# XQuery extensions: tutorial and reference

Qizx/open 1.1



## Table of Contents



This section describes the XQuest-specific extensions which can be used inside XQuery programs for querying XML Libraries and processing XML data:

- Miscellaneous extensions functions to provide desirable or missing features in the XQuery language.
- Full-text extensions.
- Particular semantics of access to XML Collections and Documents in collections.
- Java Binding mechanism, a powerful extension mechanism which allows Java methods to be called as XQuery extension functions.
- SQL Connectivity for access to relational databases.

# 1. General extension functions

These general purpose functions belong to the namespace denoted by the predefined "**x:**" prefix. The **x:** prefix refers to namespace "net.axyana.xquest.xquery.ext".

## 1.1. Serialization

Though *Serialization* - the process of converting XML nodes evaluated by XQuery into a stream of characters- is defined in the W3C specifications, there is no specific function for this purpose. The following function provides for this obvious need:

**x:serialize** can output a document or a node into XML, HTML, XHTML, or plain text, to a file or to the default output stream.

```
function x:serialize( $tree as element(),
                      $options as element(option) )
        as xs:string?
```

Serializes the tree element into marked-up text. The output can be a file, or the *default output* of the execution context.

**Parameter $tree:** a XML tree to be serialized to text.

**Parameter $options:** an element holding options in the form of attributes: see below.

**Returned value:** The path of the output file if specified, otherwise the empty sequence.

The options argument (which may be absent) has the form of an element of name "options" whose attributes are used to specify different options. For example:

```
x:serialize( $doc,
             <options output="out\doc.xml"
                      encoding="ISO-8859-1" indent="yes"/>)
```

This mechanism is similar to XSLT's xsl:output specification and is very convenient since the options can be computed or extracted from a XML document.

**Table 1. Implemented options**

| option name | values | description |
|---|---|---|
| method | XML (default) XHTML, HTML, or TEXT | output method |
| output / file | a file path | output file. If this option is not specified, the generated text is written to *default output*, which can be specified through the Java control API. |
| version | default "1.0" | version generated in the XML declaration. No validity check. |
| standalone | "yes" or "no". | No check is performed. |
| encoding | must be the name of an encoding supported by the JRE. | The name supplied is generated in the XML declaration. If different than UTF-8, it forces the output of the XML declaration. |
| indent | "yes" or "no". (default no) | output indented. |
| indent-value | integer value | (extension) specifies the number of space characters used for indentation. |
| omit-xml-declaration | "yes" or "no". (default no) | controls the output of a XML declaration. |
| include-content-type | "yes" or "no". (default no) | for XHTML and HTML methods, if the value is "yes", a META element specifying the content type is added at the beginning of element HEAD. |
| escape-uri-attributes | "yes" or "no". | for XHTML and HTML methods, escapes URI attributes. |
| doctype-public | the public ID in the DOCTYPE declaration. | Triggers the output of the DOCTYPE declaration. |
| doctype-system | the system ID in the DOCTYPE declaration. | Triggers the output of the DOCTYPE declaration. |

## 1.2. XSL Transformation

The **x:transform** function invokes a XSLT stylesheet on a node and can retrieve the results of the transformation as a tree, or let the stylesheet output the results.

This is a useful feature when one wants to transform a document (for example extracted from the XML Libraries) or a computed fragment of XML into different output formats like HTML, XSL-FO etc.

This example generates the transformed document $doc into a file out\doc.xml:

```
x:transform( $doc, "ssheet1.xsl",
             <parameters param1="one" param2="two"/>,
             <options output-file="out\doc.xml" indent="yes"/>)
```

The next example returns a new document tree. Suppose we have this very simple stylesheet which renames the element "doc" into "newdoc":

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                 version ="1.0" >
  <xsl:template match="doc">
     <newdoc><xsl:apply-templates/></newdoc>
  </xsl:template>
</xsl:stylesheet>
```

The following XQuery expression:

```
x:transform( <doc>text</doc>, "ssheet1.xsl", <parameters/> )
```

returns:

```
<newdoc>text</newdoc>
```

```
function x:transform( $source as node(),
                      $stylesheet-URI as xs:string,
                      $xslt-parameters as element(parameters)
                      [, $options as element(options)] )
    as node()?
```

Transforms the source tree through a XSLT stylesheet. If no output file is explicitly specified in the options, the function returns a new tree.

**Parameter $source:** a XML tree to be transformed. It does not need to be a complete document.

**Parameter $stylesheet-URI:** the URI of a XSLT stylesheet. Stylesheets are cached and reused for consecutive transformations.

**Parameter $xslt-parameters:** an element holding parameter values to pass to the XSLT engine. The parameters are specified in the form of attributes. The name of an attribute matches the name of a xsl:param declaration in the stylesheet (namespaces can be used). The value of the attribute is passed to the XSLT transformer.

**Parameter $options:** [optional argument] an element holding options in the form of attributes: see below.

**Returned value:** if the path of an output file is not specified in the options, the function returns a new document tree which is the result of the transformation of the source tree. Otherwise, it returns the empty sequence.

## Table 2. Available options

| option name | values | description |
|---|---|---|
| output-file | an absolute file path | output file. If this option is not specified, the generated tree is returned by the function, otherwise the function returns an empty sequence. |
| *XSLT output properties* (instruction xsl:output): version, standalone, encoding, indent, omit-xml-declaration etc. | | These options are used by the style-sheet for outputting the transformed document. They are ignored if no output-file option is specified. |
| Specific options of the XSLT engine (Saxon or default XSLT engine) | | An invalid option may cause an error. |

**Note about the efficiency of the connection with XSLT:**

- In XQuest Milestone1, the connection with an XSLT engine uses generic JAXP interfaces, and thus must copy XML trees passed in both directions. This is not as efficient as it could be and can even cause memory problems if the size of processed documents is larger then a fewdozen megabytes, depending on the available memory size.
- In later versions, XQuest will be tightly integrated with a specific XSLT2 implementation (namely Saxon). This will allow processing of very large XML documents (100Mb and more) with the benefit of indexed queries offered by XQuest.

## 1.3. Dynamic evaluation

The following functions allow dynamically compiling and executing XQuery expressions.

