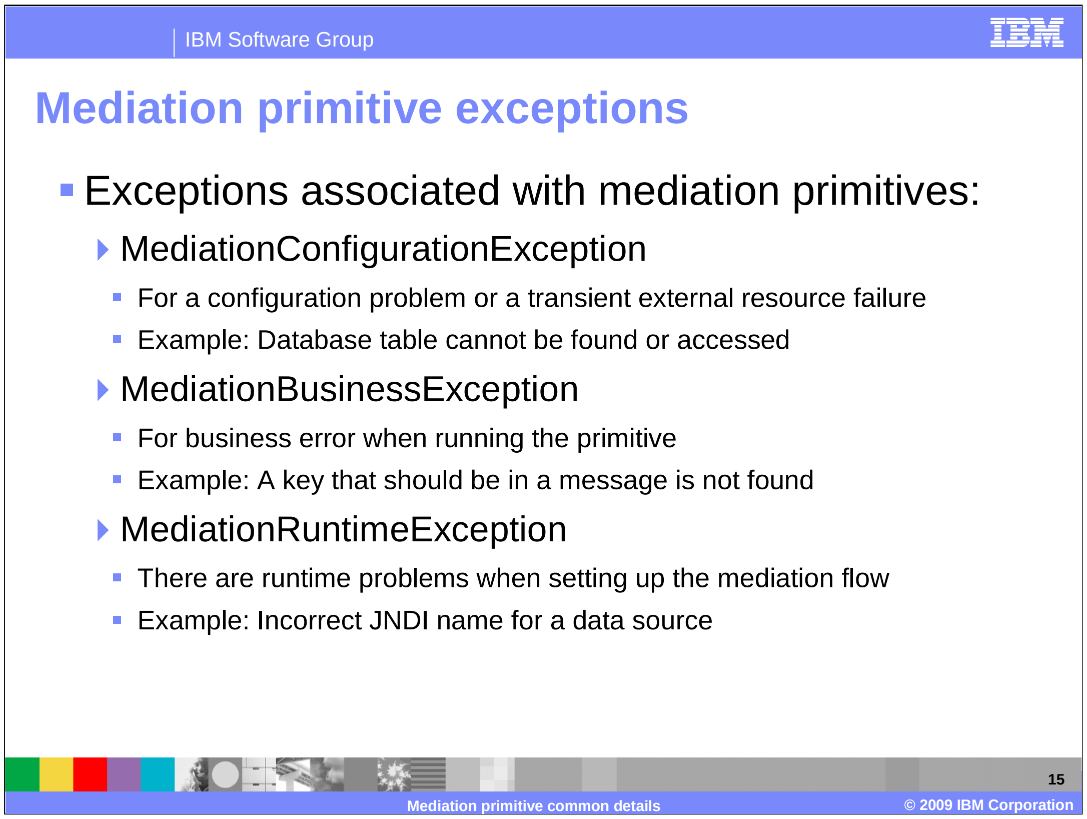
- Terminals can be given static message type values
  - Message type can be statically set in Properties View
  - Attempts to wire with terminals of unlike message type prevented

Continuing from the previous slide, the top graphic shows a wire added from the output terminal of the LogMessage primitive to the input terminal of the DBLookup primitive, both of which have undefined message types. When you do this, the terminals still have an undefined message type.

In the next graphic, the output terminal of the input node is wired to the input terminal of the LogMessage primitive. Because the output terminal of the input node has a specific message type assigned, that message type is dynamically assigned to the input terminal of the LogMessage primitive so that the wire connects terminals of like message type. Since a message logger primitive must have the same output message type as its input message type, the editor dynamically assigns the message type to the output and fail terminals of the LogMessage primitive. Since there is a wire between the output terminal of the LogMessage primitive and the input terminal of the DBLookup primitive, the message type is propagated so that the wire is connecting terminals of like type. Finally, since a database lookup primitive must have the same output message type as its input message type, the editor dynamically assigns that type to the output and fail terminals of the DBLookup primitive.

In addition, the editor resets all the terminals to have an undefined message type if the wire that started the message type propagation is removed. So you can see that the mediation flow editor makes it quite easy to manage terminal message types when wiring a flow.

It is also possible to assign a specific message type to a terminal so that it is static and overrides the dynamic assignment of message type. When you do this, the editor prevents you from wiring the terminal with the static message type to anything other than a terminal with like type or a terminal with an undefined message type.


