XQuery API for Java™ (XQJ) 1.0 Specification

Version 1.0 (JSR 225 Public Draft Specification)
October 19, 2007

Spec Lead: Jim Melton
Oracle

Editor: Marc Van Cappellen
DataDirect Technologies

This document is the Proposed Final Draft of the XQuery API for Java™ (XQJ) 1.0.
Comments on this specification should be sent to jsr-225-comments@jcp.org.

SPECIFICATION LICENSE

Oracle USA (the “Spec Lead”) for the XQuery API for Java specification (the “Specification”) hereby grant a perpetual, non-exclusive, worldwide, fully paid-up, royalty-free, irrevocable (except as explicitly set forth below) license to copy and display the Specification, in any medium without fee or royalty, provided that you include the following on ALL copies, or portions thereof, that you make:

1. A link or URL to the Specification at this location: _____”TBD”_____.
2. The copyright notice as shown herein.

In addition, to the extent that an implementation of the Specification would be considered a derivative work under applicable law requiring a license grant from the holder of the copyright in the Specification, the Spec Lead grants a copyright license solely for the purpose of making and/or distributing an implementation of the Specification that: (a) except for the RI code licensed from Oracle under the RI License, does not include or otherwise make any use of the RI; (b) fully implements the Specification including all of its required interfaces and functionality; (c) does not modify, subset, superset or otherwise extend those public class or interface declarations whose names begin with “java;” and (d) passes the TCK.

The Spec Lead also agrees, upon request, to grant a perpetual, non-exclusive, worldwide, non sub-licensable, non-transferable, royalty-free fully paid-up license, for the sole purposes of making, having made, using, selling and offering for sale, implementations of the Specification that meet the requirements of (a) - (d) above, under those respective patent claims that they own, or have the authority to license, for which there is no technically feasible way of avoiding infringement in the course of implementing the Specification (“Necessary Patent Claims”).

The licenses and agreement to license set forth above are conditional upon licensee’s offering a license, on fair, reasonable and non-discriminatory terms to Necessary Patent Claims that they own or have the authority to license to make, have made, use, sell and offer for sale, implementations of the Specification that meet the requirements of (a) - (d) above, to all other licensees to the Specification who agree to offer a similar license and not initiating any claim that either Specification Lead, has, in the course of performing its responsibilities as the Specification Lead, induced any other entity to infringe the licensee’s patent rights.

THE SPECIFICATION IS PROVIDED "AS IS," AND THE SPEC LEAD AND ANY OTHER AUTHORS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY PATENTS (INCLUDING,

WITHOUT LIMITATION, PATENTS OF THE SPECIFICATION LEADS), COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS. THE SPEC LEAD AND ANY OTHER AUTHORS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE SPECIFICATION OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of the Spec Lead or any other Authors may NOT be used in any manner, including advertising or publicity pertaining to the Specification or its contents without specific, written prior permission. Title to copyright in the Specification will at all times remain with the Authors.

No other rights are granted by implication, estoppel or otherwise.

Table of Contents

1 Introduction

The XQuery API for Java (XQJ) provides programmatic access to XQuery implementations. Applications using the XQJ API can execute XQuery queries, 'bind' data to queries and process query results.

Many Java application developers need to write Java code independent of any particular XQuery implementation. XQJ is a generic XQuery data access framework, which provides a uniform interface on top of a variety of different XQuery implementations. This allows programmers to write to a single interface, enables XQuery engine-independent Java application development tools and products, and allows XQuery vendors to provide standards based connectivity on the Java platform.

1.1 Getting Started with XQJ

The following sample Java code is meant to convey a first look and feel of the style and usage of the XQJ API. It is by no means exhaustive or complete; *e.g.*, no error handling is shown and it is assumed that `xqds` is an `XQDataSource` object representing a given data source. It illustrates the basic steps that an application would perform to execute an XQuery expression at a given XQuery implementation.

```
...
// establish a connection to the XQuery engine
XQConnection conn = xqds.getConnection();

// create an expression object that is later used
// to execute an XQuery expression
XQExpression expr = conn.createExpression();

// the XQuery expression to be executed
String es = "for $n in fn:doc('catalog.xml')//item " +
            "return fn:data($n/name)";

// execute the XQuery expression
XQResultSequence result = expr.executeQuery(es);

// process the result (sequence) iteratively
while (result.next()) {
      // retrieve the current item of the sequence as a String
      String str  = result.getAtomicValue();
      System.out.println("Product name: " + str);
}

// free all resources allocated for the result
result.close();

// free all resources allocated for the expression
expr.close();

// free all resources allocated for the connection
conn.close();
...
```

Example 1 – A first look at XQJ.

1.2 Definitions

- **XQJ-implementation** – a software component implementing all the XQJ interfaces as outlined in this specification and accompanying Javadoc.
- **XQJ driver** – a synonym for XQJ-implementation.
- **Implementation-defined** – an aspect possibly differing between XQJ-implementations, but must specified by the implementer for each particular XQJ-implementation.
- **Implementation-dependent** – an aspect possibly differing between XQJ-implementations, but not further specified by this specification, and not required to be specified by the implementer for any particular XQJ-implementation.
- **XDM** – the XQuery 1.0 and XPath 2.0 Data Model as defined in [XQuery DM].
- **XDM instance** – as described in [XQuery], the term XDM instance is used, to denote an unconstrained sequence of nodes and/or atomic values in the XDM.

1.3 Platforms

This version of the XQJ API targets Java 2 Platform, Standard Edition (J2SE) 1.4 and higher versions.

The following interfaces and classes are not available in J2SE 1.4 and need to be obtained separately:

- javax.xml.datatype.Duration
- javax.xml.datatype.XMLGregorianCalendar
- javax.xml.namespace.QName
- javax.xml.stream.XMLStreamReader

Similar for J2SE 1.5, the following interfaces and classes are not available:

- javax.xml.stream.XMLStreamReader

1.4 Target Audience

This specification targets primarily software architects and developers of the following type of systems:

- XQJ-implementations
- Application servers providing middle-tier services on top of XQJ
- Tools that use the XQJ API, like XML IDEs
- Java-based applications querying XML data using XQuery

1.5 Submitting Feedback

Please send any comments and questions concerning this specification to:

`jsr-225-comments@jcp.org`

1.6 Acknowledgements

The JSR 225 Expert Group wishes to gratefully acknowledge the contributions of several people who were unfortunately not part of the Expert Group when this work was completed.

- Andrew Eisenberg provided tremendous leadership and many technical contributions that significantly improved the quality of the specification.
- Muralidhar Krishnaprasad not only contributed greatly to the technical content of XQJ through numerous technical proposals, but he also created the Javadoc component of this work and maintained it for most of the project's duration.
- Jan-Eike Michels, in addition to numerous technical contributions, created and maintained the specification document for most of the lifetime of this project.
- Karuna Muthiah was responsible for creating and maintaining the Reference Implementation for much of the process, in addition to his technical contributions to XQJ.
- Basuki Soeterman made numerous technical contributions that significantly aided the initial stages of the activity.
- Per Bothner frequently raised concerns with both XQJ's design principals and the details of the specification and Javadoc, contributing greatly to the quality of the result.

In addition, the Spec Lead wishes to acknowledge the crucial contributions made by several Expert Group members.

- Jason Hunter spent significant time reviewing many versions of the specification and Javadoc, reporting technical and editorial problems and proposing solutions to them.
- Zhen Hua Liu (Oracle) made many technical contributions that focused largely on ensuring that XQJ remained aligned with XQuery as that latter specification evolved.
- Max Orgiyan (Oracle), in addition to his technical contributions, took on the job of completing the Reference Implementation on schedule (and succeeded); Max has also maintained and enhanced the Javadoc after its original author departed the group.
- Yi Feng (Oracle) implemented and tested the serialization functionality in the RI.
- Ying Lu(Oracle) consulted Max and Karuna on the use of the XQuery implementation embedded in the RI.
- Marc Van Cappellen (DataDirect Technologies) was a principal contributor of technical proposals for much of the lifetime of JSR 225, and accepted responsibility for maintaining and completing the PDF specification after its original owner left the Expert Group.

2 Goals and Non-Goals

2.1 Goals

This section outlines the goals and high-level design philosophy for the XQuery API for Java.

- Ensure consistency with XQuery 1.0
  XQuery 1.0 is the fundamental basis on which the XQJ API is built. The XQJ API seeks to support the broadest possible feature set in XQuery 1.0
- Offer a vendor-neutral access to any XQuery data source
  The API is built with the intent of supporting a broad spectrum of XQuery implementations and architectures, which by offering a programmatic consistency supplies a sufficiently flexible basis for a wide range of applications.
- Simple to use API
  The XQJ API is intended to be simple-to-use and a straightforward approach for use by a variety of applications.
- Promote Adoption of XQuery on the Java platform
  Position the XQJ API as the springboard for innovation and the foundation for higher level APIs.
  In addition, this specification must find its place amongst established Java XML functionality by addressing developer needs and integrate with well-known APIs such as DOM, SAX, and StAX.
- Model XQuery language and concepts for Java Audience
  A primary focus of this and future XQJ specifications is to establish and maintain XQJ as the primary API for access XQuery data sources from the Java programming language. This is achieved by providing a set of fundamental Java interfaces and classes to model key XQuery concepts.
- Relationship to the JDBC API
  Although similar in many respects to the intents and goals of the JDBC specification, XQJ is built to accommodate specific orthogonal requirements. The core connection-orientated concept that is the basis of the JDBC specification, can be understood as applicable to XQJ, but is designed in such a manner to not preclude the full spectrum of XQuery implementation possibilities.
  Similarly, the XQJ API seeks to build upon established JDBC concepts that are rooted in SQL and XQuery commonalities. Consistencies such as declarative syntax, ability to support external variably bindings, static query compilation and dynamic query processing models converge to provide a long established programming model that is familiar to many developers. XQJ capitalizes on these similarities to provide a known model to developers already familiar to JDBC.

2.2 Non-Goals

This section outlines some non-goals of the XQuery API for Java.

- XQJ intends to provide a general “least common denominator” API to interact with variety of XQuery implementations in variety of operating environments. Although it does not preclude enhancements to make it operate more efficiently for certain specific XQuery execution environments, it does however not intend to provide API capabilities that assumes particular XQuery execution environment.
- JAXP is a Java API that provides XML schema and XPath 1.0 processing. It assumes a co-located architecture which is not a design assumption that XQJ can adopt. Furthermore, there are key differences between XPath 1.0 supported by JAXP and XQuery 1.0 supported by XQJ in terms of the underlying query data model, type system, etc. Indeed, XQJ provides a set of new Java interfaces to model the new concepts in XQuery 1.0 that are not defined in XPath 1.0

3 Compliance

This chapter lists the features that an XQJ-implementation must support to claim compliance with the XQJ 1.0 specification.

3.1 Definitions

- optional feature – functionality an XQJ-implementation is not required to offer. Often a Java method, for which the implementation can raise a hard-coded `XQException`, ignore certain arguments, or simply implement it as a no-op. Which of these approaches can be used is outlined throughout the XQJ specification for each of the specific methods.
- implement the interface – implement all methods defined in the interface, except those explicitly listed as optional, according to the XQJ specification.

3.2 Guidelines and Requirements

- The XQuery implementation accessed through the XQJ driver must be minimal conformant, as outlined in [XQuery].
- All implementation-defined items must be explained in the accompanying documentation of the XQJ-implementation. Here is the list of implementation-defined aspects,
    - the class name of the `XQDataSource` implementation(s)
    - all properties defined on the `XQDataSource` implementation(s)
    - the interaction between the JDBC and XQJ connection, assuming XQJ connections created through a JDBC connection is supported
    - the syntax and semantics for commands, assuming executing commands through `XQExpression` is supported
    - is canceling of query execution supported
    - the default and supported values for each parameter described in [XQuery Serialization]
    - additional StAX or SAX event types being reported, beside the event types documented in this specification
    - support for XDM-instances and types based on user-defined schema types
    - the semantics with respect to node identity, document order and full node context, when a node is bound to an external variable
    - is login-timeout supported
    - are transactions supported
    - behavior of the `getNodeUri` method, defined on `XQItemAccessor`, for other than document nodes

  - behavior of the `getTypeName` method, defined on `XQItemType`, for anonymous types
  - behavior of the `getSchemaURI` method, defined on `XQItemType`
  - behavior of the `createItemFromDocument` methods, defined on `XQDataFactory`, if the specified value is not a well-formed XML document
  - behavior of the `bindDocument` methods, defined on `XQDynamicContext`, if the specified value is not a well-formed XML document
  - the error codes, reported through `XQQueryException`, in addition to the standard error codes listed in [XQuery] and its associated specifications

- An XQJ implementation is required to be thread safe. That is, all operations on all objects implementing an interface of the javax.xml.xquery package are thread safe and able to handle simultaneous requests from several threads on the same object. Some XQJ implementations may offer more concurrent execution than others, the exact level of concurrency is implementation dependent.

  Applications can assume fully concurrent execution. If the XQJ implementation requires some form of synchronization, it will provide it under the covers. The only difference to the application is that it will run with reduced concurrency.

  For example, two `XQExpression` objects on a single `XQConnection` can be executed concurrently, i.e. invoke the `execute` method on each of the `XQExpression` objects from within 2 threads. Some implementations are providing full concurrency, others may execute one query and wait until it completes before evaluating the other.

  In general it is assumed to be bad practice to access a single `XQExpression`, `XQPreparedExpression` or for example an `XQSequence` instance from within two threads simultaneously.
  As such, it might still require the application to perform synchronization in these scenarios. For example, two threads accessing the same `XQExpression` object simultaneously might not be deterministic. Depending on the thread scheduling the external variable bindings might interfere with some of the operations performed from the other thread.

3.3 XQJ 1.0 Compliance

An XQJ-implementation that is compliant with the XQJ 1.0 specification must:

- comply with the previous guidelines and requirements
- implement the `XQDataSource` interface, with the exception of the following optional methods

    - `getConnection(java.sql.Connection)`
    - `getConnection(java.lang.String, java.lang.String)`
- implement the `XQConnection` interface
  transaction support is optional, if not supported the XQJ implementation must raise an error if auto-commit is turned off.
- implement the `XQExpression` interface, with the exception of the following optional methods
    - `executeCommand(java.io.Reader)`
    - `executeCommand(java.lang.String)`
    - `cancel()`
- implement the `XQPreparedExpression` interface, with the exception of the following optional methods
    - `cancel()`
- implement the `XQDynamicContext` and `XQStaticContext`.
- implement the `XQItemAccessor`, `XQItem, XQResultItem`, `XQSequence` and `XQResultSequence` interfaces.
- implement the `XQMetaData` interface.
- implement the `XQItemType` and `XQSequenceType` interfaces
  XQJ-implementations are not required to support item and sequence types based on user-defined XML Schema types
- implement the interface `XQDataFactory`
  XQJ-implementations are not required to support item and sequence types based on user-defined XML Schema types
- connection pooling and its supporting interfaces and classes are optional,
    - `ConnectionPoolXQDataSource`
    - `PooledXQConnection`
    - `XQConnectionEvent`
    - `XQConnectionEventListener`

4 Overview

This chapter introduces some of the key concepts of XQJ. In addition, in addition, provides an introductory tutorial of common usage scenarios of XQJ using Java code examples. The examples range from simple ones that show the one-time execution of an XQuery expression (incl. processing the result) to more complex ones that show the repeated execution of prepared XQuery expressions with different values bound for each execution, and for example how to make the items contained in a returned result sequence outlive the life time of the connection.

In a typical scenario, the basic building blocks of an XQJ application are:

1) connecting to an XQuery implementation,
2) creating an expression (or prepared expression) from the connection,
3) possibly binding values for external variables used in an expression (or prepared expression),
4) executing the expression (or prepared expression),
5) processing the result,
6) optionally repeating steps 2-6 as needed, and
7) closing the connection and thereby freeing all resources allocated for this connection. This includes also freeing the resources allocated for the expressions and results. Optionally, the resources allocated for the expressions and results can be explicitly freed earlier, as soon as they are no longer needed.

In later sections, a detailed description of the classes and interfaces that make up the XQJ API will be provided..

4.1 Establishing a Connection

The XQJ API defines the `XQConnection` interface, it represents a connection, or session, with a specific XQuery implementation. `XQConnection` objects are obtained through an `XQDataSource` object.

Every XQJ-implementation provides an `XQDataSource` class, in this introduction we'll describe two scenarios for XQJ applications to load an `XQDataSource` object.

The application can directly create an `XQDataSource` object, using the default constructor, and set the necessary properties needed to create an `XQConnection` object.

```
...
// create a VendorXQDataSource object and set some properties
VendorXQDataSource xqds = new VendorXQDataSource();
xqds.setServerName("my_database_server");
xqds.setDatabaseName("my_database");
xqds.setDescription("data source for personnel");
...
```

Example 2 – Create an XQDataSource and set the necessary properties.

Second option is to load the `XQDataSource` object through JNDI, using such approach makes the application code vendor neutral.

```
...
// get the initial JNDI naming context
Context ctx = new InitialContext();

// get the XQDataSource object associated with the logical name
// "xqj/personnel" and use it to obtain a database connection
XQDataSource xqds = (XQDataSource)ctx.lookup("xqj/personnel");
...
```

Example 3 – Get an XQDataSource object through JNDI.

Once the application has an `XQDataSource` object, it can create one or more `XQConnection` instances.

```
...
// establish a connection to the XQuery engine
XQConnection xqc = xqds.getConnection();
...
// free all resources allocated for the connection
xqc.close();
...
```

Example 4 – Establish a connection.

Possibly a user name and password needs to be specified to make the connection, as shown in the next example.

```
...
// establish a connection to the XQuery engine
XQConnection xqc = xqds.getConnection("john","secret");
...
// free all resources allocated for the connection
xqc.close();
...
```

Example 5 – Establish a connection specifying a user name and password.

Note that in the examples above the `XQConnection` object is explicitly closed, this guarantees that all resources are properly released.

4.2 Executing Queries and Retrieving Results

Once the application has established a connection, it can execute queries against the XQuery implementation and retrieve the query results.

The XQJ API provides access to all [XQuery] features. Because [XQuery] defines a number of optional features, the XQJ API defines the `XQMetaData` interface. It allows the application to determine whether the XQuery implementation supports a particular XQuery optional feature.

The `XQConnection` can be used to create `XQExpression` or `XQPreparedExpression` objects. The former allows to execute multiple queries one after the other, the later allows to execute a single query multiple times.

The result of a query execution is an `XQSequence` object. It represents a sequence, as defined in [XQuery], with in addition a cursor. The XQJ API provides a rich set of methods to navigate through an `XQSequence` and the ability to convert the data into Java objects. Also does XQJ provides the ability to serialize the query results according to [XQuery Serialization]

4.3 Introductory Examples

This section provides introductory examples of the most commonly used functionality in the XQJ API

4.3.1 Example – One-time execution of an XQuery Expression

The example given in section 2.1, "Getting Started with XQJ", illustrates how an application might connect to an XQuery engine, execute a single XQuery expression, process the result, and cleanup afterwards. It is copied here for the convenience of the reader. No error handling is shown (see Example 17 for the appropriate error handling).

Assume that `xqds` (representing a data source) already exists.

```
...
// establish a connection to the XQuery engine
XQConnection conn = xqds.getConnection();

// create an expression object that is later used
// to execute an XQuery expression
XQExpression expr = conn.createExpression();

// the XQuery expression to be executed
String es = "for $n in fn:doc('catalog.xml')//item " +
            "return fn:data($n/name)";

// execute the XQuery expression
XQResultSequence result = expr.executeQuery(es);

// process the result (sequence) iteratively
while (result.next()) {
  // retrieve the current item of the sequence as a String
  String str  = result.getAtomicValue();
  System.out.println("Product name: " + str);
}

// free all resources allocated for the result
result.close();

// free all resources allocated for the expression
expr.close();

// free all resources allocated for the connection
conn.close();
...
```

Example 6 – One-time execution of an XQuery expression.

4.3.2 Example – Binding a value to an external variable

Example 7 shows how a value can be bound to an external variable in an XQuery expression.

Assume that a connection `conn` already exists.

```
...
XQExpression expr = conn.createExpression();

// the XQuery expression to be executed
String es = "declare variable $x as xs:integer external;" +
            " for $n in fn:doc('catalog.xml')//item" +
            " where $n/price <= $x" +
            " return fn:data($n/name)";

// bind a value (21) to an external variable with the QName x
QName var1 = new QName("x");
expr.bindInt(var1, 21, null);

// execute the XQuery expression
XQResultSequence result = expr.executeQuery(es);

// process the result (sequence) iteratively
while (result.next()) {
  // process the result ...
}

// free the allocated resources
...
```

Example 7 – Binding a value to an external variable.

4.3.3 Example – Multiple Executions with Different Input Values

The following example illustrates how an application executes the same XQuery expression multiple times, each time with different values bound to the external variable(s).

Since this is a typical scenario, XQJ has the concept of *prepared expressions*, in addition to the expressions that were featured in the previous examples. Prepared expressions are only statically analyzed (*i.e.*, prepared) once and can then be executed multiple times without the need to statically analyze them again.

