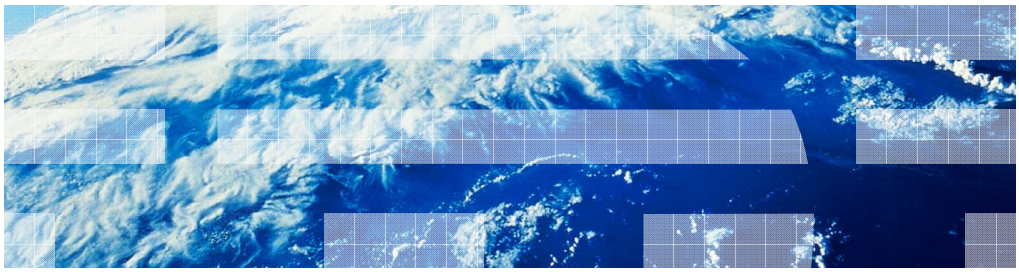
IBM WebSphere Application Server Feature Pack for XML

V1.0.0.3 updates



WebSphere. software

This presentation will go through the Feature Pack for XML V1.0.0.3 updates.

Table of contents

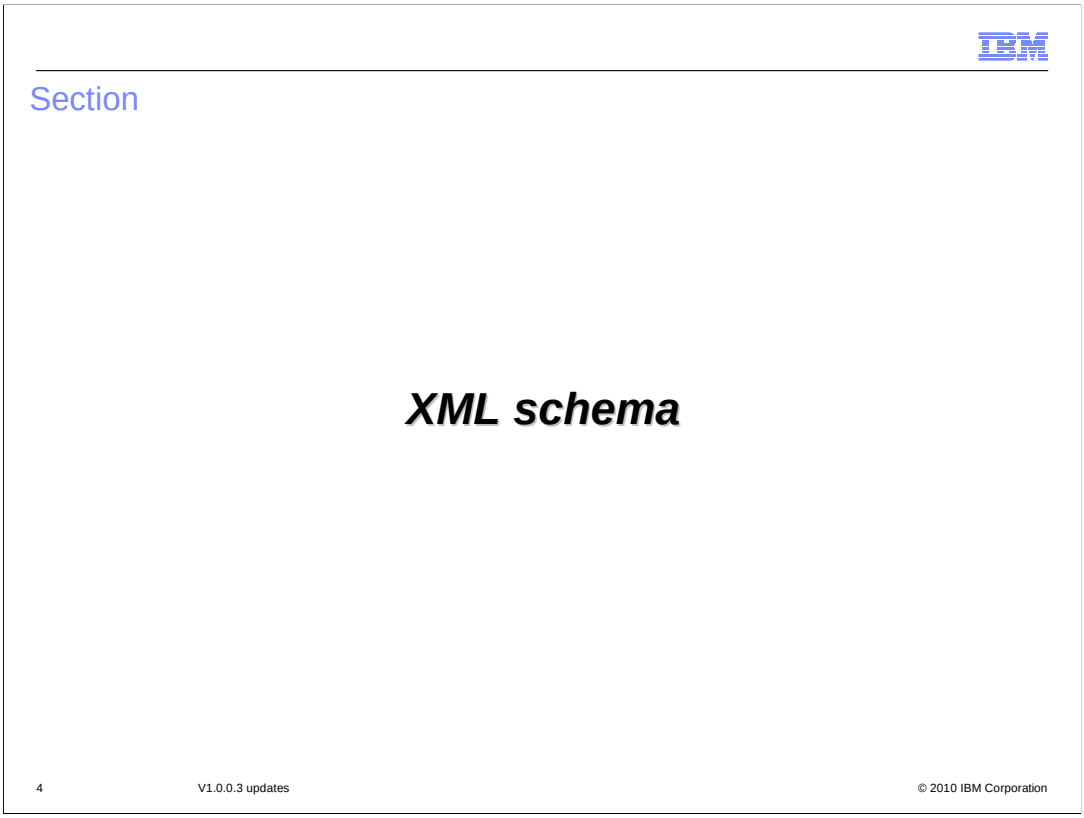
Here is the agenda for this presentation.

Feature Pack for XML support for XQuery

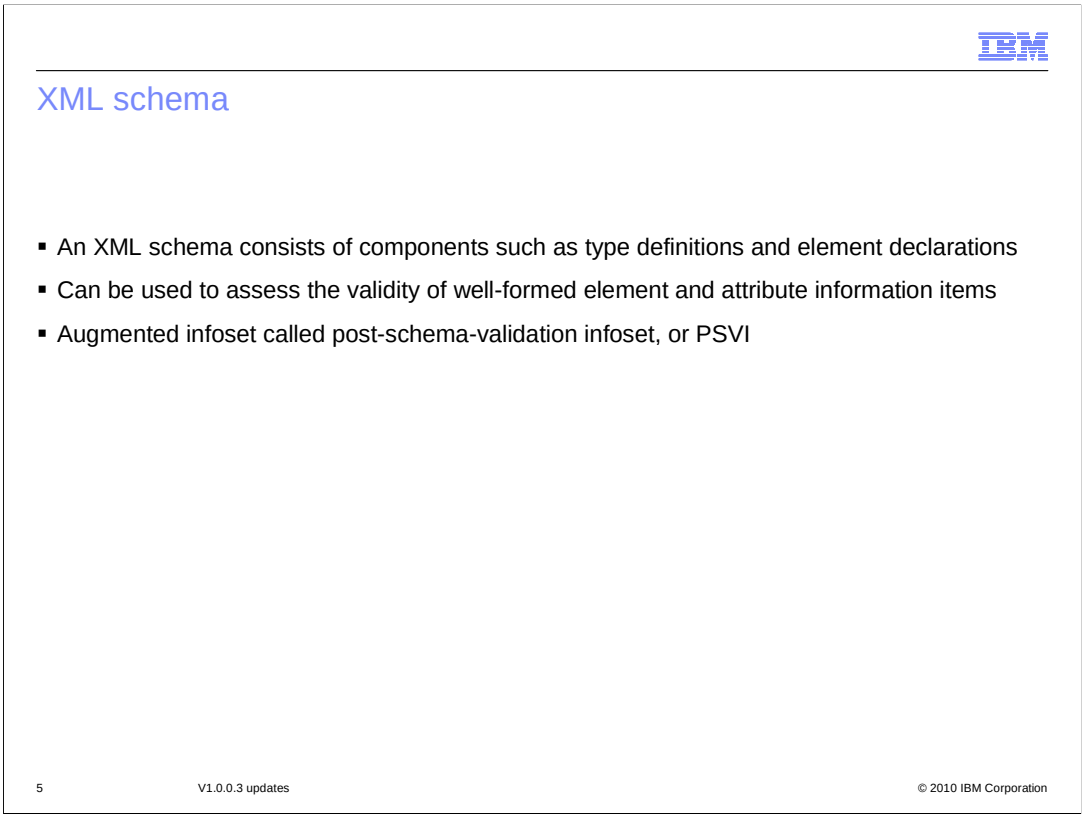
- The IBM WebSphere Application Server for XML has minimal conformance based on the XQuery 1.0 specification
- It supports the data model conformance based on XQuery 1.0 and XPath 2.0 Data Model (XDM)
- Version 1.0 of the feature pack offered the XQuery specific optional features:
  - Full axis feature
  - Serialization feature
- **Version 1.0.0.3 of the feature pack supports the optional features of V1.0 and adds XQuery optional features of schema awareness which includes:**
  - **Schema import feature**
  - **Schema validation feature**
- **Note**: XSLT already supported schema awareness in version 1.0 of the feature pack