IBM Software Group

Mediation primitive exceptions

- Exceptions associated with mediation primitives:
  - MediationConfigurationException
    - For a configuration problem or a transient external resource failure
    - Example: Database table cannot be found or accessed
  - MediationBusinessException
    - For business error when running the primitive
    - Example: A key that should be in a message is not found
  - MediationRuntimeException
    - There are runtime problems when setting up the mediation flow
    - Example: Incorrect JNDI name for a data source

15

Mediation primitive common details © 2009 IBM Corporation

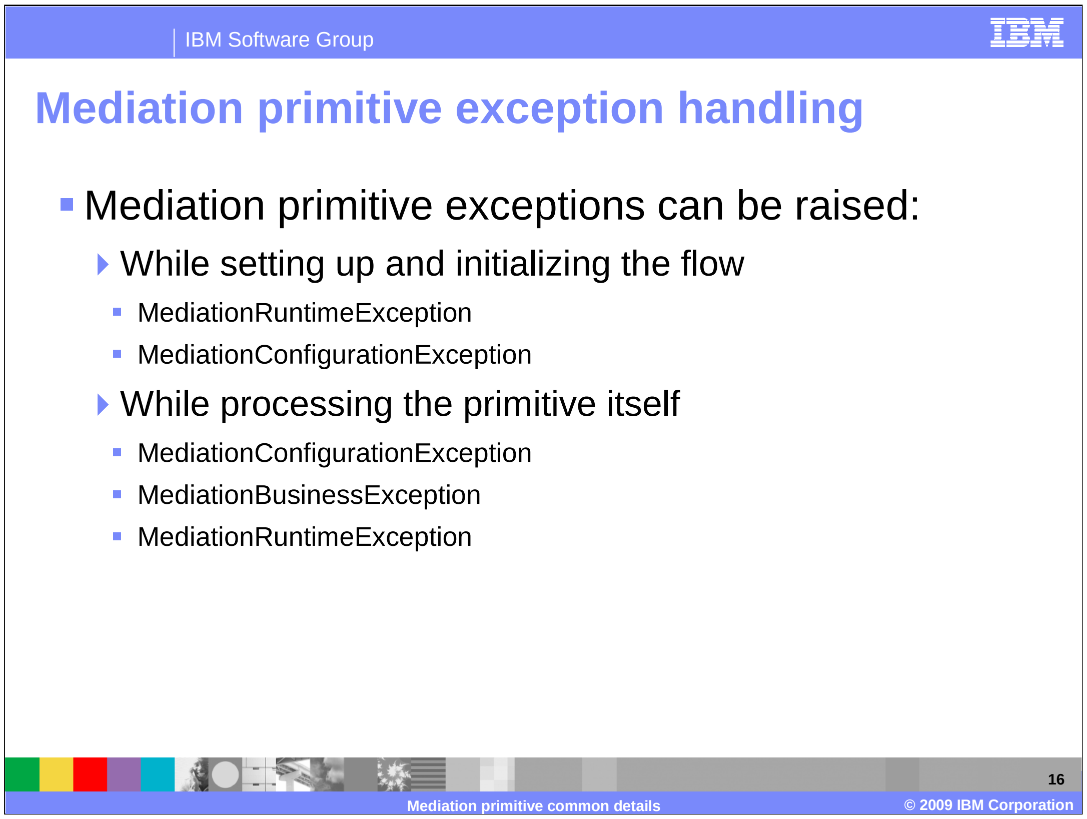
The specific exceptions that mediation primitives can raise are described here.

A **MediationConfigurationException** is used when there is a configuration problem. It is also used for a transient problem with an external resource, such as not being able to find or access a database.

A **MediationBusinessException** is used when an error that appears to be a business logic problem occurs while running a primitive. An example of this kind of problem is when the database key value configured for a primitive cannot be found in the message.

A **MediationRuntimeException** occurs when there is some kind of problem initializing a mediation flow. An example of this is when the JNDI name for a data source is incorrect.

These exceptions and associated error processing are described in more detail in the next few slides.