Assume that a connection `conn` already exists.

```
...
// the XQuery expression to be executed multiple times
String es = "declare variable $x as xs:integer external;" +
            " for $n in fn:doc('catalog.xml')//item" +
            " where $n/price <= $x" +
            " return fn:data($n/name)";

XQPreparedExpression expr = conn.prepareExpression(es);

// bind a value (21) to an external variable with the QName x
QName var1 = new QName("x");
expr.bindInt(var1, 21, null);
```

```
// execute the previously prepared expression and process the
// result
XQResultSequence result = expr.executeQuery();
while (result.next()) {
  // process the result ...
}

// free all resources allocated for the result
result.close();

// bind a different value (42) to the external variable
expr.bindInt(var1, 42, null);

// execute the previously prepared expression again and process
// the result
result = expr.executeQuery();
while (result.next()) {
  // process the result ...
}

// free the allocated resources
...
```

Example 8 – Multiple executions and external variable bindings.

4.3.4 Example – Binding a value to the context item

Similar to Example 7, the next example shows how a value can be bound to the context item.

Assume that a connection `conn` already exists.

```
...
XQExpression expr = conn.createExpression();

// the XQuery expression to be executed
String es = " for $n in fn:doc('catalog.xml')//item" +
            " where $n/price <= ." +
            " return fn:data($n/name)";

// bind a value (21) to the context item
expr.bindInt(XQDynamicContext.CONTEXT_ITEM, 21, null);

// execute the XQuery expression
XQResultSequence result = expr.executeQuery(es);

// process the result (sequence) iteratively
while (result.next()) {
  // process the result ...
}

// free the allocated resources
...
```

Example 9 – Binding a value to the context item of an XQuery expression.

Example 10 shows how a DOM document can be bound to the context item.

Assume that a connection `conn` already exists.

```
...
// the XQuery expression to be executed
// Get all employees named John
String es = "//employee[name='John']";

// prepare the query
XQPreparedExpression expr = conn.prepareExpression(es);

// bind a DOM document to the context item
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder parser = factory.newDocumentBuilder();
Document d = parser.parse("employees.xml");
expr.bindNode(XQDynamicContext.CONTEXT_ITEM, d, null);

// execute the XQuery expression
XQResultSequence result = expr.executeQuery();

// process the result (sequence) iteratively
while (result.next()) {
  // process the result ...
}

// free the allocated resources
...
```

Example 10 – Binding a DOM document to the context item of an XQuery expression.

Some XQuery implementations have support for the Static Typing Feature as defined in [XQuery]. This requires implementations to detect and report type errors during the *static analyses phase*.

In order to perform static typing, the implementation has to know the static type of the context item. The application has to provide the static type, and failing to do so, will result in an error being reported during the static analyses phase. In order to make Example 10 work with an implementation supporting the Static Typing Feature, it needs to specify a document item type as static type for the context item, as shown in Example 11.

```
...
// the XQuery expression to be executed
// Get all employees named John
String es = "//employee[name='John']";

// prepare the query, and specify the static type for the context item:
// document-node(element(*,xs:untyped))
XQItemType ciType = conn.createDocumentElementType(
      conn.createElementType(null, XQItemType.XQBASETYPE_UNTYPED)));
XQStaticContext cntxt = conn.getStaticContext();
cntxt.setContextItemStaticType(ciType);
XQPreparedExpression expr = conn.prepareExpression(es, cntxt);

// bind a DOM document to the context item
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder parser = factory.newDocumentBuilder();
Document d = parser.parse("employees.xml");
expr.bindNode(XQDynamicContext.CONTEXT_ITEM, d, null);
```

```
// execute the XQuery expression
XQResultSequence result = expr.executeQuery();

// process the result (sequence) iteratively
while (result.next()) {
  // process the result ...
}

// free the allocated resources
...
```

Example 11 – Set the static type of the context item.

4.3.5 Example – Retrieving results from an XQuery

The result of executing an XQuery expression through XQJ is an `XQSequence` object.

The `XQSequence` interface allows iterating through the results and reading the items one by one into your Java application. In the next example, we iterate over the `XQSequence` object using `next`, and as we know that all items are atomic values they are retrieved using `getAtomicValue`.

```
...
// prepare the XQuery expression
String es = "for $n in fn:doc('catalog.xml')//item " +
            "return fn:data($n/name)";
XQPreparedExpression expr = conn.prepareExpression(es);

// execute the previously prepared expression
XQResultSequence result = expr.executeQuery();

// process the result
while (result.next()) {
  System.out.println(result.getAtomicValue());
}

// free the allocated resources
...
```

Example 12 – Retrieve atomic values out of the query result.

XQJ provides a rich set of method to retrieve query result, in the next example the results are retrieved as DOM objects.

```
...
// prepare the XQuery expression
String es = "fn:doc('catalog.xml')//item"
XQPreparedExpression expr = conn.prepareExpression(es);

// execute the previously prepared expression
XQResultSequence result = expr.executeQuery();

// process the result
while (result.next()) {
  Node domNode = result.getNode());
  // do something with the DOM nodes
}
```

```
// free the allocated resources
...
```

Example 13 – Retrieve the query results as DOM nodes.

As a last example, XQJ also supports [XQuery Serialization]. In the example below the query results are serialized in a file result.xml.

```
...
// prepare the XQuery expression
String es = "<result>{fn:doc('catalog.xml')//item}</result>"
XQPreparedExpression expr = conn.prepareExpression(es);

// execute the previously prepared expression
XQResultSequence result = expr.executeQuery();

// serialize the query result into a file
// specify the indent serialization option to get the results
// pretty printed
Properties props = new java.util.Properties();
props.setProperty("method", "xml");
props.setProperty("indent", "yes");

FileOutputStream file = new FileOutputStream("result.xml")
result.writeSequence(file, props);

// free the allocated resources
...
```

Example 14 – Serialize the query result in a file.

4.3.6 Example – Using a Result Sequence as Input

The following example shows how the output (an `XQResultSequence`) of one `XQExpression` can be used as input for another `XQExpression`.

Note: In this example, `expr1` and `expr2` are both created from the same connection. This example would work equally well if the expressions were created from different connections.

Assume that a connection `conn` already exists.

```
...
XQExpression expr1 = conn.createExpression();
XQExpression expr2 = conn.createExpression();

String es1 = "declare variable $x as xs:integer external;" +
             " for $n in fn:doc('catalog.xml')//item" +
             " where $n/price <= $x" +
             " return $n";

String es2 = "declare variable $y external;" +
             " for $n in $y" +
             " return fn:data($n/name)";

// bind a value (21) to a local external variable of expr1 with
// the QName x
QName var1 = new QName("x");
```

```
expr1.bindInt(var1, 21, null);

// expr1 executes with value 21 bound to x
XQResultSequence result1 = expr1.executeQuery(es1);
// result1 is a sequence of items

// bind result1 to a local external variable of expr2 with the QName y
QName var2 = new QName("y");
expr2.bindSequence(var2, result1);
// After the execution of the bindSequence() method above, result1
// is completely consumed since it is by default forward only.
// To use the result sequence from expr1 more than once, it can
// either be copied into an XQSequence as shown in Example 16 or
// the result sequence result1 could have been declared as scrollable

// expr2 executes with result1 bound to y
XQResultSequence result2 = expr2.executeQuery(es2);

// process result2 ...

// free the allocated resources
...
```

Example 15 – Chaining XQExpressions.

4.3.7 Example – Making the Result available after closing the Connection

All of the previous examples made the implicit assumption that the result sequence would be processed completely while the connection during which it was returned was still active. Since a result sequence becomes inaccessible when the connection is closed, the following example shows how a sequence can be used to have the result of an XQuery expression available even after the connection was closed. A practical application of this feature is that it enables, for example, application server modules to release the connection, which is typically a shared resource, as early as possible.

Assume that a connection `conn` and a result sequence `result` already exist.

```
...
// create a new sequence object independent of the connection
// and populate it with the items from "result"
XQSequence sequence = conn.createSequence(result);

// free all resources allocated for the result and connection
result.close();
conn.close();


// work with the sequence
sequence.beforeFirst();
while (sequence.next()) {
  // process the sequence ...
}

// work some more with the sequence
while (sequence.previous()) {
```

```
  // process the sequence ...
}

// free all resources allocated for the sequence
sequence.close();
...
```

Example 16 – Using a sequence independent of a connection.

4.3.8 Example – Error Handling

To make the teaching examples not overly complex, the error handling in those is either omitted or minimal. However, catching an exception when it is thrown is a responsibility of the application. Equally important, it is the responsibility of the application to free the resources in a finally block to ensure the resources are freed whether or not an exception is thrown. The following example rewrites Example 6 to show how error handling can be done.

```
...
// declare variables needed in the finally block
XQConnection conn = null;
XQExpression expr = null;
XQResultSequence result = null;

try {
  // establish a connection to the XQuery engine
  conn = xqds.getConnection();

  // create an expression object that is later used
  // to execute an XQuery expression
  expr = conn.createExpression();

  // the XQuery expression to be executed
  String es = "for $n in fn:doc('catalog.xml')//item " +
              "return fn:data($n/name)";

  // execute the XQuery expression
  result = expr.executeQuery(es);

  // process the result (sequence) iteratively
  while (result.next()) {
    // retrieve the current item of the sequence as a String
    String str = result.getAtomicValue();
    System.out.println("Product name: " + str);
  }
}
catch (XQException e) {
  // handle the exception in some manner
  e.printStackTrace();
}
// free the resources whether or not there was an exception
finally {
  // free all resources allocated for the result
  if (result != null) {
    try {
      result.close();
    }
```

```
    catch (XQException ignored) {
      //write to log
    }
  }

  // free all resources allocated for the expression
  if (expr != null) {
    try {
      expr.close();
    }
    catch (XQException ignored) {
      //write to log
    }
  }

  // free all resources allocated for the connection
  if (conn != null) {
    try {
      conn.close();
    }
    catch (XQException ignored) {
      //write to log
    }
  }
}
...
```

Example 17 – Handling exceptions.

5 Classes and Interfaces

The following list contains the classes and interfaces that are contained in the `javax.xml.xquery` package. Classes are highlighted in bold; interfaces are in normal type.

`javax.xml.xquery.ConnectionPoolXQDataSource`

`javax.xml.xquery.PooledXQConnection`

`javax.xml.xquery.XQCancelledException`

`javax.xml.xquery.XQConnection`

`javax.xml.xquery.XQConnectionEvent`

`javax.xml.xquery.XQConnectionEventListener`

`javax.xml.xquery.XQConstants`

`javax.xml.xquery.XQDataFactory`

`javax.xml.xquery.XQDatasource`

`javax.xml.xquery.XQDynamicContext`

`javax.xml.xquery.XQException`

`javax.xml.xquery.XQExpression`

`javax.xml.xquery.XQItem`

`javax.xml.xquery.XQItemAccessor`

`javax.xml.xquery.XQItemType`

`javax.xml.xquery.XQMetaData`

`javax.xml.xquery.XQPreparedExpression`

`javax.xml.xquery.XQQueryException`

`javax.xml.xquery.XQResultItem`

`javax.xml.xquery.XQResultSequence`

`javax.xml.xquery.XQSequence`

`javax.xml.xquery.XQSequenceType`

`javax.xml.xquery.XQStackTraceElement`

`javax.xml.xquery.XQStackTraceVariable`

`javax.xml.xquery.XQStaticContext`

The next diagrams show the interaction and relationship between some of the key interfaces of XQJ. Figure 1 shows the interaction of the objects and methods involved in creating expressions and retrieving query results.

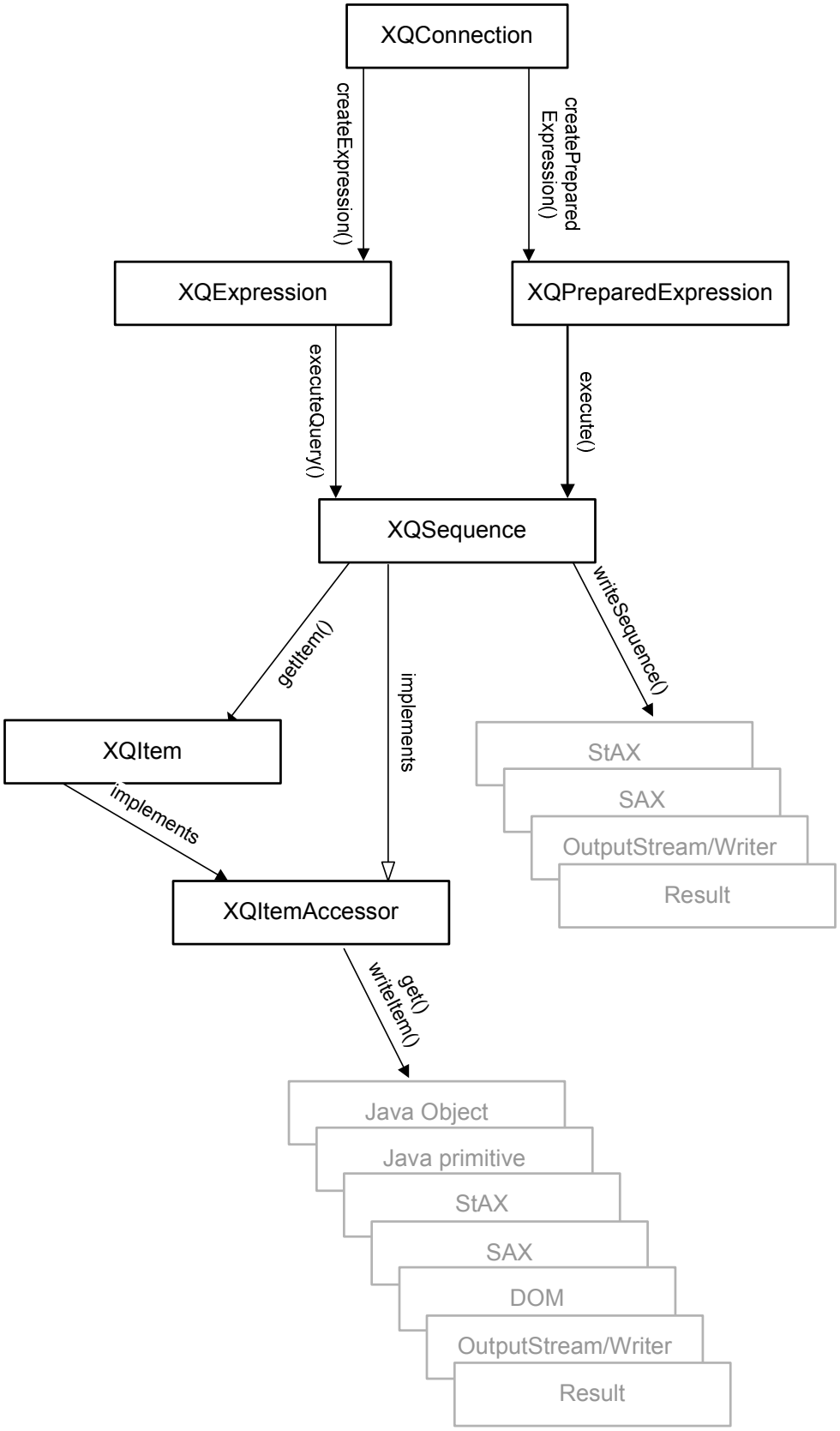

Figure 1 – Relationship Between Major XQJ Interfaces

Figure 2 shows the interaction of the objects and methods involved in creating items and sequences, and binding to dynamic contexts.

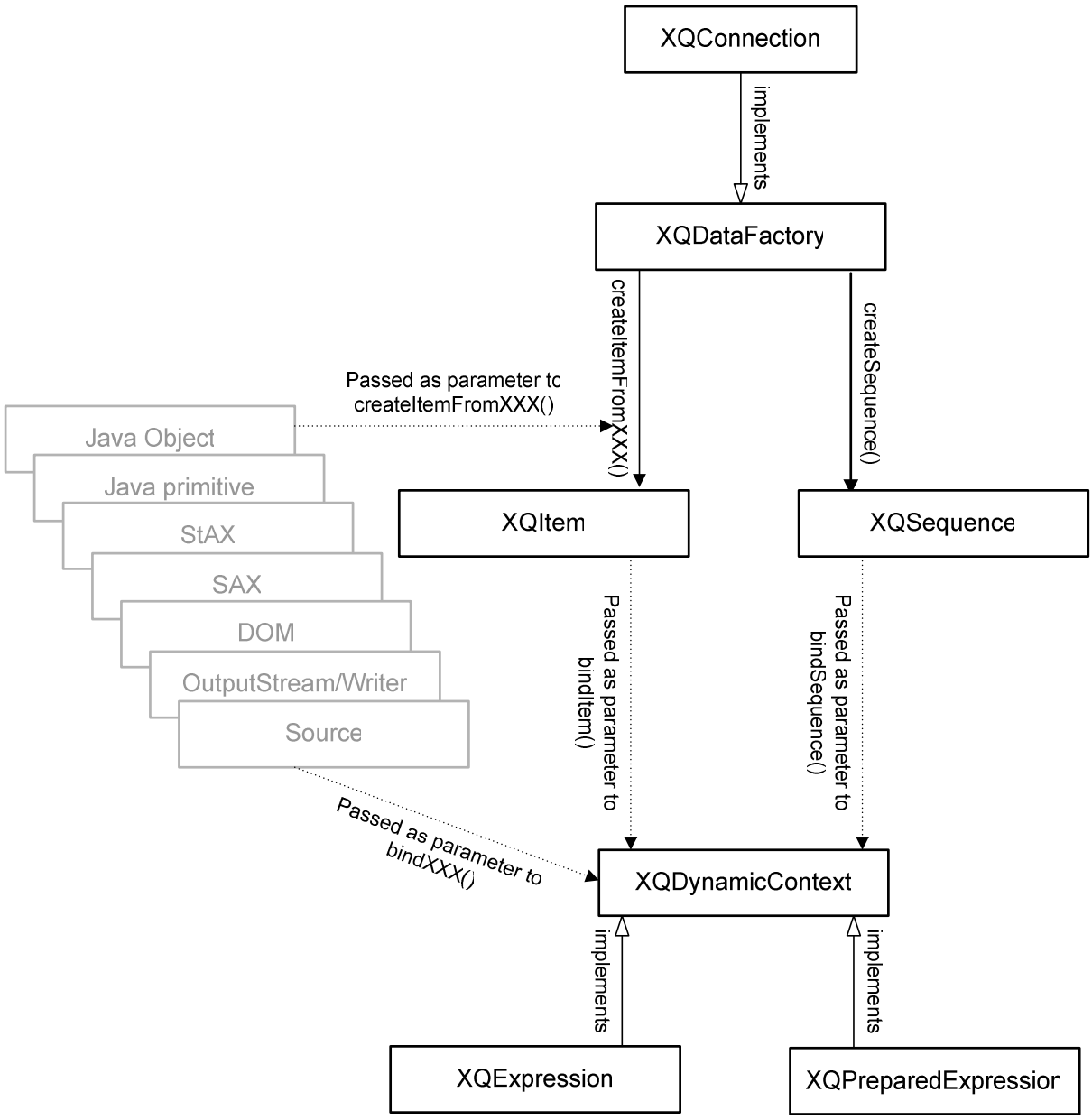

Figure 2 – Relationship Between Major XQJ Interfaces

6 Exceptions

As described in [XQuery], XQuery defines several kinds of errors:

- Static errors detected during the static analysis phase of an XQuery expression. An example of a static error is a syntax error.
- Dynamic errors detected during the evaluation phase of an XQuery expression. A dynamic error may be detected during the static analysis phase.
- Type errors caused by type mismatch. A type error is raised during the static analysis or evaluation phases.
- Warnings raised by an XQuery implementation during the static analysis or evaluation phases.
- Errors raised by XQuery function `fn:error()`. These errors are dynamic errors that are raised in the evaluation phase. These errors must not be raised during the static analysis phase.

In addition to the XQuery errors, XQJ also defines erroneous scenarios, and finally errors might also be reported due to implementation-dependent failures.

XQJ provides exception classes and a mechanism to handle the above kinds of errors. It defines the following exception classes.

- `XQException`.
- `XQCancelledException`.
- `XQQueryException`.

An XQJ exception records information such as an error code, an error message, a sequence of zero or more items associated with the exception, an error position, and a module and function name where the error occurred. The XQJ exception classes provide methods to access this information. XQJ exceptions may be chained to form a list of exceptions.

6.1.1 Navigating XQExceptions

It is possible that during the analysis or evaluation of a query one or more errors occur. This means that when an XQJ application catches an `XQException`, there is a possibility that there may be additional `XQException` objects chained to the original thrown `XQException`. Imagine for example an implementation reporting multiple errors detected during the static analysis phase. To access the additional chained `XQException` objects, an application would recursively invoke the `getNextException` method until a `null` value is returned.
Catching an `XQException`, one can also test if it is an instance of `XQQueryException`, and retrieve any of the additional properties.
An `XQException` may have a causal relationship, which consists of one or more `Throwable` instances that caused the `XQException` to be thrown. This is the so called Java SE chained exception facility, also known as the cause facility.
The application may recursively call the method `getCause`, until a `null` value is returned, to navigate the chain of causes.

