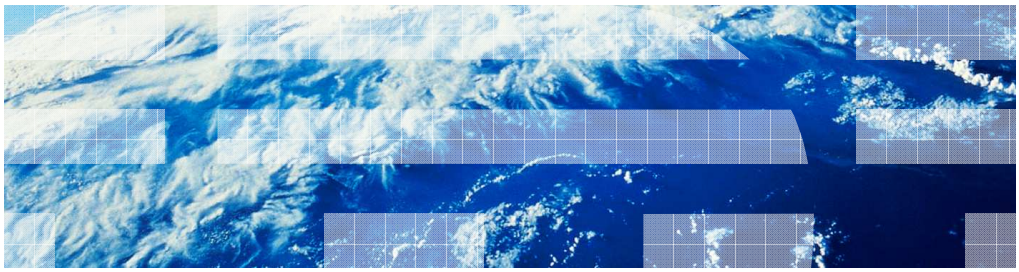
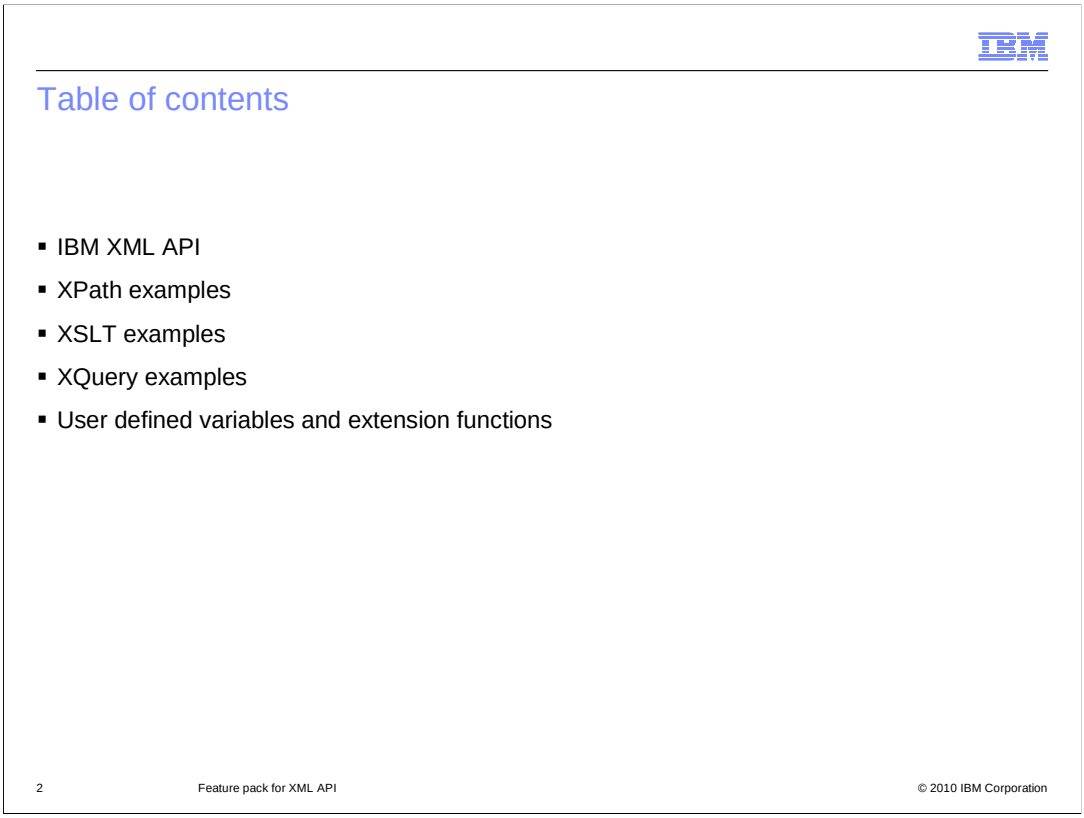
IBM WebSphere Application Server Feature Pack for XML

Feature pack for XML API

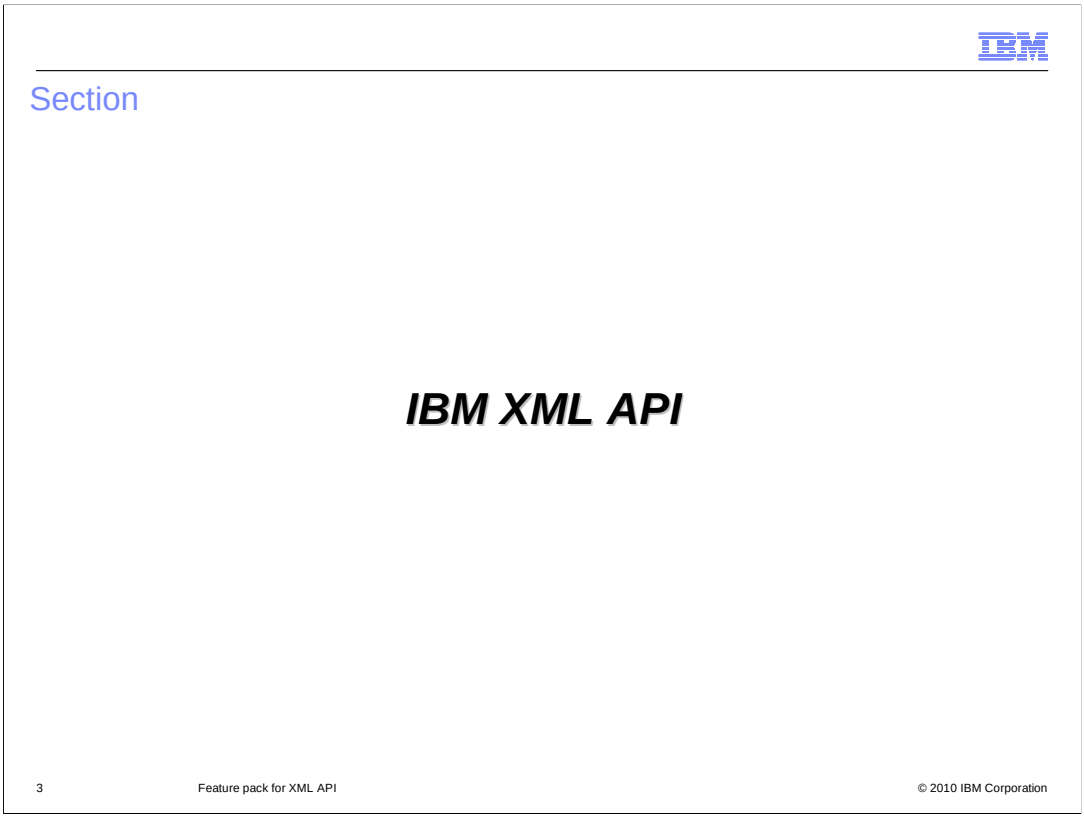


WebSphere. software

This presentation will go through the feature pack for XML APIs.



This presentation will start off going over the IBM XML API and then go through some examples using the API.



The next slides are an overview of the IBM XML API.

Basic feature pack for XML API usage

Path	XSLT	XQuery

/Create the factory
Factory factory = XFactory.*newInstance();*
/Could be from StreamSource
tring xpathString = SOME_STRING;
	//Create the factory
	XFactory factory = XFactory.newInstance();
	//Create source from file
	Source xslt = new
	StreamSource(getResourceAsStream(xsltfile));
		//Create the factory
		XFactory factory = XFactory.newInstance();
		//Could be from StreamSource
		String queryString = SOME_STRING;

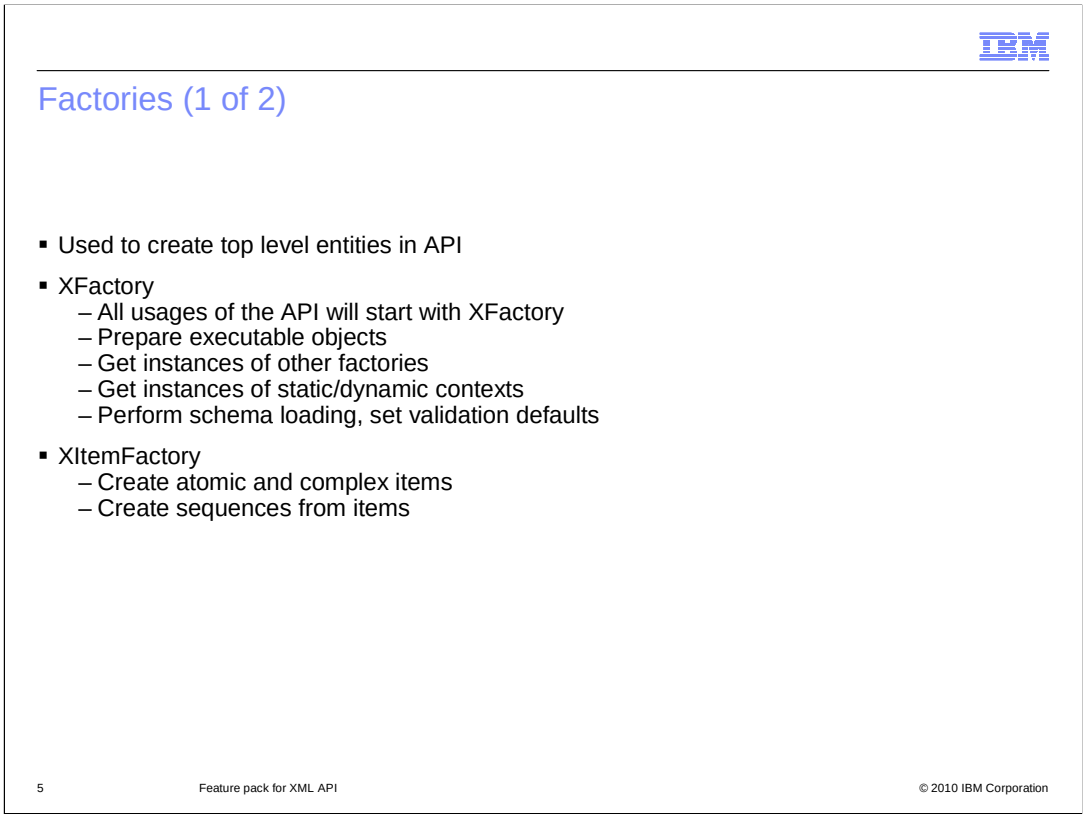
/Create an XPath executable object
PathExecutable xpath =
actory.prepareXPath(xpathString);
	//Create an XSL transform
	XSLTExecutable xslTransform
	= factory.prepareXSLT(xslt);
		//Create XQuery executable object
		XQueryExecutable xquery =
		factory.prepareXQuery(queryString);

/Create the input source
ource source = new
treamSource(getResourceAsStream(inputfile));
	//Create the input source
	Source source = new
	StreamSource(getResourceAsStream(inputfile));
	//Create the output source
	Result result = new
	StreamResult(new ByteArrayOutputStream());
		//Create the input source
		Source source = new
		StreamSource(getResourceAsStream(inputfile));

/Execute the XPath
SequenceCursor sequence =
xpath.execute(source);
	//Execute the transformation
	xslTransform.execute(source, result);
		//Execute the XQuery
		XSequenceCursor sequence =
		xquery.execute(source);

/Print out the result
f (sequence != null) {
o{System.out.println(sequence.getStringValue());
while (sequence.toNext());}
	//Serialized result is available in the
	ByteArrayOutputStream
		//Print out the result
		if (sequence != null) {
		do{System.out.println(sequence.getStringValue())
		}while (sequence.toNext());}

The table here shows the consistent steps across executing XPath/XSLT/XQuery with the XML API usage. Going through the table you first create a factory for executable objects and context item. You then create an executable object. XPath, XSLT, and XQuery all use the common XExecute base class. The next steps are to create a source for the executable object, run the executable object, and then navigate and print the results.



The XFactory class is the main factory class for creating executable objects for XPath, XQuery, and XSLT. It is also the means for creating instances of other classes and factories such as the XStaticContext, XDynamicContext, XItemFactory, and XSequenceTypeFactory classes. An instance of XFactory maintains a set of registered schemas as well and can be validating or non-validating. An XFactory instance can be created by calling the static newInstance() method on the XFactory class. The instance is thread safe as long as the settings remain stable.