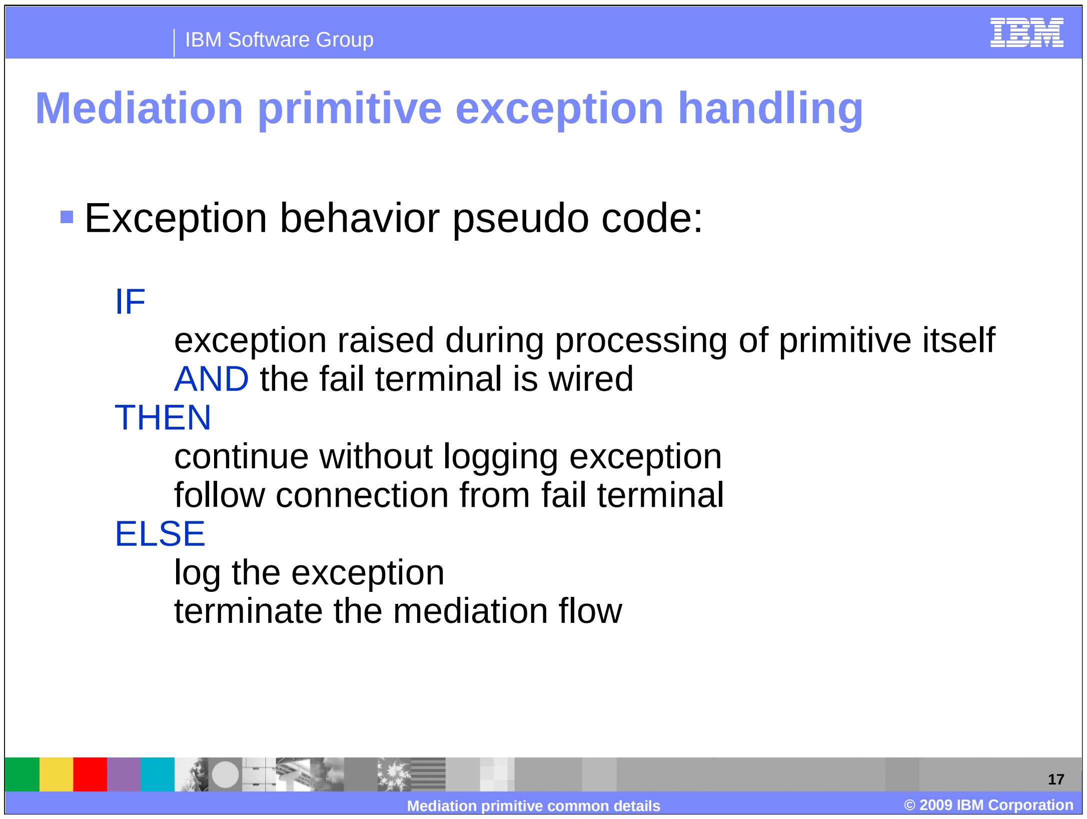
Mediation primitive exception handling

- Mediation primitive exceptions can be raised:
  - While setting up and initializing the flow
    - MediationRuntimeException
    - MediationConfigurationException
  - While processing the primitive itself
    - MediationConfigurationException
    - MediationBusinessException
    - MediationRuntimeException

In order to provide an understanding of the exception processing behavior, it is important to know the different points at which a mediation exception can be raised.

First of all, some initialization is done by the runtime to set up a mediation flow. This initialization occurs before control is given to any mediation primitives. Exceptions that might be raised at this time are the MediationRuntimeException or the MediationConfigurationException, with the MediationRuntimeException being the most common.

Secondly, after initialization of the flow, an exception can be raised during the processing of a mediation primitive. Normally these are a MediationConfigurationException or a MediationBusinessException, but in some cases a MediationRuntimeException can also be raised.