The following code demonstrates how an application could navigate the chain of `XQException` objects and their causes:

```
try {
  // do something...
}
catch(XQException ex) {
  while(ex != null) {
    System.out.println("Message:" + ex.getMessage());
    System.out.println("VendorCode:" + ex.getVendorCode());
    if(ex instanceof XQQueryException)
      System.out.println("ErrorCode:" +
                    ((XQQueryException)ex).getErrorCode());
    Throwable t = ex.getCause();
    while(t != null) {
      System.out.println("Cause:" + t);
      t = t.getCause();
    }
    ex = ex.getNextException();
  }
}
```

Example 18 – Navigating multiple XQExceptions.

Example 18 checks if the `XQException` objects are an instance of `XQQueryException`, and retrieve some of the additional properties. Alternatively, the application can catch first `XQQueryException` and then `XQException`, as shown in Example 19.

```
try {
  // do something...
}
catch (XQQueryException ex) {
  // handle the XQQueryException
}
catch(XQException ex) {
  // handle the XQException when it is not an XQQueryException
}
```

Example 19 – Catch and handle XQQueryExceptions different from XQExceptions.

6.1.2 The XQException class

The `XQException` class provides information on XQJ, XQuery or other errors reported by an XQJ-implementation.
Each `XQException` provides several kinds of information:

- A string describing the error. This is used as the Java Exception message, available via the method `getMessage`.
- The cause of the error. This is used as the Java Exception cause, available via the method `getCause`.
- The vendor code identifying the error, available via the method `getVendorCode`.

- A chain of `XQException` objects. If more than one error occurred, the exceptions in the chain are referenced via the method `getNextException`.

Note that `XQException` has a subclass `XQQueryException` providing more detailed information about errors occurred during the execution of a query. An implementation throws a base `XQException` when an error occurs in the XQJ-implementation. Implementations are encouraged to use the more detailed `XQQueryException` in case of an error reported by the XQuery engine.

6.1.3 The XQQueryException class

The `XQQueryException` class provides information on errors occurring during the static analysis and evaluation phase of an XQuery expression. Each `XQQueryException` provides several kinds of optional information, in addition to the properties inherited from `XQException`:

- An error code. This `QName` identifies the error according to [XQuery]; in addition, implementation-defined errors may be raised by a given implementation.
- A position. This identifies the character position of the failing expression in the query text.
- A line and column number. These identify the position of the failing expression in the query text.
- The module URI. This identifies the module in which the error occurred, null when the error is located in the main module.
- The XQuery error object of this exception. This is the `$error-object` argument specified through the `fn:error()` function.
- The XQuery stack trace. This provides additional dynamic information where the exception occurred in the XQuery expression.

6.1.4 The XQCancelledException class

`XQCancelledException` extends `XQQueryException` and differs only in its name. Thus allowing easy differentiation between general errors and halting due to cancellation. This exception is thrown, if a running XQuery expression is successfully cancelled.

Example 20 shows how `XQCancelledException` can be used in an application.

```
try {
  XQSequence result = m_expr.executeQuery(m_query);
  while ( result.next() ) {
    // processing query result
  }
} catch ( XQCancelledException e) {
  System.out.println("query execution is cancelled");
} catch ( XQException e) {
  System.out.println(e.getMessage());
}
...
```

Example 20 – Processing XQCancelledExceptions.

Example 47 is a more complete example on how an XQuery evaluation can be cancelled.

6.1.5 Serializing XQException objects

Java specifies that every `Throwable` object is serializable, implying that every `XQException` object is serializable.

However, every `XQQueryException` has an optional error object property. This is an `XQSequence` object, and is not required to be serializable.

When an `XQQueryException` is serialized, the error object property will be added to the serialized representation if the `XQSequence` object itself is serializable. As such the `getErrorObject` method will return the `XQSequence` object when the `XQQueryException` was obtained by deserialization. However, when the `XQSequence` object is not serializable, the error object property will not be preserved during the serialization process. And as such `getErrorObject` will return `null` when the `XQQueryException` was obtained by deserialization.

7 DataSources

The `XQDataSource` interface is the approach to obtaining data source connections. In other words, an application uses an `XQDataSource` object to initiate an XQJ session with a specific XQJ-implementation and to create `XQConnection` objects.

An `XQDataSource` object identifies a physical data source (or data sources) which can be a DBMS, a legacy file system, some other source of data, or possibly a combination of those. To identify the physical data source and to create `XQConnection` objects, each XQJ-implementation defines a number of implementation-defined properties.

Using an `XQDataSource` object also enhances application portability if used in conjunction with the Java Naming and Directory Interface™ (JNDI) API. A logical name is mapped to an `XQDataSource` object via a naming service, which avoids the need of having to supply information specific to a particular implementation in your application.

This chapter describes in detail the `XQDataSource` interface, connection pooling is described later in chapter 17 “Connection Pooling”.

7.1 XQDataSource Properties

The XQJ API defines 2 standard properties as shown in Table 1.

Property name	Description
`user`	The user name to use for creating a connection. If a user name is appropriate for an implementation, this property must be supported.
`password`	The password to use for creating a connection. If a password is appropriate for an implementation, this property must be supported.

Table 1 – Standard XQDataSource properties

`XQDataSource` implementations may augment this set with implementation-defined properties. Such properties must be given names that do not conflict with the standard property names.

`XQDataSource` properties must follow the convention specified for properties of JavaBeans™ components in the JavaBeans 1.01 Specification. `XQDataSource` implementations must provide “getter” and “setter” methods for each property they support. These properties typically are initialized when the `XQDataSource` object is deployed.

```
VendorXQDataSource vds = new VendorXQDataSource();
vds.setServerName("my_database_server");
String name = vds.getServerName();
```

Example 21 – Setting and getting an XQDataSource property.

In addition to the implementation-defined JavaBeans-like “getter” and “setter” methods, the `XQDataSource` interface defines methods to set and get these properties in a more generic way using the `setProperty` and `getProperty` methods.

```
...
vds.setProperty("serverName","my_database_server");
String name = vds.getProperty("serverName");
```

Example 22 – Setting and getting an XQDataSource property.

The properties of an `XQDataSource` can also be set through a `Properties` object using the `setProperties` method.

```
Properties props = new Properties();
...
xqds.setProperties(props);
```

Example 23 – Setting multiple properties.

7.2 Creating an XQDataSource and XQConnection

A first approach to create an `XQDataSource`, is to use the default constructor and call the appropriate JavaBeans "setter" methods. With such approach, the XQJ implementation is referenced and hard-coded into the application.

```
VendorXQDataSource vds = new VendorXQDataSource();
vds.setServerName("my_database_server");
vds.setUser("john");
vds.setPassword("topsecret");
XQConnection conn = vds.getConnection()
```

Example 24 – Create an XQDataSource and XQConnection.

As shown in Example 24, the `getConnection` method is used to create `XQConnection` objects. The `XQConnection` represents a session in which XQuery expressions are executed, as further explained in 8 "Connections".

7.3 Application Portability

Applications can improve the portability by removing the hard-coded reference to a specific XQJ implementation. One approach is to specify the `XQDataSource` class in a Java property. As every `XQDataSource` must implement the default constructor, the application can use a generic reflection-based approach to create an `XQDataSource` object.

For example, a Java system property like `javax.xml.xquery.XQDataSource` can be used. Note that this system property is not initialized by XQJ, it must be set externally, using the –D option of the Java command for example.

Next, the necessary `XQDataSource` properties can be set through the `setProperties` method. How the application obtains the `Properties` object is out of scope of the XQJ specification.

```
String dataSourceClassName =
              System.getProperty("javax.xml.xquery.XQDataSource");
Class dataSourceClass = Class.ForName(dataSourceClassName);
XQDataSource xqds = dataSourceClass.newInstance();
...
xqds.setProperties(props);
```

Example 25 – Use reflection to create an XQDataSource and set the properties.

7.4 The JNDI API and Application Portability

Using an `XQDataSource` object and registering it in a naming service through JNDI, will further increase the application portability. The JNDI API provides a uniform way for applications to access naming and directory services. See [JNDI] for a complete description of this API.

It makes it possible for an application to use a logical name for a data source instead of having any hard-coded information specific to a particular XQJ-implementation. The logical name is mapped to an `XQDataSource` object via a naming service using JNDI. If the data source or information about it changes, the properties of the `XQDataSource` object can simply be modified in the naming service to reflect the changes; no change in application code is necessary. In fact, the application can be re-directed to a different underlying data source or switch XQJ-implementation in a completely transparent fashion. This is particularly useful in the three-tier environment, where an application server hides the details of accessing different data sources.

In such setup the `XQDataSource` properties are not intended to be directly accessible by XQJ applications. This design is reinforced by defining the access methods on the implementation class rather than on the public `XQDataSource` interface used by applications

Management tools that need to manipulate the properties of an `XQDataSource` implementation can access those properties using the Java reflection API or through the property-based methods. Example 26 illustrates the use of a JNDI-based naming service to deploy a new `VendorXQDataSource` object.

```
// Create a VendorXQDataSource object and set some properties
VendorXQDataSource vds = new VendorXQDataSource();
vds.setServerName("my_database_server");
vds.setDatabaseName("my_database");
vds.setDescription("data source for inventory and personnel");

// Use the JNDI API to register the new VendorXQDataSource object.
// Reference the root JNDI naming context and then bind the
// logical name "xqj/AcmeDB" to the new VendorXQDataSource object.
Context ctx = new InitialContext();
ctx.bind("xqj/AcmeDB", vds);
```

Example 26 – Registering an XQDataSource object with a JNDI-based naming service.

Once an `XQDataSource` object has been registered with a JNDI-based naming service, an application can use it to obtain a connection to the physical data source that it represents, as is done in Example 27.

```
// Get the initial JNDI naming context
Context ctx = new InitialContext();

// Get the XQDataSource object associated with the logical name
// "xqj/AcmeDB" and use it to obtain a database connection
XQDataSource ds = (XQDataSource)ctx.lookup("xqj/AcmeDB");
```

```
XQConnection con = ds.getConnection("user", "pwd");
```

Example 27 – Getting an XQConnection object using an XQDataSource object.

8 Connections

An `XQConnection` object represents a session in which XQuery expression are executed to query a data source via a specific XQJ-implementation. The data source can be a DBMS, a legacy file system, some other source of data, or possibly a combination of those, with a corresponding XQJ driver. A single application using the XQJ API may maintain multiple connections. These connections may access multiple data sources, or they may all access a single data source.

From the application perspective, an `XQConnection` object represents a client session in which XQuery expressions are executed. It has associated state information such as a set of XQuery expressions and results being used in that session, and what transaction semantics are in effect. To obtain a connection, the application interacts with an `XQDataSource` implementation.

8.1 Creating an XQConnection

In a typical scenario, an XQJ application will connect to a target data source using the `XQDataSource` interface. As explained in 7 “DataSources”, this interface allows details about the underlying data source to be transparent to the application. An `XQDataSource` object can be obtained through JNDI lookup or can explicitly be created by calling a vendor's XQJ-implementation class for `XQDataSource` object. The latter one is not a portable way of creating an `XQDataSource` object. The former one is the recommended way of obtaining an `XQDataSource` object and it conforms to the standard Java EE approach of using JNDI to write portable application code that works with different vendors’ XQJ-implementations.

An `XQDataSource` object’s properties are set so that it represents a particular data source. When its `getConnection` method is invoked, the `XQDataSource` instance will return a connection to that data source. An application can be directed to a different data source by simply changing the `XQDataSource` object’s properties; no change in application code is needed. Likewise, an `XQDataSource` implementation can be changed without changing the application code that uses it.

Example 28 shows how to create an `XQConnection` object.

```
// Assume the application has obtained an XQDataSource object
XQDataSource ds = ...;
XQConnection conn = ds.getConnection();
```

Example 28 – Create an XQConnection.

If a user name and password are needed to create the `XQConnection` object, the application can specify these as argument to the `getConnection` method, as shown in Example 29.

```
// Assume the application has obtained an XQDataSource object
XQDataSource ds = ...;
XQConnection conn = ds.getConnection("john", "topsecret");
```

Example 29 – Create an XQConnection, specifying a user name and password.

If no authentication is needed for a specific data source, the XQJ-implementation can ignore the specific user name and password. In such case calling `getConnection` with user name and password is equivalent to the parameter-less `getConnection` flavor.

8.2 Using an XQConnection

Once a connection is established, it can be used to:

- create `XQExpression` objects
- create `XQPreparedExpression` objects
- get and set the default `XQStaticContext`

The lifetime of the `XQExpression` and `XQPreparedExpression` objects are dependent on the `XQConnection` object from which they are created. *I.e.*, they become invalid when the connection is closed.

In addition, an `XQConnection` object also implements the `XQDataFactory` interface, it can be used to create:

- `XQItem` objects
- `XQSequence` objects
- `XQItemType` objects
- `XQSequenceType` objects

`XQItem`, `XQSequence`, `XQItemType`, and `XQSequenceType` objects create through the `XQDataFactory` interface are independent of any `XQConnection` object. *I.e.*, they will still be valid after the connection is closed.

Besides the aforementioned functionality, the `XQConnection` interface also provides methods to manipulate the transactional semantics.

Subsequent chapters of this document describe each of these functionalities in more detail.

8.3 Interacting with JDBC Connections

XQJ provides the ability to create an `XQConnection` based on an existing JDBC Connection, as shown in Example 30.

```
...
Connection jdbcConn = DriverManager.getConnection("jdbc:odbc:DSN",
                                                  "top",secret");
// Assume the application has obtained an XQDataSource object
XQDataSource ds = ...;
XQConnection conn = ds.getConnection(jdbcConn);
...
```

Example 30 – Create an XQConnection from a JDBC Connection.

This is an optional feature, an XQJ-implementation is not required to support this functionality, in which case an `XQException` is thrown.

How the JDBC Connection object is subsequently used in the context of the XQJ connection object is implementation-defined. However, XQJ mandates that both connections operate under the same transaction context, as is further explained in paragraph 16.3 “XQConnections on top of JDBC connections”.

Once the `XQConnection` object is closed, the JDBC Connection is dereferenced by the XQJ-implementation and the application can safely close the JDBC Connection object. To continue with Example 30,

```
...
// close the XQJ connection
conn.close();
// the XQJ implementation doesn't reference the JDBC
// connection anymore. It can be safely closed now.
jdbcConn.close();
...
```

Example 31 – Closing XQJ and JDBC connection objects.

9 Static Contexts

[XQuery] defines two phases of processing an XQuery expression called the *static analysis phase* and the *dynamic evaluation phase*. During the static analysis phase, the query is parsed into an internal representation called the operation tree. The static analysis phase depends on the expression itself and on the *static context*. [XQuery] defines the static context of an expression to be the information that is available during static analysis of the expression, prior to its evaluation. The static context is initialized by the XQuery implementation and can be changed and augmented by the XQJ application through the `XQStaticContext` interface. Finally, the static context is then changed and augmented based on information in the prolog of the XQuery expression. For more details on the static context, see [XQuery], section 2.1.1 "Static Context" and appendix C.1 "Static Context Components".

XQJ also defines a number of properties to change the behavior of `XQExpression` and `XQPreparedExpression` objects, for example the query timeout property. In addition to the XQuery static context components, also these XQJ expression properties can be changed through the `XQStaticContext` interface.

9.1 XQStaticContext

The `XQStaticContext` interface represents default values for various XQuery static context components and in addition the expression properties defined by XQJ. Each of these components and properties can be retrieved and changed through an `XQStaticContext` object.

The `XQStaticContext` interface supports the following XQuery Static Context Components:

- Statically known namespaces
- Default element/type namespace
- Default function namespace
- Context item static type
- Default collation
- Construction mode
- Ordering mode
- Default order for empty sequences
- Boundary-space policy
- Copy-namespaces mode
- Base URI

In accordance with [XQuery], the static context, which is initialized by the XQuery implementation, which includes the XQJ API, can be changed and augmented by the prolog of an XQuery expression too.

In addition `XQStaticContext` includes the XQJ properties for `XQExpression` and `XQPreparedExpression` objects:

- Binding mode
- Holdability of the result sequences
- Scrollability of the result sequences
- Query language
- Query timeout

9.2 Retrieving the implementation's default values

An application can retrieve the implementation's default values of the XQuery Static Context Components and XQJ expression properties through the `XQConnection` object. After having created an `XQConnection` object, the `getStaticContext` method returns an `XQStaticContext` holding the implementation's default values.

The following example determines the implementation's default value for the "base uri" XQuery Static Context Component.

```
...
// assume we have an XQDataSource object xqds
XQConnection conn = xqds.getConnection();

// get a static context object with the default values
XQStaticContext cntxt = conn.getStaticContext();

// report the default base uri
System.out.println(cntxt.getBaseURI());
...
```

Example 32 – Retrieve the implementation's default static context values.

9.3 Changing the implementation's default

XQJ allows the application to change the implementation's default values for a specific `XQConnection` object.

As explained in the previous paragraph, the application gets access to an `XQStaticContext`, this is a value object changing any of the XQuery static context components or XQJ expression properties doesn't yet affect the implementation. Only calling `setStaticContext` on the `XQConnection` object will make the new values in the `XQStaticContext` effective. One can say that `XQStaticContext` objects are *passed by value* from the XQJ driver to the application and vice-versa.

```
...
// assume we have an XQDataSource object xqds
XQConnection conn = xqds.getConnection();

// get a static context object with the default values
XQStaticContext cntxt = conn.getStaticContext();

cntxt.setBaseURI("file:///root/user/john/");
```

```
// make the changes effective
conn.setStaticContext(cntxt);
```

Example 33 – Change the implementation's defaults static context values.

If the implementation's default static context values are changed, existing `XQExpression` and `XQPreparedExpression` objects are not affected. Only subsequently created `XQExpression` and `XQPreparedExpression` objects will assume these new defaults. Consider the following example, the first `XQPreparedExpression` object will assume file:///root/user/john/ as base uri, the second file:///root/user/jessy/.

```
...
// assume we have an XQDataSource object xqds
XQConnection conn = xqds.getConnection();

// the XQuery expression to execute
String xquery = "fn:doc('books.xml')//title"

// get a static context object with the default values
XQStaticContext cntxt = conn.getStaticContext();

// change the base uri
cntxt.setBaseURI("file:///root/user/john/");

// make the changes effective
conn.setStaticContext(cntxt);

// prepare expression, it will query
// file:///root/user/john/books.xml
XQPreparedExpression expr1 = conn.prepareExpression(xquery);

// change the base uri
cntxt.setBaseURI("file:///root/user/jessy/");

// make the changes effective
conn.setStaticContext(cntxt);

// prepare expression, it will query
// file:///root/user/jessy/books.xml
XQPreparedExpression expr2 = conn.prepareExpression(xquery);

// get all titles in john's books.xml
XQSequence seq1 = expr1.execute();

// get all titles in jessy's books.xml
XQSequence seq2 = expr2.execute();
...
```

Example 34 – Changing defaults static context values doesn't affect existing expressions.

9.4 Setting the static context for a specific expression

Rather than changing the implementation's default values through the `XQConnection` object, an application can also create an `XQExpression` or

`XQPreparedExpression` with explicitly specifying an `XQStaticContext`. In such case, the specified `XQStaticContext` will be used, rather than the implementation's default values, for that specific `XQExpression` or `XQPreparedExpression` object.

During the creation of the `XQExpression` or `XQPreparedExpression`, the implementation will read all static context components and XQJ expression properties out of the `XQStaticContext`. Subsequent changes to this `XQStaticContext` object will not affect previously created `XQExpression` and `XQPreparedExpression` objects. One can say that `XQStaticContext` objects are *passed by value* from the application to the XQJ driver.

Consider the previous code example, but rather than changing the defaults on the `XQConnection` object, the `XQStaticContext` is specified when the `XQPreparedExpression` objects are created.

```
...
// assume we have an XQDataSource object xqds
XQConnection conn = xqds.getConnection();

// the XQuery expression to execute
String xquery = "fn:doc('books.xml')//title"

// get a static context object with the default values
XQStaticContext cntxt = conn.getStaticContext();

// change the base uri
cntxt.setBaseURI("file:///root/user/john/");

// prepare expression, it will query
// file:///root/user/john/books.xml
XQPreparedExpression expr1 = conn.prepareExpression(xquery, cntxt);

// change the base uri
cntxt.setBaseURI("file:///root/user/jessy/");

// prepare expression, it will query
// file:///root/user/jessy/books.xml
XQPreparedExpression expr2 = conn.prepareExpression(xquery, cntxt);

// get all titles in john's books.xml
XQSequence seq1 = expr1.execute();

// get all titles in jessy's books.xml
XQSequence seq2 = expr2.execute();
...
```

Example 35 – Specify the static context when creating expressions.

Such approach is especially interesting if some XQuery static context components or XQJ expression properties need to be changed for a specific `XQExpression` or `XQPreparedExpression`, but want to keep the default values for most other expression being executed.

9.5 Retrieving the static context of an expression

Both `XQExpression` and `XQPreparedExpression` have functionality to return an `XQStaticContext`, representing the XQuery static context components and XQJ expression properties used when the object was created.