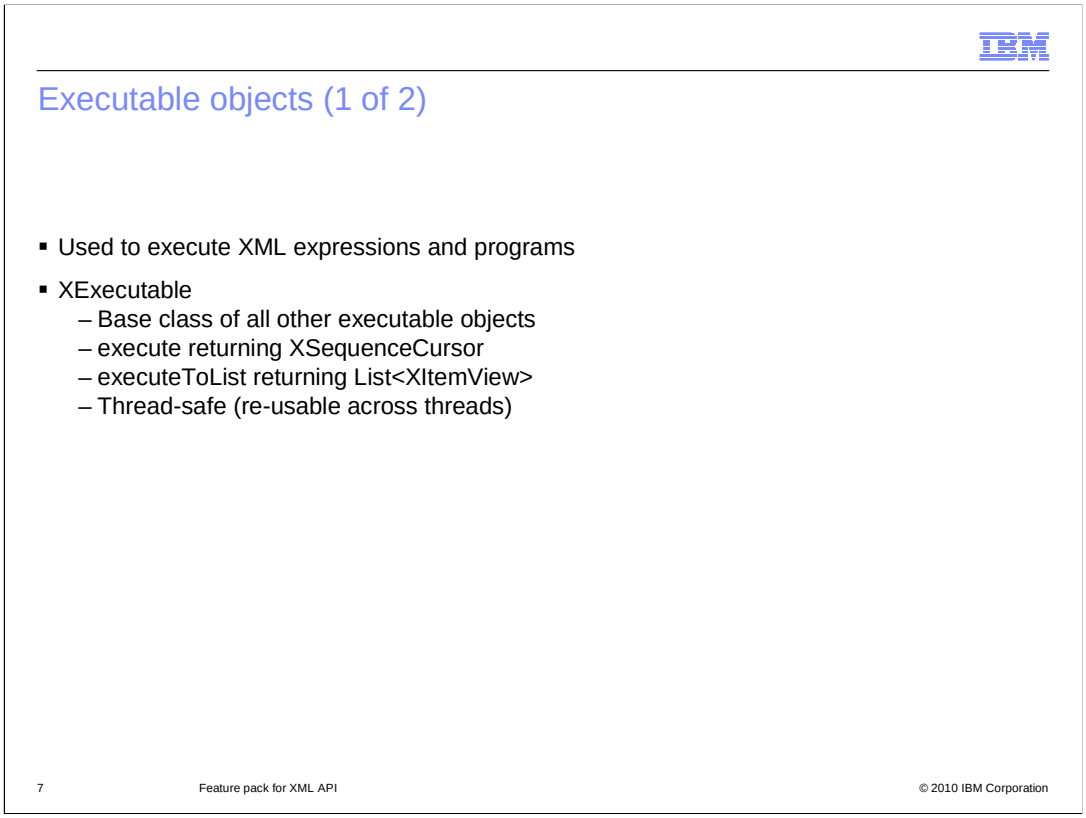
You can use XItemFactory to create new items and sequences of items of different types. This factory can be used to create items and sequences from Java™ objects and primitives. Many of the methods in the API accept Java objects and primitives directly (such as the bind methods in XDynamicContext for binding variable values), however, if a heterogeneous sequence is required this factory can be used to create it by first creating the individual items using the item method and then using the sequence method that takes an array of items.

Factories (2 of 2)

- XSequenceTypeFactory
  - Create sequence types to represent types and cardinality of sequence items
- XCompilationFactory
  - Compile XML artifacts and load (from pre-compiled classes) compiled artifacts
  - Used with XCompilationParameters

XSequenceTypeFactory is used for creating sequence types. The sequence type factory associated with an XFactory instance can be obtained by calling the getSequenceTypeFactory method on the XFactory instance.

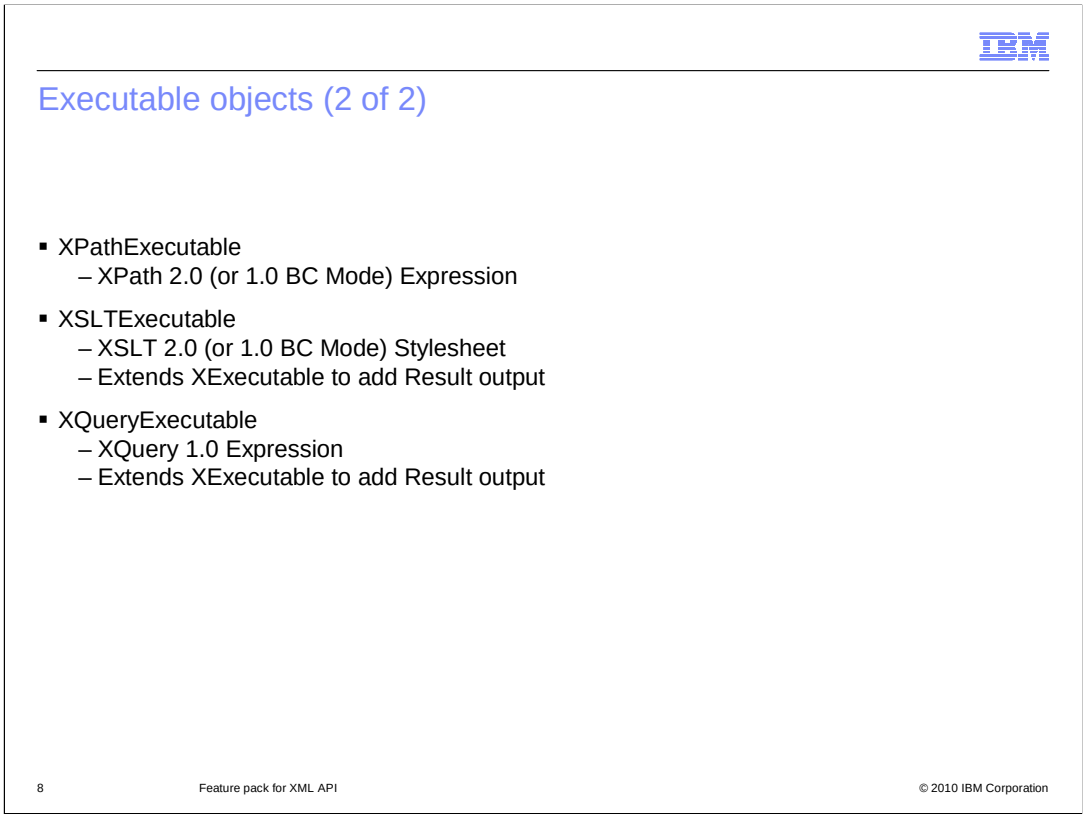
XCompiliationFactory for compiling expressions, queries and stylesheets into Java classes. Load methods are provided for loading the compiled Java classes and instantiating the executable object. Compiling the expression, query or stylesheet ahead of time means that the cost of preparation can be avoided during the application runtime. The getCompilationFactory method on XFactory can be used to get an XCompilationFactory instance. XCompiliationFactory is used with **XCompilationParameters, which is an** interface for compilation parameter settings that is used when generating precompiled Java classes for an expression, query or stylesheet and when loading precompiled classes using the XCompilationFactory compile and load methods. New compilation parameters objects can be created through the XCompilationFactory newOutputParameters method.



XExecutable is the base executable class. It provides common execute methods. Parameters to the execute methods include either a JAXP Source object or an XItemView object for the initial context item. Valid Source types are StreamSource, SAXSource, DOMSource, and StAXSource. XExecutable also can include an XDynamicContext object containing the execution-time settings. If none is provided the default settings are used.

The return value is an XSequenceCursor which is a cursor view of the resulting sequence of items. A List of XItemView return type is also provided for random access. If the result is the empty sequence and the return type is XSequenceCursor then the value is null. If the result is empty and the return type is List then the value is an empty list. Note that XSequenceCursor implements XItemView allowing the result of one execution to be passed in as the context item for another.

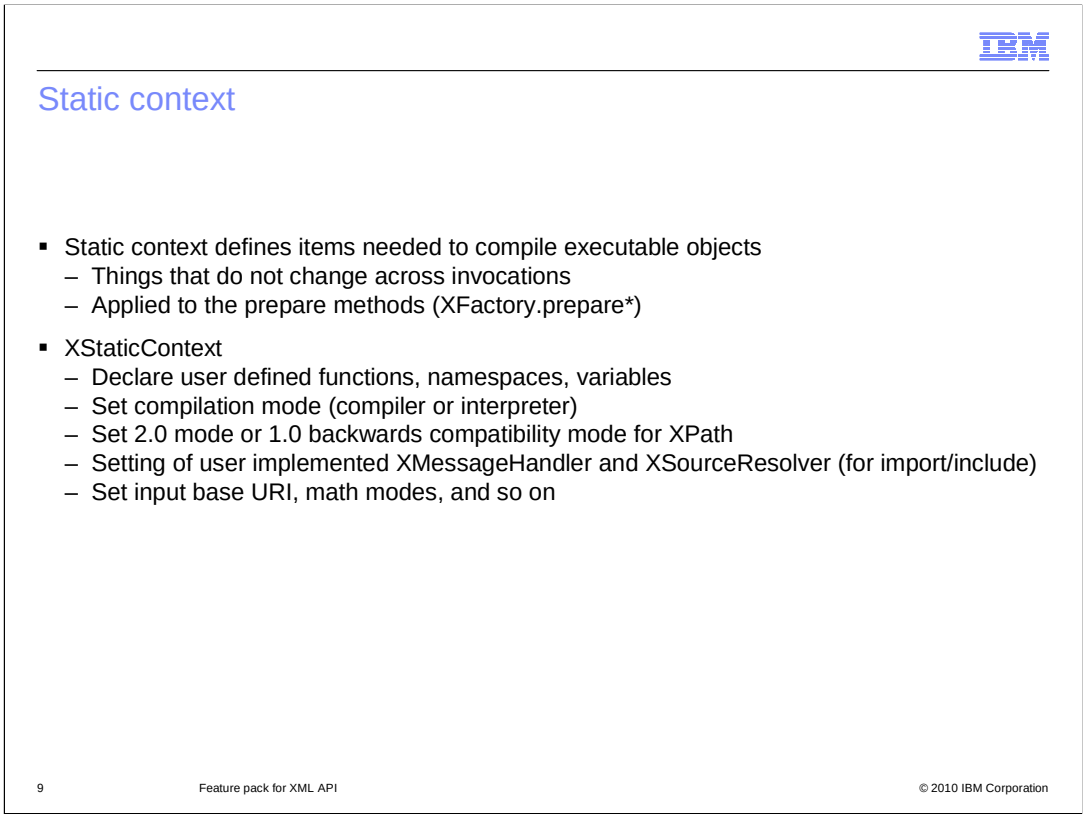
All XExecutable objects are thread safe.



XPathExecutable extends XExecutable and represents a prepared XPath expression. All XPathExecutable objects are thread safe. This interface is intentionally empty and you should refer to the XExecutable for available methods.

XSLTExecutable extends XExecutable and represents a prepared XSLT stylesheet. The parameters to the execute method is the same as XExecutable. With an XSLTExecutable you need to pass in an JAXP Result object to serialize the result to. Valid Result types are StreamResult, SAXResult, DOMResult, and StAXResult. All XSLTExecutable objects are thread safe.