```
function x:eval( $expression as xs:string ) as xs:any
```

Compiles and evaluates a simple expression provided as a string.

The expression is executed in the context of the current query: it can use global variables, functions and namespaces of the current static context. It can also use the current item '.' if defined in the evaluation context.

However there is no access to the local context (for example if x:eval is invoked inside a function, the arguments or the local variables of the function are not visible.)

**Parameter $expression:** a simple expression (cannot contain prologue declarations).

**Returned value:** evaluated value of the expression.

Example:

```
declare variable $x := 1;
declare function local:fun($p as xs:integer) { $p * 2 };

let $expr := "1 + $x, local:fun(3)"
return x:eval($expr)
```

This should return the sequence (2, 6).

```
function x:expression (
          $module-uri as xs:string
        [ , $option-name as xs:string, $option-value as xs:any ]* )
     as xs:any
```

Loads and executes a XQuery expression from an external URI.

**Note: this function is not available in XQuest Milestone 1.**

**Parameter $module-uri:** the URI of a XQuery main module. It is is resolved in the same way as XQuery modules (n the "at" clause of **import module**). The "Module Manager" of the underlying implementation is in charge of loading and caching dynamically loaded expressions.

**Parameter $option-name:** [repeatable argument] a string containing the name of an option or a global-variable to pass to the invoked expression. See below for details. This can be a QName with a namespace prefix: *the prefix is resolved in the context of the invoked expression*.

**Parameter $option-value:** [repeatable argument] any value passed for the option whose name precedes.

**Returned value:** the result of the executed expression.

Example:

```
x:expression( "query1.xq",
              "x:input", $node,
              "global1", 1,   "global2", "value2" )
```

Invokes the XQuery scripts contained in query1.xq, passing $node as default input for the query, and intializaing its global variable $global1 to the value 1 and $global2 to "value2".

## 1.4. Pattern-matching

The following functions match the string-value of nodes (elements and attributes) with a pattern.

Example 1: this expression returns true if the value of the attribute @lang matches the SQL-style pattern:

```
x:like( "en%", $node/@lang )
```

Example 2: this expression returns true if the content of the element 'NAME' matches the pattern:

```
$p/NAME[ x:like( "Theo%" ) ]
```

```
function x:like( $pattern as xs:string
                 [, $context-nodes as node()* ])
     as xs:boolean
```

Returns true if the pattern matches the string-value of at least one node in the node sequence argument.

**Parameter $pattern:** a SQL-style pattern: the wildcard '_' matches any single character, the wildcard '%' matches any sequence of characters.

**Parameter $context-nodes:** optional sequence of nodes. The function checks sequentially the string-value of each node against the pattern. If absent, the argument default to '.', the current item. This makes sense inside a predicate, like in the example 2 above.

**Returned value:** a boolean.

```
function x:ulike( $pattern as xs:string
                  [, $context-nodes as node()* ])
     as xs:boolean
```

This function is very similar to x:like, except that the pattern has syntax à la Unix ("glob pattern"). The character '?' is used instead of '_' (single character match), and '*' instead of '%' (multi-character match).

**Note**: these functions - as well as the standard fn:matches function, and the full-text functions - are automatically recognized by the query optimizer which uses library indexes to boost their execution whenever possible.

## 1.5. Date and Time

### 1.5.1. Differences with W3C specifications

XQuest 1.0 and Qizx/open 1.1 are compliant with the W3C Recommendation, and therefore slightly incompatible with former versions of Qizx/open.

The only differences at present are extensions of the cast operation: Qizx/open can directly cast date, time, dateTime and durations to and from double values representing seconds, and keeps the extended "constructors" that build date, dateTime etc. from numeric components like days, hours, minutes etc.

### 1.5.2. Cast Extensions

In order to make computations easier, Qizx/open can:

- Cast xdt:yearMonthDuration to numeric values: this yields the number of months. The following expression returns 13:

  ```
  xdt:yearMonthDuration("P1Y1M") cast as xs:integer
  ```

- Conversely, cast numeric value representing months to xdt:yearMonthDuration. The following expression holds true:

```
xdt:yearMonthDuration(13) = xdt:yearMonthDuration("P1Y1M")
```

- Cast xdt:daytimeDuration to double: this yields the number of seconds. The following expression returns 7201:

```
xdt:dayTimeDuration("PT2H1S") cast as xs:double
```

- Conversely, cast a numeric value representing seconds to xdt:daytimeDuration.
- Cast xs:dateTime to double. This returns the number of seconds elapsed since "the origin", i.e. 1970-01-01T00:00:00Z. If the timezone is not specified, it is considered to be UTC (GMT).
- Conversely, cast a numeric value representing seconds from the origin to a dateTime with GMT timezone.
- cast from/to the xs:date type in a similar way (like a dateTime with time equal to 00:00:00).