This is shown in the following code example.

```
...
// assume we have an XQDataSource object xqds
XQConnection conn = xqds.getConnection();

// the XQuery expression to execute
String xquery = "fn:doc('books.xml')//title"

// get a static context object with the default values
XQStaticContext cntxt = conn.getStaticContext();

// change the base uri
cntxt.setBaseURI("file:///root/user/john/");

// prepare expression, it will query
// file:///root/user/john/books.xml
XQPreparedExpression expr1 = conn.prepareExpression(xquery, cntxt);

// change the base uri
cntxt.setBaseURI("file:///root/user/jessy/");

// prepare expression, it will query
// file:///root/user/jessy/books.xml
XQPreparedExpression expr2 = conn.prepareExpression(xquery, cntxt);

// returns "file:///root/user/john/ "
expr1.getStaticContext().getbaseURI

// returns "file:///root/user/jessy/ "
expr2.getStaticContext().getbaseURI
...
```

Example 36 – Retrieve the static context of an expression.

10 Expressions

[XQuery] defines two phases of processing an XQuery expression called the *static analysis phase* and the *dynamic evaluation phase*. The static analysis phase depends on the static context, as described in 9 “Static Contexts”. The dynamic evaluation phase, which computes the value of a given expression, occurs after completion of the static analysis phase. The dynamic evaluation phase depends on the operation tree of the expression being evaluated, on the input data, and on the *dynamic context*, which in turn draws information from the external environment and the static context (for more details on the dynamic context, see [XQuery], section 2.1.2 “Dynamic Context”).

The external variables declared in the dynamic context are initialized through the XQJ interface `XQDynamicContext`. The `XQDynamicContext` interface is also the base for both `XQExpression` and `XQPreparedExpression`.

`XQExpression` and `XQPreparedExpression` are the two interfaces that represent an XQuery expression to be evaluated by an XQuery implementation. In general, a prepared expression can be used to execute the same XQuery expression multiple times, while an unprepared expression (*i.e.*, an `XQExpression` object) can be used to evaluate different XQuery expressions once. While both interfaces have a great commonality, they are also sufficiently different and serve different purposes, so that a different treatment is warranted.

The commonalities include:

- accepting the same representation of the expression to be evaluated (*i.e.*, XQuery, XQueryX, or an implementation-defined format),
- being created from, and therefore dependent, on a connection,
- being used to model and evaluate an XQuery expression, and
- allowing binding values to external variables (by extending the `XQDynamicContext` interface).

The differences include:

- an `XQPreparedExpression` is used to evaluate the same XQuery expression multiple times (possibly with different bindings for external variables), while an `XQExpression` is generally used to execute a given XQuery expression only once.
- a given `XQPreparedExpression` object can only be used to evaluate exactly one XQuery expression, while a given `XQExpression` object can be reused to execute different XQuery expressions.
- the `XQPreparedExpression` interface models the two phases of the XQuery expression processing model directly by differentiating the preparation phase (*i.e.*, when an `XQPreparedExpression` object is created from the connection) and the execution phase (*i.e.*, when the `executeQuery` method is invoked) used to evaluate the expression, while the `XQExpression` interface combines these two phases.
- the `XQExpression` interface provides methods to evaluate implementation-defined commands (non-XQuery) through `executeCommand`.

10.1 XQDynamicContext

The `XQDynamicContext` interface allows an application to retrieve information about the dynamic context and to manipulate it. This interface is extended by the `XQExpression` and `XQPreparedExpression` interfaces. The `XQDynamicContext` interface provides methods:

- to retrieve the value of the implicit time zone,
- to change the value of the implicit time zone,
- to bind a value to the context item, and
- to bind a value to an external variable with a given QName.

There is a relationship between the values and data types for external variables that are supplied by an XQJ-implementation and the XQuery dynamic context. The value of an external variable that is declared in an XQuery expression can be set through the `XQDynamicContext` interface.

XQJ specifies how a Java value is converted into an XDM instance., according to the rules described in section 14 “Data Type Conversions”.

Binding a value to an external variable is further also governed by [XQuery]. As specified in [XQuery] section 4.14 “Variable Declaration”, if the external variable declaration in the query includes a declared type, then that XDM instance must match the declared type according to the rules for Sequence Type matching.

XQJ specifies it is implementation-dependent whether the Sequence Type matching rules are performed during the bind process, or later during the *dynamic evaluation phase*.

Example 37 shows how a value can be bound to an external variable in an XQuery expression. As a variable name in XQuery is a QName, the binding is performed by name, the application specifies a `javax.xml.namespace.QName`.

```
...
XQExpression expr = conn.createExpression();

// the XQuery expression to be executed
String es = "declare variable $i as xs:integer external; " +
            "$i + 1 ";

// bind a value to the external variable i
expr.bindInt(new QName("i"), 21, (XQType)null);

// execute the XQuery expression
XQResultSequence result = expr.executeQuery(es);
...
```

Example 37 – Bind a value to an external variable.

In Example 37 a java `int` is bound to the external variable `$i`. Note that the third parameter of `bindInt` is `null`. Which means that the default Java to XDM instance mapping is used. In this particular case this results in an `xs:int` instance being bound to `$i`.

Example 38 is similar to Example 37, but rather than assuming the default Java to XDM instance mapping, the type[1] of the XDM instance is explicitly specified, as such an `xs:integer` instance is bound to the external variable.

```
...
XQExpression expr = conn.createExpression();

// the XQuery expression to be executed
String es = "declare variable $i as xs:integer external; " +
            "$i + 1 ";

// bind a value to the external variable i
expr.bindInt(new QName("i"), 21,
             conn.createAtomicType(XQItemType.XQBASETYPE_INTEGER));

// execute the XQuery expression
XQResultSequence result = expr.executeQuery(es);
...
```

Example 38 – Bind a value, and explicitly specify the type of the XDM instance to be created.

In XQJ, the context item can be bound like any other variable, using the predefined constant `XQDynamicContext.CONTEXT_ITEM` as variable name.

```
...
XQExpression expr = conn.createExpression();

// the XQuery expression to be executed
String es = ". + 1 ";

// bind a value to the external variable i
expr.bindInt(XQDynamicContext.CONTEXT_ITEM, 21, (XQType)null);

// execute the XQuery expression
XQResultSequence result = expr.executeQuery(es);
...
```

Example 39 – Bind a value to the context item.

10.2 XQPreparedExpression

An `XQPreparedExpression` object is created from an `XQConnection` object by supplying the XQuery expression that needs to be evaluated, using the `prepareExpression` method.

```
String es = "declare variable $x as xs:integer external;" +
            " for $n in fn:doc('catalog.xml')//item" +
            " where $n/price <= $x" +
            " return fn:data($n/name)";

XQPreparedExpression expr = conn.prepareExpression(es);
```

Example 40 – Create an XQPreparedExpression.

A variant of this `prepareExpression` method allows to specify an `XQStaticContext` object as shown in Example 35.

[1] More on XQuery types in XQJ can be found in section 13 “Representing the XQuery Type System”.

The XQuery expression is immutable for the lifetime of the `XQPreparedExpression` object. This step is equivalent to the static analysis phase defined by [XQuery]. The actual evaluation (corresponding to the dynamic evaluation phase defined by [XQuery]) of the `XQPreparedExpression` is initiated when the `executeQuery` method on the `XQPreparedExpression` object is invoked. An `XQPreparedExpression` object can be executed multiple times without the need to statically re-analyze the expression again. The actual evaluation of the XQuery expression is governed by the rules defined by [XQuery]. Before the `XQPreparedExpression` object is evaluated, an application can bind values to external variables using several `bindXXX` methods. These external variables need to be declared in the XQuery prolog for the `bindXXX` methods to be successful. If an application tries to bind a value to a variable that was not declared in the prolog, then an exception is thrown.

The `executeQuery` method returns the result of the expression evaluation as an `XQResultSequence` object, whose holdability and scrollability characteristics are determined by the corresponding settings in the `XQStaticContext` object linked to the `XQPreparedExpression` object when it was created.

The `XQPreparedExpression` object is valid until the `close` method is explicitly called on this object, or until it is implicitly closed. It is implicitly closed when the `XQConnection` object that brought this `XQPreparedExpression` object into existence is closed (*i.e.*, the lifetime of the `XQPreparedExpression` object cannot be longer than the lifetime of the connection that brought it into existence). In all cases, all associated resources are freed.

Besides the aforementioned functionality, the `XQPreparedExpression` interface provides methods:

- to cancel the execution of a given prepared expression,
- to close an `XQPreparedExpression` object and release all resources associated with this prepared expression,
- to test whether an `XQPreparedExpression` object is closed,
- to get the `XQStaticContext` used to create the `XQPreparedExpression`,
- to retrieve the static type information of an external variable with a given QName, and
- to retrieve the static type information of the result sequence.

10.3 XQExpression

An `XQExpression` object is created from an `XQConnection` object. The `XQExpression` interface combines the static analysis phase and the dynamic evaluation phase defined by [XQuery]) in a single step. This step is initiated when the `executeQuery` method is invoked. This method takes an XQuery expression as input and returns an `XQResultSequence` object, whose holdability and scrollability characteristics are determined by the corresponding settings in the `XQStaticContext` object linked to the `XQExpression` object when it was created.

Example 41 shows the creation of an `XQExpression` object, and how to subsequently execute an XQuery expression using this `XQExpression`.

```
...
XQExpression expr = conn.createExpression();

XQResultSequence result = expr.executeQuery(
  "fn:doc('catalog.xml')//item");
...
```

Example 41 – Create an XQExpression and execute it.

Before the `XQExpression` object is evaluated, an application can bind values to external variables using the `bindXXX` methods. These external variables need to be declared in the XQuery prolog in order to be referenced and used in the XQuery expression. The actual evaluation of the XQuery expression is governed by the rules defined by [XQuery]. Once an application has processed the results of the execution of an XQuery expression, the `XQExpression` object that was used for that XQuery expression can be reused for the execution of a different XQuery expression.

An `XQExpression` object can also be used to execute a *statement* that is not an XQuery expression as defined by [XQuery]. The `XQExpression` interface provides the `executeCommand` method for this purpose. This method takes the statement to be executed as parameter. The syntax for such a statement and semantics of executing it are implementation-defined. If an error occurs during the execution of the statement, then an `XQException` is thrown.

The `XQExpression` object is valid until the `close` method is explicitly called on this object, or until it is implicitly closed. It is implicitly closed when the `XQConnection` object that brought this `XQExpression` object into existence is closed (*i.e.*, the lifetime of the `XQExpression` object cannot be longer than the lifetime of the connection that brought it into existence). In all cases, all associated resources are freed.

Besides the aforementioned functionality, the `XQExpression` interface provides methods:

- to cancel the execution of a given expression,
- to close the `XQExpression` and release all resources associated with this object,
- to test whether an `XQExpression` object is already closed,
- to get the `XQStaticContext` used to create the `XQExpression`.

10.4 Binding mode

The default binding mode for an `XQExpression` or `XQPreparedExpression` is *immediate*. In other words, the external variable value specified by the application is consumed during the execution of the `bindXXX` method. And the binding stays valid for multiple executions, as shown in Example 42.

```
...
XQExpression expr = conn.createExpression();
XQResultSequence result;
```

```
// the XQuery expression to be executed
String es = "declare variable $i as xs:integer external; " +
            "$i + 1 ";

// bind a value to the external variable i
expr.bindInt(new QName("i"), 21, null);

// execute the XQuery expression a first time
// integer 21 is bound to $i
result = expr.executeQuery(es);

// execute the XQuery expression a second time
// integer 21 is still bound to $i
result = expr.executeQuery(es);

...

// bind a different value to the external variable i
expr.bindInt(new QName("i"), 31, null);

// execute the XQuery expression a third time
// integer 21 is bound to $i
result = expr.executeQuery(es);
```

Example 42 – Bind a value to an external variable and execute multiple times.

An application has the ability to set the binding mode to *deferred*. With deferred binding mode, the application gives a hint to the XQJ-implementation and underlying XQuery processor to take advantage of any query streaming capabilities. The actual, performance and scalability improvements of deferred binding mode are implementation-dependent. The cost for the application writer is a slightly more complex programming model compared to the default immediate binding mode.

In deferred binding mode, bindings are only active for a single execution cycle. The application is required to explicitly re-bind values to every external variable before each execution. Failing to do so will result in an `XQException`, as the implementation will assume during the next execution that none of the external variables is bound. Going back to Example 42, in deferred mode it would fail during the second execution.

As explained, the default binding mode is immediate, Example 43 shows how to set it to deferred. After having changed the default binding mode to deferred, all subsequent created `XQExpression` and `XQPreparedExpression` objects will operate in deferred mode.

```
...
// get a static context object with the default values
XQStaticContext cntxt = conn.getStaticContext();

// change the binding mode
cntxt.setBindingMode(XQConstants.BINDING_MODE_DEFERRED);

// make the changes effective
conn.setStaticContext(cntxt);
```

Example 43 – Change the default binding mode to deferred.

As explained in 9.4 “Setting the static context for a specific expression”, the application can also change `XQStaticContext` properties for specific `XQExpression` and `XQPreparedExpression` objects, and as such switch binding mode more selectively.

In deferred mode the application cannot assume that the bound value will be consumed during the invocation of the `bindXXX` method. In such scenarios, the order in which the bindings are evaluated is implementation-dependent, and an implementation does not necessarily need to consume a binding if it can evaluate the query without requiring the external variable. The XQJ-implementation is also free to read the bound value either at bind time or during the subsequent evaluation and processing of the query results. In addition, in deferred binding mode, any error condition specified to throw an exception during the `bindXXX` methods, may be thrown later during the evaluation of the query itself.

As a consequence, in deferred mode, the application must guarantee to extend the lifetime of the bound values, for example `java.io.InputStream` objects, bound to external variables can only be closed after completing the query processing, which includes consuming the query results. Example 44 demonstrates this.

```
...
// assume an XQConnection c
// and set the binding mode to deferred
c.setBindingMode(XQConstants.BINDING_MODE_DEFERRED);

// prepare an expression with external variable $schema
XQPreparedExpression p = c.prepareExpression(
    "declare variable $schema external; $schema//xs:annotation");

// get the input stream
URL schema = new URL("http://www.w3.org/2001/XMLSchema.xsd");
InputStream input = schema.openStream()

// might be that the complete stream is read during bind time.
// but this is not guaranteed ...
p.bindDocument(new QName("schema"),input, null, null);

// execute the query
XQSequence s = p.execute();

// consume the results
while (s.next())
  s.writeItem(System.out);
s.close();

// ... a streaming processor might have only finished reading the
// input stream now that all results have been read.

// only now we can close the input stream, doing so earlier
// could result in a failure
input.close();
```

Example 44 – Extended lifetime of bound values with deferred binding mode.

Remember, in deferred mode an implementation does not necessarily need to consume a binding if it can evaluate the query without requiring the external variable. In Example 45, we execute two queries, and bind those results to external variables in a third query, which conditionally evaluates one or the other variable.

```
...
// assume an XQConnection c
// and set the binding mode to deferred
c.setBindingMode(XQConstants.BINDING_MODE_DEFERRED);

XQExpression e1 = c.createExpression();
XQSequence s1 = e1.execute("1,2,3,4,5");

XQExpression e2 = c.createExpression();
XQSequence s2 = e2.execute("6,7,8,9,10");

XQPreparedExpression p = c.prepareExpression(
    "declare variable $v1 external; " +
    "declare variable $v2 external; " +
    " if (fn:current-time() < xs:time('12:00:00')) then $v1 else $v2");

p.bindSequence(new QName("v1"), s1);
p.bindSequence(new QName("v2"), s2);
XQSequence s = p.execute():
while (s.next())
  System.out.println(s.getInt());
s.close();
...
```

Example 45 – Variable bindings are not always consumed in deferred binding mode.

In the previous example, Example 45, either `s1`, `s2` or both the `XQSequence` objects are consumed. It depends on the implementation details of the XQJ-implementation and on the current time.
In any case, all the application must do subsequently is close both `XQSequence` object to make sure all resources are freed.

```
...
s1.close();
s2.close();
```

Example 46 – Continue, variable bindings are not always consumed in deferred binding mode.

10.5 Canceling the Execution of an Expression

An application can invoke the `cancel` method to abort the execution on an `XQExpression` or `XQPreparedExpression` object. `cancel` attempts to cancel the execution if both the XQJ driver and data source support aborting the execution. Typically `cancel` is invoked from one thread to abort the execution on another thread.

It is implementation-defined whether the invocation of this method has any effect. But if an XQJ driver does not support this functionality, then this method is effectively a no-op. Even if an XQJ driver supports this functionality, this method might still be a no-op if the `XQExpression` or `XQPreparedExpression` is not currently executing any XQuery expression.

If the cancellation of the execution of the `XQExpression` or `XQPreparedExpression` in another thread is successful, then an `XQCancelledException` is thrown in that other thread during the invocation of the `executeQuery` method, or any method accessing the `XQResultSequence` returned from `executeQuery`, or any method accessing an `XQResultItem` contained in that `XQResultSequence`. It is implementation-dependent, which invocation of which such method causes the `XQCancelledException` to be thrown (this includes the case where an `XQCancelledException` is never thrown). However, if the `XQCancelledException` is thrown, then any `XQResultSequence` is implicitly closed (closing the `XQResultSequence` implicitly closes any `XQResultItem` contained in this `XQResultSequence` as well).

Assuming that the cancellation of the execution of an `XQExpression` is supported by both, the XQJ driver and the data source, the following example shows how to cancel a currently executing `XQExpression`. The `CancelActionThread` and `QueryActionThread` classes are used to show how the `cancel` method can be used to abort the execution of an XQuery expression in a GUI-style XQuery tool. When users press the "run" button to run an XQuery, the GUI tool creates an `XQExpression` object and a `QueryActionThread` object to execute an XQuery. When users press the "cancel" button, the GUI tool creates and starts a `CancelActionThread` and passes in the currently being executed `XQExpression` to cancel the currently running query.

```
// CancelActionThread is used to cancel a query being
public class CancelActionThread extends Thread
{
      XQExpression m_expr;
      public CancelActionThread(XQExpression expr)
       {
        m_expr = expr;
       }
      public void run()
       {
        try {
            // cancel the query being executed in another thread
            m_expr.cancel();
       } catch ( XQException e) {
          System.out.println("Exception in cancel");
          e.printStackTrace();
         }
       }
}

// run a query and check if the query execution is cancelled or not.
public class QueryActionThread extends Thread
{
      XQExpression m_expr;
      String m_query;
      public QueryActionThread(XQExpression expr, String query)
       {
        m_expr = expr;
        m_query  = query;
```

```
      }
     // run a query and check whether the query execution was cancelled.
      public void run()
       {
        try {
            XQSequence result = m_expr.executeQuery(m_query);
            while ( result.next() ) {
                                    // processing query result
                                    }
            } catch ( XQCancelledException e) {
                  System.out.println("query execution is cancelled");
             }
            catch ( XQException e) {
                  System.out.println(e.getMessage());
            }

       }
}
```

Example 47 – Canceling the execution of an XQExpression.

11 Items and Sequences

XQuery operates on the abstract, logical structure of an XML document, rather than its surface syntax. This logical structure is known as the XQuery 1.0 and XPath 2.0 Data Model (XDM), which is defined in [XQuery DM], briefly summarized as follows:

- Every value in the data model is a *sequence*.
- A sequence is an ordered collection of zero or more *item*s.
- An item may be an *atomic value* or a *node*.
- An atomic value is a value in the value space of an *atomic type* labeled with that atomic type.
- An atomic type is a primitive type as defined by [XML Schema Part 2], `xs:untypedAtomic`, `xs:anyAtomicType`, or a type derived by restriction from another atomic type. Types derived by list or union are not atomic.
- Every node is one of the following six kinds: document, element, attribute, text, processing instruction, and comment (note that XQuery doesn't have a namespace axis and therefore namespace nodes are not supported through XQJ and thus omitted from this list).
- When a node is added to a sequence its identity remains the same. Consequently a node may occur in more than one sequence and a sequence may contain duplicate items.
- An important characteristic of the data model is that there is no distinction between an item (a node or an atomic value) and a singleton sequence containing that item. An item is equivalent to a singleton sequence containing that item and vice versa.
- Sequences never contain other sequences; if sequences are combined, the result is always a "flattened" sequence.

XQJ provides several interfaces to mirror XQuery's data model concepts of item and sequence. The most important ones are `XQItem` and `XQSequence`.

11.1.1 XQItem

The `XQItem` interface represents the item in the XQuery data model. An `XQItem` is an immutable object; *i.e.*, once it is created its internal state cannot be changed.

An application gets hold of an `XQItem` object either by invoking the `getItem` method of an `XQSequence` object or by invoking the `createItem` or one of the `createItemFromXXX` methods of `XQDataFactory`.

An `XQItem` object is valid until either the close method is explicitly called on this object, or, if any, the `XQSequence` object that brought this item into existence is closed. In either case, all associated resources are freed. Using the explicit `close` method is advisable to release the resources as soon as possible.

The `XQItem` interface provides methods:

- to retrieve the type of the item,
- to check the type of an item against a given type,
- to retrieve the value of the item, if the item is an atomic value, and

- to retrieve the node, if the item is a node.