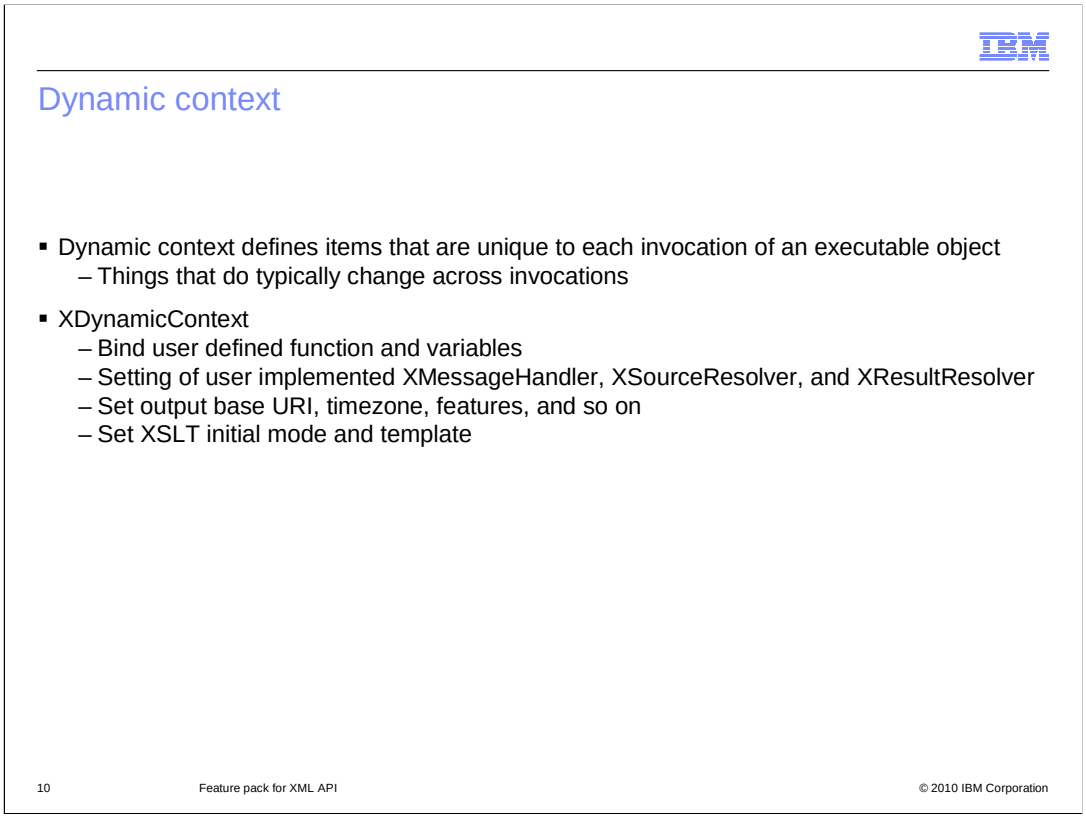
XQueryExecutable extends XExecutable and represents a prepared XQuery expression. The parameters to the execute method is the same as XExecutable. You can pass in a result object to serialize the result to. Valid Result types are StreamResult, SAXResult, DOMResult, and StAXResult. All XQueryExecutable objects are thread safe.



XStaticContext is used to override prepare time settings. These are settings used when preparing an expression, query or stylesheet. The XFactory class has the method newStaticContext for creating new instances of XStaticContext. Prepare time settings such as whether to use the interpreter or the compiler, the integer math mode to use, the names and types of external variables and functions, backwards compatibility mode for XPath and so on, are built directly into the executable object and cannot be changed at execution time. Execution-time settings, alternatively, such as the values of external variables and parameters, and the implementation of external functions are set using the XDynamicContext and can be different for each execution.

The following are examples of methods on XStaticContext used for setting prepare-time characteristics are.

The setUseCompiler method is used to set if the executable object is compiled or interpreted. The declareVariable method is used to declare variables. The type of variable affects how the expression, query, or stylesheet gets compiled and therefore is a prepare-time characteristic. Alternatively, binding the value of a variable is an execution-time characteristic, and the value can be different for each execution. The setSourceResolver method registered at prepare time is used to resolve includes and imports. A source resolver can also be registered at execution time, but it is used for a different purpose.



XDynamicContext is used to override execution-time settings. These are settings used when executing a prepared expression, query or stylesheet. The XFactory class has the method newDynamicContext for creating new instances. Execution time settings such as the values of external variables and resolvers for source, collections and results can be different for each call to the execute methods on an executable object.

Examples of execution-time settings: Bind values to external variables (XPath and XQuery) or parameters (XSLT), Bind an external function implementation, Set source, collection and result resolvers, and Bind a Collator or a Locale to a collation URI.

Settings in XDynamicContext map to settings in the XPath, XQuery and XSLT dynamic context as defined in the specifications:

Examples of methods on XDynamicContext used for setting execution-time characteristics include bind methods used to supply a value for a variable. The value can be different for each execution.

The setSourceResolver method registered at execution time is used to resolve input documents loaded with the XPath fn:doc function or the XSLT fn:document function.

The setXSLTInitialTemplate method identifies the initial template to invoke for an XSL transformation.

Resolvers (1 of 2)

- Used to resolve input stylesheets, input documents and output documents
- XSourceResolver
  - Under static context - used to resolve imported and included stylesheets
  - Under dynamic context -used to resolve dynamically resolved input documents (ex. fn:doc)
- XResultResolver
  - Under dynamic context –used to resolve dynamically resolved output documents (ex. xsl:result-document)
- XCollectionResolver
  - Used at runtime by collection() function to retrieve a collection of nodes from arbitrary sources

Resolvers are used to resolve input stylesheets, input documents, and output documents.

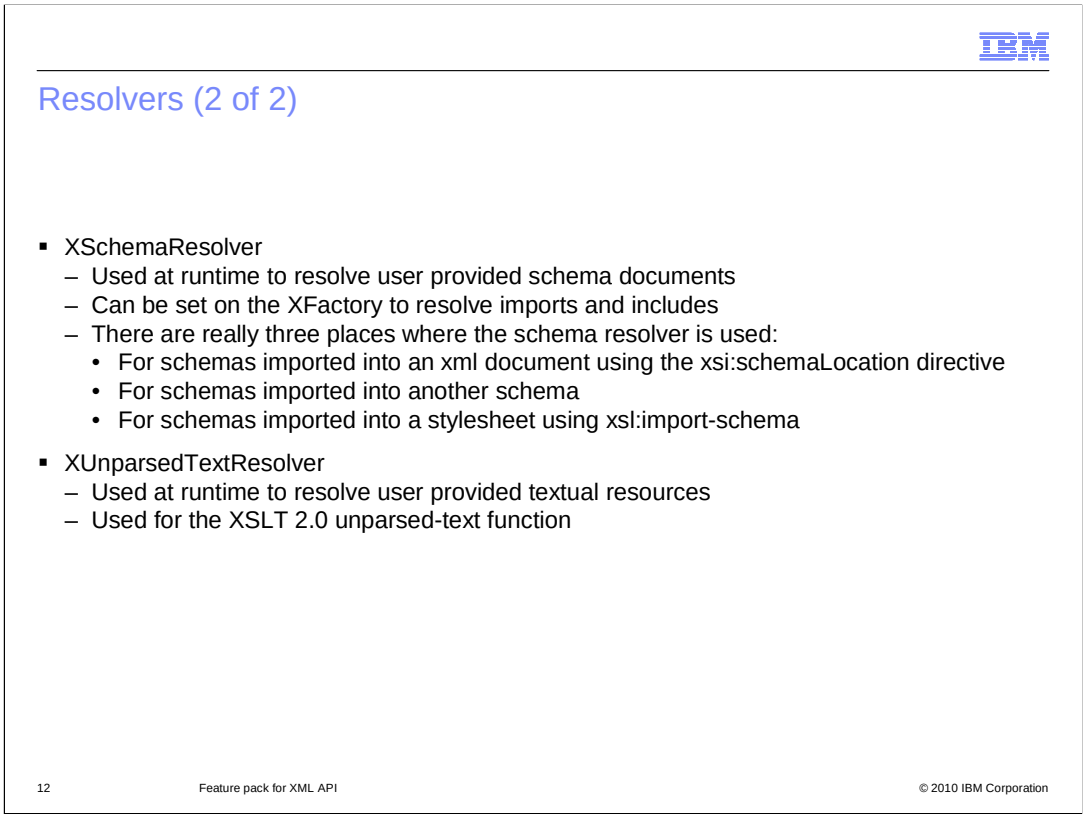
The **XSourceResolver** is used to override the default source document resolution behavior. Register with the static context to resolve imports and includes. Register with the dynamic context to resolve documents loaded using the XPath fn:doc function and the XSLT document function. The default source resolution behavior for imports and includes is to use the base URI of the expression, query, or stylesheet to resolve relative URIs. If the base URI is not available, the current working directory is used. Absolute URIs are used unchanged. The default source resolution behavior for documents loaded using the XPath fn:doc function is to resolve relative URIs based on the base URI from the static context. If the base URI is not available, the current working directory is used. Absolute URIs are used unchanged.

The default source resolution behavior for documents loaded using the XSLT document function is described in the XSLT 2.0 specification. If no base URI is available, the current working directory is used.

XResultResolver is used to override the default resolution behavior for the URI reference in XSLT xsl:result-document instructions. Resolution depends on the URI reference from the xsl:result-document instruction and the base output URI. The base output URI can be set in the XDynamicContext using the setBaseOutputURI method. If it is not set then the base URI of the main result document (as passed in to one of the execute methods on the executable object) is used. If this is not available then the current working directory is used. The default resolution behavior is to use the base output URI to resolve result documents if the URI reference is relative. Absolute URIs are used unchanged.

Supported Result types are StreamResult, SAXResult, DOMResult, StAXResult, and XSequenceCursorResult.

XCollectionResolver is used to resolve the URI provided on a call to fn:collection to a sequence of nodes. If no collection resolver is registered with the XDynamicContext then calls to fn:collection will result in a recoverable error and the empty sequence is used for the collection.



XSchemaResolver is used to override the default schema resolution behavior. Implementations should be registered with the XFactory and are used to resolve imports for schemas that are registered with the XFactory using the registerSchema method as well schemas that are imported in XSLT using the xsl:import-schema declaration. The default behavior for resolving imports within a schema is to use the base URI of the schema to resolve the imported schema's location. The default behavior for XSLT schema imports is to use the base URI of the xsl:import-schema declaration to resolve the location specified in the declaration.

XUnparsedTextResolver is used to override the default unparsed text resolution behavior. Register with the dynamic context to resolve resources loaded using the XSLT unparsed-text function.

The default unparsed text resolution behavior for resources loaded using the XSLT unparsed-text function is to resolve relative URIs based on the base URI from the static context. If the base URI is not available, the current working directory is used. Absolute URIs are used unchanged.

Compiler and interpreter

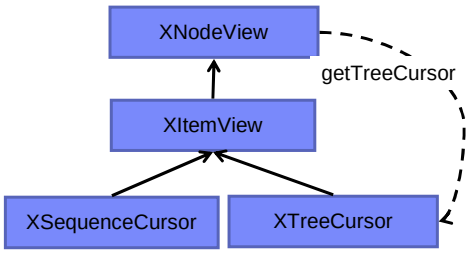
- You can use either the compiler or the interpreter for preparing and executing an XQuery expression, XPath expression, or XSLT stylesheet
  - setUseCompiler(true) method on the XStaticContext to use the compiler
  - setUseCompiler(false) method to use the interpreter (default)
- Expressions, queries, and stylesheets can also be prepared ahead of time (precompiled using the XCompilationFactory)
- It takes longer to prepare a compiled executable object than an interpreted executable object, but a compiled executable object generally runs faster

You can use either the compiler and the interpreter for preparing and executing an XQuery expression, XPath expression, or XSLT stylesheet. Choosing which one to use is very application specific and depends on several factors. Use the setUseCompiler(true) method on the XStaticContext to use the compiler, and use the setUseCompiler(false) method to use the interpreter. The default is to use the interpreter for preparing an XQuery expression, XPath expression, or XSLT stylesheet.