```
xdt:date("1970-01-02") cast as xs:double = 86400
```

- cast from/to the xs:time type in a similar way (seconds from 00:00:00).

```
xdt:time("01:00:00") cast as xs:double = 3600
```

### 1.5.3. Additional constructors

These constructors allow **date, time, dateTime** objects to be built from numeric components (this is quite useful in practice).

```
function xs:date( $year as xs:integer, $month as xs:integer,
                  $day as xs:integer )
         as xs:date
```

builds a date from a year, a month, and a day in integer form. The implicit timezone is used.

For example xs:date(1999, 12, 31) returns the same value as xs:date("1999-12-31").

```
function xs:time( $hour as xs:integer, $minute as xs:integer,
                  $second as xs:double )
         as xs:time
```

builds a xs:time from an hour, a minute as integer, and seconds as double. The implicit timezone is used.

```
function xs:dateTime( $year as xs:integer, $month as xs:integer,
                      $day as xs:integer, $hour as xs:integer,
                      $minute as xs:integer, $second as xs:double
                      [, $timezone as xs:double] )
         as xs:dateTime
```

builds a dateTime from the six components that constitute date and time.

A timezone can be specified: it is expressed as a signed number of hours (ranging from -14 to 14), otherwise the implicit timezone is used.

### 1.5.4. Additional accessors

These functions are kept for compatibility. They are slightly different than the standard functions:

- they accept several date/time and durations types for the argument (so for example we have get-minutes instead of get-minutes-from-time, get-minutes-from-dateTime etc.),

- but they do not accept untypedAtomic (node contents): such an argument should be cast to the proper type before being used. So the standard function might be as convenient here.

```
function get-seconds( $moment )
     as xs:double?
```

Returns the "second" component from a xs:time, xs:dateTime, and xs:duration.

Can replace fn:seconds-from-dateTime, fn:seconds-from-time, fn:seconds-from-duration, except that the returned type is double instead of decimal, and an argument of type xdt:untypedAtomic is not valid.

```
function get-all-seconds( $duration )
     as xs:double?
```

Returns the total number of seconds from a xs:duration. This does not take into account months and years, as explained above.

For example `get-all-seconds( xs:duration("P1YT1H"))` returns 3600.

```
function get-minutes( $moment )
     as xs:integer?
```

Returns the "minute" component from a xs:time, xs:dateTime, and xs:duration.

```
function get-hours( $moment )
     as xs:integer?
```

Returns the "hour" component from a xs:time, xs:dateTime, and xs:duration.

```
function get-days( $moment )
     as xs:integer?
```

Returns the "day" component from a xs:date, xs:dateTime, xs:day, xs:monthDay and xs:duration.

```
function get-months( $moment )
   as xs:integer?
```

Returns the "month" component from a xs:date, xs:dateTime, xs:yearMonth, xs:month, xs:monthDay and xs:duration.

```
function get-years( $moment )
    as xs:integer?
```

Returns the "year" component from a xs:date, xs:dateTime, xs:year, xs:yearMonth and xs:duration.

```
function get-timezone( $moment )
    as xs:duration?
```

Returns the "timezone" component from any date/time type and xs:duration.

The returned value is like timezone-from-* except that the returned type is xs:duration, not xdt:dayTimeDuration.

## 1.6. Error handling

XQuery has currently no mechanism to handle run-time errors.

Actually the language is such that an error handling is not absolutely mandatory: many errors need not be recovered (for example type errors); the `doc()` function which, can generate a dynamic error, is now protected by a new function doc-available().

However, extensions (namely the Java binding mechanism) can generate errors. It is not possible to provide a protection auxiliary like doc-available() for every functionality.

Qizx/open provides a try/catch construct, which is a syntax extension. This construct has several purposes.

```
try { expr } catch($error, ) { fallback-expr }
```

The **try/catch** extended language construct first evaluates the body *expr*. If no error occurs, then the result of the try/catch is the return value of this expression.

If an error occurs, the local variable *$error* receives a string value which is the error message, and *fallback-expr* is evaluated (with possible access to the error message). The resulting value of the try/catch is in this case the value of this fallback expression. An error in the evaluation of the fallback-expression is not caught.

The type of this expression is the type that encompasses the types of both arguments.

### Important

The body (first expression) is guaranteed to be evaluated completely before exiting the try/catch - unless an error occurs. In other terms, lazy evaluation, which is used in most Qizx/open expressions, does not apply here.

This is specially important when functions with side-effects are called in the body. If such functions generate errors, these errors are caught by the try/catch, as one can expect. Otherwise lazy evaluation could produce strange effects.

Example: tries to open a document, returns an element `'error'` with an attribute containing the error message if the document cannot be opened.