11.1.2 XQResultItem

An application gets hold of an `XQResultItem` object by invoking the `getItem` method of an `XQResultSequence` object. An `XQResultItem` object represents an immutable item, which is valid until either the `close` method is explicitly called on this object, or the `XQResultSequence` object that brought this item into existence is explicitly or implicitly closed. In either case, all associated resources are freed.

11.1.3 XQSequence

The `XQSequence` interface represents a sequence in the XQuery data model. Similar to the XQuery data model, an `XQSequence` object contains zero or more `XQItem` objects. An `XQSequence` object has a *current position* that points to the *current item*, unless the current position is before the first item or after the last item in the sequence. An application can work with the current item of the sequence directly using the same methods that are available for an `XQItem` object. There is no need to clone or extract the item out of the sequence to work with the item.

An `XQSequence` object is either scrollable (*i.e.*, any item can be accessed independently of its location with respect to the current item) or forward-only (*i.e.*, an item that is previous to the current item in the sequence cannot be accessed anymore). Some methods are only applicable for scrollable sequences and will throw an exception if invoked on a forward-only sequence.

An application gets hold of an `XQSequence` object either by executing a query or by invoking any of the `createSequence` method of an `XQDataFactory` object.

An `XQSequence` as result of a query execution is further discussed in section 11.1.4 “XQResultSequence”.

After an `XQSequence` object is created from an `XQDataFactory` object, it is independent of that `XQDataFactory`, respectively. The object is valid until the `close` method is explicitly called on this object, closing the resource also guarantees that all associated resources are freed.

Chapter 15 “Data Factories” describes how an `XQDataFactory` is used to create `XQSequence` objects.

The `XQSequence` interface provides methods:

- to move the current position to
  - before the first item (if the sequence is scrollable),
  - the first item (if the sequence is scrollable),
  - the last item (if the sequence is scrollable),
  - after the last item (if the sequence is scrollable),
  - an absolute position counting from before the first item (if the sequence is scrollable),
  - a relative position counting from the current position (if the sequence is scrollable),

    - the next item,
    - the previous item (if the sequence is scrollable)
- to test whether the current position is
    - before the first item (if the sequence is scrollable),
    - the first item (if the sequence is scrollable),
    - the last item (if the sequence is scrollable),
    - after the last item (if the sequence is scrollable)
- to test whether the `XQSequence` object is scrollable,
- to return the number of items in the sequence (if the sequence is scrollable),
- to return the current position (if the sequence is scrollable),
- to close an `XQSequence` object (*i.e.*, free all resources associated with the `XQSequence` object and invalidate all future references to the `XQSequence` object), and
- to test whether an `XQSequence` object is closed.

11.1.4 XQResultSequence

An application gets hold of an `XQResultSequence` object by invoking the `executeQuery` method of either an `XQExpression` or an `XQPreparedExpression` object. An `XQResultSequence` object represents the result of such an execution. The object is valid until the `close` method is explicitly called on this object, or until it is implicitly closed. It is implicitly closed when the `XQExpression` or `XQPreparedExpression` object that brought this `XQResultSequence` object into existence is implicitly or explicitly closed or re-executed (*i.e.*, the lifetime of the `XQResultSequence` object cannot be longer than the lifetime of the expression that brought it into existence). In all cases, all associated resources are freed. Since an `XQResultSequence` object may potentially represent a large XQuery result, it is recommended to call the `close` method as soon as the result is no longer needed to free the resources.

Besides the methods inherited from the `XQSequence` interface, the `XQResultSequence` interface provides methods:

- to retrieve a reference to the `XQConnection` object, on which this `XQResultSequence` object depends.

12 Serialization

The `XQSequence` and `XQItem` interfaces provide several methods that allow the sequence or item to be serialized as, for example, a character string. This serialization is done according to [XQuery Serialization]. [XQuery Serialization] specifies a number of parameters that influence how the serialization is performed. The parameters can be specified by an application as follows:

- The serialization parameters are specified in a `java.util.Properties` object, where the property key specifies the serialization parameter name and the property value represents the serialization parameter value.
- Specifying null for the serialization properties, is equivalent to specifying an empty `java.util.Properties` object.
- The serialization parameter names are case sensitive. In order to be recognized by an XQJ-implementation, a parameter name must match literally the parameter name defined in [XQuery Serialization].
- An unknown serialization parameter is silently ignored.
- If the serialization parameter is not applicable for a given output method, it is silently ignored.
- Parameter values must conform to the rules outlined in [XQuery Serialization]. For parameters where the specification enumerates permitted values, the specified value must match literally one of the enumerated values.
- If the parameter is not ignored, an invalid value according to [XQuery Serialization] for a given parameter must result in an `XQException`. Note that an XQJ implementation is not required to support all values for every serialization parameter. Any restrictions on the supported values are implementation-defined.
- If the parameter value is an expanded QName, it must be formatted using the representation defined by [Clark]:
  "{" + Namespace URI + "}" + local part
  In case of an empty Namespace URI, only the local part is used.
- If the parameter is a list of expanded QNames, it is formatted as a semi-colon separated list of expanded QNames.
- The default value for the method parameter is xml. An implementation may support optionally additional output methods.
- The default value for the encoding parameter must be UTF-8 or UTF-16. All implementations are required to support both UTF-8 and UTF-16.
- The default value for any other parameter than xml or encoding, is implementation-defined.
- XQJ methods serializing into a `java.lang.String` or `java.io.Writer`, must support the encoding parameter as outlined in [XQuery Serialization]. Each character that cannot be represented in the specified encoding must be represented as an XML character entity reference.
  For example assuming that the value for the encoding parameter is US-ASCII, the copyright character, ©, which is not in the US-ASCII character set, must be escaped and serialized as a sequence of the following 6 Java characters, ©

If the application wants to serialize into a sequence of octets using the specified encoding, the application must serialize to a `java.io.OutputStream`.

- As specified in [XQuery], an XQJ-implementation must support the combination of method=xml and version=1.0.

12.1 Serializing an XDM instance into a StAX event stream (XMLStreamReader)

Serializing an XDM instance (*i.e.*, a sequence of zero or more items) into an `XMLStreamReader` event stream is conceptually implemented through the following steps:

1) The XDM instance is normalized as described in [XQuery Serialization]. Note that the normalization process can fail, in which case an `XQException` is thrown.

2) The resulting XDM instance is transformed into an `XMLStreamReader` event stream. The following describes, which events are generated (*i.e.*, the event type returned by the `XMLStreamReader.getEventType` method). The *dm:xxx* accessors referenced below are defined in [XQuery DM].

   a) Document Node:

      i) `XMLStreamReaderConstants.START_DOCUMENT`

      ii) Process the children as returned by *dm:children* in order.

      iii) `XMLStreamReaderConstants.END_DOCUMENT`

   b) Element Node:

      i) `XMLStreamReaderConstants.START_ELEMENT`

         (1) All namespace nodes returned by *dm:namespace-nodes* are reported as part of the START_ELEMENT event.

         (2) All attribute nodes returned by *dm:attributes* are reported as part of the START_ELEMENT event.

      ii) Process the children as returned by *dm:children* in order.

      iii) `XMLStreamReaderConstants.END_ELEMENT`

         (1) All namespace nodes returned by *dm:namespace-nodes* are reported as part of the END_ELEMENT event.

   c) Processing Instruction Node:

      i) `XMLStreamReaderConstants.PROCESSING_INSTRUCTION`

   d) Comment Node:

      i) `XMLStreamReaderConstants.COMMENT`

   e) Text Node:

      i) `XMLStreamReaderConstants.CHARACTERS`

Note that the value of dm:string-value can be reported as consecutive CHARACTERS events.

f) It is implementation-defined whether any other event types are reported through the event stream. For example, an implementation may report a CDATA or SPACE event instead of a CHARACTERS event where appropriate.

12.2 Serializing an XDM instance into a SAX event stream

Serializing an XDM instance (*i.e.*, a sequence of zero or more items) into a SAX event stream is conceptually implemented through the following steps:

1) The XDM instance is normalized as described in [XQuery Serialization]. Note that the normalization process can fail, in which case an `XQException` is thrown.

2) The resulting XDM instance is transformed into SAX events. The following describes, which `org.xml.sax.ContentHandler` and `org.xml.sax.LexicalHandler` methods are called. The *dm:xxx* accessors referenced below are defined in [XQuery DM].

   a) Document Node:

      i) `ContentHandler.startDocument`

      ii) Process the children as returned by *dm:children* in order.

      iii) `ContentHandler.endDocument`

   b) Element Node:

      i) For each namespace node returned by *dm:namespace-nodes*: `ContentHandler.startPrefixMapping`

      ii) `ContentHandler.startElement`
      The attributes parameter specifies all attributes as returned by *dm:attributes*

      iii) Process the children as returned by *dm:children* in order.

      iv) `ContentHandler.endElement`

      v) For each namespace node returned by *dm:namespace-nodes*: `ContentHandler.endPrefixMapping`

   c) Processing Instruction Node:

      i) `ContentHandler.processingInstruction`

   d) Comment Node:

      i) `LexicalHandler.comment`

   e) Text Node:

      i) One or more `ContentHandler.characters`

      Note that all `characters` events together must represent the value returned by dm:string-value.

f) It is implementation-defined whether any additional `ContentHandler` or `LexicalHandler` methods besides the ones referenced above are called. For example, an implementation may report CDATA sections (`LexicalHandler.startCDATA` and `LexicalHandler.endCDATA`) or ignorable whitespaces (`ContentHandler.ignorableWhitespace`).

13 Representing the XQuery Type System

13.1 XQSequenceType

The `XQSequenceType` interface represents a sequence type as defined in [XQuery]. The interface provides three methods that allow an application to retrieve:

- the string representation of the sequence type,
- the occurrence indicator (exactly one item, zero or one item, zero or more items, one or more items), and
- the item type (an `XQItemType`).

The occurrence indicator is one of the following

- `XQSequenceType.OCC_ZERO_OR_ONE`
  denote an empty or singleton sequence.
- `XQSequenceType.OCC_EXACTLY_ONE`
  denote a singleton sequence
- `XQSequenceType.OCC_ZERO_OR_MORE`
  denote a sequence with zero or more items
- `XQSequenceType.OCC_ONE_OR_MORE`
  denote a sequence with one or more items
- `XQSequenceType.OCC_EMPTY`
  denote an empty sequence

In case of an empty sequence (`OCC_EMPTY`), the item type of the `XQSequence` object is `null`.

13.2 XQItemType

The `XQItemType` interface represents an item type as defined in [XQuery]. `XQItemType` extends `XQSequenceType` but restricts the occurrence indicator to be exactly one. This derivation allows passing an `XQItemType` object wherever an `XQSequenceType` object is expected, but not the other way. The `XQItemType` interface contains methods to represent information about the following aspects of an XQuery item type:

- the kind of the item (one of the `XQITEMKIND_*` constants, see below),
- for atomic types the closest matching built-in [XML Schema] type, for element and attributes the closest matching built-in [XML Schema] type this node is based on (one of the `XQBASETYPE_*` constants, see below),
- the name of the node, if any,
- the processing-instruction name, if any,
- the QName representing the type name, if any. If present, then also whether the type is an anonymous type,

- the XML Schema URI associated with the type, if any,
- the nillability characteristics, if any, and
- whether the item type represents an XML Schema element.

Node Name: Represents the name of the node. This can be used to represent a type such as element ("foo"). This attribute is only defined for XQITEMKIND_DOCUMENT_ELEMENT, XQITEMKIND_DOCUMENT_SCHEMA_ELEMENT, XQITEMKIND_ELEMENT, XQITEMKIND_SCHEMA_ELEMENT, XQITEMKIND_ATTRIBUTE or XQITEMKIND_SCHEMA_ATTRIBUTE. A null value indicates a wildcard entry. E.g. element (*).

Processing-instruction name: Represents the name of a processing-instruction. This can be used to represent a type such as processing-instruction("foo"). This is only valid if the Item Kind is XQITEMKIND_PI. A null value indicates a wildcard entry. E.g. processing-instruction().

Schema Type Name: Represents a type name (global or local). This can be used to represent a specific type name such as, element foo of type hatsize. A null value indicates a wildcard entry. The schema type name is represented as a single QName.

Schema URI: Represents an URI to the XML Schema document that contains the element or type.

Schema Element: (Boolean property) Represents whether the item type represented is a schema element or not. This is only relevant for static types.

Anonymous Type (Boolean property) Represents whether the item type is an anonymous type in the schema. This property is only valid if the schema type name is specified.

Nillability: Represents the nillability of the element. Valid only if the item kind is XQITEMKIND_DOCUMENT_ELEMENT or XQITEMKIND_ELEMENT.

13.2.1 XQITEMKIND_* Constants

XQJ defines the following constants which specifies the kind of an `XQItemType` object

XQITEMKIND_* constants	**Description**
`XQITEMKIND_ITEM`	Represents any kind of item, i.e. the abstract type `item()` as defined in [XQuery]
`XQITEMKIND_ATOMIC`	Represents an atomic type, i.e. xs:anyAtomicType or one of the types derived from it. The base type property of the `XQItemType` object specifies the exact atomic type being represented
`XQITEMKIND_NODE`	Represents any kind of node, i.e. the abstract type `node()` as defined in [XQuery]

XQITEMKIND_* constants	**Description**
XQITEMKIND_DOCUMENT	Represents a document node, i.e. the type `document-node()` as defined in [XQuery]
XQITEMKIND_DOCUMENT_ELEMENT	Represents a document node that contains exactly one element node, optionally accompanied by one or more comment and processing instruction nodes. I.e. the type `document-node(element(…))` as defined in [XQuery]
XQITEMKIND_DOCUMENT_SCHEMA_ELEMENT	Represents a document node that contains exactly one element node for a particular schema, optionally accompanied by one or more comment and processing instruction nodes. I.e. the type `document-node(schema-element(…))` as defined in [XQuery]
XQITEMKIND_COMMENT	Represents a comment node, i.e. the type `comment()` as defined in [XQuery]
XQITEMKIND_ELEMENT	Represents an element node, i.e. the type `element(…)` as defined in [XQuery]
XQITEMKIND_SCHEMA_ELEMENT	Represents an element node whose type annotation matches a schema type, i.e. the type `schema-element(…)` as defined in [XQuery]
XQITEMKIND_ATTRIBUTE	Represents an attribute node, i.e. the type `attribute(…)` as defined in [XQuery]
XQITEMKIND_SCHEMA_ATTRIBUTE	Represents an attribute node whose type annotation matches a schema type, i.e. the type `schema-attribute(…)` as defined in [XQuery]
XQITEMKIND_PI	Represents a processing instruction node, i.e. the type `processing-instruction(…)` as defined in [XQuery]
XQITEMKIND_TEXT	Represents a text node, i.e. the type

XQITEMKIND_* constants	Description
	`text()` as defined in [XQuery]

Table 2 – XQITEMKIND _* constants

13.2.2 XQBASETYPE_* Constants

The following table lists all defined `XQBASETYPE_*` constants. XQJ defines a constant for each of the built-in schema types defined in [XML Schema] and [XQuery].

XQBASETYPE_* constants	Schema Type
XQBASETYPE_UNTYPED	xs:untyped [XQuery]
XQBASETYPE_ANYTYPE	xs:anyType [XML Schema]
XQBASETYPE_ANYSIMPLETYPE	xs:anySimpleType [XML Schema]
XQBASETYPE_ANYATOMICTYPE	xs:anyAtomicType [XQuery]
XQBASETYPE_UNTYPEDATOMIC	xs:untypedAtomic [XQuery]
XQBASETYPE_DAYTIMEDURATION	xs:dayTimeDuration [XQuery]
XQBASETYPE_YEARMONTHDURATION	xs:yearMonthDuration [XQuery]
XQBASETYPE_ANYURI	xs:anyURI [XML Schema]
XQBASETYPE_BASE64BINARY	xs:base64Binary [XML Schema]
XQBASETYPE_BOOLEAN	xs:boolean [XML Schema]
XQBASETYPE_DATE	xs:date [XML Schema]
XQBASETYPE_INT	xs:int [XML Schema]
XQBASETYPE_INTEGER	xs:integer [XML Schema]
XQBASETYPE_SHORT	xs:short [XML Schema]
XQBASETYPE_LONG	xs:long [XML Schema]
XQBASETYPE_DATETIME	xs:dateTime [XML Schema]
XQBASETYPE_DECIMAL	xs:decimal [XML Schema]
XQBASETYPE_DOUBLE	xs:double [XML Schema]
XQBASETYPE_DURATION	xs:duration [XML Schema]
XQBASETYPE_FLOAT	xs:float [XML Schema]
XQBASETYPE_GDAY	xs:gDay [XML Schema]
XQBASETYPE_GMONTH	xs:gMonth [XML Schema]
XQBASETYPE_GMONTHDAY	xs:gMonthDay [XML Schema]
XQBASETYPE_GYEAR	xs:gYear [XML Schema]
XQBASETYPE_GYEARMONTH	xs:gYearMonth [XML Schema]
XQBASETYPE_HEXBINARY	xs:hexBinary [XML Schema]
XQBASETYPE_NOTATION	xs:NOTATION [XML Schema]
XQBASETYPE_QNAME	xs:QName [XML Schema]
XQBASETYPE_STRING	xs:string [XML Schema]
XQBASETYPE_TIME	xs:time [XML Schema]
XQBASETYPE_BYTE	xs:byte [XML Schema]
XQBASETYPE_NONPOSITIVE_INTEGER	xs:nonPositiveInteger [XML Schema]
XQBASETYPE_NONNEGATIVE_INTEGER	xs:nonNegativeInteger [XML Schema]
XQBASETYPE_NEGATIVE_INTEGER	xs:negativeInteger [XML Schema]
XQBASETYPE_POSITIVE_INTEGER	xs:positiveInteger [XML Schema]
XQBASETYPE_UNSIGNED_LONG	xs:unsignedLong [XML Schema]

XQBASETYPE_* constants	Schema Type
XQBASETYPE_UNSIGNED_INT	xs:unsignedInt [XML Schema]
XQBASETYPE_UNSIGNED_SHORT	xs:unsignedShort [XML Schema]
XQBASETYPE_UNSIGNED_BYTE	xs:unsignedByte [XML Schema]
XQBASETYPE_NORMALIZED_STRING	xs:normalizedString [XML Schema]
XQBASETYPE_TOKEN	xs:token [XML Schema]
XQBASETYPE_LANGUAGE	xs:language [XML Schema]
XQBASETYPE_NAME	xs:Name [XML Schema]
XQBASETYPE_NCNAME	xs:NCName [XML Schema]
XQBASETYPE_NMTOKEN	xs:NMTOKEN [XML Schema]
XQBASETYPE_IDREF	xs:IDREF [XML Schema]
XQBASETYPE_ENTITY	xs:ENTITY [XML Schema]
XQBASETYPE_IDREFS	xs:IDREFS [XML Schema]
XQBASETYPE_NMTOKENS	xs:NMTOKENS [XML Schema]
XQBASETYPE_ENTITIES	xs:ENTITIES [XML Schema]

Table 3 – XQBASETYPE_* constants

13.2.3 Retrieving static type information

A static type may be obtained from the `getStaticResultType` or the `getStaticVariableType` methods on the `PreparedExpression` object.

In both cases, an implementation is allowed to give a generic type that is correct but not the most specific. For example, consider a user-defined primitive type foo:hatsize defined in the namespace http://www.foo.com and described by a schema "http://www.fooschema.com". If the query expression is "hatsize(21)", then an implementation may return the static result type in one of many forms.

For example, an implementation that does not perform static type checking may return `XQITEMKIND_ITEM` denoting that the result is some item. An implementation may also perform partial typing and return it as `XQITEMKIND_ATOMIC` with `XQBASETYPE_ANYATOMICTYPE`. Alternatively, an implementation may perform full static typing and return the type as `XQITEMKIND_ATOMIC` with `XQBASETYPE_INTEGER`. Further, it may also include the schema URI (http://www.fooschema.com) and the name of the type "foo:hatsize (namespace http://www.foo.com)". All three types shown above are correct but the last form is the most specific. Note that there may be many more possible valid type representations.

In addition, when the inferred expression type is a complex formal semantics type [XQuery FS], it is up to the XQuery engine to create the sequence type that best represents it. For instance the formal semantics type might infer a result of `(xs:integer | xs:date)*`, but `item()*` or `xs:anyAtomicType*` could be returned via the XQJ API.

13.2.4 Retrieving dynamic type information

The type of an item can be retrieved using the `getItemType` method, defined in the `XQItemAccessor` interface, and as a consequence available both on `XQSequence` as well as `XQItem`.

In contrast with the static type, which might not be precise, the dynamic type returned by the `getItemType` method must always be exact. As such, `getItemType` will never return an `XQItemType` representing an abstract XQuery item type line `item()`, `node()` or for example `xs:anyAtomicType`.

13.2.5 User-defined XML Schema types

The XQuery type system is based on XML Schema. This implies that an XQJ-implementation needs to be able to support the same XML Schema as that of the XQuery engine it communicates with in order for the XQJ API users to fully understand and interpret the XQuery result. However, this may impose significant overhead for an XQJ-implementation depending on how tightly it is integrated with the XQuery engine.