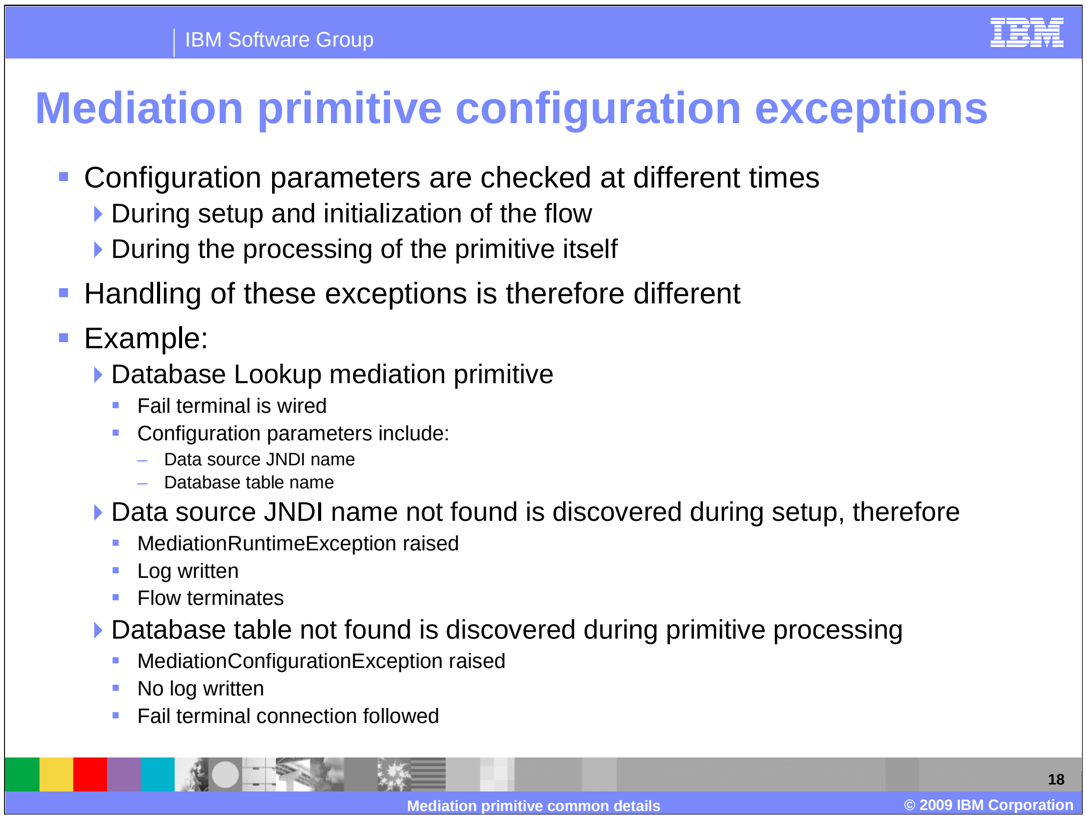
Mediation primitive exception handling

- Exception behavior pseudo code:

```
IF
    exception raised during processing of primitive itself
    AND the fail terminal is wired
THEN
    continue without logging exception
    follow connection from fail terminal
ELSE
    log the exception
    terminate the mediation flow
```

The behavior for processing exceptions is different based on a couple of factors. The first factor is whether the exception is raised during initialization processing or during the processing of a primitive. The second factor is whether the fail terminal is wired.

Looking at the slide you can see pseudocode describing the actual behavior. If the exception is raised during the processing of the primitive and the fail terminal is wired to some other primitive or node, then the exception is not logged. The mediation flow continues, following the wire from the fail terminal. In all other cases, the exception causes a log message to be written and the mediation flow to terminate.


IBM Software Group

Mediation primitive configuration exceptions

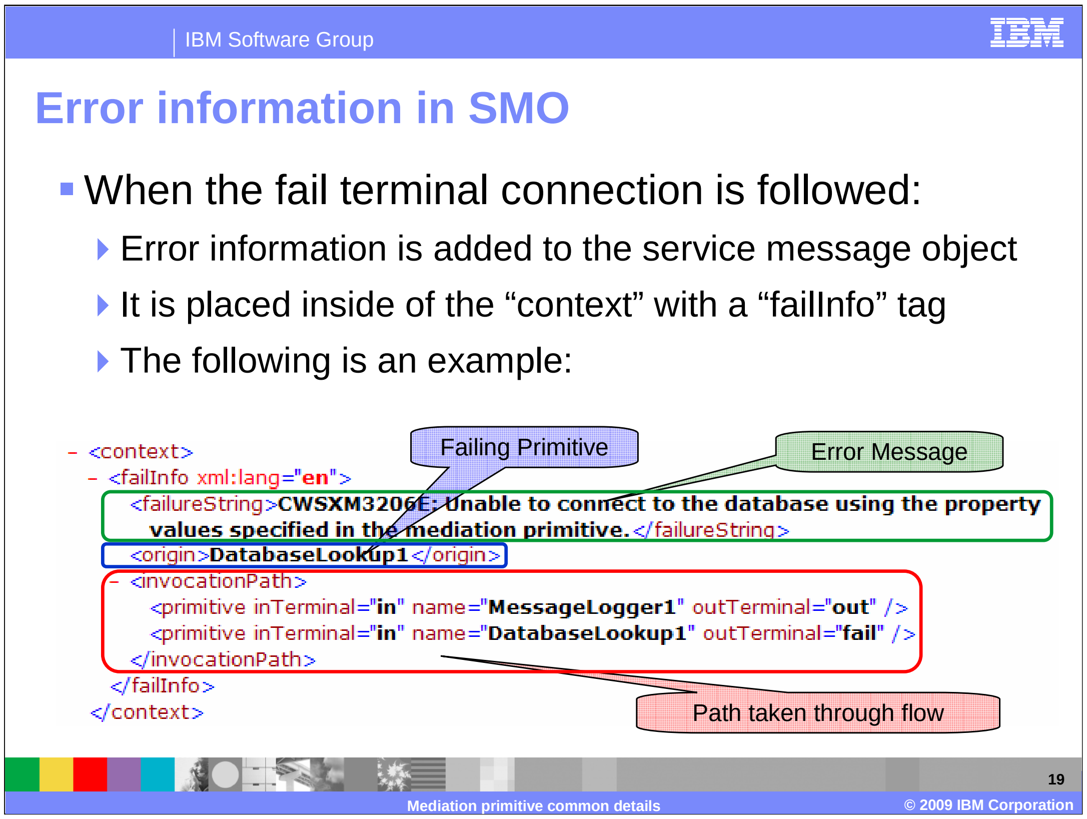
- Configuration parameters are checked at different times
  - During setup and initialization of the flow
  - During the processing of the primitive itself
- Handling of these exceptions is therefore different
- Example:
  - Database Lookup mediation primitive
    - Fail terminal is wired
    - Configuration parameters include:
      - Data source JNDI name
      - Database table name
  - Data source JNDI name not found is discovered during setup, therefore
    - MediationRuntimeException raised
    - Log written
    - Flow terminates
  - Database table not found is discovered during primitive processing
    - MediationConfigurationException raised
    - No log written
    - Fail terminal connection followed