```
try {
    doc("unreachable.xml")
}
catch($err) {
    <error msg="{$err}"/>
}
```

## 1.7. Miscellaneous functions

```
function x:system-property( $name as xs:string )
         as item()?
```

Returns the value of a "system" or application property. Similar to the function with same name in XSLT.

Additional properties can be defined through the Java API.

**Parameter $name:** name of the system property.

**Returned value:** value of the property, or empty sequence if unknown property name.

Predefined properties:

- **"vendor"** : name of the vendor.
- **"vendor-url"** : URL of the vendor's site, here "http://www.axyana.com/xquest".
- **"product-name"** : here "XQuest".

- "**product-version**" : the current version in string form.

Note: the set of predefined properties can be enriched through the Java API (see the related documentation).

# 2. Full-text functions

XQuest supports contextual full-text search through extensions functions.

Contextual (or context-sensitive) full-text search means that text patterns can be searched in the context of specific XML elements. For example, it is possible to express queries like:

- find the SECTION elements whose TITLE child element contains the word "hazard":

```
collection(...)//SECTION [ x:fulltext(" hazard ", TITLE) ]
```

- find the TABLE elements where the phrase "first try" occurs in a cell of the first column:

```
collection(...)//TABLE [ x:fulltext(" 'first try' ", ROW/CELL[1]) ]
```

When used to search XML Libraries, full-text search functions are index-based and therefore very efficient.

These functions can also be used to search parsed documents or even constructed fragments, but in this case the execution speed is significantly lower (as an indication, the entire works of Shakespeare - 8 Mb of XML - can be scanned in about one second on a 3 GHz processor).

Full-text functions return a boolean value (is there a match for the full-text expression?).

They are typically used inside a XPath predicate.

## 2.1. General full-text function

Signatures:

```
function x:words( $query as xs:string
               [, $context-nodes as node()* ] )
     as xs:boolean
```

```
function x:fulltext( $query as xs:string
                   [, $context-nodes as node()* ] )
     as xs:boolean
```

Note: `x:words` and `x:fulltext` are aliases of the same function.

This function implements context-sensitive full-text search: it can search boolean combinations of words, word patterns and phrases, in the context of specific elements. It is typically used inside a predicate.

For example the following expression returns SPEECH elements which contain both words "romeo" and "juliet":

```
//SPEECH [ x:words(" romeo AND juliet ") ]
```

**Returned value:** The function returns true if the string-value of at least one node of the *`context-nodes`* parameter matches the full-text query. Matching is therefore not affected by element substructure (mixed content). For example the phrase 'to be or not to be' would be found in `<line>To be <b>or not to be</b> ..</line>`.

**Parameter $query:** a string containing a full-text pattern (see the syntax below).

**Parameter $context-nodes:** A node sequence of nodes inside which the full-text expression is searched for.

When *context-nodes* is not specified (it must be inside a predicate), the current context node '.' is used implicitly like in the example above. When *context-nodes* parameter is present, it can be relative to the current context node: for example this expression finds SPEECH elements which contain a LINE element which in turn contains both words "romeo" and "Juliet":

```
//SPEECH [ x:words(" romeo AND juliet ", LINE) ]
```

**Syntax of full text queries:**

**Simple term**
A word without the wildcard characters '*' and '?'. By default case and accents are ignored (i.e. "café is equivalent with "CAFE").

**Term with wildcard**
Wildcard characters '%' and '_' make a SQL-style pattern. The underscore matches a single character, the percent sign matches the longest possible sequence of characters.

For example "intern%" would match intern, internal, internals etc.

**Approximate term**
Notation: *word~*. Uses a generic phonetic distance algorithm (somewhat similar to Soundex). Experimental feature.

**Term alternative**
Notation: *term1 OR term2*. The operator *OR* or the sign '|' can be used. It *has precedence over* AND (see below).

**Term conjunction**
Notation: *term1 AND term2*. The operator *AND*, the sign '&', or even simple juxtaposition can be used: thus "romeo AND juliet", "romeo & Juliet" and "Roméo Juliet" are equivalent.

**Term exclusion**
Notation: sign *'-'* or keyword *NOT*. For example "Romeo -Juliet" is equivalent to "Romeo AND NOT Juliet".

**Phrase**
Ordered sequence of terms (simple words or patterns), surrounded by single or double quotes. By default, terms must appear exactly in the order specified.

It is possible to specify a tolerance or distance, which is the maximum number of words interspersed among the terms of the phrase query. The notation is *phrase~N* where N is a optional count of words (4 by default). The two following examples match the phrase "to be or not to be, that is the question":

```
//SPEECH [ x:words(" 'to be that question'~ ", LINE) ]
//SPEECH [ x:words(" 'to be or question'~6 ", LINE) ]
```

Notice that there are some limitations in this syntax: the OR cannot combine AND clauses or phrases, however this limitation can be solved by a boolean combination of calls to x:words, for example:

```
doc("r_and_j.xml")//LINE [ x:words("name AND rose")
                           or x:words(" 'smell as sweet' ") ]
```

would yield the two lines (Romeo and Juliet, act II scene 2):

```
<LINE>What's in a name? that which we call a rose</LINE>
<LINE>By any other name would smell as sweet;</LINE>
```

## 2.2. Convenience full-text functions

These functions are used for helping specialized text search.