On one end of the spectrum where XQJ client and XQuery engine can be tightly integrated and co-located, for example, in the case that XQJ and the XQuery engine are fully embedded in an application, the XML schema is controlled and shared by both the XQJ-implementation and XQuery engine. In this case, the XQJ-implementation not only knows the name and schema URL of a user-defined schema type but also knows the structure information of the user-defined types based on the full knowledge of the underlying XML Schema, which defines that user-defined type.

On the other end of the spectrum, the XQJ client and the XQuery engine could be completely decoupled and they do not share any central XML schema repository. The XQJ client only knows the name and the schema URL of the user defined schema type but has no knowledge of its structure. There could also be loosely coupled client and server system where either client and server share the same XML schema repository remotely or client queries back to the server for XML schema information.

To accommodate variety of XQJ-implementation strategies and use cases for all of them, the design of XQJ API is flexible and portable. To find out whether an XQJ-implementation, in general, knows more than the name and the schema URI of a user-defined type, the application can make use of the `XQMetaData.IsUserDefinedXMLSchemaTypeSupported` method. If this method returns true, then the XQJ-implementation is assumed to understand all the XML schema that the XQuery engine uses and must be able to answer questions on user defined schema type based on the knowledge of the XML schema that define the user defined type. If this meta-data is FALSE, then it is implementation-dependent if the XQJ-implementation can answer questions on user defined schema type.

In this way, XQJ API users can write portable applications based on the query on that metadata. However, unlike user-defined types, for XML Schema built-in types, such as `xs:integer`, `xs:time` and new XQuery built-in types, such as `xs:untypedAtomic`, all XQJ-implementations are required to understand them.

Also refer to 19 “Interoperability” on possible behavioral restrictions with user-defined XML Schema types.

13.2.6 Additional Examples

The following examples illustrate additional uses of the `XQItemType`.

```
...
XQConnection conn = datasrc.getConnection();
XQPreparedExpression expr = conn.prepareExpression("xs:integer(1)");
XQSequence res = expr.executeQuery();
res.next();
XQItemType itype = res.getItemType();
// itype MUST be an item type with kind XQITEMKIND_ATOMIC
int basetype = itype.getBaseType();
// The base type MUST be XQBASETYPE_INTEGER
...
```

Example 48 - Result is an atomic type.

```
...
XQConnection conn = datasrc.getConnection();
XQPreparedExpression expr = conn.prepareExpression("<A>33</A>");
XQSequence res = expr.executeQuery();
res.next();
XQItemType itype = res.getItemType();
// itype MUST be an item type with kind XQITEMKIND_ELEMENT
int basetype = itype.getBaseType();
// The base type MUST be XQBASETYPE_ANYTYPE
QName name = itype.getNodeName();
// name must be the QName "A"
QName tname = itype.getTypeName();
// tname must be the QName "xs:anyType"
...
```

Example 49 - Result is a node.

```
...
// Assume hatsize is a derived type of xs:positiveInteger
XQConnection conn = datasrc.getConnection();
XQPreparedExpression expr = conn.prepareExpression(
            "import schema namespace foo='http://www.foo.com' "+
            " at 'http://www.fooschema.com'; " +
            "for $i in (foo:hatsize(1),foo:hatsize(4)) return $i");
XQSequenceType seq = expr.getStaticResultType();
XQItemType itype = seq.getItemType();
int kind = itype.getItemKind();
   // May be XQItemType.XQITEMKIND_ATOMIC or XQItemType.XQITEMKIND_ITEM
if (kind == XQItemType.XQITEMKIND_ATOMIC)
{
      int basetype = itype.getBaseType();
      // the base type can be XQItemType.XQBASETYPE_ANYATOMICTYPE,
      // XQItemType.XQBASETYPE_POSITIVEINTEGER or any of its
      // ancestor types. The most specific one will be
      // XQItemType.XQBASETYPE_POSITIVEINTEGER.

      int occ = seq.getItemOccurrence();
            // Could be XQSequenceType.OCC_ZERO_OR_MORE or
            // XQSequenceType.OCC_ONE_OR_MORE
```

```
      URI schm = itype.getSchemaURI();
            // Can be NULL or http://www.fooschema.com

      QName typename = itype.getTypeName();
            // The typename must be the QName foo:hatsize
            // or a QName identifying the built-in type returned
            // by getBaseType()
}

XQSequence res = expr.executeQuery();
res.next();
XQItemType itype = res.getItemType();
   // itype MUST be an item type with kind XQItemType.XQITEMKIND_ATOMIC
   // The base type MUST be XQItemType.XQBASETYPE_POSITIVEINTEGER
   // and it may include optional schema URI (http://www.fooschm.com)
   // and an optional type name foo:hatsize (namespace
   // http://www.foo.com)
...
```

Example 50 - Result with simple and derived atomic types.

```
...
// Assume PurchaseOrder is a global element of type
// PurchaseOrderType belonging to the namespace http://www.po.com and
// defined in the schema at http://www.poschema.com. PurchaseOrder has
// a local definition of ShippingAddress inside the type definition.
XQConnection conn = datasrc.getConnection();
XQPreparedExpression expr = conn.prepareExpression(
            "import schema namespace po='http://www.po.com' "+
            " at 'http://www.poschema.com'; "+
            "for $i in doc('po.xml') "+
            " return $i/po:PurchaseOrder/po:ShippingAddr ");

// Get the static result type.
XQSequenceType seq = expr.getStaticResultType();
XQItemType itype = seq.getItemType();
int kind = itype.getItemKind();
  // May be XQItemType.XQITEMKIND_ITEM or XQItemType.XQITEMKIND_NODE,
  // XQItemType.XQITEMKIND_ELEMENT or
  // XQItemType.XQITEMKIND_SCHEMA_ELEMENT
  // If it is XQItemType.XQITEMKIND_SCHEMA_ELEMENT, then you can
  // get more information.
  // Assuming that there is enough type information available
  // about doc('po.xml') and the implementation does static typing
  // it might have more information on the result.

if (kind == XQItemType.XQITEMKIND_SCHEMA_ELEMENT)
{
      int basetype = itype.getBaseType();
      // The base type must be XQItemType.XQBASETYPE_ANYTYPE

      int occ = seq.getItemOccurrence();

      URI schm = itype.getSchemaURI();
            // Can be NULL or http://www.poschema.com

      QName name = itype.getNodeName();
```

```
            // Can be NULL or po:ShippingAddr
            // (namespace http://www.po.com)
}

XQSequence seq = expr.executeQuery();
seq.next();
XQItemType itype= seq.getItemType();
  // itype MUST be an item type with kind
  // XQItemType.XQITEMKIND_SCHEMA_ELEMENT
  // The base type MUST be XQItemType.XQBASETYPE_ANYTYPE with node
  // name po:ShippingAddr (namespace po=http://www.po.com) and it may
  // include optional schema URI (http://www.poschm.com) and an
  // optional type name (po:PurchaseOrder/po:ShippingAddr)
  // (namespace po=http://www.po.com)
  // This is an anonymous type and will return true for the
  // isAnonymousType() method invocation.
...
```

Example 51 - Result with complex values.

14 Data Type Conversions

An important requirement for XQJ 1.0 is to define how data type conversions are performed. And this in both directions:

- Converting from Java to XQuery, a Java object or primitive value is converted into an XDM instance
- Converting from XQuery to Java, an XDM instance is converted into a Java object or primitive value.

14.1 The XDM and Data Types

In order to fully grasp data type conversion in XQJ, it is important to comprehend the XQuery data model and type system. The next diagram outlines the XQuery type hierarchy taken from [XQuery F&O].

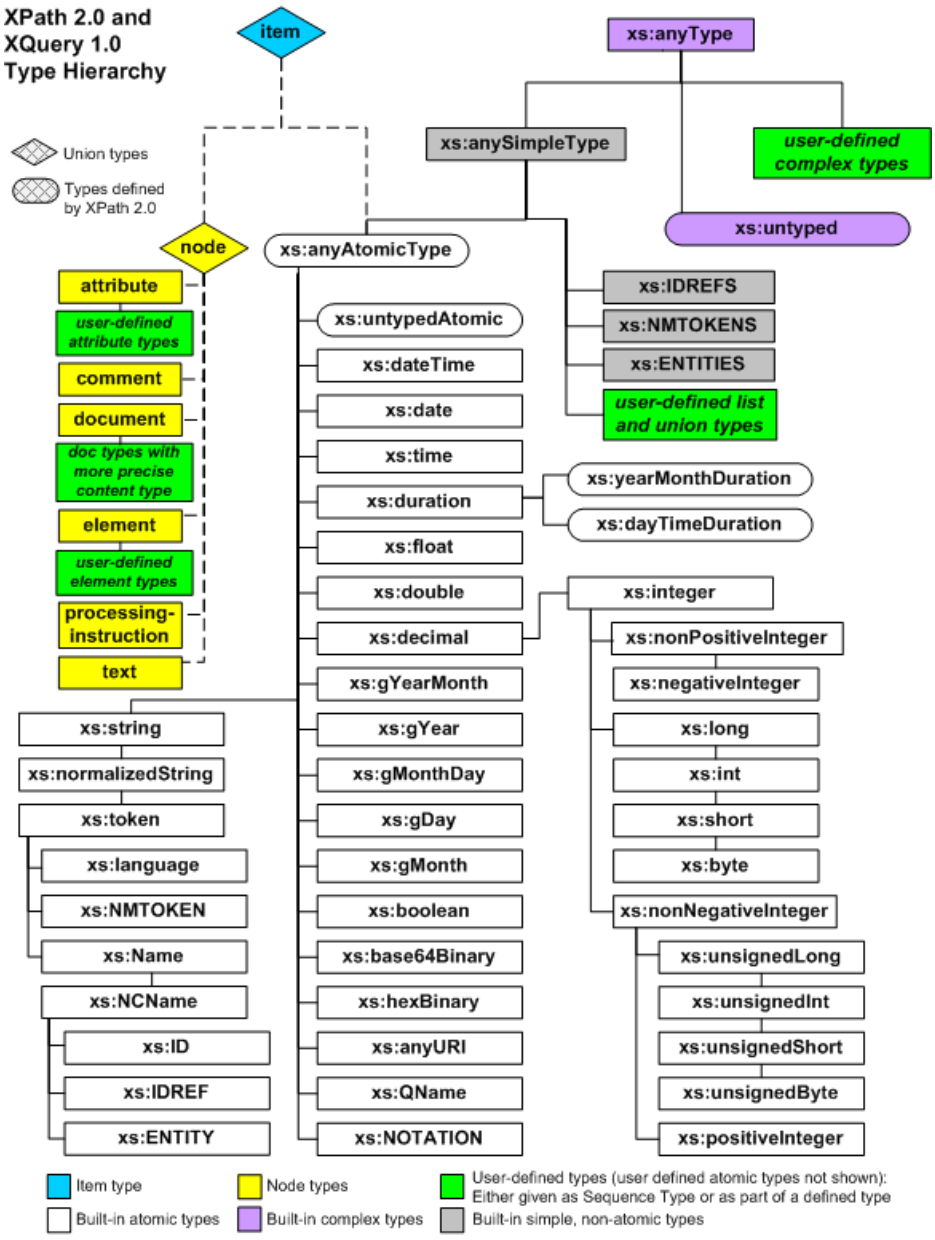

Figure 3 – XQuery 1.0 type hierarchy taken from [XQuery F&O].

More information on the XQuery data model and type system can be found in [XQuery] and its associated specifications.

14.2 Mapping a Java Data Type to an XQuery Data Type

This paragraph outlines the rules for mapping Java data types to XQuery data types, and which conversions are supported. These rules are used to convert a Java value to and XDM instance, and are applied in the following scenarios

- When a value of a Java data type is used as an input value for an XQuery expression, it needs to be mapped. This mapping is performed either during the actual bind or during the subsequent query execution.

- When a value of a Java date type is used to create an XDM instance through the `XQDataFactory` interface

A Java value (*JV*) with data type *JDT* is mapped to an XQuery data type (*XDT*) as follows:

- If the application does not specify an XQuery data type to map *JV* to, then *XDT* defaults to the XQuery data type that corresponds to *JDT* that can be found in Table 4 – Java Data Types and default XQuery Data Types. If no default XQuery data type exists for *JDT*, then the behavior is implementation-defined, which may include throwing an exception.
- Table 5 – Java Data Types and supported XQuery Data Types, specifies the supported mapping for Java data types.
  If *JDT* is not listed in Table 5 or if *XDT* is not listed as being supported for *JDT*, then the behavior is implementation-defined, this may include throwing an exception.
  Also note that an XQJ-implementation is not required to support types dependent on user defined XML Schema types, as explained in 19 “Interoperability”.
- *JV* is converted to an instance of *XDT*.
  Note that this conversion might fail, if *JV* does not conform to the value space of *XDT*, in which case an `XQException` is thrown. For example, converting a `java.lang.Integer` 10000 value to `xs:byte` will fail.
  Note also that a successful mapping of *JV* to an instance of *XDT*, does not necessarily mean that the binding to an XQuery external variable succeeds. As specified in http://www.w3.org/TR/xquery/#id-variable-declarations, if the external variable declaration in the query includes a declared type, then the value provided by the external environment must match the declared type according to the rules for SequenceType matching.

Java Data Type	**Default XQuery Data Type(s)**
boolean	xs:boolean
byte	xs:byte
byte[]	xs:hexBinary
double	xs:double
float	xs:float
int	xs:int
long	xs:long
short	xs:short
java.lang.Boolean	xs:boolean
java.lang.Byte	xs:byte
java.lang.Float	xs:float
java.lang.Double	xs:double
java.lang.Integer	xs:int
java.long.Long	xs:long
java.lang.Short	xs:short
java.lang.String	xs:string
java.math.BigDecimal	xs:decimal

Java Data Type	Default XQuery Data Type(s)
java.math.BigInteger	xs:integer
javax.xml.datatype.Duration	xs:dayTimeDuration (object represents an xs:dayTimeDuration) , xs:yearMonthDuration (object represents an xs:yearMonthDuration), xs:duration
javax.xml.datatype.XMLGregorianCalendar	xs:date (Java object represents an xs:date), xs:dateTime (Java object represents an xs:dateTime), xs:gDay (Java object represents an xs:gDay), xs:gMonth (Java object represents an xs:gMonth), xs:gMonthDay (Java object represents an xs:gMonthDay), xs:gYear (Java object represents an xs:gYear), xs:gYearMonth (Java object represents an xs:gYearMonth), xs:time (Java object represents an xs:time)
javax.xml.namespace.QName	xs:QName
org.w3c.dom.Document	document-node(element(*, xs:untyped))
org.w3c.dom.DocumentFragment	document-node(element(*, xs:untyped))
org.w3c.dom.Element	element(*, xs:untyped)
org.w3c.dom.Attr	attribute(*, xs:untypedAtomic)
org.w3c.dom.Comment	comment()
org.w3c.dom.ProcessingInstruction	processing-instruction()
org.w3c.dom.Text	text()

Table 4 – Java Data Types and default XQuery Data Types

Java Data Type	XQuery Data Type(s)
boolean	xs:boolean
byte	xs:decimal (or any of its built-in subtypes derived by restriction)
byte[]	xs:hexBinary, xs:base64Binary
double	xs:double
float	xs:float
int	xs:decimal (or any of its built-in subtypes derived by restriction)
long	xs:decimal (or any of its built-in subtypes derived by restriction)

Java Data Type	XQuery Data Type(s)
short	xs:decimal (or any of its built-in subtypes derived by restriction)
java.lang.Boolean	xs:boolean
java.lang.Byte	xs:decimal (or any of its built-in subtypes derived by restriction)
java.lang.Float	xs:float
java.lang.Double	xs:double
java.lang.Integer	xs:decimal (or any of its built-in subtypes derived by restriction)
java.long.Long	xs:decimal (or any of its built-in subtypes derived by restriction)
java.lang.Short	xs:decimal (or any of its built-in subtypes derived by restriction)
java.lang.String	xs:untypedAtomic, xs:string (or any of its built-in subtypes derived by restriction), xs:NOTATION, xs:anyURI.
java.math.BigDecimal	xs:decimal (or any of its built-in subtypes derived by restriction)
java.math.BigInteger	xs:decimal (or any of its built-in subtypes derived by restriction)
javax.xml.datatype.Duration	xs:duration (or any of its built-in subtypes derived by restriction)
javax.xml.datatype.XMLGregorianCalendar	xs:date (Java object represents an xs:date), xs:dateTime (Java object represents an xs:dateTime), xs:gDay (Java object represents an xs:gDay), xs:gMonth (Java object represents an xs:gMonth), xs:gMonthDay (Java object represents an xs:gMonthDay), xs:gYear (Java object represents an xs:gYear), xs:gYearMonth (Java object represents an xs:gYearMonth), xs:time (Java object represents an xs:time)
javax.xml.namespace.QName	xs:QName
org.w3c.dom.Document	document-node(element(*, xs:untyped)) document-node(schema-element(...)), strict validation is applied
org.w3c.dom.DocumentFragment	document-node(element(*, xs:untyped))

Java Data Type	XQuery Data Type(s)
	document-node(schema-element(...)), strict validation is applied
org.w3c.dom.Element	element(*, xs:untyped) schema-element(...), strict validation is applied
org.w3c.dom.Attr	element(*, xs:untyped)
org.w3c.dom.Comment	attribute(*, xs:untypedAtomic)
org.w3c.dom.ProcessingInstruction	comment()
org.w3c.dom.Text	processing-instruction()

Table 5 – Java Data Types and supported XQuery Data Types

14.3 Mapping a Java XML document to an XQuery document node

In addition to the conversion rules of "14.2 Mapping a Java Data Type to an XQuery Data Type", XQJ also offers the ability to create an XQuery document node for the following Java objects:

- `java.lang.String`
- `java.io.Reader`
- `java.io.InputStream`
- `javax.xml.stream.XMLStreamReader`
- `org.xml.sax.XMLReader`
- `javax.xml.transform.Source`

The following mapping is performed. Assume a Java value (*JV*) of one of the above listed types. Optionally the application specifies an XQuery data type (*XDT*).

- If *XDT* is not of kind `XQITEMKIND_DOCUMENT_SCHEMA_ELEMENT` or `XQITEMKIND_DOCUMENT_ELEMENT`, behavior is implementation defined and may raise an exception.
  Otherwise continue with the mapping as outlined in the following steps.
- An untyped XQuery document (*UXD*) is created based on *JV*, i.e. *UXD* is an instance of type `document-node(element(*, xs:untyped))`.
- If the application did not specify *XDT*, the result of the conversion is *UXD*.
- If the specified *XDT* is not of kind `XQITEMKIND_DOCUMENT_SCHEMA_ELEMENT`, the sequence type matching rules are applied. I.e. if *UXD* is not an instance of *XDT*, the conversion fails and an exception is thrown.
  If the sequence type matching rules, as outlined in [XQuery], are applied successfully the conversion results in *UXD*.
- The *XDT* is of kind `XQITEMKIND_DOCUMENT_SCHEMA_ELEMENT`, *UXD* is validated strict against the schema. The referenced element declaration in XDT must be top-level in the XML Schema.

Also note that an XQJ-implementation is not required to support types dependent on user defined XML Schema types, as explained in 19 “Interoperability”.
The conversion results in a schema validated document or in case of a failure in an exception being thrown.

14.4 Mapping an XQuery Atomic Type to a Java Object Type

Table 6 - XQuery Atomic Types and Corresponding Java Object Types – specifies the mapping of XQuery atomic types to Java object types. For an arbitrary XQuery atomic value, the closest matching atomic type in the table below specifies the Java object type returned by the `XQItemAccessor.getObject` method.

XQuery Data Type	Corresponding Java Object Type
xs:anyURI	java.lang.String
xs:base64Binary	byte[]
xs:boolean	java.lang.Boolean
xs:byte	java.lang.Byte
xs:date	javax.xml.datatype.XMLGregorianCalendar
xs:dateTime	javax.xml.datatype.XMLGregorianCalendar
xs:decimal	java.math.BigDecimal
xs:double	java.lang.Double
xs:duration	javax.xml.datatype.Duration
xs:ENTITY	java.lang.String
xs:float	java.lang.Float
xs:gDay	javax.xml.datatype.XMLGregorianCalendar
xs:gMonth	javax.xml.datatype.XMLGregorianCalendar
xs:gMonthDay	javax.xml.datatype.XMLGregorianCalendar
xs:gYear	javax.xml.datatype.XMLGregorianCalendar
xs:gYearMonth	javax.xml.datatype.XMLGregorianCalendar
xs:hexBinary	byte[]
xs:ID	java.lang.String
xs:IDREF	java.lang.String
xs:int	java.lang.Integer
xs:integer	java.math.BigInteger
xs:language	java.lang.String
xs:long	java.lang.Long
xs:Name	java.lang.String
xs:NCName	java.lang.String
xs:negativeInteger	java.math.BigInteger
xs:NMTOKEN	java.lang.String
xs:nonNegativeInteger	java.math.BigInteger
xs:nonPositiveInteger	java.math.BigInteger
xs:normalizedString	java.lang.String
xs:NOTATION	java.lang.String
xs:positiveInteger	java.math.BigInteger
xs:QName	javax.xml.namespace.QName

XQuery Data Type	Corresponding Java Object Type
xs:short	java.lang.Short
xs:string	java.lang.String
xs:time	javax.xml.datatype.XMLGregorianCalendar
xs:token	java.lang.String
xs:unsignedByte	java.lang.Short
xs:unsignedInt	java.lang.Long
xs:unsignedLong	java.math.BigInteger
xs:unsignedShort	java.lang.Integer
xs:daytimeDuration	javax.xml.datatype.Duration
xs:untypedAtomic	java.lang.String
xs:yearMonthDuration	javax.xml.datatype.Duration

Table 6 - XQuery Atomic Types and Corresponding Java Object Types

14.5 Mapping an XQuery Node Type to a Java Object Type

Table 7 - XQuery Node Types and Corresponding Java Object Types, specifies the mapping of XQuery nodes to Java object types. For an arbitrary XQuery node, the matching node kind in the table below specifies the Java object type returned by the `XQItemAccessor.getObject` method.