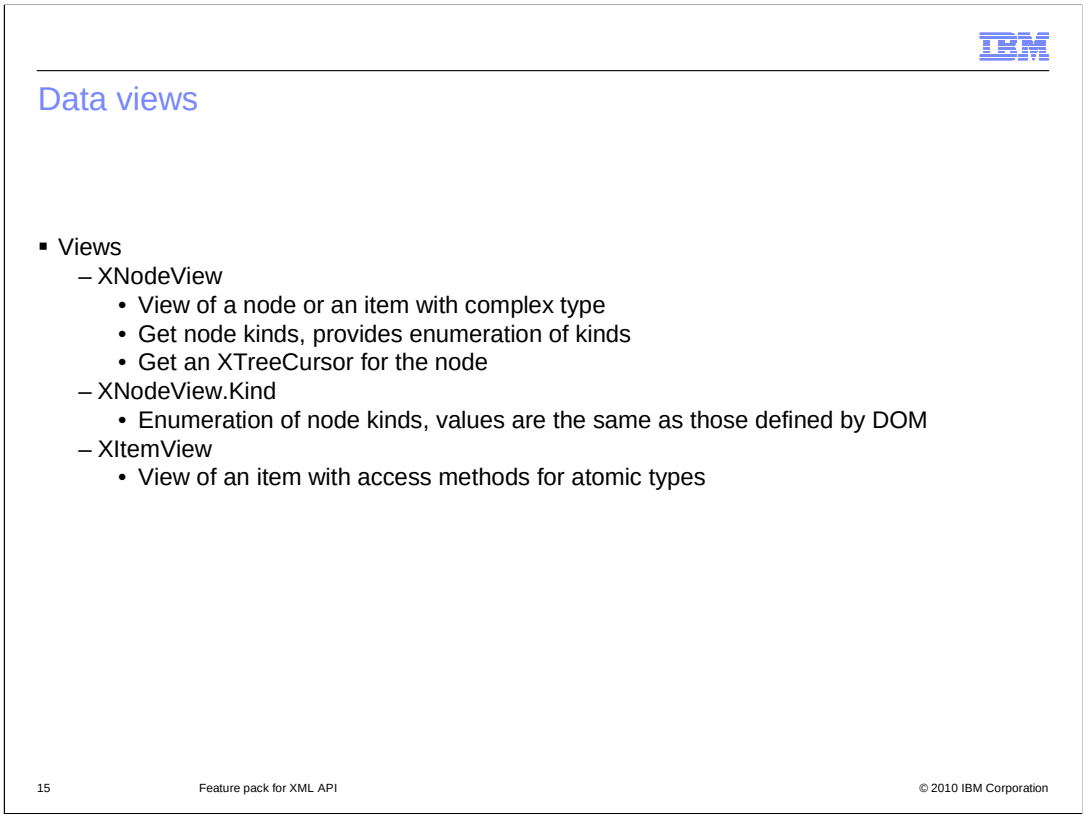
Expressions, queries, and stylesheets can also be prepared ahead of time (precompiled). This is the most efficient option because the preparation is done ahead of time instead of during the application run time, but precompiling might not be applicable to all applications. Remember that it takes longer to prepare a compiled executable object than an interpreted executable object, but a compiled executable object generally runs faster; therefore, there is a trade off between the cost of preparing a compiled executable object and the improved execution-time efficiency.

Data views and cursors

- Used for navigating (cursors) and viewing (views) data



Data views are used to view data and cursors are used to navigate data. The next few slides will talk about the available data views and cursors included in the XML API.



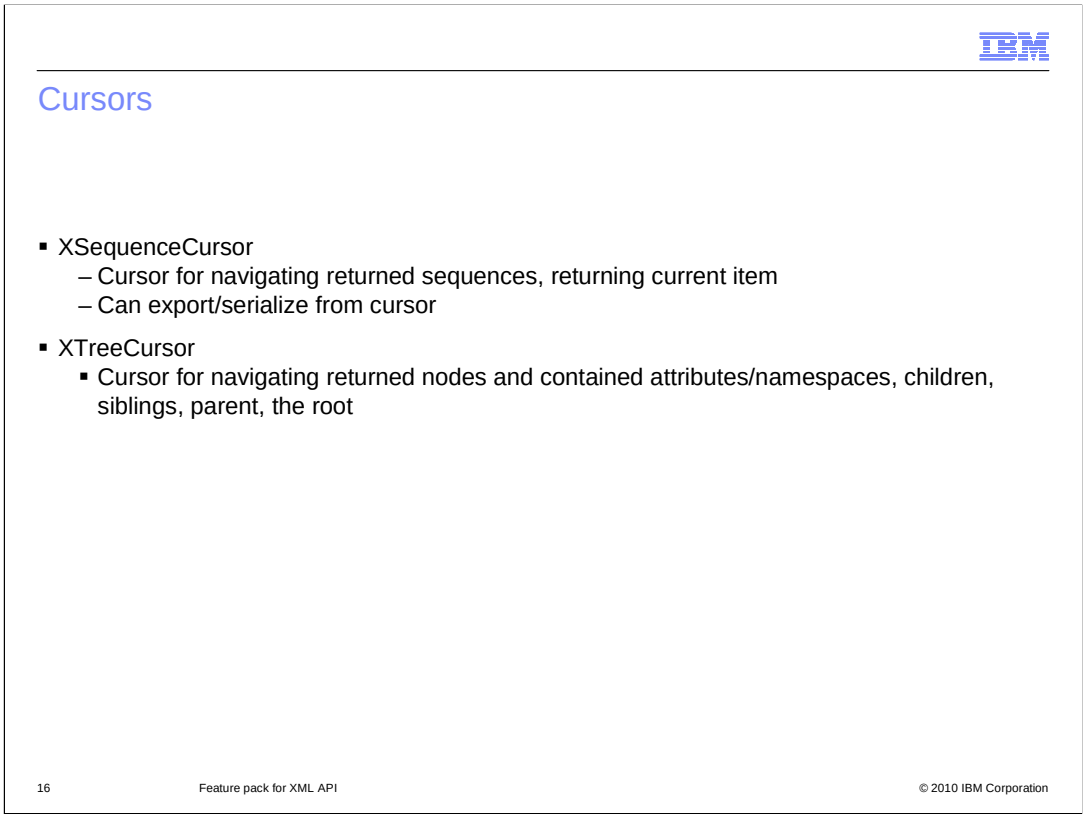
XNodeView represents a node in the data model (an item with complex type). An object of type XNodeView should never occur, rather XItemView extends XNodeView so that items of complex type can be examined as nodes. The isAtomic method should be called on an XItemView object before any of the XNodeView methods are called as they are not valid for atomic items. If an XNodeView method is invoked on an atomic item, an XViewException is thrown.

XNodeView.Kind is an Enumeration of node kinds. Values are the same as those defined by DOM.

XItemView extends XNodeView and represents an item in the data model. You can access each item in a sequence using the methods on the XItemView interface. You can use the XItemView.isAtomic() method to determine whether the item is an atomic value or a node. If the item is an atomic value, you can use the getValueType() method on the XItemView interface. This method returns an instance of the enumerated type XTypeConstants.Type. If the atomic value is an instance of a built-in atomic type or a user-defined type derived from a built-in atomic type, the result of the getValueType() method is the enumerated value corresponding to that type. For an atomic value that can be of a user-defined derived type, you also might find it handy to use the XItemView.getValueTypeName() method to determine the precise type of the value.

Given the type of an atomic value, you then can use the appropriate method to get the value.

The XItemView interface also extends the XNodeView interface, so if the result of calling the XItemView.isAtomic() method is false – that is, the item is a node – you can use the methods inherited from the XNodeView interface to access information about the node. The XNodeView.getNodeQName() method will return the name of the node and the XNodeView.getKind() method will return a value of the enumerated type XNodeView.Kind indicating what sort of node the item is: XNodeView.DOCUMENT, XNodeView.ELEMENT, and so on.



XSequenceCursor extends XItemView and represents a sequence of items providing cursor access to the items. Cursor traversal can be achieved through the toNext and toPrevious methods. When either of these methods is used the cursor will change state as it will now be pointing to a different item in the sequence. This means that other references to the same XSequenceCursor object will also now be pointing to the next or previous item. XSequenceCursor implements XItemView so that each item can be examined as the cursor is traversed. XItemView in turn implements XNodeView so that items of complex type can be examined as well. Note that null is used to represent an empty sequence so an XSequenceCursor will always contain at least one item.

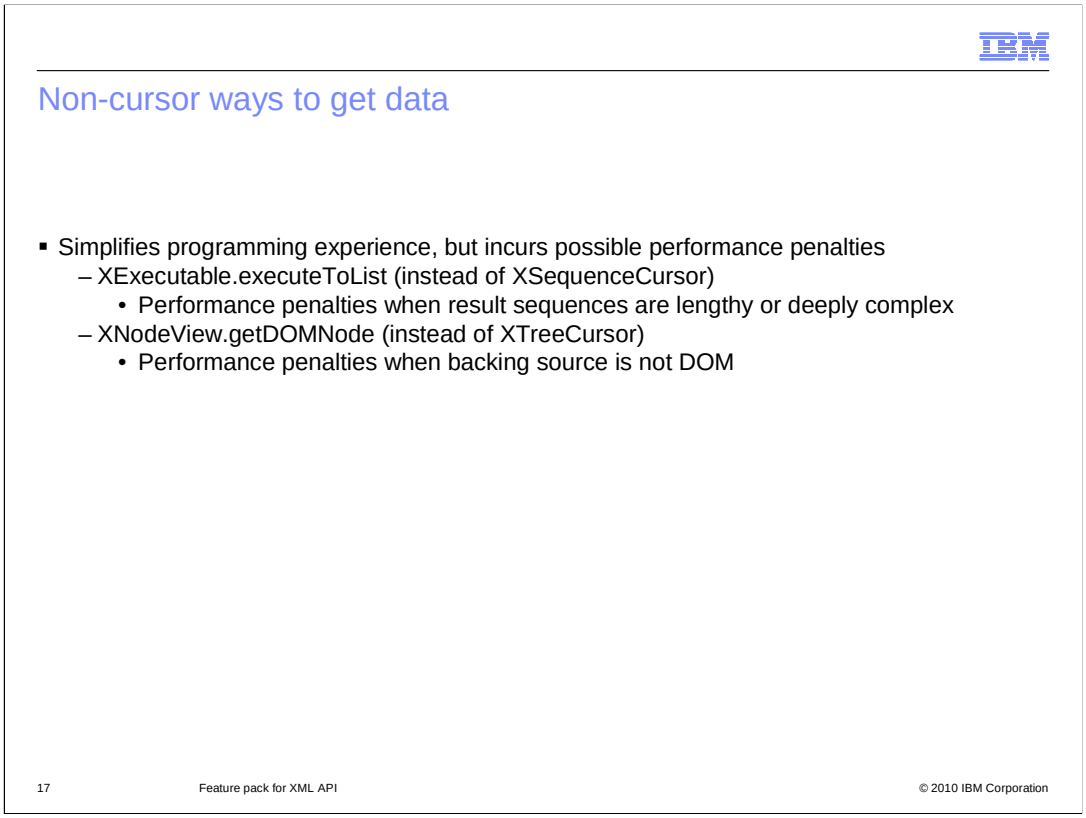
At any given time, an instance of the XSequenceCursor interface is positioned to give you access to one of the items in a sequence. It is positioned initially at the first item in the sequence; and you can move forward or backward in the sequence, one item at a time, using the toNext() or toPrevious() methods. If the toNext() method or toPrevious() method is able to position the XSequenceCursor instance to the next or previous item in the sequence—that is, if there actually is a next or previous item—the method returns true. If the XSequenceCursor instance is already positioned at the last item in the case of the toNext() method or the first item in the case of the toPrevious() method, the method returns false and the instance of the XSequenceCursor interface remains positioned at the same item as before the call.

XTreeCursor extends XItemView and represents a node providing cursor access to the attributes and children of the node. Traversal of the tree is achieved through the various "to" methods. When any of these methods is used the cursor will change state as it will now be pointing to a different node in the tree. This means that other references to the same XTreeCursor object will also now be pointing to the new node.

An XTreeCursor can be obtained by using the XNodeView getTreeCursor interface. XTreeCursor implements XItemView which in turn implements XNodeView allowing each to be examined as the cursor is traversed. The XItemView interface provides access to the schema type and typed value of a node. The XNodeView interface provides access to the node name, node kind, and so on.

The following are some examples:

To view Attributes of an Element - First use the toFirstAttribute method to move the cursor



There are non-cursor ways to get data. The executeToList method returns the sequence as an instance of the java.util.List<XItemView> interface, where each item in the sequence that results from evaluating the XPath expression is represented in the list as an instance of the XItemView interface. The entries in that list are in the same order as in the sequence that resulted from evaluating the XPath expression.