The Feature Pack for XML delivers critical technology that enables adoption of key XML standards and principles. The Feature Pack for XML provides application developers with support for the W3C XML standards XSLT 2.0, XPath 2.0, and XQuery 1.0. The Feature Pack for XML also provides the IBM XML API in support of these standards. The API invokes a runtime engine that is capable of executing XPath 2.0, XSLT 2.0, and XQuery 1.0 and manipulating the returned XML data. The XML API across the XML programming model family of languages offers consistent execution and data navigation while allowing access to existing Java logic.

The IBM WebSphere Application Server for XML has minimal conformance based on the XQuery 1.0 specification. It supports the data model conformance based on XQuery 1.0 and XPath 2.0 Data Model (XDM). XQuery specification includes optional features. The Feature Pack for XML V1.0 includes the optional features of Full Axis and Serialization. The feature pack V1.0.0.3 supports the optional features of V1.0 and includes the XQuery optional features Schema Awareness which includes Schema Import Feature, and Schema Validation Feature. Schema Awareness for XSLT is already supported in V1.0 of the feature pack.



The next few slides will go over XML Schema.



A schema is an XML document that defines the content and structure of one or more XML documents. XML Schema Definition (XSD) can be used to express a set of rules to which an XML document must conform in order to be considered 'valid' according to that schema. An XML schema consists of a set of schema components element, attribute declarations, complex and simple type definitions.

An XML Schema consists of components such as type definitions and element declarations. An XML schema can define data types for elements and attributes such as integer, date, decimal, string and user defined Types and complex Element Types. XML schema also allows the definition of length and patterns for string values, and supports type inheritance and derived data types. These can be used to assess the validity of well-formed element and attribute information items, and furthermore can specify augmentations to those items and their descendants. This augmentation makes explicit information which might have been implicit in the original document, such as normalized or default values for attributes and elements and the types of element and attribute information items. You refer to the augmented infoset which results from conformant processing as defined in this specification as the post-schema-validation infoset, or PSVI.

Schema-validity assessment has two aspects:

One is determining local schema-validity, that is whether an element or attribute information item satisfies the constraints embodied in the relevant components of an XML Schema;

The second is synthesizing an overall validation outcome for the item, combining local schema-validity with the results of schema-validity assessments of its descendants, if any, and adding appropriate augmentations to the infoset to record this outcome.

XML schema languages

- DTDs and XML schema
  - With similar goals to define
    - Types of literal (terminal) data
    - Names of elements and attribute
    - “Vertical” and “horizontal” structure of elements
- Example:

```
<xs:element name="book" type="Book"/>
<xs:complexType name="Book">
  <xs:sequence>
    <xs:element name="title" type="xs:string"/>
    <xs:element name="author" type="xs:string"/>
    <xs:element name="price" type="Price"/>
  </xs:sequence>
  <xs:attribute name="isbn" type="ISBN"/>
</xs:complexType>
```


An XML *schema* or *type* describes the grammar for a class of XML documents and serves as a contract between applications that produce and consume XML data. Typically, an XML schema specifies the grammar for XML documents in a particular application domain. There are a variety of XML schema languages such as DTDs and XML Schema. Interfaces provide varying support for accessing schema information. The schema typically specifies the types of literal data in the documents, the names of elements and attributes, and the permissible vertical (ancestor) and horizontal (sibling) structure between elements.

Schemas are critical for validation as they check that a given document corresponds to some expected structure. Schemas are useful to help humans to understand the structure of XML documents. They are like a map to make the navigation easier. Schemas provide information about the structure of the document (nesting relationships, nature of the element data, and so on.). This information can be used to store the data more efficiently. This slide shows an example of an XML schema. You can see the constructs that reveal the permissible structure of a book element. The operators ‘sequence’, ‘choice’ and repeated sequences are used to construct complex structures. Attributes are implicitly unordered. They typically follow the definition of an element’s children elements and instances of attributes can occur in any order in a document.

Validated XML

- Validation : Document and Schema -> PSVI
  - Elements and attributes “annotated” with types
  - PSVI input to XQuery 1.0, XPath 2.0, XSLT 2.0
  - Well-formed, un-validated documents annotated with default types
- Example:

```
<book[Book] isbn[ISBN]="ISBN 1565114302">
   <title[xs:string]>No Such Thing as a Bad Day</title>
   <author[xs:string]>Hamilton Jordan</author>
<price[Price] currency[Currency] ="USD">17.60</price>
</book>
```


XQuery 1.0, XPath 2.0, and XSLT 2.0 are all “type-aware” query languages, meaning the languages can observe, preserve and respect the types associated with input and output documents. Validation is the process that takes an (untyped) input document and an associated schema, and checks that the document conforms to the specified schema. A more formal definition is that the document is an instance of the languages denoted by the grammar of the schema. As output, validation produces a “post-schema validated infoset” or PSVI.A PSVI is an instance of the document in which each element and attribute node has been annotated with the Schema type against which it was successfully validated. The XPath 2.0 data model (which is also the data model of XQuery 1.0 and XSLT 2.0) is defined in terms of the PSVI.

The type annotations in red are ***not*** visible in the XML document – they are used here and in the rest of the presentation to denote the types that are associated with the corresponding values after validation.

Well-formed documents with no corresponding schemas can also be queried. These documents are labeled with default types – well-formed elements are labeled with xs:untyped and well-formed attributes are labeled with xs:untypedAtomic, which indicate that the elements (attributes) contain unvalidated content(text).