XQuery Node Type	Corresponding Java Object Type
Attribute	org.w3c.dom.Attr
Comment	org.w3c.dom.Comment
Document	org.w3c.dom.Document org.w3c.dom.DocumentFragment If the document node represents a well-formed XML document (it has a single element node child and no text node children, then it is mapped to org.w3c.dom.Document; otherwise, it is mapped to org.w3c.dom.DocumentFragment.
Element	org.w3c.dom.Element
Processing Instruction	org.w3c.dom.ProcessingInstruction
Text	org.w3c.dom.Text

Table 7 - XQuery Node Types and Corresponding Java Object Types

15 Data Factories

XQJ offers the ability to create objects representing XDM instances and XQuery types, through the `XQDataFactory` interface. Such `XQDataFactory` implementation is used to create:

- `XQItem` objects
- `XQSequence` objects
- `XQItemType` objects
- `XQSequenceType` objects

XQJ 1.0 mandates one concrete implementation of `XQDataFactory`, every `XQConnection` objects is by definition also an `XQDataFactory`. Future versions of XQJ might introduce different means to create and access `XQDataFactory` objects.

Different from the items and sequences obtained through query evaluation, e.g. the method `execute` defined on `XQPreparedExpression` returns an `XQSequence`, the `XQItem` and `XQSequence` objects obtained through an `XQDataFactory` are independent of any expression or connection.

15.1 Creating XQItemType objects

`XQDataFactory` provides a variety of methods to create `XQItemType` objects.

- `createAtomicType`
- `createAttributeType`
- `createCommentType`
- `createDocumentElementType`
- `createDocumentType`
- `createElementType`
- `createItemType`
- `createNodeType`
- `createProcessingInstructionType`
- `createSchemaAttributeType`
- `createSchemaElementType`

In this first example we show how to create an `XQItemType` representing `xs:string`.

```
...
// create an XQItemType representing xs:string()
XQItemType itemType = conn.createAtomicType(
                              XQItemType.XQBASETYPE_STRING);
```

```
...
```

Example 52 – create an XQItemType representing xs:string.

Not only does `XQDataFactory` offer the ability to create atomic types but also `XQItemType` objects representing a node kind. As shown in Example 53, which demonstrates the creating of an `element(foo:bar, xs:integer)`.

```
...
// create an XQItemType representing element(foo:bar, xs:integer)
XQItemType itemType = createElementType(
               new QName("http://www.foo.com", "bar","foo"),
               XQItemType.XQBASETYPE_INTEGER);
...
```

Example 53 – create an XQItemType representing element(foo:bar, xs:integer).

Or as in Example 54, which demonstrates the creating of the abstract type `element()`.

```
...
// create an XQItemType representing element()
XQItemType itemType = createElementType(
               null, XQItemType.XQBASETYPE_ANYTYPE);
...
```

Example 54 – create an XQItemType representing element().

In addition, other abstract types can be created through `XQDataFactory` too. Like in Example 55, showing the creation of an `XQItemType` object representing `item()`.

```
...
// create an XQItemType representing item()
XQItemType itemType = createItemType();
...
```

Example 55 – create an XQItemType representing item().

15.2 Creating XQSequenceType objects

Creating an `XQSequenceType` object is accomplished with the `createSequenceType` method defined on `XQDataFactory`, specifying an `XQItemType` and occurrence indicator as arguments.

If the specified XQItemType object is not null, one of the following occurrence indicatiors must be specified

- `XQSequenceType.OCC_ZERO_OR_ONE`
- `XQSequenceType.OCC_EXACTLY_ONE`
- `XQSequenceType.OCC_ZERO_OR_MORE`
- `XQSequenceType.OCC_ONE_OR_MORE`

In order to create an `empty-sequence()`, `null` is specified as XQItemType for the first parameter and an occurrence indicator `OCC_EMPTY` as second argument.

Example 56 creates an `XQSequenceType` representing `node()*`.

```
...
// create an XQItemType representing node()
XQItemType itemType = conn.createNodeType();

// create an XQSequenceType representing node()*
XQSequenceType sequenceType = conn.createSequenceType(
                                    itemType,
                                    XQSequenceType.OCC_ZERO_OR_MORE);
...
```

Example 56 – create an XQSequenceType representing node()*.

Example 57 shows how to create an `XQSequenceType` representing `empty-sequence()`.

```
...
// create an XQSequenceType representing empty-sequence
XQSequenceType sequenceType = conn.createSequenceType(
                                    null,
                                    XQSequenceType.OCC_EMPTY);
...
```

Example 57 – create an XQSequenceType representing empty-sequence().

15.3 Creating XQItem objects

`XQItem` objects created by one of the `XQDataFactory` methods are standalone. I.e. different from `XQItem` objects resulting from query evaluation, which are invalidated when the `XQResultSequence` is explicitly or implicitly closed, `XQItem` object created through an `XQDataFactory` are valid until explicitly closed by the application using the `close` method.

The `XQDataFactory` provides an extensive list of methods to create `XQItem` objects.

- `createItem`
- `createItemFromAtomicValue`
- `createItemFromBoolean`
- `createItemFromByte`
- `createItemFromDocument`
- `createItemFromDouble`
- `createItemFromFloat`
- `createItemFromInt`
- `createItemFromLong`
- `createItemFromNode`
- `createItemFromObject`
- `createItemFromShort`

- `createItemFromString`

Example 58 creates a deep copy from a specified `XQItem` object.

```
...
// execute an xquery
XQExpression expr = conn.createExpression();
XQSequence result = expr.executeQuery("fn:doc('doc1.xml')");

// creates an XQItem, copying the first one in the result
result.next();
XQItem item = conn.createItem(result.getItem());

// free all resources allocated for the connection, this includes
// the expression and result
conn.close();

// the connection is closed, still the XQItem is valid
item.getObject();
...
```

Example 58 – create an XQItem, making a deep copy of another XQItem.

Through the `XQDataFactory`, one can also create atomic values, specifying the atomic type and lexical value. The next example shows to create twice an `xs:float` value (100) and in addition an `xs:string`.

```
...
// create a first xs:float
XQItem item1 = conn.createItemFromAtomicValue(
                   "100",
                   conn.createAtomicType(XQItemType.XQBASETYPE_FLOAT));
// create a second xs:float
XQItem item2 = conn.createItemFromAtomicValue(
                   "1E2",
                   conn.createAtomicType(XQItemType.XQBASETYPE_FLOAT));
// create an xs:string
XQItem item3 = conn.createItemFromAtomicValue(
                   "Hello world!",
                   conn.createAtomicType(XQItemType.XQBASETYPE_STRING));
...
```

Example 59 – specify lexical values to create XQItem objects.

Rather than creating `xs:string` XDM instances through the `createItemFromAtomicValue` method, and thus forcing to specify an `XQItemType`, there is also a convenience method to create `xs:string` instances, as shown in Example 60.

```
...
// create an xs:string
XQItem item = conn.createItemFromString("Hello world!", null);
...
```

Example 60 – create an XQItem from a java.lang.string.

Specifying null for the second argument, defaults to create an `xs:string` instance. This second argument is either null, an `XQItemType` representing `xs:string` or any type derived by restriction from `xs:string`.

Further `XQDataFactory` also provides a method to create an `XQItem` specifying any of the Java primitive types. The actual type of the resulting XQItem is governed by 14.2 “Mapping a Java Data Type to an XQuery Data Type”. Example 61 shows the creation of an `XQItem` representing a `xs:int`.

```
...
// create an xs:int
XQItem item = conn.createItemFromInt(123, null);
...
```

Example 61 – create an XQItem from a java int.

`XQDataFactory` also provides the ability to create `XQItem` objects representing a node, specifying a DOM object. The conversion rules are outlined in 14.2 “Mapping a Java Data Type to an XQuery Data Type”. Example 62 shows how to create a document node.

```
...
org.w3c.dom document = ...; // a DOM document

// create a document node
XQItem item = conn.createItemFromNode(document, null);
...
```

Example 62 – create an XQItem representing a document node.

An `XQDataFactory` also offers the ability to create an `XQItem` representing a document node through the following Java object types

- `java.lang.String`
- `java.io.Reader`
- `java.io.InputStream`
- `javax.xml.stream.XMLStreamReader`
- `org.xml.sax.XMLReader`
- `javax.xml.transform.Source`

Example 63 creates a document node from a file doc.xml.

```
...
// create a document node based on doc.xml
FileInputStream input = new FileInputStream("doc.xml");
XQItem item = conn.createItemFromDocument(input, null, null);
...
```

Example 63 – create an XQItem representing a document node.

15.4 Creating XQSequence objects

An `XQSequence` object created through an `XQDataFactory` is standalone, e.g. independent of any `XQConnection` object. Important to note is that such `XQSequence` object is always scrollable.

There are two methods in `XQDataFactory` to create `XQSequence` objects. First, there is a copy method, creating an `XQSequence` starting from another one. As shown in Example 64.

```
...
// execute an xquery
XQExpression expr = conn.createExpression();
XQSequence result = expr.executeQuery(
                      "fn:doc('doc1.xml'), fn:doc('doc2.xml'");

// create a new sequence object independent of the connection
// and populate it with the items from "result"
XQSequence sequence = conn.createSequence(result);

// free all resources allocated for the connection, this includes
// the expression and result
conn.close();

// the connection is closed, still the XQSequence is valid
sequence.next();
...
```

Example 64 – create an XQSequence by copying another XQSequence.

Second, an `XQSequence` can be created through a `java.util.Iterator`. The `Iterator` object should either return

- `XQItem` objects, in which case a copy of the item is added to the `XQSequence`
- other objects are added as items according to the rules described in 14.2 "Mapping a Java Data Type to an XQuery Data Type"

Example 64 shows the creation of an `XQSequence` starting from an `Iterator` returning `XQItem` objects; the `XQSequence` contains 3 `xs:int` items

```
...
ArrayList items = new ArrayList()
items.add(conn.createItemFromInt(123,null));
items.add(conn.createItemFromInt(456,null));
items.add(conn.createItemFromInt(789,null));
XQSequence sequence = conn.createSequence(items.iterator());
```

Example 65 – create an XQSequence through an Iterator of XQItem objects.

In the next example, we create a similar `XQSequence` object, using an `Iterator` returning `java.lang.Integer` objects.

```
...
ArrayList items = new ArrayList()
items.add(new Integer(123));
items.add(new Integer(456));
items.add(new Integer(789));
XQSequence sequence = conn.createSequence(items.iterator());
```

Example 66 – create an XQSequence through an Iterator of Integer objects.

Note that the `XQSequence` does not need to be homogeneous, it can have items of different types. The next example, according to the rules of 14.2 "Mapping a Java Data Type to an XQuery Data Type", creates an `XQSequence` with an `xs:int` atomic value, `xs:double` and a document node.

```
...
```

```
org.w3c.dom document = ...; // a DOM document
ArrayList items = new ArrayList()
items.add(new Integer(123));
items.add(new Double(456));
items.add(document);
XQSequence sequence = conn.createSequence(items.iterator());
```

Example 67 – create an XQSequence with items of different types.

16 Transactions

Transactions are used by applications to ensure data integrity and in case of concurrent access by multiple applications, to get a consistent view of the data. As such XQJ provides transactional support through its API.

This chapter describes the transaction semantics associated with a single `XQConnection` object. This version of XQJ does not support transactions involving multiple `XQConnection` objects or any other type of data sources, so called "distributed transactions".

16.1 Transaction Boundaries and Auto-commit

By default, a connection operates in auto-commit mode, which means that each xquery is executed and committed in an individual transaction. When a query is executed a new transaction is automatically started, and it completes when all the query results have been processed and the `XQResultSequence` is closed.

If auto-commit mode is disabled, a transaction must be ended explicitly by the application calling the `commit` or `rollback` method on the `XQConnection` object. In case of such event a new transaction is automatically started.

The default is for auto-commit mode to be enabled when the `XQConnection` object is created. The auto-commit connection attribute is controlled using the `setAutoCommit` method.

```
...
XQConnection con = dataSource.getConnection("myUID","myPWD");
// The connection operates in auto-commit mode.

con.setAutoCommit(false);
// The connection operates in manual-commit mode, the application
// must explicitly end the transactions
...
con.commit();
```

Example 68 – Disabling auto-commit and committing the transaction.

If the value of auto-commit is changed in the middle of a transaction, the transaction is committed. If `setAutoCommit` is called and the auto-commit attribute is not changed from its current value, it is treated as a no-op.

16.2 Transactions are an optional feature

XQJ-implementations are not required to support transactions. An application can determine if transactions are supported using the `isTransactionSupported` method on `XQMetaData`.

Implementations which do not support transactions, must raise an error if auto-commit is turned off.

16.3 XQConnections on top of JDBC connections

As described in paragraph 8.3 “Interacting with JDBC Connections”, an `XQConnection` object can be created on top of an existing JDBC connection. If an `XQConnection` is created on top of a JDBC connection, it inherits the transaction context from the JDBC connection. As such the `XQConnection` object does not necessarily operate in auto-commit mode by default, Also, in this case, if the auto-commit mode is changed, or a transaction is ended using `commit` or `rollback`, it also changes the underlying JDBC connection.

17 Connection Pooling

In a basic `XQDataSource` implementation, there is a 1:1 correspondence between the client's `XQConnection` object and the physical connection to the data source. When the `XQConnection` object is closed, the physical connection is dropped. Thus, the overhead of opening, initializing, and closing the physical connection is incurred for each client session.

A connection pool solves this problem by maintaining a cache of physical connections to the data source that can be reused across client sessions. Connection pooling greatly improves performance and scalability, particularly in a three-tier environment where multiple clients can share a smaller number of physical connections.

In Figure 4, the XQJ driver provides an implementation of `ConnectionPoolXQDataSource` that the application server uses to build and manage the connection pool.

The algorithm used to manage the connection pool is implementation-dependent and varies with application servers. The application server provides its clients with an implementation of the `XQDataSource` interface that makes connection pooling transparent to the client. As a result, the client gets better performance and scalability while using the same JNDI and `XQDataSource` APIs as before.

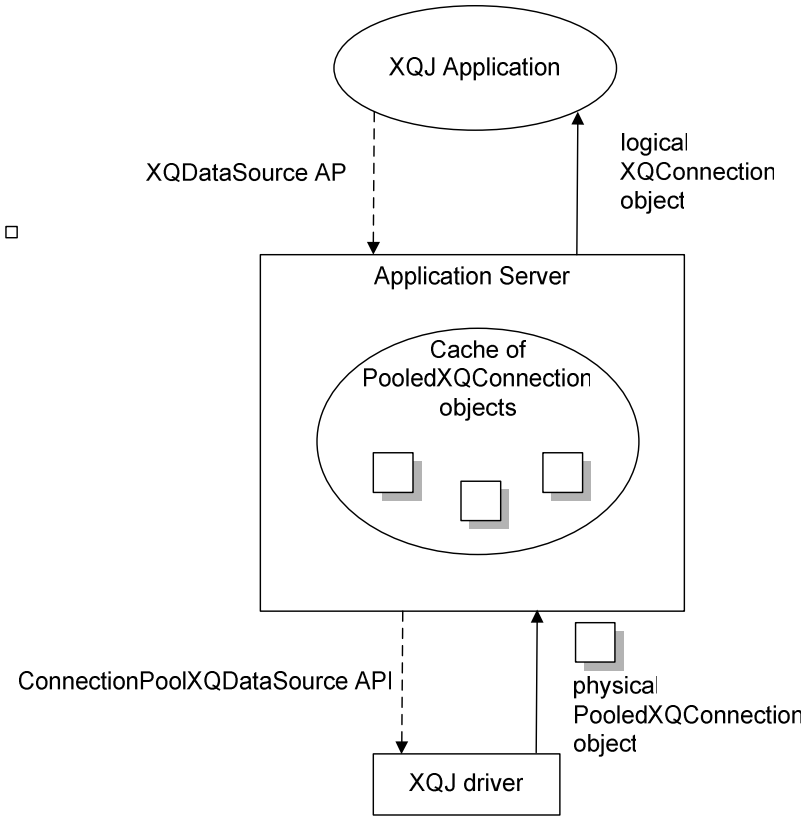

Figure 4 – Relationship Between Major XQJ Interfaces

The following sections introduce the `ConnectionPoolXQDataSource` interface, the `PooledXQConnection` interface, and the `XQConnectionEvent` class. These pieces, which operate beneath the `XQDataSource` and `XQConnection` interfaces used by the client, are incorporated into a step-by-step description of a typical connection pooling implementation. This chapter also describes some important differences between a basic `XQDataSource` object and one that implements connection pooling.

Although much of the discussion in this chapter assumes a three-tier environment, connection pooling is also relevant in a two-tier environment. In a two-tier environment, the XQJ driver implements both the `XQDataSource` and `ConnectionPoolXQDataSource` interfaces. This implementation allows an application that opens and closes multiple connections to benefit from connection pooling.

17.1 ConnectionPoolXQDataSource and PooledXQConnection

Typically, an XQJ driver implements the `ConnectionPoolXQDataSource` interface, and the application server uses it to obtain `PooledXQConnection` objects.

The next code example shows the signatures for the two versions of the `getPooledConnection` method.

```
public interface ConnectionPoolXQDataSource {
      PooledXQConnection getPooledConnection()
            throws XQException;
      PooledXQConnection getPooledConnection(String user,
            String password) throws XQException;
      ...
}
```

Example 69 – The ConnectionPoolXQDataSource interface.

A `PooledXQConnection` object represents a physical connection to a data source. The XQJ driver's implementation of `PooledXQConnection` encapsulates all of the details of maintaining that connection.

An application server caches and reuses `PooledXQConnection` objects within its implementation of the `XQDataSource` interface. When a client calls the method `XQDataSource.getConnection`, the application server uses the physical `PooledXQConnection` object to obtain a logical `XQConnection` object.

The following example shows the `PooledXQConnection` interface definition.

```
public interface PooledXQConnection {
      XQConnection getConnection() throws XQException;
      void close() throws XQException;
      void addConnectionEventListener(
            XQConnectionEventListener listener);
      void removeConnectionEventListener(
            XQConnectionEventListener listener);

}
```

Example 70 – The PooledXQConnection interface.

When an application is finished using a connection, it closes the logical connection using the method `XQConnection.close`. This closes the logical connection but does not close the physical connection. Instead, the physical connection is returned to the pool so that it can be reused.

Connection pooling is completely transparent to the client: A client obtains a pooled connection and uses it just the same way it obtains and uses a non pooled connection.

17.2 Connection Events

Recall that when an application calls the method `XQConnection.close`, the underlying physical connection—the `PooledXQConnection` object—becomes available for reuse. JavaBeans-style events are used to notify the connection pool manager (the application server) that a `PooledXQConnection` object can be recycled.

In order to be notified of an event on a `PooledXQConnection` object, the connection pool manager must implement the `XQConnectionEventListener` interface and then be registered as a listener by that `PooledXQConnection` object. The `XQConnectionEventListener` interface defines the following two methods, which correspond to the two kinds of events that can occur on a `PooledXQConnection` object:

- `connectionClosed` — triggered when the logical `XQConnection` object associated with this `PooledXQConnection` object is closed, that is, the application called the method `Connection.close`
- `connectionErrorOccurred` — triggered when a fatal error, such as the server crashing, causes the connection to be lost

A connection pool manager registers itself as a listener for a `PooledXQConnection` object using the `PooledXQConnection.addConnectionEventListener` method. Typically, a connection pool manager registers itself as an `XQConnectionEventListener` before returning an `XQConnection` object to the application.

The driver invokes the `XQConnectionEventListener` methods `connectionClosed` and `connectionErrorOccurred` when the corresponding events occur. Both methods take an `XQConnectionEvent` object as a parameter, which can be used to determine which `PooledXQConnection` object was closed or had an error. When the XQJ application closes its logical connection, the XQJ driver notifies the connection pool manager (the listener) by calling the listener's implementation of the method `connectionClosed`. At this point, the connection pool manager can return the `PooledXQConnection` object to the pool for reuse.

When an error occurs, the XQJ driver notifies the listener by calling its `connectionErrorOccurred` method and then throws an `XQException` object to the application to notify it of the same error. In the event of a fatal error, the bad `PooledXQConnection` object is not returned to the pool. Instead, the connection pool

manager calls the `close` method on the `PooledXQConnection` object to close the physical connection.