18

Mediation primitive common details © 2009 IBM Corporation

When there is a problem with something in the configuration of a mediation primitive, you see the different behaviors described on the previous slide. This is because configuration parameters are checked at different times, some being checked at the initialization of the flow and others being checked during the processing of the mediation primitive. For example, assume you have a database lookup primitive that has its fail terminal wired. Two of the configuration parameters for a database lookup are the data source JNDI name and the database table name. The data source JNDI name is checked during the setup of the flow, therefore the result of an incorrect JNDI name is a MediationRuntimeException raised with a log written and the mediation flow terminated. However, the database table name is checked during the processing of the primitive itself. Therefore, when the MediationConfigurationException is initially raised it is caught, no log is written, and the mediation flow continues by following the wire from the fail terminal of the database lookup. Only if the fail terminal is not wired will the MediationConfigurationException be logged and the flow terminated.

In the presentations for each of the individual mediation primitives, there is specific information about error conditions that can occur for that primitive type. Which of these behaviors you can expect to see is described.



Error information in SMO

- When the fail terminal connection is followed:
  - Error information is added to the service message object
  - It is placed inside of the “context” with a “failInfo” tag
  - The following is an example:

```
- <context>
  - <failInfo xml:lang="en">
      <failureString>CWSXM3206E: Unable to connect to the database using the property
        values specified in the mediation primitive.</failureString>
      <origin>DatabaseLookup1</origin>
    - <invocationPath>
        <primitive inTerminal="in" name="MessageLogger1" outTerminal="out" />
        <primitive inTerminal="in" name="DatabaseLookup1" outTerminal="fail" />
      </invocationPath>
    </failInfo>
  </context>
```