Extend schema

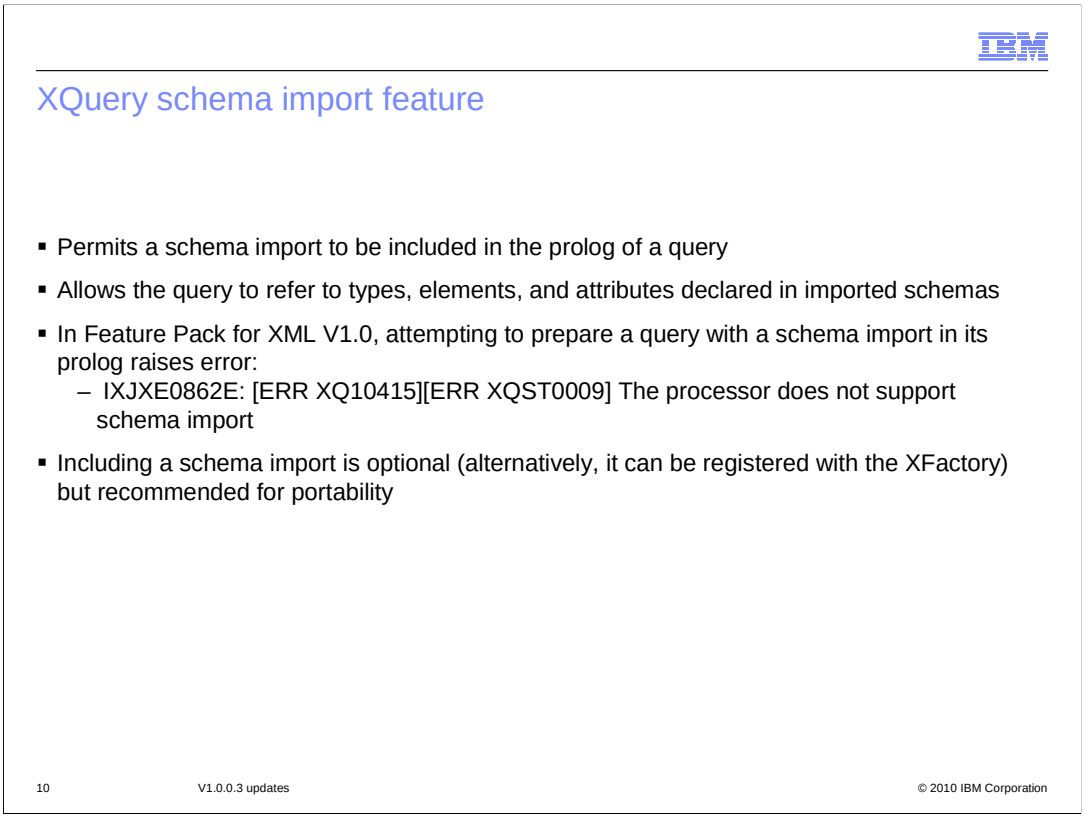
- Define and extend industry standard schemas
- XQuery is a superset of XPath 2.0 and benefits from the same types of schema-aware expressions as in XPath 2.0:
  - In XPath 1.0, search for every possible substitution in the group and every possible complex type derivation type
  - In XPath 2.0, due to its schema awareness you can search for the substitution group head and complex type base type
  - Example:
    - `//element(*, tns:AddressType)/postalCode` returns the postalCode of any elements with a type of AddressType
    - If the document had both BillingAddress and MailingAddress, both are returned
    - With XPath 1.0, you have to look for both //tns:BillingAddress and //tns:MailingAddress

It is common to define and extend industry standard schemas through XML schema constructs of substitution groups and complex type derivation by extension. Since XQuery is a superset of XPath 2.0 it benefits from the same types of schema-aware expressions as in XPath 2.0. In XPath 1.0, you have to search for every possible substitution in the group and every possible complex type derivation type. This is impossible if you're writing an application that processes data on the group and base types without knowledge of how eventual users might extend the groups and types. In XPath 2.0, due to its schema awareness you can search for the substitution group head and complex type base type. This will work even if users later add to the substitution group and extend the base complex type. An example of this is shown above.

Section

XQuery schema import feature

The next few slides will go over the XQuery schema import feature.



A query consists of a prolog followed by a query body. A **prolog** is an optional section that is a series of declarations and imports that define the processing environment. A prolog is organized into two parts. The first part of the prolog consists of setters, imports, namespace declarations, and default namespace declarations. The Feature Pack for XML V1.0.0.3 supports imports of schemas. The second part of the prolog consists of declarations of variables, functions, and options. These declarations appear at the end of the prolog because they can be affected by declarations and imports in the first part of the prolog. In V1.0 of the feature pack a query with a schema import in the prolog raises an error. In version 1.0.0.3 of the feature pack, schema import is supported. This allows the query to refer to types, elements and attributes that are declared in the imported schemas. Including a schema import is optional. You can also register the schema with the XFactory.

XML file needed for examples

```
<?xml version="1.0" encoding="UTF-8"?>
<lib:mycollection
    xmlns:lib="http://www.example.org/library"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.example.org/library library.xsd ">
    <lib:book title="The Complete Works of William Shakespeare">
        <lib:author>William Shakespeare</lib:author>
        <lib:editor>Jonathan Bate</lib:editor>
    </lib:book>
    <lib:dvd title="Hamlet">
        <lib:actor>Judi Dench</lib:actor>
        <lib:actor>Kenneth Branagh</lib:actor>
        <lib:director>Kenneth Branagh</lib:director>
        <lib:writer>William Shakespeare</lib:writer>
        <lib:writer>Kenneth Branagh</lib:writer>
    </lib:dvd>
</lib:mycollection>
```

borrowed_items.xml

Here is an XML file. It lists a library collection that includes both books and DVDs. The book element includes child elements such as author and editor and a title attribute. The DVD element includes child elements such as actor, director, and writer and a title attribute.

Schema file needed for examples

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.example.org/library"
elementFormDefault="qualified" xmlns:lib="http://www.example.org/library">
<element name="mycollection">
  <complexType>
    <sequence maxOccurs="unbounded">
      <choice>
        <element name="book" type="lib:resourceType" />
        <element name="dvd" type="lib:resourceType" />
      </choice>
    </sequence>
  </complexType>
</element>
<complexType name="resourceType">
  <sequence maxOccurs="unbounded">
    <element ref="lib:contributor" />
  </sequence>
  <attribute name="title" type="string" />
</complexType>

<element name="contributor" type="string" />
<element name="author" type="string" substitutionGroup="lib:contributor" />
<element name="actor" type="string" substitutionGroup="lib:contributor" />
<element name="editor" type="string" substitutionGroup="lib:contributor" />
<element name="writer" type="string" substitutionGroup="lib:contributor" />
<element name="director" type="string" substitutionGroup="lib:contributor" />
</schema>
```

library.xsd

Here is the schema file associated with the XML file on the previous slide. Notice the declaration of the mycollection and resourceType. The mycollection element can have child elements of book or DVD that are type resourceType. The resourceType will have an attribute title and contributor child elements such as author, actor, editor, writer and director.

Example 1: Schema import in XQuery and result

```
import schema namespace lib="http://www.example.org/library" at "library.xsd";

for $title in lib:mycollection/element(*,lib:resourceType)/@title
order by $title
return <title type="{ local-name($title/..) }">{ string($title) }</title>
```

get_titles.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<title type="dvd">Hamlet</title>
<title type="book">The Complete Works of William Shakespeare</title>
```

get_titles.out

This slide shows the query and result. The query includes the import schema of the schema on the last slide. The query uses a type from the imported schema to select lib:resourceType elements. This is useful in case types of resources other than books or DVDs are added to the schema later. The schema import sets up the prefix “lib” for use in the query and specifies a location hint to help the processor locate the schema. Notice that both the titles of the DVD and book are in the output.