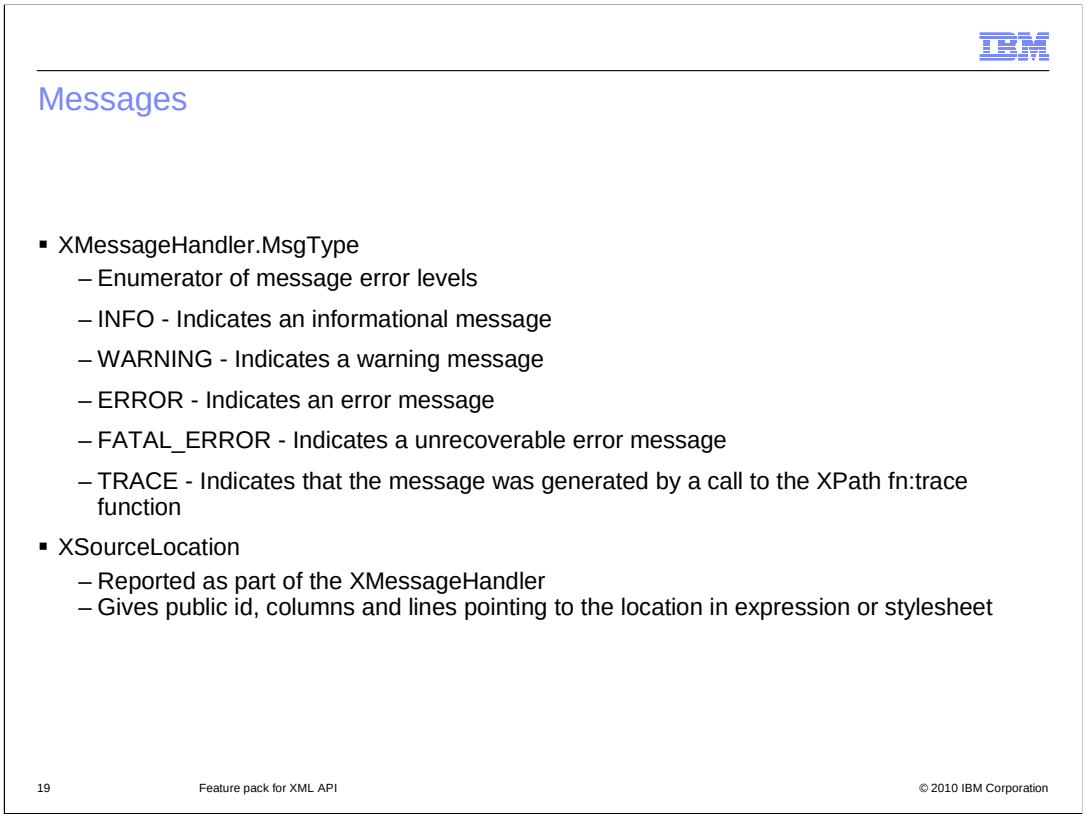
XNodeView.getDOMNode is a non-curor way to get a DOM Node for an item, it can be used instead of XTreeCursor. This can have some performance penalties when the backing source is not DOM.

XMessageHandler

- Implement this class to override the default error and message handling behavior
- Set message handler on the XFactory instance, but it is recommended to create and set in Static and Dynamic Context
  - Message handlers set on the XStaticContext or XDynamicContext will take precedence over a message handler set on the XFactory
- Message handlers are also used for:
  - The XSLT xsl:message instruction
  - The XPath fn:error function
  - The XPath fn:trace function
- At execution time the source location of an error is generally not available
  - Use the xsl:message instruction or the fn:trace function to help locate errors at runtime

XMessageHandler class is used to override the default error and message handling behavior. The default behavior is to print the message to System.err and to also create an XProcessException for unrecoverable errors. The default behavior can be overridden in general by setting the message handler on the XFactory instance. This will affect errors and messages that occur while processing schemas (XFactory.registerSchema), processing source files (XItemFactory.item(Source)) and for errors and messages that occur when preparing or executing expressions, queries or stylesheets (XFactory.prepare, XExecutable.execute). The message handling behavior can also be overridden for individual prepare invocations by setting the message handler on the XStaticContext and for individual execute invocations by setting the message handler on the XDynamicContext. Message handlers set on the XStaticContext or XDynamicContext will take precedence over a message handler set on the XFactory.

You do not have to register a message handler as all messages will go to System.err, but say you want to register a message handler so that you have a chance of changing how a message is handled. For example, warnings are printed to System.err and the processor tries to continue, however, you want to stop processing when there is a warning.



XMessageHandler.MsgType returns an enumerator of message error levels.

INFO - Indicates an informational message. These messages are informational and will not affect the result. This is also the message type that is used for the XSLT xsl:message instruction when the terminate attribute evaluates to "no".

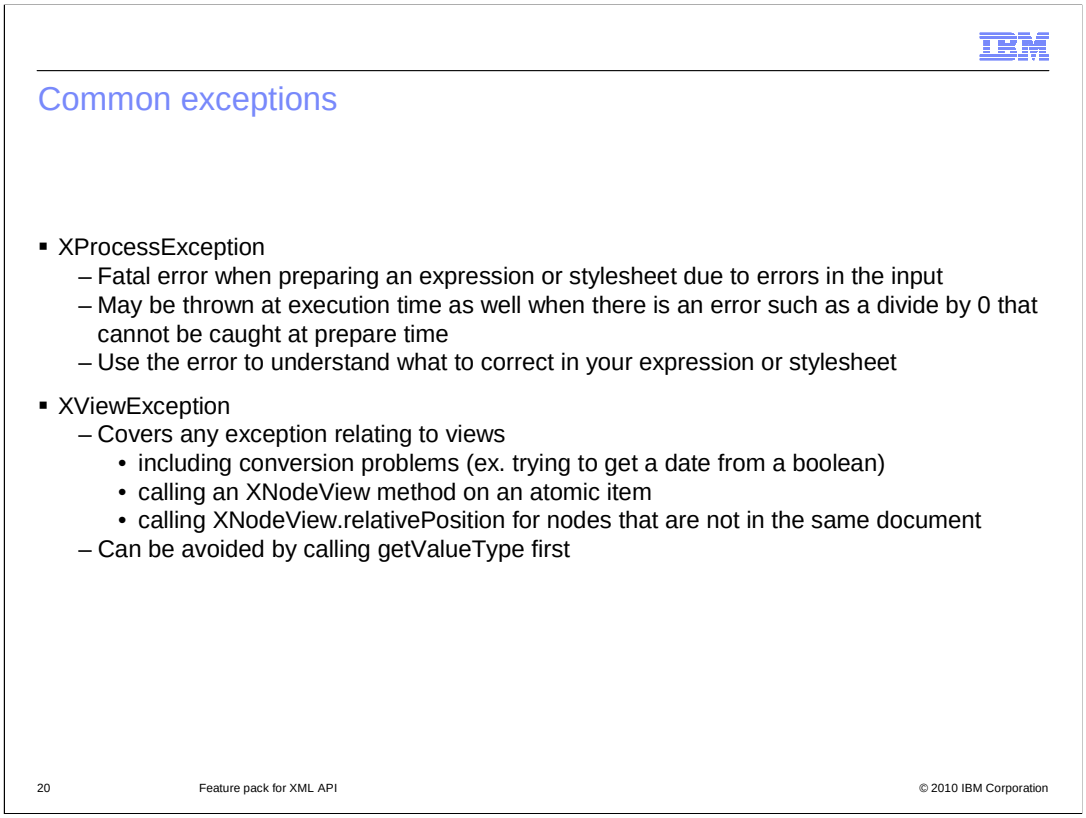
WARNING - Indicates a warning message. The processor will recover from a warning situation but the output can not be what was expected. For example if the value of the xsl:sort data-type attribute is unsupported, a warning is issued and the data-type attribute is ignored.

ERROR - Indicates an error message. The processor can recover from the error for the purpose of signaling additional errors but no result is produced.

FATAL_ERROR - Indicates a unrecoverable error message. The processor cannot recover from these errors and no result is produced. This is also the message type that is used for the XSLT xsl:message instruction when the terminate attribute evaluates to "yes".

TRACE - Indicates that the message was generated by a call to the XPath fn:trace function.

XSourceLocation represents a location in a document. It is passed to the XMessageHandler report method to indicate the location of the error or message.

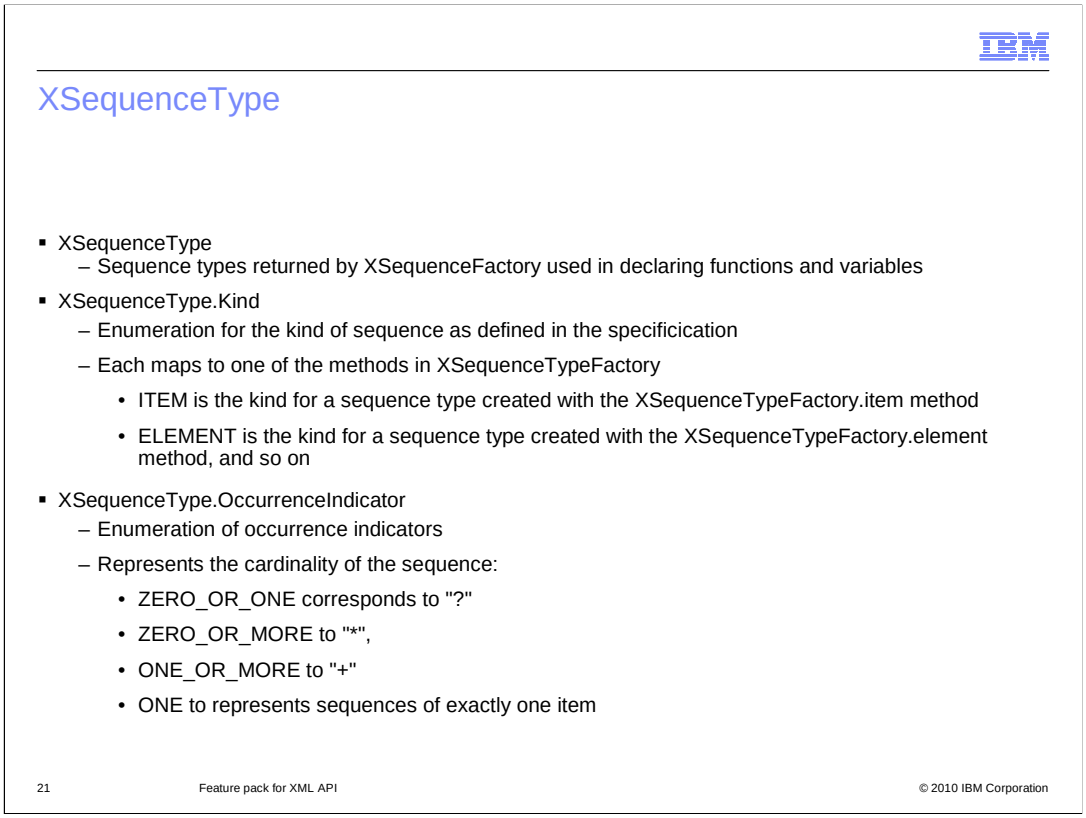


XProcessException extends RuntimeException and is the base exception for XML processing. This exception is thrown when the processor finds a non-recoverable error when preparing or executing an expression, query, or stylesheet as described in the specifications for each language. If there are multiple errors at prepare time, the processor attempts to report all of the errors and only creates an XProcessException at the end of preparation or if it reaches a point where it cannot continue. At execution time, however, the first error results in an XProcessException and the end of execution.

In general, XProcessExceptions should not occur if the expression, query, or stylesheet is syntactically and semantically valid and valid for the types of input documents that it is meant to process.

An XProcessException is also thrown for an XSLT message instruction where the terminate attribute evaluates to "yes." This is the default behavior for handling processing errors and other messages. Applications can register an implementation of the XMessageHandler interface on the XFactory or at prepare time or execution time to modify the default behavior.