Failing Primitive
Error Message
Path taken through flow

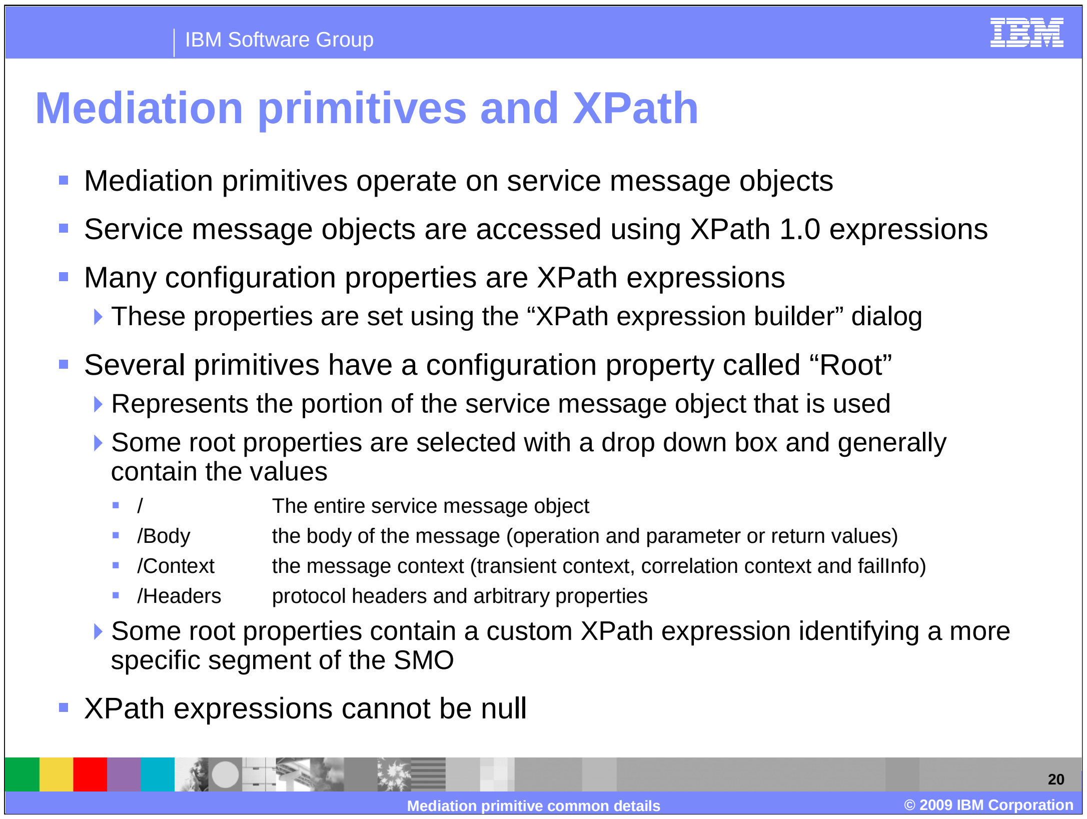
When the fail terminal is wired and an exception occurs within a mediation primitive, information about the error is added to the failInfo section of the context section of the service message object. The following information is added:

The **failureString** contains a text description of the error that occurred.

The **origin** contains the name of the mediation primitive in which the exception occurred.

The **invocation Path** contains a list of every mediation primitive that was encountered in the message flow, up to and including the primitive in which the error occurred. In addition, the names of the terminals through which the message passed are also listed with each primitive.

With this information, logic in the flow might be able to determine what action to take in response to the failure.



Mediation primitives and XPath

- Mediation primitives operate on service message objects
- Service message objects are accessed using XPath 1.0 expressions
- Many configuration properties are XPath expressions
  - These properties are set using the "XPath expression builder" dialog
- Several primitives have a configuration property called "Root"
  - Represents the portion of the service message object that is used
  - Some root properties are selected with a drop down box and generally contain the values

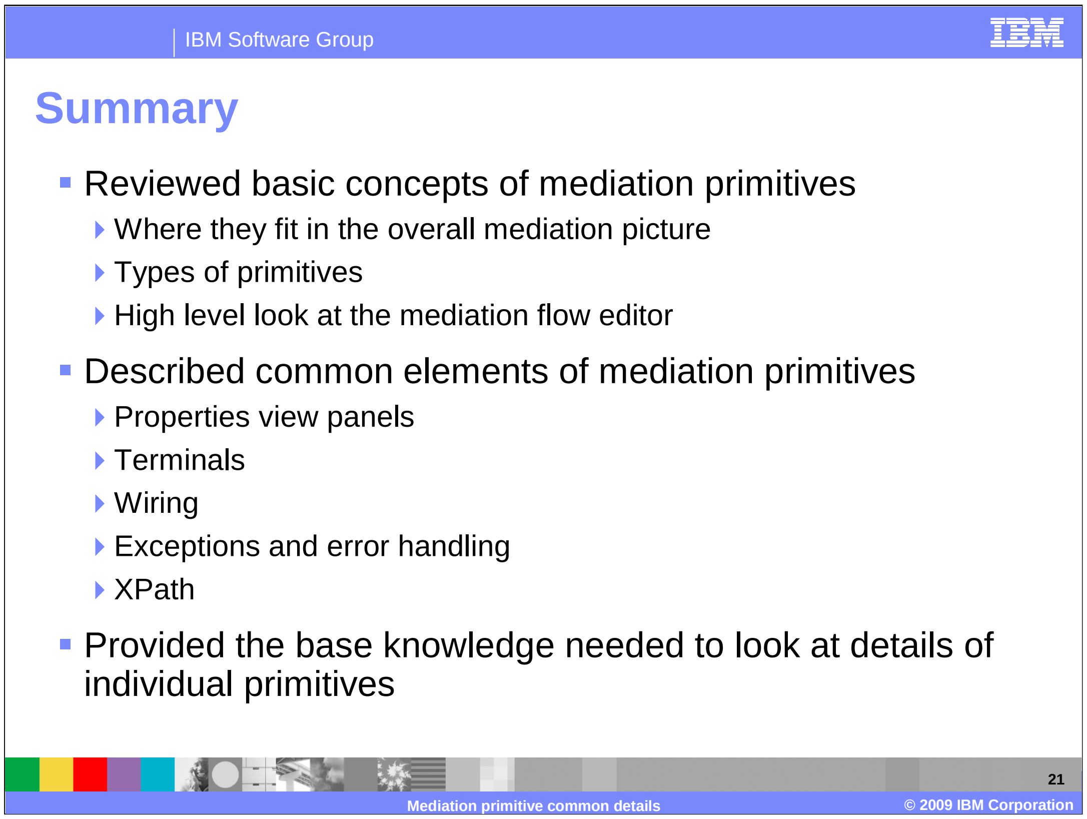
/	The entire service message object
/Body	the body of the message (operation and parameter or return values)
/Context	the message context (transient context, correlation context and failInfo)
/Headers	protocol headers and arbitrary properties