Example 2: Schema import in XQuery

```
import schema default element namespace "http://www.example.org/library";

for $contribName in distinct-values(mycollection/element(*,resourceType)/schema-
  element(contributor))
let $contrib := mycollection/element(*,resourceType)/schema-
  element(contributor)[.=$contribName]

order by $contribName
return
    <contributor name="{ $contribName }">
    {
        for $resource in $contrib/..
        return
            element { local-name($resource) }
            {
                $resource/@*,
                for $contribution in $resource/schema-element(contributor)
                where $contribution = $contribName
                return
                    <role type="{ local-name($contribution) }"/>
            }
    }
    </contributor>
```

group_contributors.xq

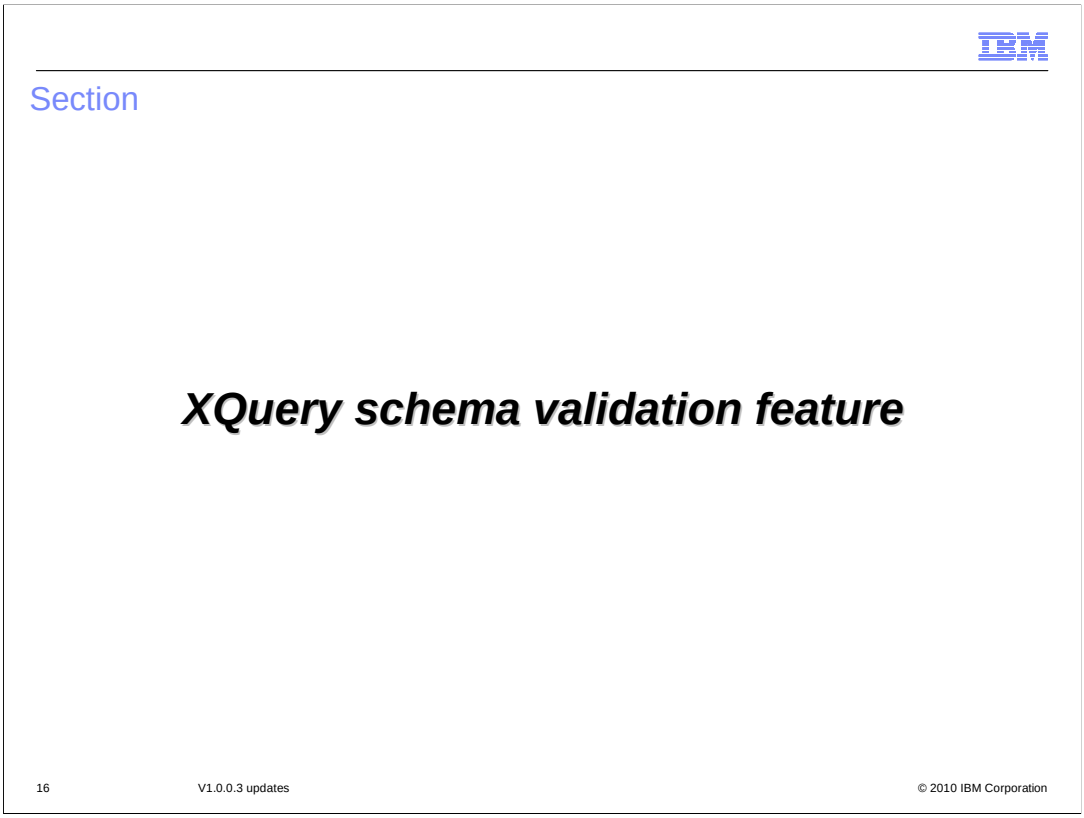
Here is another example of a query that shows the use of import schema. In this case, the query uses an element declaration from the imported schema to select any elements that belong to its substitution group. The schema import sets up the schema namespace as default so that no prefix is required in the query (this is necessary for schemas that have no target namespace). The schema import does not provide a location hint, so the schema must be registered or supplied by an XSchemaResolver. This will later be explained in the API section of this presentation.