```
function x:phrase( $words as xs:string+
                  [, $spacing as xs:integer ]
                  [, $context-nodes as node()* ] )
    as xs:boolean
```

Convenience function: A variant of x:fulltext specialized in phrase search, which allows words to be specified as a sequence of strings.

For example:

```
x:phrase( ("to", "be", "or", "not"), 5 )
```

as well as:

```
x:phrase( ("to be", "or not"), 5 )
```

are equivalent to:

```
x:fulltext(" 'to be or not'~5 ")
```

```
function x:all-words( $words as xs:string+
                    [, $context-nodes as node()* ] )
    as xs:boolean
```

Convenience function: a variant of x:fulltext which allows words to be specified as a sequence of strings.

For example:

```
x:all-words( ("romeo", "juliet"), LINE )
```

is equivalent to

```
x:fulltext("romeo AND juliet", LINE)
```

```
function x:any-word( $words as xs:string+
                   [,$context-nodes as node()* ] )
    as xs:boolean
```

Convenience function: a variant of x:fulltext which allows words to be specified as a sequence of strings.

For example:

```
x:any-word( ("romeo", "juliet"), LINE )
```

is equivalent to

```
x:fulltext("romeo OR juliet", LINE)
```

```
function x:highlighter( $query as xs:string,
                        $fragment as element(),
                        $parts as node()*,
                        $options as element(option) ] )
    as element()
```

This function is a companion of the full text search functions, which "highlights" matched terms.

Precisely it returns a copy of a document fragment where matched terms are surrounded by generated elements (which are typically used for displaying).

By default a generated element has the name 'span' and an attribute 'class' with a value equal to the prefix 'hi' followed by the rank of the term in the query.

Applied to a LINE in the example LINE [ x:words("name OR rose") ], this would produce something like:

```
<LINE>What's in a <span class='hi0'>name</span>?
 that which we call a <span class='hi1'>rose</span></LINE>
```

The first argument of this function is a full text query.

The second argument is the root of the document fragment to process.

The optional third argument $parts is a list of sub-elements of the root which must be specifically highlighted (if empty, the whole root fragment is highlighted, otherwise only the specified parts are highlighted).

The 4th argument specifies options: it allows redefining of the generated elements. For example:

```
<options element='frag' attribute='style' prefix='st'/>
```

would surround terms with `<frag style="st0"></frag>` instead of `<span class='hi0'></span>`.

# 3. Java binding mechanism

The *Java binding* feature is a powerful extensibility mechanism which allows direct calling of Java methods bound as XQuery functions and manipulation of wrapped Java objects.

This opens a tremendous range of possibilities since nearly all the Java APIs become accessible. The implementation performs many automatic conversions, including Java arrays and some Java collections.

Qizx Java Binding is similar to the mechanism introduced by several other XQuery or XSLT engines like XT or Saxon: a qualified function name where the namespace URI starts with "java:" is automatically treated as a call to a Java method.

- The namespace URI must be of the form *java:fullyQualifiedClassName*. The designated class will be searched for a method matching the name and arguments of the XQuery function call.
- The XQuery name of the function is modified as follows: hyphens are removed while the character following an hyphen is upper-cased (producing 'camelCasing'). So "get-instance" becomes "getInstance".

In the following example the **getInstance()** method of the class **java.util.Calendar** is called:

```
declare namespace cal = "java:java.util.Calendar"
cal:get-instance()    (:  or cal:getInstance()  :)
```

The mechanism is actually a bit more flexible: a namespace can also refer to a package instead of a class name. The class name is passed as a prefix of the function name, separated by a dot. For example:

```
declare namespace util = "java:java.util"
util:Calendar.get-instance()
```

The following example invokes a constructor, gets a wrapped File in variable $f, then invokes the non-static method mkdir():

```
declare namespace file = "java:java.io.File"

let $f := file:new("mynewdir")
return file:mkdir($f)
```

In this example we list the files of the current directory with their sizes and convert the results into XML :

```
declare namespace file = "java:java.io.File"

for $f in file:listFiles( file:new(".") )    (: or list-files() :)
return
    <file name="{ $f }" size="{ file:length($f) }"/>
```

**Static and instance methods:**

A static Java method must be called with the exact number of parameters of its declaration.

A non-static method is treated like a static method with an additional first argument ('this'). The additional first actual argument must of course match the class of the method.

**Constructors:**

A constructor of a class is invoked by using the special function name "`new`". A wrapped instance of the class is returned and can be handled in XQuery and passed to other Java functions or to user-defined XQuery functions. For example:

```
declare namespace file = "java:java.io.File";
 file:new("afile.txt")
```

Overloading on constructors is possible in the same way as on other methods.

**Wrapped Java objects**

Bound Java functions can return objects of arbitrary classes which can then be passed as arguments to other functions or stored in variables. The type of such objects is `xdt:object` (formerly `xs:wrappedObject`). It is always possible to get the string value of such an object [invokes the Java method `toString()`.]

**Type conversions:**

Parameters are automatically converted from XQuery types to Java types. Conversely, the return value is converted from Java type to a XQuery type.

Basic Java types are converted to/from corresponding XQuery basic types.

Since the XQuery language handles *sequences* of items, special care is given to Java arrays which are mapped to and from XQuery sequences. In addition, a Vector, ArrayList or Enumeration returned by a Java method is converted to a XQuery sequence (each element is converted individually to a XQuery object).

See below the type conversion chart.

**Overloading**

Overloaded Java methods are partially supported:

- When two Java methods differ by the number of arguments, there is no difficulty. XQuery allows functions with the same name and different number of arguments.

- when two Java methods have the same name and the same number of arguments, there is no absolute guaranty which method will be called, because XQuery is a weakly typed language, so it is not always possible to resolve the method based on static XQuery types (Resolution at run-time would be possible but much more complex and possibly fairly inefficient).

  However, from Qizx/db 0.2p5and Qizx/open 1.1p4, static argument types can be used to find the best matching Java method. For example, assume you bind the following class:

  ```
  class MyClass {
      String myMethod(String sarg) ...
      int myMethod(double darg) ...
  }
  ```

  Then you can call the **myMethod** (or **my-method**) function in XQuery with arguments which have a known static type and be sure which Java method is actually called:

  ```
  declare namespace myc = "java:MyClass"
  myc:my-method(1)   (: second Java method is called :)
  myc:my-method("string")   (: first Java method is called :)
  ```

  1. in the first call, the argument type is xs:integer for which the closest match is Java double, so the second method is called.

  2. In the second call, the argument type is xs:string which matches String perfectly, so the first method is called.

Of course it is possible to use XQuery type declarations or constructs like **cast as** or **treat as** to statically specify the type of arguments:

```
declare function local:fun($s as xs:string) {
   myc:my-method($s)   (: first Java method is called :)
}
```

or:

```
 myc:my-method($s treat as xs:string)   (: first Java method is called :)
```

**Limitations**: there are still some limitations when in both methods the argument types is any non-mappable Java class (xdt:object in XQuery):

```
class MyClass {
    Object myMethod2(ClassA arg) ...
    int    myMethod2(ClassB arg) ...
}
```

In that case there is currently no way in Qizx to specify the static type of the actual argument, so the result is inpredictable and may result in a run-time error.

### Table 3. Types conversions

| Java type | XML Query type |
|---|---|
| void (return type) | empty() |
| String | xs:string |
| boolean, Boolean | xs:boolean |
| double, Double | xs:double |
| float, Float | xs:float |
| long, Long | xs:integer |
| int, Integer | xs:int |
| short, Short | xs:short |
| byte, Byte | xs:byte |
| char, Char | xs:integer |
| net.axyana.xquest.xquery.dm.Node | node() |
| other class | xdt:object |
| String[ ] | xs:string * |
| double[ ], float[ ] | xs:double * |
| long[ ], int[ ], short[ ], byte[ ], char[ ] | xs:integer * |
| net.axyana.xquest.xquery.dm.Node[ ] | node()* |
| other array | xdt:object * |
| java.util.Enumeration, java.util.Vector, java.util.ArrayList (*return value only*) | xdt:object * |

The Java binding mechanism is widely used in the XQuest extensions hereafter.

# 4. Collections and Documents

In the XQuery standard, Documents and Collections are the basic entities that can contain XML data.

The standard defines two functions fn:doc and fn:collection that allow access to the nodes of a document or a collection respectively. However the semantics of these functions are largely implementation-dependent.

This section describes the particular semantics of Collections and Documents in XQuest.

## 4.1. Generalities

This section is simply a reminder of how documents and collections are used in XQuery and XPath2.

Tree queries in the XQuery language are based on *Path Expressions* (called *Location Paths* in XPath 1). Paths are built from *steps* composed with the slash operator.

- The expression at the root of a Path Expression is generally a document, a collection, or a node obtained from an expression:

  ```
  for $invoice in collection("invoices")/invoice
  for $client in doc("clients.xml")/client
   where $invoice/client-id = $client/@id
  return ....
  ```

  In the example above we have four path expressions, one with a collection at its root, one with a document, the other two with nodes obtained from the first two paths.

- The root can also be a *sequence* of documents, collections, or nodes. The following expressions are also valid paths, though less usual:

  ```
  (collection("invoices2003"), collection("invoices2004"))
      /invoice

  (doc("doc1.xml"), doc("doc2.xml")//clients)/client-id
  ```

- Finally, the root can be implicit: the path begins with '/' or '//', or with a simple "step".

  There are two cases here:

  - If there is a *current item* (which must be a node), then the current item is used as origin (or its document-node if the path starts with '/').
  - **XQuest extension**: if no current item is defined but a *Default Input* is specified through the API, then this default input is used as a root.

    Note: if a current item is defined, it is still possible to access the default input through the function **input()**.

    Example: in the command-line tool xquestcli, the option -input <path> parses the XML document and sets it as the default input. It can be accessed by a path (for example `/DOC`) or by the function input().

The XQuery Specifications define two basic functions for accessing XML collections and documents:

```
fn:doc($uri as xs:string?)
    as document?
```

Retrieves a document using an xs:anyURI supplied as an xs:string.