  - Some root properties contain a custom XPath expression identifying a more specific segment of the SMO
- XPath expressions cannot be null

Another element common to mediation primitives is that they operate on service message objects and that XPath 1.0 expressions can be used to access the data within the SMO. Many of the configuration properties used by primitives are expressed as XPath expressions. There is a dialog in WebSphere Integration Developer called the XPath expression builder that can be used to construct XPath expressions.

Several of the primitives have a property called root, which defines the portion of the SMO that is to be used by the primitive during its processing. The valid values for root properties are not consistent across all of the primitives. In some cases the root is specified with a drop down box with four choices. The choice "/" refers to the entire SMO and the choices "/body", "/context" and "/headers" refer to each of the three major sections of the SMO. In other cases, the ability exists to use the XPath expression builder to construct the expression for root. This enables a finer granularity in identifying the portion of the SMO that is the root for the primitive.

Null XPath expressions cause an exception at runtime.


IBM Software Group

Summary

- Reviewed basic concepts of mediation primitives
  - Where they fit in the overall mediation picture
  - Types of primitives
  - High level look at the mediation flow editor
- Described common elements of mediation primitives
  - Properties view panels
  - Terminals
  - Wiring
  - Exceptions and error handling
  - XPath
- Provided the base knowledge needed to look at details of individual primitives