Example 2: Output

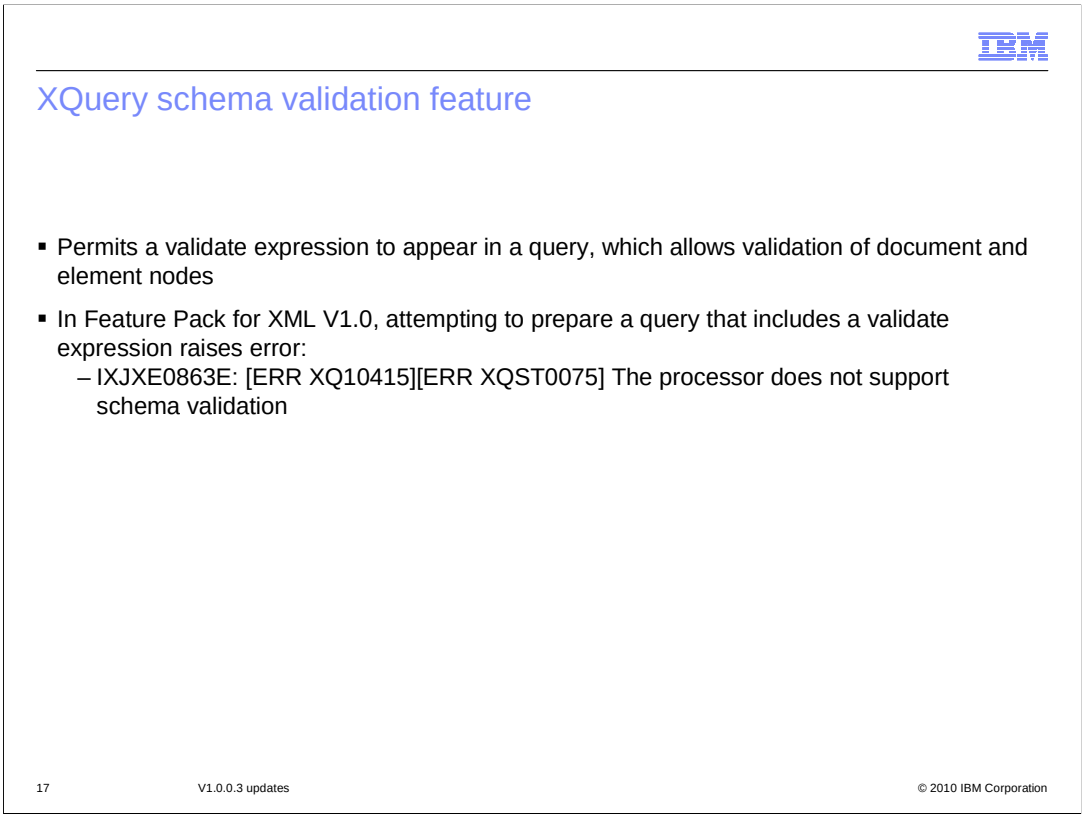
```
<?xml version="1.0" encoding="UTF-8"?>
<contributor xmlns="http://www.example.org/library" name="Jonathan Bate">
  <book title="The Complete Works of William Shakespeare">
    <role type="editor"/>
  </book>
</contributor>
<contributor xmlns="http://www.example.org/library" name="Judi Dench">
  <dvd title="Hamlet">
    <role type="actor"/>
  </dvd>
</contributor>
<contributor xmlns="http://www.example.org/library" name="Kenneth Branagh">
  <dvd title="Hamlet">
    <role type="actor"/>
    <role type="director"/>
    <role type="writer"/>
  </dvd>
</contributor>
<contributor xmlns="http://www.example.org/library" name="William Shakespeare">
  <book title="The Complete Works of William Shakespeare">
    <role type="author"/>
  </book>
  <dvd title="Hamlet">
    <role type="writer"/>
  </dvd>
</contributor>
```

group_contributors.out

The result of the query on the last slide using the XML and schema earlier in the presentation is shown. (pause for time to review)



The next few slides will go over the XQuery Schema Validation Feature.



In V1.0 of the feature pack a query that attempted to use the validation feature raises an error. In version 1.0.0.3 of the feature pack schema validation for XQuery is supported. This allows a validate expression to appear in a query, which in turn allows validation of document and element nodes.

XML file needed for examples

```
<?xml version="1.0" encoding="UTF-8"?>
<lib:mycollection
    xmlns:lib="http://www.example.org/library"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.example.org/library library.xsd ">
    <lib:book title="The Complete Works of William Shakespeare">
        <lib:author>William Shakespeare</lib:author>
        <lib:editor>Jonathan Bate</lib:editor>
    </lib:book>
    <lib:dvd title="Hamlet">
        <lib:actor>Judi Dench</lib:actor>
        <lib:actor>Kenneth Branagh</lib:actor>
        <lib:director>Kenneth Branagh</lib:director>
        <lib:writer>William Shakespeare</lib:writer>
        <lib:writer>Kenneth Branagh</lib:writer>
    </lib:dvd>
</lib:mycollection>
```

borrowed_items.xml

Here is an XML file. It lists a library collection that includes both books and DVDs. The book element includes child elements such as author and editor and a title attribute. The DVD element includes child elements such as actor, director, and writer and a title attribute.

Schema file needed for examples

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.example.org/library"
elementFormDefault="qualified" xmlns:lib="http://www.example.org/library">
<element name="mycollection">
  <complexType>
    <sequence maxOccurs="unbounded">
      <choice>
        <element name="book" type="lib:resourceType" />
        <element name="dvd" type="lib:resourceType" />
      </choice>
    </sequence>
  </complexType>
</element>
<complexType name="resourceType">
  <sequence maxOccurs="unbounded">
    <element ref="lib:contributor" />
  </sequence>
  <attribute name="title" type="string" />
</complexType>

<element name="contributor" type="string" />
<element name="author" type="string" substitutionGroup="lib:contributor" />
<element name="actor" type="string" substitutionGroup="lib:contributor" />
<element name="editor" type="string" substitutionGroup="lib:contributor" />
<element name="writer" type="string" substitutionGroup="lib:contributor" />
<element name="director" type="string" substitutionGroup="lib:contributor" />
</schema>
```

library.xsd

Here is the schema file associated with the XML file on the previous slide. Notice the declaration of the mycollection and resourceType. The mycollection element can have child elements of book or DVD that are type resourceType. The resourceType will have an attribute title and contributor child elements such as author, actor, editor, writer and director.

Example: Schema validation in XQuery – Validate entire output document

```
import schema namespace lib="http://www.example.org/library" at "library.xsd";

declare variable $resourceType external;
declare variable $resourceTitle external;
declare variable $contributorType external;
declare variable $contributorName external;

validate strict
{
    element lib:mycollection
    {
        element { concat("lib:", $resourceType) }
        {
            attribute title { $resourceTitle },
            element { concat("lib:", $contributorType) } { $contributorName }
        }
    }
}
```

group_contributors.xq

The example here will validate the entire output document. The query here creates a lib:mycollection element and its content and validates it against the schema file on the previous slide. It assumes that variables resourceType, resourceTitle, contributorType, contributorName are bound in the dynamic context. If resourceType is not one of “book” or “dvd” or contributorType is not the local name of one of the elements defined to be in the substitution group of lib:contributor, an error is raised.IXJXE1005E: [ERR 0770][ERR XQDY0027]. The 'validate' expression has the mode 'strict', and schema validity assessment concludes that the validity of the element is not valid or unknown.

Variable values bound in the dynamic context

Variable	Value	Type
resourceType	book	xs:string
resourceTitle	The Complete Works of William Shakespeare	xs:string
contributorType	author	xs:string
contributorName	William Shakespeare	xs:string

If you bind these variable values in the dynamic context: resourceType="book", resourceTitle="War and Peace", contributorType=author, contributorName="Leo Tolstoy, you will get the result on the next slide from running the query on the pervious slide.

Example: Output