XViewException extends RuntimeException and occurs when there is a View exception. This exception is raised for incorrect use of the API such as calling one of the XNodeView methods on an item that is atomic. An XViewException is thrown when there are conversion problems (for example, trying to get a date from a boolean), calling an XNodeView method on an atomic item, and calling XNodeView.relativePosition for nodes that are not in the same document.



In the XML API, sequence types are represented using XSequenceType objects. XSequenceType objects can represent the same set of sequence types as the SequenceType syntax defined in XPath 2.0. XSequenceType objects are created through the XSequenceTypeFactory interface. XSequenceTypeFactory has several methods for creating XSequenceType instances.

The XSequenceType interface represents a sequence type. To examine the type, a user should first call the getKind method. The other accessor methods can be applicable depending on the Kind of the sequence type. **XSequenceType.Kind** is an enumeration for the kind of sequence as defined in the SequenceType syntax. Each maps to one of the methods in XSequenceTypeFactory. ITEM is the kind for a sequence type created with the XSequenceTypeFactory.item method, ELEMENT is the kind for a sequence type created with the XSequenceTypeFactory.element method, and so on.

Each method on the XSequenceTypeFactory interface has an XSequenceType.OccurrenceIndicator parameter except for emptySequence(). OccurrenceIndicator is an enum that represents the cardinality of the sequence; ZERO_OR_ONE corresponds to "?" in SequenceType syntax, ZERO_OR_MORE to "*", ONE_OR_MORE to "+", and ONE to no occurrence indicator for sequences of exactly one item.

Miscellaneous

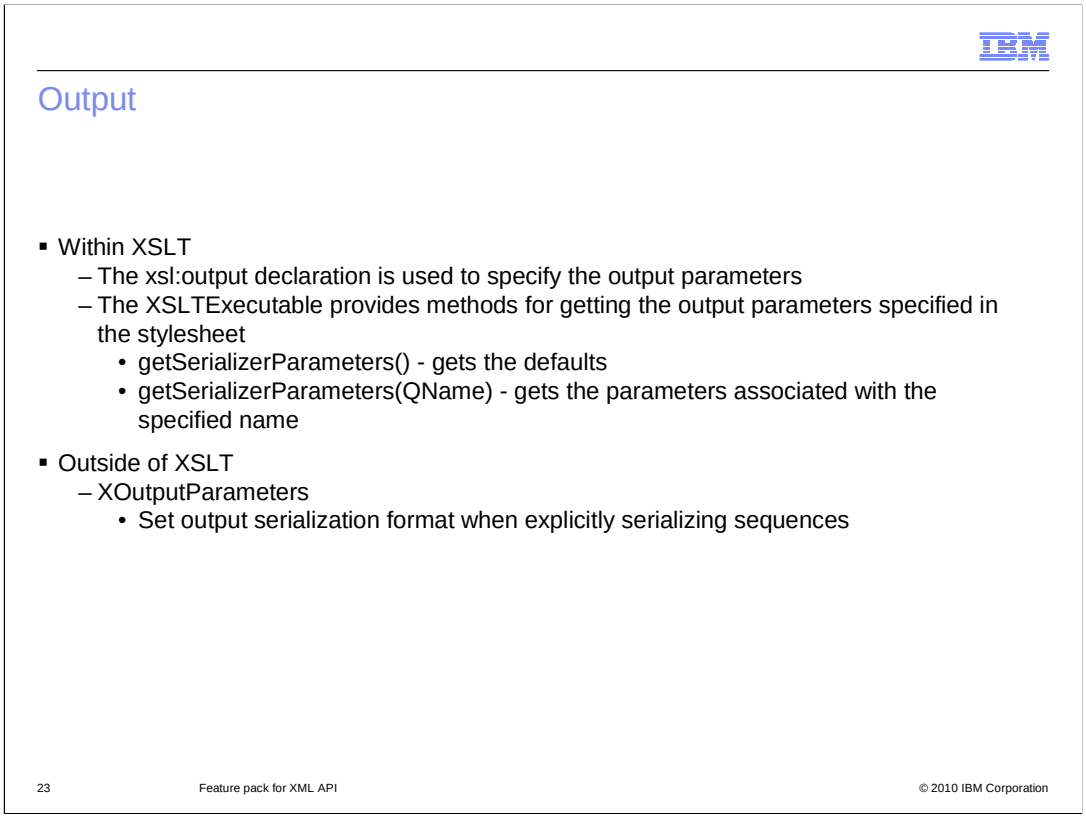
- XTypeConstants
  - QName and enum constants for “built-in” types
- XTypeConstants.Type
  - Enumerator of built-in types
    - COMPLEX is used for all non-atomic types
- XSequenceCursorResult
  - A JAXP result that can then be navigated using XSequenceCursor, typically returned from XResultResolver
- XItemSource
  - A way to provide XItemViews as sources to be used as input data, typically useful when using an XSourceResolver

XTypeConstants represents predefined constants for the built-in types including Integer values for the built-in types. The XValueView.getValueType method will return one of these types or -1 for a user defined type. XTypeConstants can be used when declaring variables and functions on the XStaticContext.

XTypeConstants.Type represents an enumerator of built-in types. COMPLEX is used for all non-atomic types.

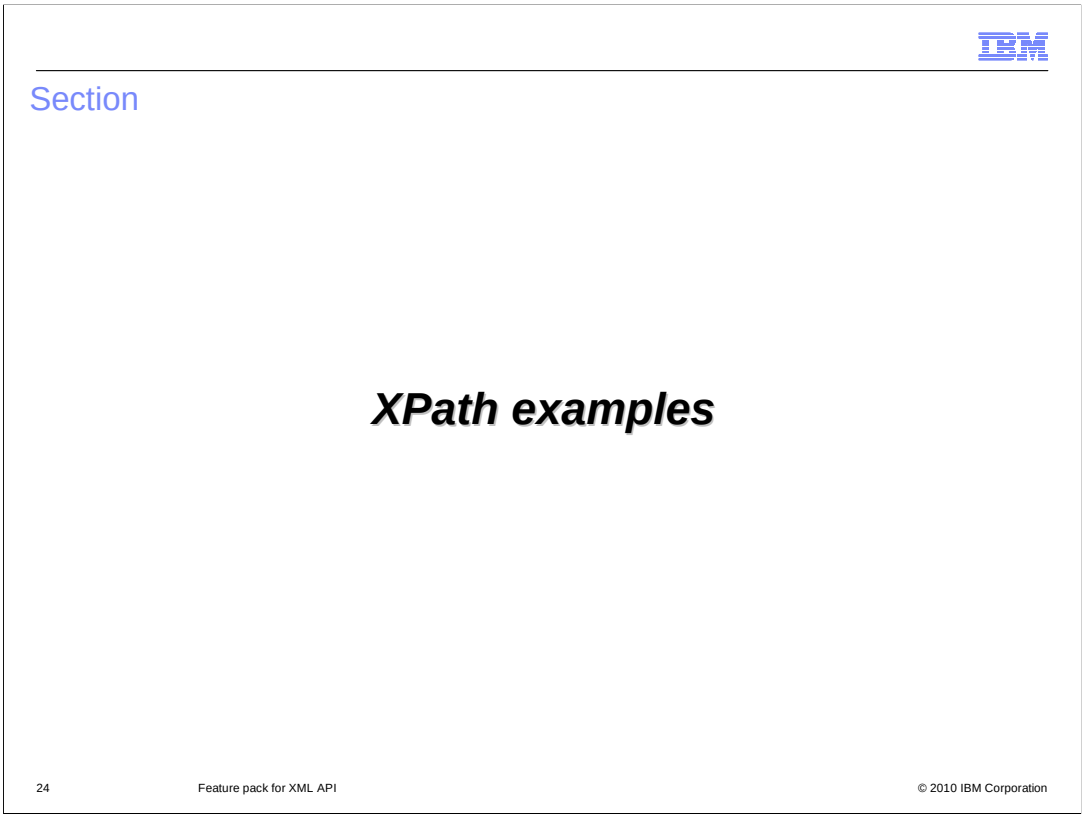
XSequenceCursorResult is an implementation of result for XSequenceCursor. It can be used by implementations of XResultResolver to allow for XSequenceCursor results. This result can be navigated using XSequenceCursor, and is typically returned from XResultResolver.

XItemSource is an implementation of Source for XItemView objects. This allows results from a previous expression, query or transformation to be used where a Source object is required (such as by implementations of XSourceResolver) without first serializing to a DOM or a stream. Note that an XItemSource can only be used at execution time for providing input documents and not at prepare time for imports and includes.



Within XSLT the xsl:output declaration is used to specify the output parameters. The XSLTExecutable provides methods for getting these output parameters specified within the stylesheet. These methods are getSerializerParameters() to gets the default parameters, and getSerializerParameters(QName) to get the parameters associated with a specified name.

Outside of XSLT you will want to use the interface **XOutputParameters** for output parameter settings. Used when exporting sequences and items. New output parameters objects can be created through the XFactory.newOutputParameters method. These parameters correspond to the ones in the XSLT 2.0 and XQuery 1.0 Serialization specification.



The next few slides will show how to use the XML API to execute XPath.

Preparing and executing an interpreted XPath expression

- You can use the XPathExecutable instances that are created using XFactory.prepareXPath methods to evaluate XPath expressions
- Basic example of preparing and executing an interpreted XPath expression:

```
//Create a string for the XPath expression
String expression = "/doc/something";
//Create the factory
XFactory factory = XFactory.newInstance();
//Create an XPath executable object for the expression
XPathExecutable xPathExec = factory.prepareXPath(expression);
//Create the input XML source
StreamSource input = new StreamSource("myxml.xml");
//Execute expression and store results in an XSequenceCursor
XSequenceCursor xSequenceCursor = xPathExec.execute(input);
```


You can use the XPathExecutable instances that are created using XFactory.prepareXPath methods to evaluate XPath expressions. The example here shows a basic example of preparing and executing an interpreted XPath expression. XPath expressions can be passed to the XFactory.prepareXPath method using a JAXP StreamSource object or using a plain Java string object. The resulting XPathExecutable instance is thread safe and can be reused to evaluate an XPath expression on multiple XML input documents.

Preparing and executing a compiled XPath

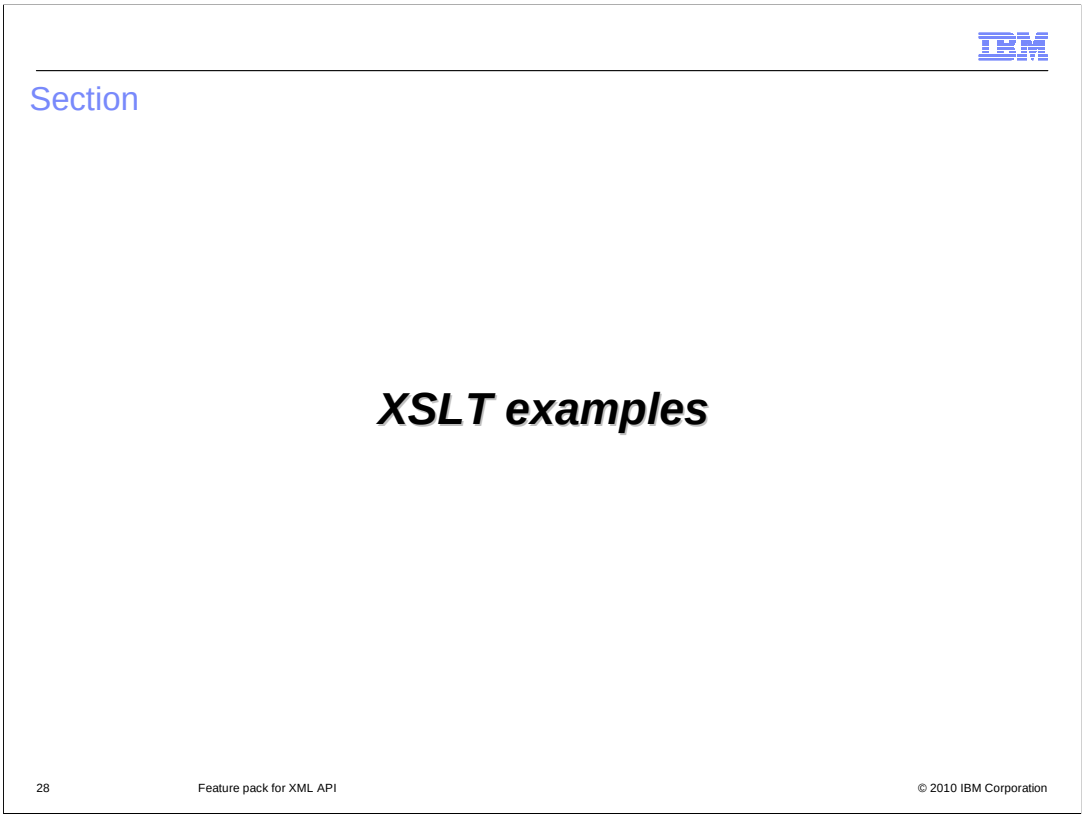
```
// Create a string for the XPath expression
String expression = "/doc/something";
// Create the factory
XFactory factory = XFactory.newInstance();
// Create a new static context from the factory
XStaticContext sc = factory.newStaticContext();
// Set the mode to compile for the processor
sc.setUseCompiler(true);
// Create an XPath executable object for the expression
XPathExecutable xPathExecutable =
  factory.prepareXPath(expression,sc);
//Create the input XML source
StreamSource input = new StreamSource("myxml.xml"); //Execute
expression, store results in XSequenceCursor
XSequenceCursor xSequenceCursor =
  xPathExecutable.execute(input);
```


Here is a basic example of preparing and executing a compiled XPath expression. Notice how the XStaticContext interface is used to setUseCompiler to true.