21

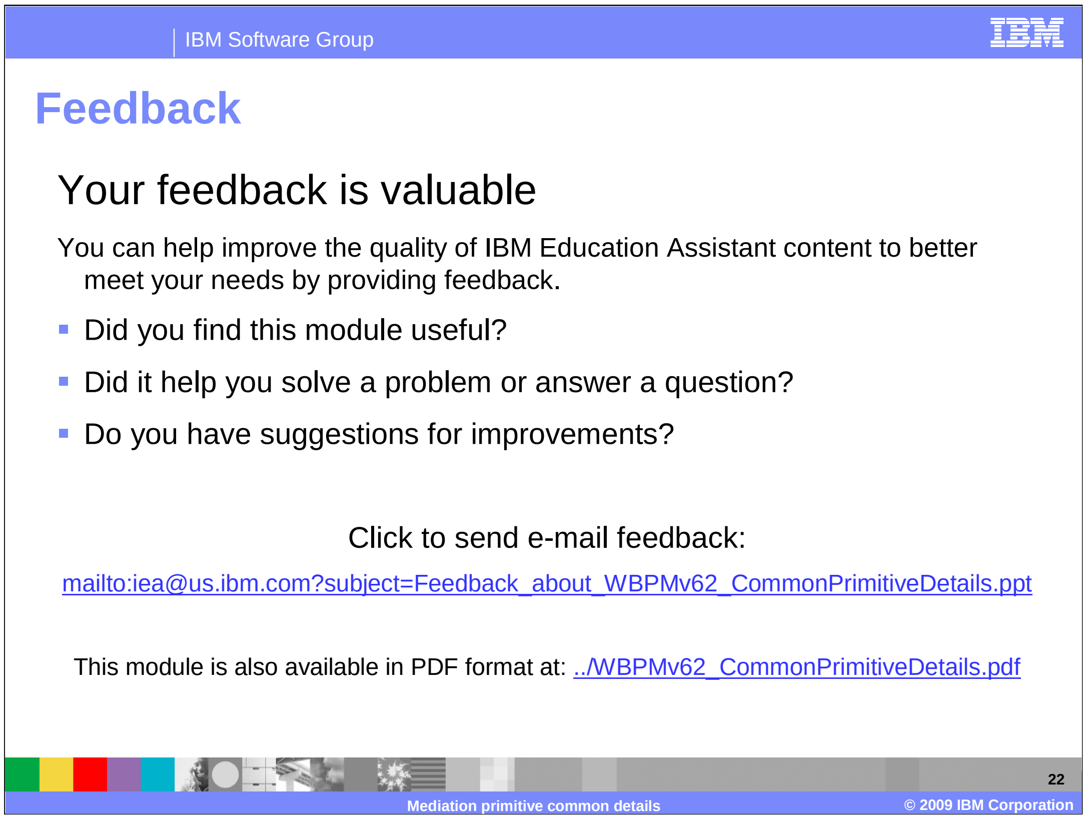
Mediation primitive common details

© 2009 IBM Corporation

In this presentation, the basic concepts of mediation primitives were reviewed. This included a description of where mediation primitives fit into the overall mediation picture, what types of mediation primitives there are and an introduction to how they are edited in the mediation flow editor.

Some of the common elements of mediation primitives were examined. These include the properties view panels, terminals, wiring, exceptions, error handling, and XPath usage.

With the understanding provided by this presentation, you should now be better prepared to understand the specifics of each individual mediation primitive.



Feedback

Your feedback is valuable

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

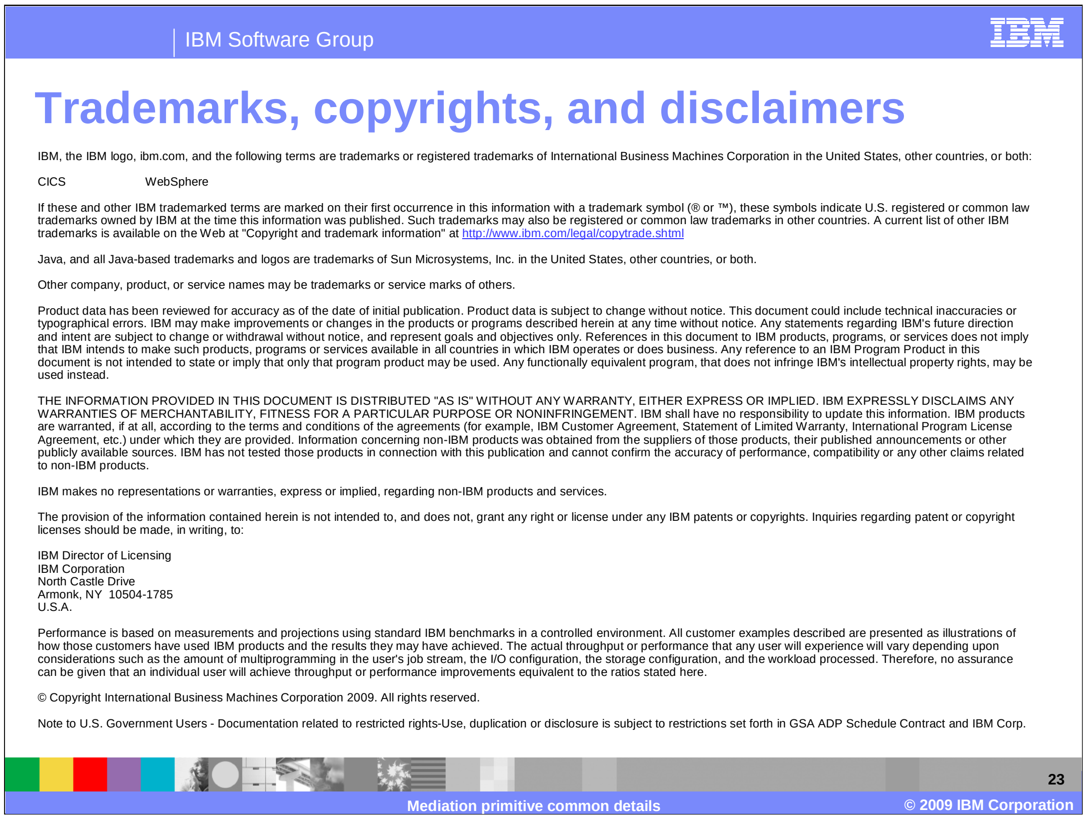
- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send e-mail feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_WBPMv62_CommonPrimitiveDetails.ppt

This module is also available in PDF format at: ../WBPMv62_CommonPrimitiveDetails.pdf

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



Trademarks, copyrights, and disclaimers