```
<?xml version="1.0" encoding="UTF-8"?>
<lib:mycollection xmlns:lib="http://www.example.org/library">
  <lib:book title="War and Peace">
    <lib:author>Leo Tolstoy</lib:author>
  </lib:book>
</lib:mycollection>
```

create_valid_collection.out

Here is the result based on the variables, query, schema and XML on the previous slides.

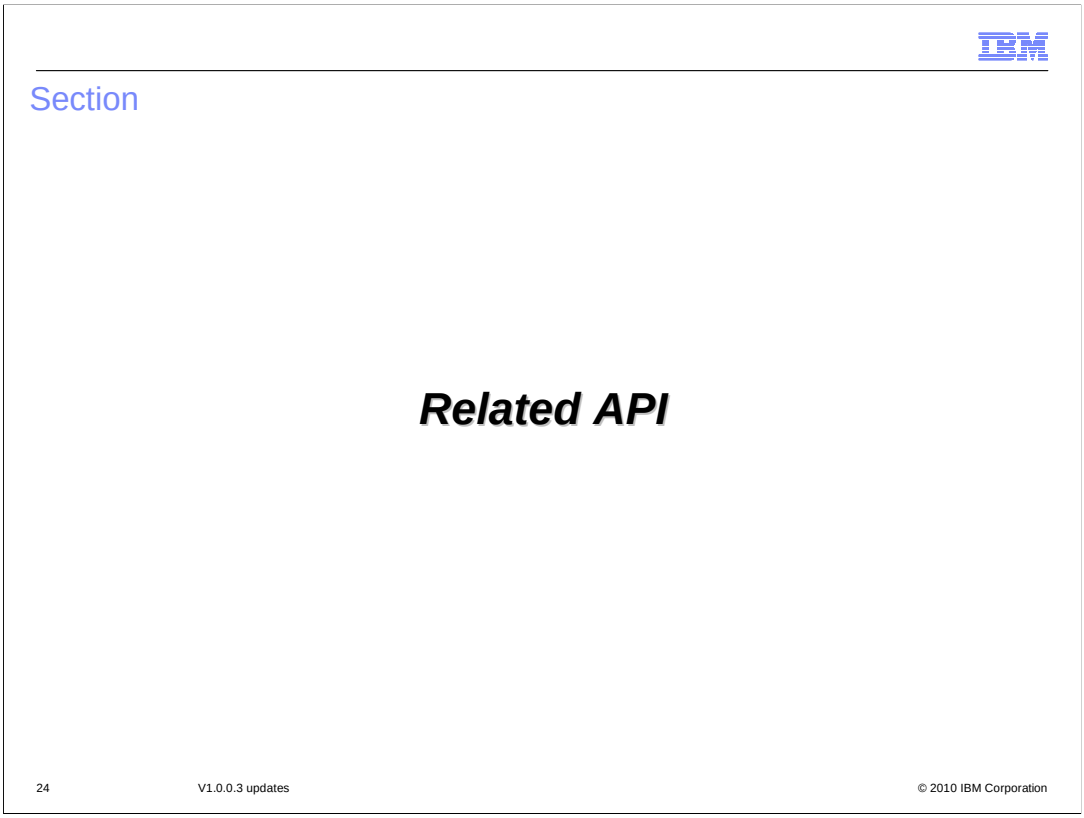
Example: Schema validation in XQuery – Validate element

```
import schema namespace lib="http://www.example.org/library" at "library.xsd";

element lib:mycollection
    {
        element { concat("lib:", "book") }
        {
            attribute title {"The Complete Works of William Shakespeare" },
            validate strict {element concat("lib:", “author") } {"William
                 Shakespeare" }
        }
    }
```

group_contributors.xq

The example here will validate an element. The query here creates a lib:mycollection element and its content and validates it against the schema file. If there is not an author element with the value of William Shakespeare, an error is raised.IXJXE1005E: [ERR 0770][ERR XQDY0027] The 'validate' expression has the mode 'strict', and schema validity assessment concludes that the validity of the element is not valid or unknown.



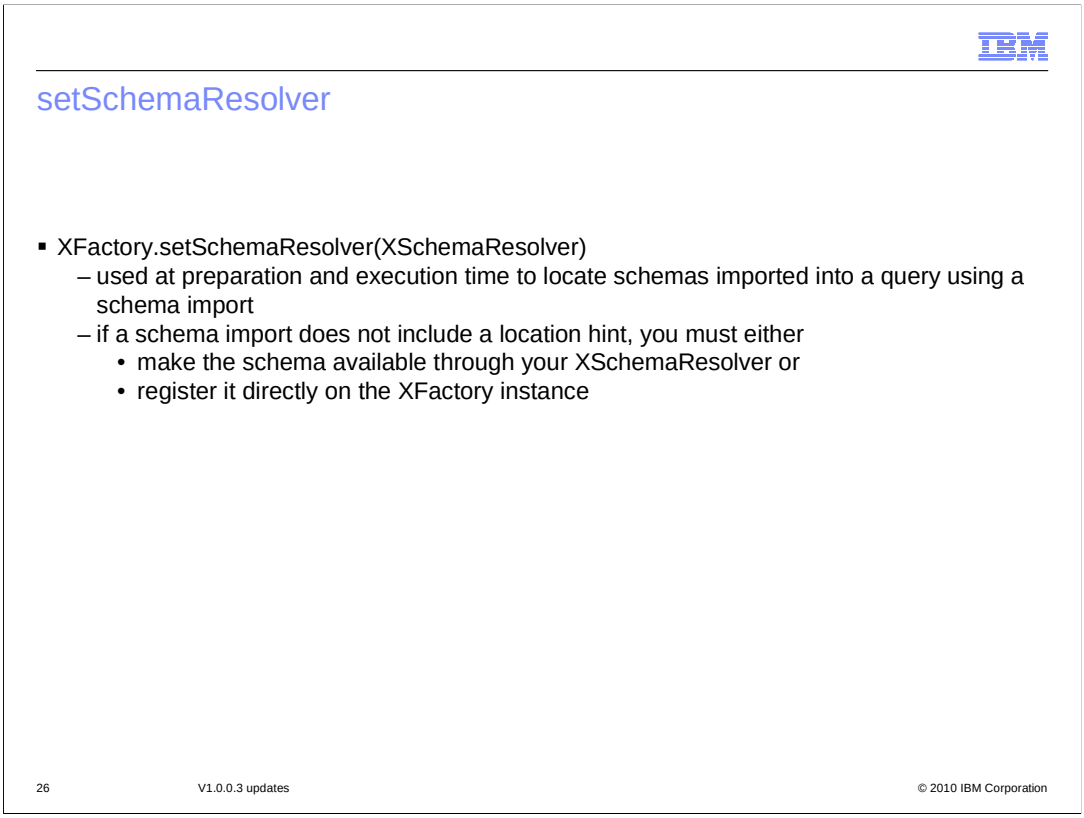
The next section talks about the related API that you can use with the Schema Import and Schema Validation Features.

setValidating and registerSchema

- XFactory.setValidating(int)
  - if set to XFactory.FULL_VALIDATION, input documents are validated against registered schemas
- XFactory.registerSchema(Source) and XFactory.registerSchemas(List<? extends Source>)
  - makes the schema available for use in validating input documents (including those without xsi:schemaLocation attributes)

In order to set validation on the input documents against registered schemas you need to set the setValidating method of XFactor to XFactory.FULL_VALIDATION. A validating factory produces schema-aware executable objects and ensures that source documents get validated against the set of registered schemas before they are processed.

Resolve imports for schemas that are registered with the XFactory using the registerSchema method. XFactory.registerSchema makes the schema available for use to validate input documents. Register a schema has no effect if validating is not enabled. Valid Source types are StreamSource, SAXSource, DOMSource, and StAXSource.



XFactory.setSchemaResolver(XSchemaResolver) can implement the XSchemaResolver interface to help the processor locate schemas. It is used at preparation and execution time to locate schemas imported into a query using a schema import. It is used at execution time to locate schemas applied to an XML document using xsi:schemaLocation or imported into other schemas. If a schema import does not include a location hint, you must either make the schema available through your XSchemaResolver or register it directly on the XFactory instance. Pass in a schema resolver implementation or null to revert to the default schema resolution behavior.

registerImportedSchemas

- XQueryExecutable.registerImportedSchemas
  - registers schemas referenced in schema imports in the query
  - Performance consideration:
    - Call if the XQueryExecutable is used repeatedly and the input documents should be validated against the imported schemas
- Schemas imported into the query are not used for validating the input document by default
  - Use registerImportedSchemas method to register schemas imported so that the types in schemas imported into the query are used for the validation

XQueryExecutable.registerImportedSchemas, registers schemas referenced in schema imports in the query. The schema must be locatable through a location hint or XSchemaResolver. If the XQueryExecutable is used repeatedly and the input documents should be validated against the imported schemas, it is advisable to register the imported schema, otherwise the schema is loaded on each execution. By default, schemas imported into the query are not used for validating the input document, the input document will still be validated but the types in schemas imported into the query are not used for the validation. You must use the registerImportedSchemas method to register schemas imported in the query so that they are used for validating the input document. This also relates to the performance because if registerImportedSchemas is not called then the imported schemas are reprocessed on each execution.

Preparing and executing an interpreted XQuery expression with schema awareness