Preparing and executing an interpreted XPath expression with schema validation

```
// Create a string for the Xpath expression
String expression = "/doc/byte cast as my:derived1-byte-enumeration-
Type";
// Create the factory
XFactory factory = XFactory.newInstance();
// Create the schema source
StreamSource schema = new StreamSource("schema.xsd");
// Load schema
factory.registerSchema(schema);
// Turn on validation
factory.setValidating(XFactory.FULL_VALIDATION);
// Create a new static context from the factory
XStaticContext sc = factory.newStaticContext();
// Add new namespace
sc.declareNamespace("my", "http://www.schematype.ibm.com/UDSimple");
// Create an XPath executable object for the expression
XPathExecutable xPathExec=factory.prepareXPath(expression,sc);
//Create the input XML source
StreamSource input = new StreamSource("myxml.xml");
//Execute the expression, store the results in XSequenceCursor
XSequenceCursor xSequenceCursor = xPathExec.execute(input);
```


Here is a basic example of preparing and executing an interpreted XPath expression with schema validation. An instance of XFactory maintains a set of registered schemas as well and can be validating or non-validating. 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. If different sets of stylesheets or expressions need different sets of schemas, these can be kept separate by using more than one XFactory instance.



The next few slides will show how to use the XML API to execute XSLT.

Preparing and executing an interpreted transformation

- You can use the XSLTExecutable instances that are created using XFactory.prepareXSLT methods to perform XSLT transformations
- Basic example of preparing and executing an interpreted transformation:

```
// Create the factory
XFactory factory = XFactory.newInstance();
// Create a StreamSource for the stylesheet
StreamSource stylesheet = new
  StreamSource("simple.xsl");
// Create an XSLT executable object for the stylesheet
XSLTExecutable xsltExec =
  factory.prepareXSLT(stylesheet);
// Create the input source
Source input = new StreamSource("simple.xml");
// Create the result
Result result = new StreamResult(System.out);
// Execute the transformation
xsltExec.execute(input, result);
```


You can use the XSLTExecutable instances that are created using XFactory.prepareXSLT methods to perform XSLT transformations. The example on this page shows a basic example of preparing and executing an interpreted transformation.

Preparing and executing a compiled transformation

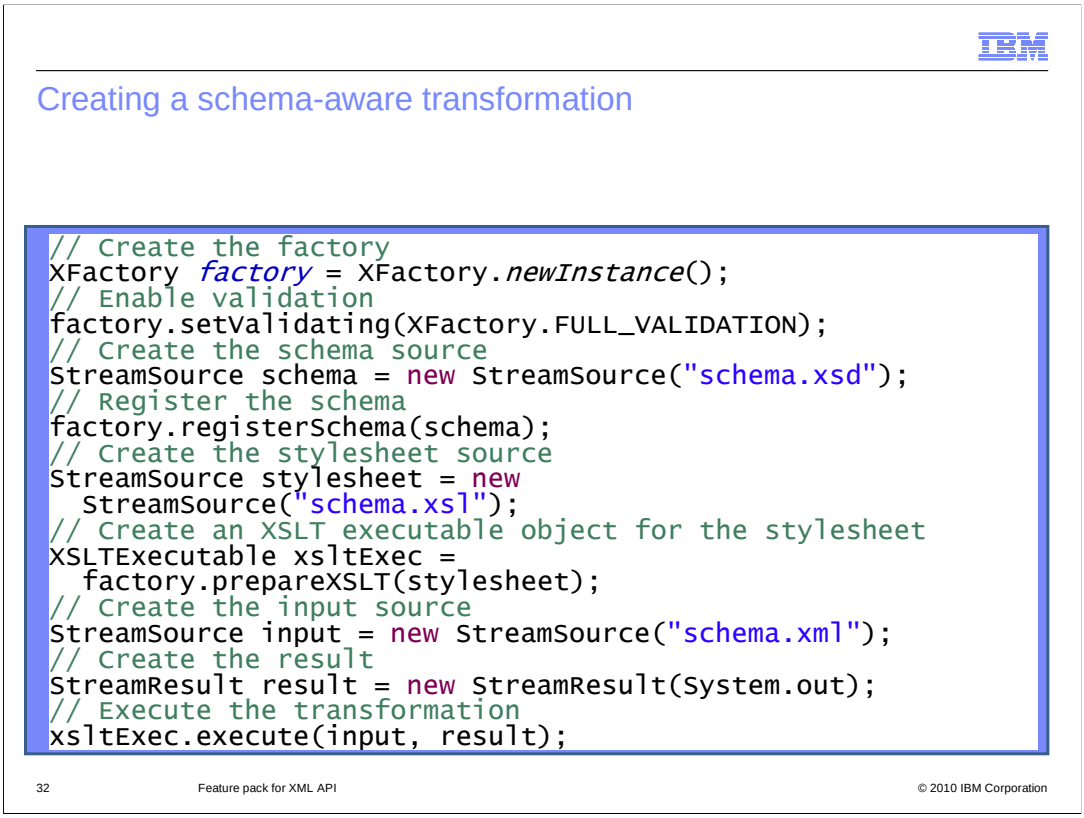
```
// Create the factory
XFactory factory = XFactory.newInstance();
// Create a StreamSource for the stylesheet
StreamSource stylesheet=new StreamSource("simple.xsl");
// Create a new static context
XStaticContext sc = factory.newStaticContext();
// Enable the compiler
sc.setUseCompiler(true);
// Create an XSLT executable object for the stylesheet
XSLTExecutable xsltExec =
  factory.prepareXSLT(stylesheet, sc);
// Create the input source
Source input = new StreamSource("simple.xml");
// Create the result
Result result = new StreamResult(System.out);
// Execute the transformation
xsltExec.execute(input, result);
```


Here is a basic example of preparing and executing a compiled XPath expression. Notice the use of the XStaticContext and setting the compiler to true.

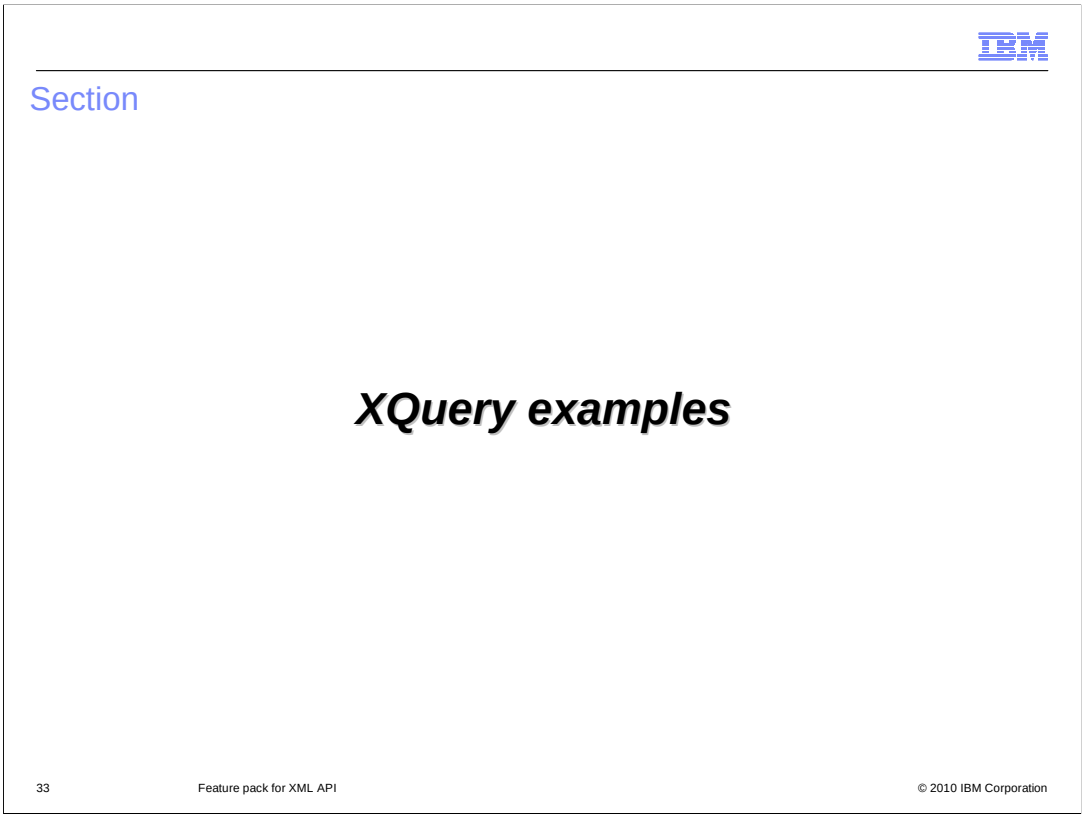
Creating identity transformation

```
// Create the factory
XFactory factory = XFactory.newInstance();
// Create the item factory
XItemFactory itemFactory = factory.getItemFactory();
// Create the input source
Source input = new StreamSource("simple.xml");
// Create the XItemView object from the input source
XItemView item = itemFactory.item(input);
// Create an XOutputParameters object
XOutputParameters params =
  factory.newOutputParameters();
// Set parameters
params.setMethod("xml");
params.setEncoding("UTF-8");
params.setIndent(true);
// Create the result
Result result = new StreamResult(System.out);
// Serialize to the result
item.exportItem(result, params);
```


The example on this slide shows how to create an identity transformation. Notice the use of XItemView and the setting of parameters for the output.



The following is an example of creating a schema-aware transformation. An instance of XFactory maintains a set of registered schemas as well and can be validating or non-validating. 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. If different sets of stylesheets or expressions need different sets of schemas, these can be kept separate by using more than one XFactory instance. An XFactory instance can be created by calling the static newInstance() method on the XFactory class. The instance is thread safe as long as the settings remain stable.



The next few slides will show how to use the XML API to execute XQuery.

Preparing and executing an interpreted XQuery expression

- You can use the XQueryExecutable instances that are created using XFactory.prepareXQuery methods to evaluate XQuery expressions
- Basic example of preparing and executing an interpreted XQuery expression:

```
// Create a string for the XQuery expression
String expression = "/doc/name[@first='David']";
// Create the factory
XFactory factory = XFactory.newInstance();
// Create the XQueryExecutable
XQueryExecutable xQueryExecutable =
  factory.prepareXQuery(expression);
//Create the input XML source
StreamSource input = new StreamSource("myxml.xml");
//Execute expression, store the results in
XSequenceCursor
XSequenceCursor xSequenceCursor =
  xQueryExecutable.execute(input);
```


This slide shows a basic example of preparing and executing an interpreted XQuery expression. You can use the XQueryExecutable instances that are created using XFactory.prepareXQuery methods to evaluate XQuery expressions. XQuery expressions can be passed to the XFactory.prepareXQuery method using a JAXP StreamSource object or using a plain Java string object. The resulting XQueryExecutable instance is thread safe and can be reused to evaluate an XQuery expression on multiple XML input documents.