```
fn:collection($uri as xs:string?)
    as node()*
```

Takes a xs:string as argument and returns a sequence of nodes obtained by interpreting $uri as an xs:anyURI and resolving it.

The way the URI are resolved into actual documents or collections is considered implementation-dependent by the XQuery standard.

The particular semantics of these two function in Qizx/open are explained below.

## 4.2. Function fn:doc

```
fn:doc( $uri as xs:string? )
    as document?
```

Retrieves a document using an xs:anyURI supplied as an xs:string.

The document is parsed from XML source text.

- If the URI specified by the argument has a protocol supported by Java (for example http:), it is directly opened for parsing.
- If the URI is an absolute file path (without protocol), it is opened as a file.
- Otherwise, the Document Manager is in charge of resolving the URI: the default implementation uses a base URI to resolve the relative document URI. Java applications may plug other implementations using a different policy (for example a multiple path, similar to Java class paths.)

In any case, if the document cannot be found, or if a parsing error occurs, the function raises an error (which can be caught using the try/catch extension).

The execution environment uses an object called *Document Manager* to parse and cache documents. Therefore the doc function will not always parse the document, if it found in the cache. The cache can be configured through the Java API.

## 4.3. Function fn:collection

This standard function is implemented in a simple way: the argument can be a list of document URI's, separated by commas or semicolons. The collection (in the sense of XQuery) is a sequence formed by the root-nodes of these documents, in the specified order.

```
fn:collection($paths as xs:string)
    as node()*
```

**Parameter $path-pattern:** Comma- or semicolon- separated list of document URI's. Each document is resolved as per the doc() function. Whitespace is significant.

**Returned value:** a sequence of document-nodes.

Example:

```
xlib:collection("/2005/IMOLA/EVENT.XML;/2006/MONTREAL/EVENT.XML;/2006/MAGNY-COURS/EVENT.XML")
```

or alternately:

```
xlib:collection("/2005/IMOLA/EVENT.XML,/2006/MONTREAL/EVENT.XML,/2006/MAGNY-COURS/EVENT.XML")
```

# 5. SQL Connectivity

The "SQL Connectivity" is an extension which allows querying of data from relational databases, using SQL, and transform it on-the-fly into XML, providing a simple facility to merge relational data into XML documents.

This is a rough-and-ready implementation, mainly based on JDBC (Java Database Connectivity) and the Java binding mechanism, which does not attempt to compile XQuery into SQL.

We assume in this section that the reader has some knowledge of SQL and JDBC.

Basically, the SQL Connection provides functions which take a SQL query statement as an argument and return the result set as a sequence of "row" XML elements. These elements have sub-elements corresponding with the columns/fields returned by the SQL query.

**A simple example**: we assume the existence of a MySQL database "db1" with a table "pets" containing the description of a few animals. Here is a XQuery expression which opens a connection to the database and queries the table:

```
let $conn := sqlx:get-connection("jdbc:mysql://localhost/db1",
                                 "user", "password")
return
     sqlx:execute($conn, "SELECT name, species, weight FROM pets")
```

Returning results like:

```
<row>
    <name>Adalbert</name>
    <species>cat</species>
    <weight>2.5</weight>
</row>
<row>
    <name>Bertha</name>
    <species>boa constrictor</species>
    <weight>45</weight>
</row>
```

The function has returned two row elements each containing the three sub-elements name, species, and weight for the corresponding fields used in the SQL query.

You may also want to produce a HTML table with this data:

```
let $conn := sqlx:get-connection("jdbc:mysql://localhost/db1",
                                 "user", "password")
return
  <table width="80%">
   <tr><th>Name</th><th>Species</th><th>Weight</th></tr>
  {
   for $row in
     sqlx:execute($conn, "SELECT name, species, weight FROM pets")
   return <tr>
            <td>{ $row/name/text() }</td>
            <td>{ $row/species/text() }</td>
            <td>{ $row/weight/text() }</td>
          </tr>
  }
 </table>
```

**Notes:**

- To run the examples above it is assumed that your JRE has access to a JDBC driver for MySQL (or any other JDBC enabled database). For this you must have the jar of the driver in your class-path and you must register the driver (either by using the system property jdbc.drivers or by explicitly registering the driver with the helper function sqlx:register-driver() -- see below).

- The namespace prefix **sqlx** is predefined: it is used for the functions of the SQL extension. Some other namespaces are predefined for convenient access to the main JDBC classes or interfaces: **sqlc** for java.sql.Connection, **sqlr** for java.sql.ResultSet, and **sqlp** for java.sql.PreparedStatement.

- The object returned by sqlx:get-connection is simply the Connection object of JDBC, therefore all its methods are accessible through the Java binding mechanism. For example, `sqlc:commit($conn)` would call the commit method of interface java.sql.Connection (remember that **sqlc:** is a predefined namespace prefix for java.sql.Connection).

  As a consequence the connection can be obtained by other means, for example using JDBC DataSources, of course still using the Java binding mechanism.

## 5.1. Prepared and callable statements

JDBC has a notion of Prepared or Callable Statement, which is a precompiled query or update instruction with place-holders for parameterized values.

XQuest's SQL Connection API makes it easy to pass actual parameters to such statements:

```
declare variable $QUERY := text {
         SELECT * from pets WHERE name = ?
};

let $conn := sqlx:get-connection("jdbc:mysql://localhost/db1",
                                 "user", "password"),
    (: prepare a statement for later execution :)
    $prStatement := sqlx:prepare( $conn, $QUERY )
return
     sqlx:execStatement($prStatement, "Adalbert")
```

This would return the first row matching the parameter value "Albert" for the field "name":