```
// Create the factory
  XFactory factory = XFactory.newInstance();
// Enable validation
  factory.setValidating(XFactory.FULL_VALIDATION);
// Register the schema, if necessary
  factory.registerSchema(new StreamSource("library.xsd"));
// Set up an XSchemaResolver, if necessary
  factory.setSchemaResolver(...);
// Create the query source
  StreamSource query = new StreamSource("query.xq");
// Create an XQuery executable object for the query
  XQueryExecutable queryExec = factory.prepareXQuery(query);
// Register imported schemas (recommended for performance)
  queryExec.registerImportedSchemas();
// Create the input source
  StreamSource input = new StreamSource("schema.xml");
// Create the result
  StreamResult result = new StreamResult(System.out);
// Execute the query
  queryExec.execute(input, result);
```

Java file

Here is an example showing off how to prepare and execute an interpreted XQuery expression with schema awareness. Notice the enabling of validation, registering the schema, setting the schema resolver, and registering the imported schema.

Register schema

- The XML file does not have a schema location so it must be registered with the XFactory:

In XML file:

```
<lib:mycollection
   xmlns:lib="http://www.example.org/library">
```

In Java file:

```
XFactory factory = XFactory.newInstance();
factory.setValidating(XFactory.FULL_VALIDATION);
factory.registerSchema(new StreamSource("library.xsd"));
```

In XQuery file:

```
import schema default element namespace "http://www.example.org/library";
```


Here is an example of using the registerSchema method. The XML file does not have a schema location. The import in the query does not list the location, so the registerSchema method is used to register the schema to be used for validation.

XML file has schema location

- The XML has a schema location so it does not need to be registered with the XFactory:

In XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<lib:mycollection
   xmlns:lib="http://www.example.org/library"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.example.org/library library.xsd ">
```

In Java file:

```
  XFactory factory = XFactory.newInstance();
  factory.setValidating(XFactory.FULL_VALIDATION);
```

In XQuery file:

```
import schema default element namespace "http://www.example.org/library" at "library.xsd";
```


Here is an example where the XML file has a schema location, so no extra method is needed. Remember that you still need to set validating to FULL_VALIDATION..

Register imported schemas

- Use registerImportedSchemas to register the schema imported into the query so it can be used for input validation
  - The XML does not have a schema
  - The query imports the schema

In XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<lib:mycollection xmlns:lib="http://www.example.org/library">
```

In Java File:

```
XFactory factory = XFactory.newInstance();
factory.setValidating(XFactory.FULL_VALIDATION);
XQueryExecutable executable = factory.prepareXQuery(new StreamSource("library2.xq"));
executable.registerImportedSchemas();
```

In XQuery file:

```
import schema default element namespace "http://www.example.org/library" at "library.xsd";
```


Here is an example of using the registerImportedSchemas method. The XML file does not have a schema location, but the schema is imported into the XQuery file. The registerImportedSchemas method is used to register the imported schema to be used for input validation.

Section

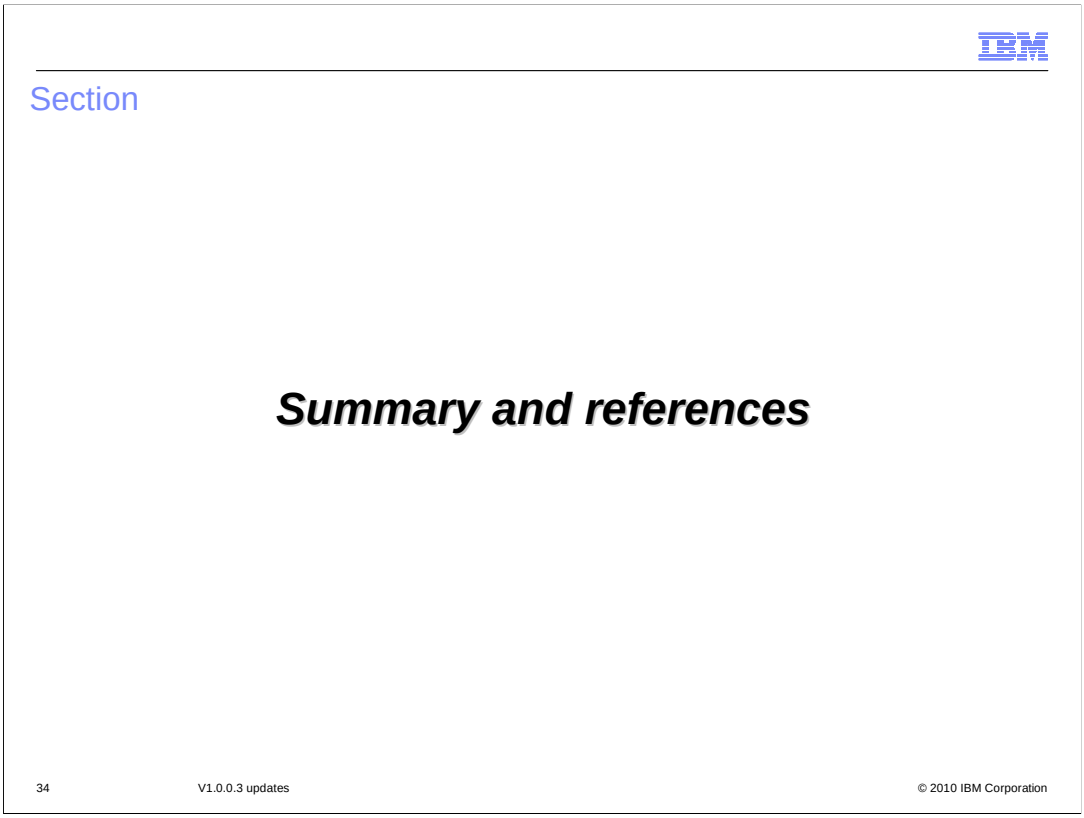
-validating flag on command-line execution tools

The next section provides a summary of the -validate flag on the command-line execution tools that is added in the Feature Pack for XML V1.0.0.3

-validating option

- -validating option for the ExecuteXPath, ExecuteXQuery, and ExecuteXSLT command line tools is added in the V1.0.0.3 of the feature pack
  - Allows users to enable schema aware processing and validation when an schema is imported and a schema location is specified
  - For the XPath there is no way to import a schema in the XPath expression itself
    - -schema option must still be used at compile time
- Valid values are
  - "full“ - turns on schema aware processing and validation
  - "none“ - turns off validation

The Feature Pack for XML V1.0.0.3 includes a -validating option for the ExecuteXPath, ExecuteXQuery, and ExecuteXSLT command line tools. The –validating option allows users who have imported the necessary schemas into their stylesheet or query and who have a schema location specified in their input documents to enable schema aware processing and validation. For the XPath case there is no way to import a schema in the XPath expression itself so the -schema option must still be used at compile time if the expression references user defined schema types. Valid values are "full" and "none". Where "full" turns on schema aware processing and validation and "none" turns it off.



The next section provides a summary and references.

Summary

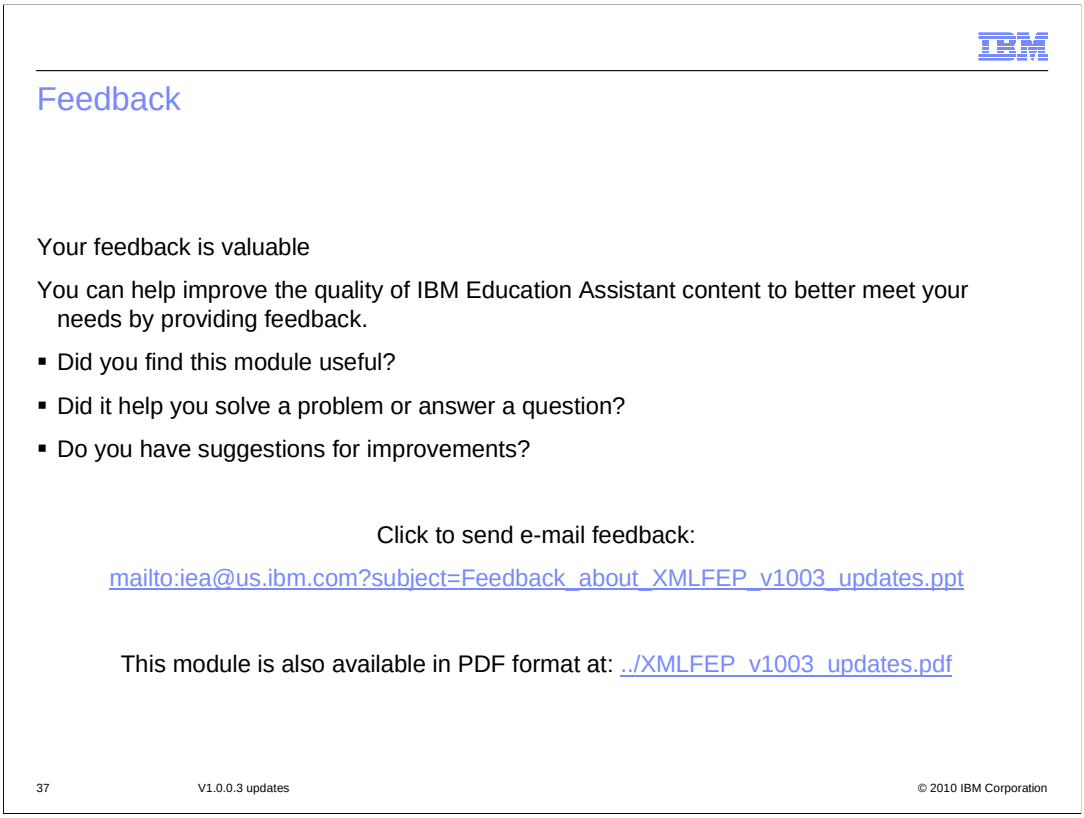
- Version 1.0.0.3 of the feature pack includes the XQuery specific optional features:
  - Schema import feature
    - Permits a schema import to be included in the prolog of a query
    - Allows the query to refer to types, elements, and attributes declared in imported schemas
    - Including a schema import is optional (alternatively, it can be registered with the XFactory) but recommended for portability
  - Schema validation feature
    - Permits a validate expression to appear in a query, which allows validation of document and element nodes

The Feature Pack for XML Version 1.0.0.3 includes the optional XQuery features Schema Import and Schema Validation. These permit a schema import to be included in the prolog of the query and permits a validate expression to appear in the query, allowing validation of document and element nodes.

References

- WebSphere Application Server Feature Pack for XML
  http://www.ibm.com/software/webservers/appserv/was/featurepacks/xml/
- Information center
  http://www14.software.ibm.com/webapp/wsbroker/redirect?version=v700xml&product=was-nd-mp
- Primary specifications
  http://www.w3.org/TR/xpath20/
  http://www.w3.org/TR/xslt20/
  http://www.w3.org/TR/xquery/

Here are some useful links.



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

Trademarks, disclaimer, and copyright information