Preparing and executing a compiled XQuery expression

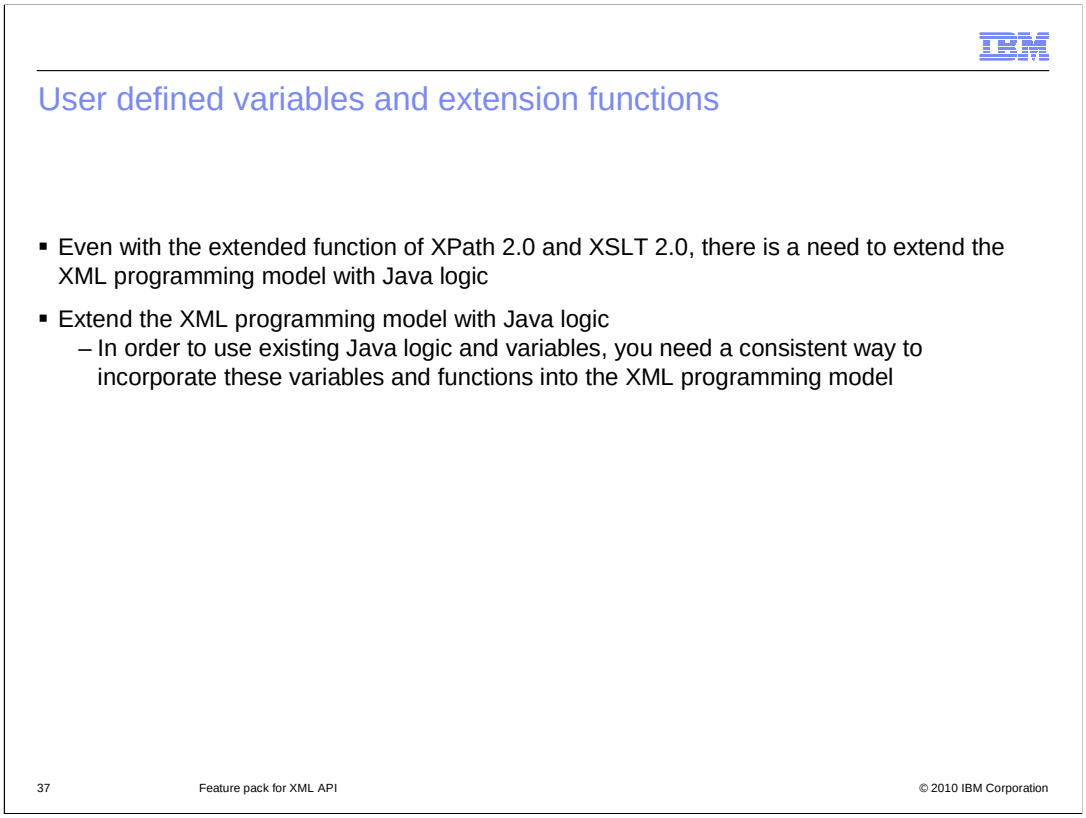
```
// Create a string for the XQuery expression
String expression = "/doc/name[@first='David']";
// Create the factory
XFactory factory = XFactory.newInstance();
// Create a new static context from the factory
XStaticContext sc = factory.newStaticContext();
// Set the mode to compile for the processor
sc.setUseCompiler(true);
// Create the XQueryExecutable
XQueryExecutable xQueryExecutable =
  factory.prepareXQuery(expression, sc);
//Create the input XML source
StreamSource input = new StreamSource("myxml.xml");
//Execute expression, store results in XSequenceCursor
XSequenceCursor xSequenceCursor =
  xQueryExecutable.execute(input);
```


The following is a basic example of preparing and executing a compiled XQuery expression. Notice the use of the XStaticContext and the setting of the compiler to true.

Section

User defined variables and extension functions

The next few slides show how to take advantage of user defined variables and extension functions.



Even with the extended function of XPath 2.0 and XSLT 2.0, there is a need to extend the XML programming model with Java logic. In order to use existing Java logic and variables, you need a consistent way to incorporate these variables and functions into the XML programming model.

Example of binding external Java variables

- 1 - Declare the variable definition (QName and type) to the preparation
- 2 - Bind the variable value (based on QName) to the execution

```
// Create a XPath string that has a user defined variable
private static String XPATH_GET_QUESTIONABLE_COMMENTS =  "…
  matches(atom:content, $my:vulgarwords, 'i') …";
// Create a QName for the Java variable
String myNamespace = "http://com.ibm.xml.samples";
QName vulgarNamesQName = new QName(myNamespace, "vulgarwords");
// Declare the variable to the static context used in the prepare
XStaticContext sc = factory.newStaticContext();
sc.declareNamespace("my", myNamespace);
sc.declareVariable(vulgarNamesQName, XTypeConstants.STRING_QNAME);
// Prepare the XPath
XPathExecutable getQuestionableCommentsXPath=
  factory.prepareXPath(XPATH_GET_QUESTIONABLE_COMMENTS, sc);
// Bind the Java variable to the dynamic context
XDynamicContext dc = factory.newDynamicContext();
String VULGAR_WORDS = "shoot|darn";
dc.bind(vulgarNamesQName, VULGAR_WORDS);
// Execute the XPath
XSequenceCursor sequence =
  getQuestionableCommentsXPath.execute(source, dc);
```

1 2

38 Feature pack for XML API

For XSLT and XQuery, variables must be declared in the stylesheet or query itself and thus should not be re-declared in the static context. However, XPath variables need to be declared in the static context. The example here shows the declaration of the variable in the static context and the binding of the variable value in the dynamic context.

Example of binding Java extension functions

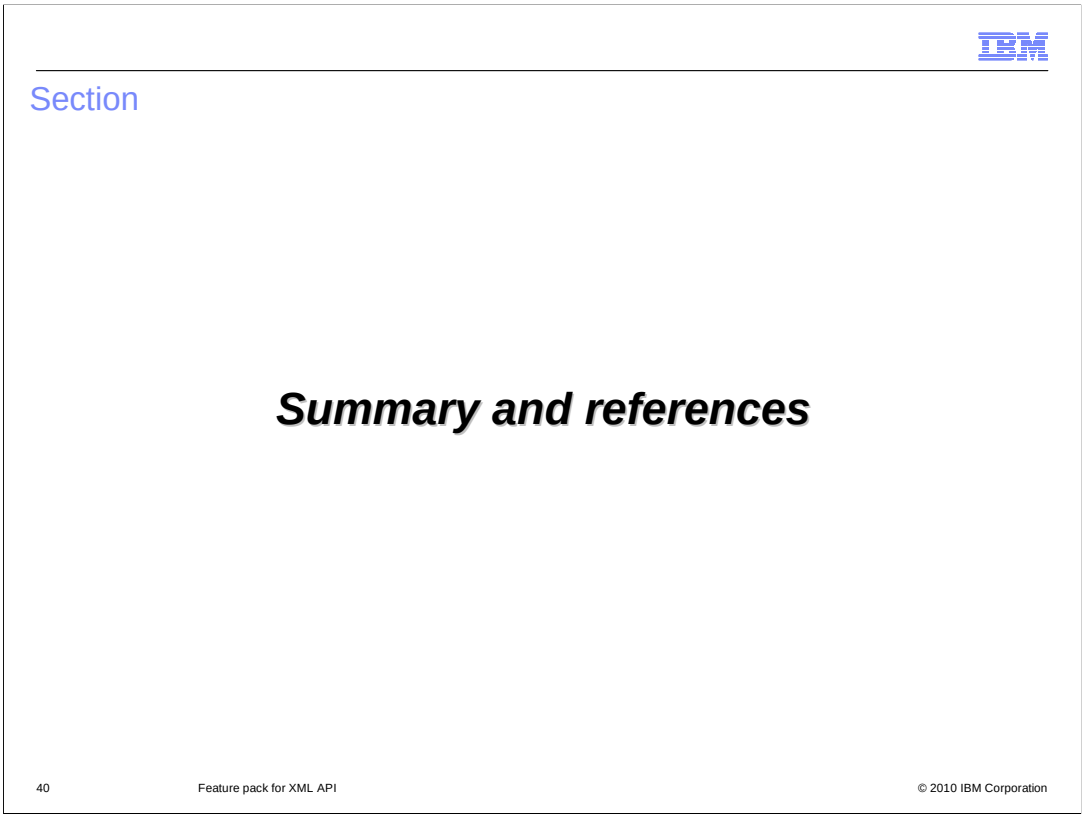
- 1 - Declare the function definition (QName and parameter/return types) to the preparation
- 2 - Bind the function (based on QName) to the execution

```
// Create a XPath string that has a user defined extension function
private static String XPATH_GET_QUESTIONABLE_COMMENTS =  "…
   my:legacyContentAnalysis(atom:content/text()) …"
// Create a QName for the Java extension function
String myNamespace = "http://com.ibm.xml.samples";
QName legacyContentAnalysisQName = new QName(myNamespace,
  "legacyContentAnalysis");
// Declare the function to the static context used in the prepare
XStaticContext sc = factory.newStaticContext();
sc.declareNamespace("my", myNamespace);
sc.declareFunction(legacyContentAnalysisQName, XTypeConstants.BOOLEAN_QNAME,
  new QName[] { XTypeConstants.STRING_QNAME });
// Prepare the XPath
XPathExecutable getQuestionableCommentsXPath =
  factory.prepareXPath(XPATH_GET_QUESTIONABLE_COMMENTS, sc);
// Bind the Java function to the dynamic context
XDynamicContext dc = factory.newDynamicContext();
dc.bindFunction(legacyContentAnalysisQName,
  ExtensionFunctions.class.getMethod("legacyContentAnalysis", String.class));
// Execute the XPath
XSequenceCursor sequence = getQuestionableCommentsXPath.execute(source, dc);
```

```
public class ExtensionFunctions {
  public static Boolean legacyContentAnalysis(String input) {  …  }
}
```

39 Feature pack for XML API

The example here shows how to extend the XML programming model by binding a Java extension function. First you will declare the function definition in the static context, then you will bind the function in the dynamic context.



The next section provides a summary and references.

Summary

- Feature Pack for XML API Support
  - A new XML runtime API across the X-* family
- Factories, executable objects, contexts, resolvers, data views and cursors, messages and exceptions
- Preparing and executing examples

This presentation went through the feature pack for XML API support and the new XML runtime API. There were also examples showing off how to use the APIs to execute XPath, XSLT, and XQuery.

References

- WebSphere® Application Server Feature Pack for XML
  - http://www.ibm.com/software/webservers/appserv/was/featurepacks/xml/
- Primary specifications
  - http://www.w3.org/TR/xpath20/
  - http://www.w3.org/TR/xslt20/
  - http://www.w3.org/TR/xquery/
- Infocenter (also contains the API documentation)
  - http://www14.software.ibm.com/webapp/wsbroker/redirect?version=v700xml&product=was-nd-mp

Here are some useful links.

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_XMLFEP_API.ppt

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

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

Trademarks, disclaimer, and copyright information

IBM, the IBM logo, ibm.com, and WebSphere are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of other IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml

THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY. in the United States, other countries, or both.

THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY. WHILE EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. IN ADDITION, THIS INFORMATION IS BASED ON IBM'S CURRENT PRODUCT PLANS AND STRATEGY, WHICH ARE SUBJECT TO CHANGE BY IBM WITHOUT NOTICE. IBM SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION. NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO, NOR SHALL HAVE THE EFFECT OF, CREATING ANY WARRANTIES OR REPRESENTATIONS FROM IBM (OR ITS SUPPLIERS OR LICENSORS), OR ALTERING THE TERMS AND CONDITIONS OF ANY AGREEMENT OR LICENSE GOVERNING THE USE OF IBM PRODUCTS OR SOFTWARE.