17.3 Connection Pooling in a Three-tier Environment

The following sequence of steps outlines what happens when an XQJ client requests a connection from an `XQDataSource` object that implements connection pooling:

- The client calls `XQDataSource.getConnection`.
- The application server providing the `XQDataSource` implementation looks in its connection pool to see if there is a suitable `PooledXQConnection` object — a physical database connection — available. Determining the suitability of a given `PooledXQConnection` object may include matching the client's user authentication information or application type as well as using other implementation-dependent criteria. The lookup method and other methods associated with managing the connection pool are specific to the application server.
- If there are no suitable `PooledXQConnection` objects available, the application server calls the `ConnectionPoolXQDataSource.getPooledConnection` method to get a new physical connection. The XQJ driver implementing `ConnectionPoolXQDataSource` creates a new `PooledXQConnection` object and returns it to the application server.
- Regardless of whether the `PooledXQConnection` was retrieved from the pool or was newly created, the application server does some internal bookkeeping to indicate that the physical connection is now in use.
- The application server calls the method `PooledXQConnection.getConnection` to get a logical `XQConnection` object. This logical `XQConnection` object is actually a "handle" to a physical `PooledXQConnection` object, and it is this handle that is returned by the `XQDataSource.getConnection` method when connection pooling is in effect.
- The application server registers itself as an `XQConnectionEventListener` by calling the method `PooledXQConnection.addConnectionEventListener`. This is done so that the application server will be notified when the `PooledXQConnection` object is available for reuse.
- The logical `XQConnection` object is returned to the application, which uses the same `XQConnection` API as in the basic `XQDataSource` case. Note that the underlying `PooledXQConnection` object cannot be reused until the client calls the method `XQConnection.close`.

Connection pooling can also be implemented in a two-tier environment where there is no application server. In this case, the XQJ driver provides both the implementation of

`XQDataSource` which is visible to the client and the underlying `ConnectionPoolXQDataSource` implementation.

17.4 XQDataSource Implementations and Connection Pooling

Aside from improved performance and scalability, an XQJ application should not see any difference between accessing an `XQDataSource` object that implements connection pooling and one that does not. However, there are some important differences in the application server and driver level implementations.

A basic `XQDataSource` implementation, that is, one that does not implement connection pooling, is typically provided by an XQJ driver vendor. In a basic `XQDataSource` implementation, the following are true:

- The `XQDataSource.getConnection` method creates a new `XQConnection` object that represents a physical connection and encapsulates all of the work to set up and manage that connection.
- The `XQConnection.close` method shuts down the physical connection and frees the associated resources.

In an `XQDataSource` implementation that includes connection pooling, a great deal happens behind the scenes. In such an implementation, the following are true:

- The `XQDataSource` implementation includes an implementation-dependent connection pooling module that manages a cache of `PooledXQConnection` objects.
  The `XQDataSource` object is typically implemented by the application server as a layer on top of the driver's implementations of the `ConnectionPoolXQDataSource` and `PooledXQConnection` interfaces.
- The `XQDataSource.getConnection` method calls `PooledXQConnection.getConnection` to get a logical handle to an underlying physical connection. The overhead of setting up a new physical connection is incurred only if there are no existing connections available in the connection pool. When a new physical connection is needed, the connection pool manager will call the `ConnectionPoolXQDataSource` method `getPooledConnection` to create one. The work to manage the physical connection is delegated to the `PooledXQConnection` object.
- The `XQConnection.close` method closes the logical handle, but the physical connection is maintained. The connection pool manager is notified that the underlying `PooledXQConnection` object is now available for reuse. If the application attempts to reuse the logical handle, the `XQConnection` implementation throws an `XQException`.
- A single physical `PooledXQConnection` object may generate many logical `XQConnection` objects during its lifetime. For a given `PooledXQConnection` object, only the most recently produced logical

`XQConnection` object will be valid. Any previously existing `XQConnection` object is automatically closed when the associated `PooledXQConnection.getConnection` method is called. Listeners (connection pool managers) are not notified in this case.
This gives the application server a way to take a connection away from a client. This is an unlikely scenario but may be useful if the application server is trying to force an orderly shutdown.

- A connection pool manager shuts down a physical connection by calling the method `PooledXQConnection.close`. This method is typically called only in certain circumstances: when the application server is undergoing an orderly shutdown, when the connection cache is being re initialized, or when the application server receives an event indicating that an unrecoverable error has occurred on the connection.

17.5 Deployment

Deploying an `XQDataSource` object that implements connection pooling requires that both a client-visible `XQDataSource` object and an underlying `ConnectionPoolXQDataSource` object be registered with a JNDI-based naming service.

The first step is to deploy the `ConnectionPoolXQDataSource` implementation, as is done in the next example.

```
// ConnectionPoolDS implements the ConnectionPoolXQDataSource
// interface. Create an instance and set properties.
com.acme.ConnectionPoolDS cpds =
      new com.acme.ConnectionPoolDS();
cpds.setServerName("bookserver");
cpds.setDatabaseName("booklist");
cpds.setPortNumber(9040);
cpds.setDescription("Connection pooling for bookserver");
// Register the ConnectionPoolDS with JNDI, using the logical name
// "xqj/pool/bookserver_pool"
Context ctx = new InitialContext();
ctx.bind("xqj/pool/bookserver_pool", cpds);
```

Example 71 – Deploying a ConnectionPoolXQDataSource object.

Once this step is complete, the `ConnectionPoolXQDataSource` implementation is available as a foundation for the client-visible `XQDataSource` implementation. The `XQDataSource` implementation is deployed such that it references the `ConnectionPoolXQDataSource` implementation, as shown in the example below.

```
// PooledDataSource implements the XQDataSource interface.
// Create an instance and set properties.
com.acme.appserver.PooledDataSource ds =
      new com.acme.appserver.PooledDataSource();
ds.setDescription("Datasource with connection pooling");
// Reference the previously registered ConnectionPoolXQDataSource
ds.setDataSourceName("xqj/pool/bookserver_pool");
// Register the XQDataSource implementation with JNDI,
// using the logical name "xqj/bookserver".
```

```
Context ctx = new InitialContext();
ctx.bind("xqj/bookserver", ds);
```

Example 72 – Deploying an XQDataSource object backed by a ConnectionPoolXQDataSource object.

The `XQDataSource` object is now available for use in an application.

18 MetaData

An `XQMetaData` object provides information about the XQJ-implementation, underlying XQuery engine and possibly the accessible data sources in an given configuration. It's primarily used by generic applications and tools, targeting multiple XQJ-implementations, to determine how to interact with a specific `XQConnection` object.

The methods included in the `XQMetaData` interface can be categorized according to the types of information they provide:

- XQuery optional features
- Version information
- Character encoding support
- General information about the XQJ-implementation and data source
- Updates and transactions

18.1 Creating a MetaData object

Every `XQConnection` object has its own metadata. Different `XQConnection` objects from the same XQJ-implementation can possibly return different metadata information – think about the scenario where two `XQConnection` objects are used to query a different data source.

As such, an `XQMetaData` object is created with the `getMetaData` method defined on `XQConnection`. Subsequently the `XQMetaData` object is used to discover information from the underlying XQJ-implementation, XQuery engine and data sources.

Example 73 retrieves the `XQMetaData` object from connection `conn`, and determines if the XQuery full access feature is supported.

```
...
// assume an XQConnection conn
XQMetaData metaData = conn.getMetaData();
boolen fullAxisSupported = metadata.isFullAxisFeatureSupported();
...
```

Example 73 – Creating an XQMetaData object.

18.2 Determining XQuery Optional Feature support

A first group of `XQMetaData` methods can be used to determine if a specific XQuery Optional Feature is supported.

- isFullAxisFeatureSupported
- isModuleFeatureSupported
- isSchemaImportFeatureSupported
- isSchemaValidationFeatureSupported

- isSerializationFeatureSupported
- isStaticTypingExtensionsSupported
- isStaticTypingFeatureSupported
- isXQueryXSupported

18.3 Retrieving Version information

Some generic application might need to determine version information.

- getProductMajorVersion
- getProductMinorVersion
- getProductName
- getProductVersion
- getXQJMajorVersion
- getXQJMinorVersion
- getXQJVersion

18.4 Detrmining supported character encodings

XQJ is designed to run on any Java platform, but the supported character encodings, e.g. used by the implementation to parse an XQuery expression, might differ from implementation or be dependent on some other environmental characteristics.

- getSupportedXQueryEncodings
- isXQueryEncodingDeclSupported
- isXQueryEncodingSupported

18.5 Retrieving General Information

A number of `XQMetaData` methods are used to retrieve general information

- getUserName
- getMaxExpressionLength
- getMaxUserNameLength
- isUserDefinedXMLSchemaTypeSupported
- wasCreatedFromJDBCConnection

18.6 Support for Updates and Transactions

Although update functionality is not part of [XQuery], XQJ expects that some implementations will include some proprietary extensions to handle update functionality.

- isReadOnly

- isTransactionSupported

19 Interoperability

This paragraph describes the interoperability of XQuery data model instances and XQuery data types between the application and XQJ-implementation and in addition between XQJ-implementations.

19.1 Issues with Interoperability

Some XQuery data types are based on an XML Schema type, for example `element(*,xs:integer)`. As such in order to understand such XQuery data type, the implementation must have the ability to interpret the XML Schema type. While the built-in XML Schema types, such as `xs:decimal`, are understood by all implementations, it is not clear what `myschema:hatSize` or `po:purchaseOrderType` means, unless the underlying XML schema that defines these user defined types is available to the implementation.

This raises the question if an `XQItemType` or `XQSequenceType` object can be passed from one implementation to another. As every item in the XQuery data model has a type, the question is also if `XQItem` and `XQSequence` objects are interoperable.

Furthermore, each node has a unique identity. Every node in an instance of the data model is unique: identical to itself, and not identical to any other node. If a node is passed to or retrieved from an XQJ-implementation, is the node identity preserved?

Next, XQuery defines the concept of document order. Document order is defined among all the nodes accessible during a given query. Document order is a total ordering, although the relative order of some nodes is implementation-dependent. If nodes are transferred from one to the other implementation, is document order then preserved?

Finally, every node, except the node at the root of a tree, has a parent. Is the parent-child relationship preserved?

Therefore, XQuery types and XQuery data model instances are not stand-alone, passing and consuming such objects between XQJ-implementations requires communication of information like XML Schema, full node context, document order and node identity. If any of this information is not preserved it affects the interoperability.

The question about interoperability goes further than only between implementations. Is the information preserved when passed from one `XQConnection` instance to another instance, or even within a single connection from one `XQExpression` to another?

19.2 Design Principle for handling interoperability

Although one can envision mechanisms to guarantee such interoperability, such as sharing XML Schema repository or passing globally unique node references around, it requires some proprietary designs among implementations that are beyond the scope of XQJ. Therefore, we use the following principles for providing interoperability of `XQItem` and `XQItemType` objects:

- Only `XQItemType` of predefined XML Schema types, `XQSequenceType` of `XQItemType` of predefined XML Schema types, `XQItem` of XQuery atomic

values whose type are of predefined XML Schema Types, `XQSequence` instances of `XQItem` objects of XQuery atomic values whose type are of predefined XML Schema Types, are understood by all XQJ-implementations. Behavior of sharing of any other `XQItemType`, `XQSequenceType`, `XQItem` and `XQSequence` among different implementations and different `XQConnection` instances are implementation-defined.

- Practically this means XQJ applications can not expect `XQItem` instances created by one `XQConnection` to be consumed in the context of the same or another `XQConnection` without loosing information. The full knowledge and information of the XML Schema type, node identity, document order and full node context is NOT guaranteed to be preserved.
- A reasonable approach for an XQJ-implementation is to consume an `XQItem` object as follows:
    - If the `XQItem` represents a node, consume it as a DOM node, in which the case type annotation and other node characteristics are not preserved.
    - Consume each atomic item of predefined XML Schema type without loosing any information
    - For atomic values of user defined type, only its base built-in type and the value can be consumed. That is, the XML schema type information is not preserved.

19.3 Interoperability specified

XQJ defines the following interoperability requirements. Every XQJ-implementation must guarantee these interoperability rules.

- Interoperability of `XQItemType` instances
  All XQJ-implementations must fully understand and support all `XQItemType` instances not dependent on a user defined XML Schema type. As a consequence, `XQItemType` objects independent of any XML Schema type are by definition also interoperable.
  It is however implementation-defined whether an implementation is able to understand `XQItemType` instance's based on user defined XML Schema types, created by another implementation.
  For example, implementation A should accept an `XQItemType` object representing an `xs:integer`, created by implementation B. An `XQItemType` instance representing `item()`is also interoperable as this item type independent of any XML Schema type.
  On the other hand, an `XQItemType` object representing `schema-element(myschema:hatSize)` is not guaranteed to be interoperable.
- Interoperability of `XQSequenceType` instances
  The interoperability of an `XQSequenceType` depends on the characteristics of the enclosed `XQItemType` instance.
  For example an `XQSequenceType object` representing `xs:integer*` is interoperable between implementations. As a special case, an empty-sequence() `XQSequenceType` instance, which has no enclosed `XQItemType`, is always

interoperable.
On the other hand, `myschema:hatSize*` is not interoperable.

- Interoperability of `XQItem` instances representing an atomic value
  For XQuery data model atomic values other than atomic value whose type is of XML Schema user defined type, all XQJ-implementations should be able to fully understand and support them.
  For example, one implementation should be able to understand an `xs:integer XQItem` object created by another implementation as the `XQItem` instance is dependent on a built-in XML Schema type.
  For XQuery data model atomic value whose type is of XML Schema user defined type, XQJ doesn't mandate interoperability. Basically the rule is the same as that of "Interoperability of `XQItemType` instances" defined above.
- Interoperability of `XQItem` instances representing a node
  For XQuery data model nodes, retrieved and passed back through the XQJ API, it is implementation-defined if the following properties are preserved: node identity, document order and full node context. XQJ does not specify that the `equals` method on `XQItem` checks for XQuery node identity equality, as specified for `java.lang.Object`, `equals` checks for object equality, and anything beyond that is implementation-defined.
  Furthermore, as with atomic values, also for nodes interoperability is only guaranteed if the node's type is not based on a user defined XML Schema type.

19.4 Examples of interoperability

In the next example, we create a sequence containing twice the same element, and bind that result into a second query. It is implementation-defined whether both items in the sequence still have the same identity during the evaluation of the second query.

```
XQExpression expr1 = conn.createExpression();
XQExpression expr2 = conn.createExpression();

String es1 = "let $doc = fn:doc('po.xml') " +
             "let $x as element():= $doc/purchaseOrder " +
             "return ($x, $x)";
String es2 = "declare variable $x as element()* external; ";
             "$x[1] is $x[2]";
XQResultSequence rslt = expr1.executeQuery(es1);
expr2.bindSequence(new QName("x"), rslt);
XQResultSequence rslt2 = expr2.executeQuery(es2);
rslt2.next();
boolean b = rslt2.getBoolean();
// value of b is implementation-defined. It can be true if the node
identity is preserved and false otherwise.
```

Example 74 – Is node identity preserved?

In the second example, an element is returned by a first query and bound to the second. It's implementation-defined if the element has a parent or has become the root of an XML fragment,

```
XQExpression expr1 = conn.createExpression();
XQExpression expr2 = conn.createExpression();
```

```
String es1 = "let $doc = fn:doc('po.xml') " +
             "let $x as element():= $doc/purchaseOrder " +
             "return $x";
String es2 = "declare variable $x as element() external; " +
             "$x/..";
XQResultSequence rslt = expr1.executeQuery(es1);
expr2.bindSequence(new QName("x"), rslt);
XQResultSequence rslt2 = expr2.executeQuery(es2);
boolean b = rslt2.next();
// if the parent relationship for the node is preserved, b will be true.
// otherwise an empty sequence is returned, in which case b is false.
```

Example 75 – Is the parent relationship preserved?

In the next example, we bind an xs:integer from a first into a second query. As this `XQItem` represents an atomic value of a built-in schema type, it is guaranteed that the bound value into the second query is of type xs:integer.

```
XQExpression expr1 = conn.createExpression();
XQExpression expr2 = conn.createExpression();

String es1 = "1";
String es2 = "declare variable $x external; " +
             "$x instance of xs:integer";
XQResultSequence rslt = expr1.executeQuery(es1);
expr2.bindSequence(new QName("x"), rslt);
XQResultSequence rslt2 = expr2.executeQuery(es2);
boolean b = rslt2.next();
// According to the XQJ interoperability rules, b must be true.
```

Example 76 – built-in atomic values and types are fully interoperable.

Suppose the scenario where an application calls the method `XQDataFactory.createItemFromLong(long value, XQItemType type)`

How could an XQJ-implementation process the specified `XQItemType`?

- `XQDataFactory.createItemFromLong(long value, XQItemType type)` is based on the conversion table in "14.2 Mapping a Java Data Type to an XQuery Data Type"
- According to the table, the creation of the item must succeed if the `XQItemType` is `xs:decimal` or any of its built-in subtypes derived by restriction.
- The method `getBaseType` on `XQItemType`, returns the closest matching built-in type.
- Using this information obtained from the method of `XQItemType`, the implementation can create an `XQItem` instance of the specified `XQItemType`, using the `long` value.
- However, the `XQItemType` might be a user defined type in an XML Schema, such as `po:hatSize` whose base type is `xs:decimal tool`. In order to detect such scenario, the implementation can invoke the getTypeName() method, to figure out if the `XQItemType` represents a built-in XML Schema type like `xs:long` or a user defined type like `po:hatSize`.

- If the `XQItemType` represents such user defined schema type, XQJ doesn't guarantee that the `XQItemType` is understood. As such it is legal to make the method invocation fail, or the implementation might consider some proprietary means and create the appropriate `XQItem` instance.

20 Releasing External Resources

Applications must explicitly or implicitly close the various XQJ objects such as `XQConnection`, `XQExpression`, `XQPreparedExpression`, `XQSequence` and `XQItem`. Objects are explicitly closed using the respective `close` methods. Application writers are strongly encouraged to invoke the `close` methods as soon as possible to release the associated resources.

In a number of scenarios object are closed implicitly:

- an `XQResultItem` is implicitly closed when the parent `XQResultSequence` is implicitly or explicitly closed,
- an `XQResultSequence` is implicitly closed when the parent `XQExpression` or `XQPreparedExpression` is implicitly or explicitly closed or is re-executed,
- an `XQExpression` is implicitly closed when the parent `XQConnection` is closed,
- an `XQPreparedExpression` is implicitly closed when the parent `XQConnection` is closed.

Failing to close the above mentioned object might result in serious resource leaks. Note that XQJ does not mandate XQJ-implementations to implement `finalize` methods to take care of automatically cleaning up external resources during garbage collection.

Applications should especially be careful to close objects in case of exception handling. A typical mistake is demonstrated in Example 77.

Assume that a connection `conn` and a string `es` containing the XQuery expression already exist.

```
XQExpression expr = null;
try {
  expr = conn.createExpression();
  XQResultSequence result = expr.executeQuery(es);
  // process the results ...
  expr.close();
} catch (XQException xe) {
  // handle the error
}
```

Example 77 – Potential resource leaks.

A potential resource leak in the previous example exists in case of exceptions thrown during the execution of the result processing. Example 78 shows how to handle this scenario by using a finally block.

```
XQExpression expr = null;
try {
  expr = conn.createExpression();
  XQResultSequence result = expr.executeQuery(es);
  // process the results ...
} catch (XQException xe) {
  // handle the error
} finally {
  if (expr != null) {
    try {expr.close())}
```

```
    catch (XQException xe) {
      // write to log
    }
  }
}
```

Example 78 – Avoiding potential resource leaks.

21 Appendix A: References

[Clark] "XML Namespaces", James Clark.
http://jclark.com/xml/xmlns.htm

[XQuery] XQuery 1.0: An XML Query Language
World Wide Web Consortium
W3C Recommendation 23 January 2007
http://www.w3.org/TR/2007/REC-xquery-20070123/

[XQuery DM] XQuery 1.0 and XPath 2.0 Data Model (XDM)
World Wide Web Consortium
W3C Recommendation 23 January 2007
http://www.w3.org/TR/2007/REC-xpath-datamodel-20070123/

[XQuery F&O] XQuery 1.0 and XPath 2.0 Functions and Operators
World Wide Web Consortium
W3C Recommendation 23 January 2007
http://www.w3.org/TR/2007/REC-xpath-functions-20070123/

[XQuery Serialization] XSLT 2.0 and XQuery 1.0 Serialization
World Wide Web Consortium
W3C Recommendation 23 January 2007
http://www.w3.org/TR/2007/REC-xslt-xquery-serialization-20070123/

[XQuery FS] XQuery 1.0 and XPath 2.0 Formal Semantics
World Wide Web Consortium
W3C Recommendation 23 January 2007
http://www.w3.org/TR/2007/REC-xquery-semantics-20070123/

[XML Schema] XML Schema, Parts 0, 1, and 2 (Second Edition)
World Wide Web Consortium
W3C Recommendation, 28 October 2004
http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/
http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/,
http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

[JNDI] Java Naming and Directory Interface (JNDI)
Sun Microsystems
http://java.sun.com/products/jndi/

-- End of Document --