```
<row>
    <name>Adalbert</name>
    <species>cat</species>
    <weight>2.5</weight>
</row>
```

More generally, several parameters can be passed this way in the same call to `sqlx:execute` (this function accepts a variable number of arguments). The parameter types must be compatible with the expected field types, otherwise an error is raised.

To illustrate this, let's introduce another function: `sqlx:execUpdate` (the equivalent of the `JDBC executeUpdate` method of PreparedStatement):

```
declare variable $STATEMENT := text {
         INSERT INTO pets VALUES (?, ?, ?)
};
let (: prepare a statement for later execution :)
    $insert := sqlx:prepare( $conn, $STATEMENT )
return (
     sqlx:execUpdate($insert, "Donald", "duck", 4),
     sqlx:execUpdate($insert, "Pooh", "bear", ())
   )
```

This piece of code inserts two new rows into the pets table.

- sqlx:execUpdate returns an integer which is the number of rows affected.
- Each parameter must evaluate as a single item or an empty sequence.
- The empty sequence stands for a NULL field value.

Notice the useful trick of typing SQL inside a text constructor: this saves escaping of quote characters.

## 5.2. Other query invocation style

The function `sqlx:execute` provides an easy way to perform the fusion of tabular data by manipulating SQL query results as XML elements.

However a finer and maybe more efficient style may also be desirable.

This is achieved by the function `sqlx:rawExec` : it allows directly dealing with the ResultSet interface and retrieving of field values as typed items, instead of plain text.

The HTML example above could be changed like this, converting a weight from kilograms to pounds:

```
let $conn := sqlx:get-connection("jdbc:mysql://localhost/db1",
                                 "user", "password")
return
  <table width="80%">
   <tr><th>Name</th><th>Species</th><th>Weight</th></tr>
  {
   for $r in
       sqlx:rawExec($conn, "SELECT name, species, weight FROM pets")
   return <tr>
            <td>{ sqlr:getString($r, 1) }</td>
            <td>{ sqlr:getString($r, "species") }</td>
            <td>{ sqlr:getDouble($r, "weight") / 0.463 }</td>
          </tr>
  }
 </table>
```

Notes:

- sqlr:getString for example is a binding of the method ResultSet.getString(int columnIndex). The prefix sqlr is predefined for ResultSet.
- The "get" methods can of course be called either with an integer columnIndex or a String column name.
- The "update" methods can be called.
- Methods which move the cursor of the result set (first, beforeFirst) will most probably have undesirable effects.

## 5.3. Function Reference

Only XQuest-specific functions are described here. For functions which are direct bindings of JDBC methods, please refer to the JDBC documentation.

```
function sqlx:get-connection(
            $dbURL as xs:string,
            $user as xs:string, $passwd as xs:string)
    as xdt:object[Connection]
```

A thin wrapper on the DriverManager.getConnection method.

**Parameter $dbURI:** a database url accepted by JDBC.

**Parameter $user:** database user name.

**Parameter $user:** database user password.

**Returned value:** a wrapped Connection object. If no connection can be opened, an error is raised.

Note: the connection can be closed by `sqlc:close($conn)`. There is no specific function for this purpose.

```
function sqlx:prepare( $connection as xdt:object,
                       $statement as xs:string )
    as xdt:object[PreparedStatement]
```

A thin wrapper on the Connection.prepareStatement(String) method.

**Parameter $connection:** a valid database connection.

**Parameter $statement:** a SQL statement.

**Returned value:** a wrapped PreparedStatement object. An error can be raised on an erroneous statement.

```
function sqlx:execute( $connection as xdt:object,
                       $statement as xs:string )
    as element()*
```

```
function sqlx:execStatement( $preparedStatement as xdt:object,
                             $param1 as item()?, $param2... )
    as element()*
```

Query execution, accepting simple or prepared statement.

**Parameter $connection:** a valid database connection.

**Parameter $preparedStatement:** a prepared SQL statement.

**Parameters $param1 ...:** values passed as parameters of the SQL statement.

**Returned value:** a sequence of nodes. Each node has the name "row" (without namespace) and contains as many children elements as there are fields in the query result. Each child element has the name of a queried field.

A field with a NULL value appears as an empty child element.

```
function sqlx:execUpdate( $connection as xdt:object,
                          $statement as xs:string )
    as xs:integer
```

```
function sqlx:execUpdateStatement( $preparedStatement as xdt:object,
                                   $param1 as item()?, $param2... )
    as xs:integer
```

Update statement execution, accepting a simple or prepared statement.

**Parameter $connection:** a valid database connection.

**Parameter $preparedStatement:** a prepared SQL statement.

**Parameters $param1 ...:** values passed as parameters of the SQL statement.

**Returned value:** an integer which is the number of rows affected.

```
function sqlx:rawExec( $connection as xdt:object,
                       $statement as xs:string )
    as xdt:object*
```

```
function sqlx:rawExecStatement( $preparedStatement as xdt:object,
                                $param1 as item()?, $param2... )
    as xdt:object*
```

Raw statement execution, accepting a simple or prepared statement.

This function returns a wrapped ResultSet sequence. In order to iterate on this result set, *it is compulsory to use a **for** loop*: for $rs in sqlx:rawExec(...) return ...

**Parameter $connection:** a valid database connection.

**Parameter $preparedStatement:** a prepared SQL statement.

**Parameters $param1 ...:** values passed as parameters of the SQL statement.

**Returned value:** a sequence of ResultSet states.